



Performance and programmability comparison of the thick control flow architecture and current multicore processors

Martti Forsell¹ · Sara Nikula¹ · Jussi Roivainen¹ · Ville Leppänen² · Jesper Larsson Träff³

Accepted: 1 July 2021
© The Author(s) 2021

Abstract

Commercial multicore central processing units (CPU) integrate a number of processor cores on a single chip to support parallel execution of computational tasks. Multicore CPUs can possibly improve performance over single cores for independent parallel tasks nearly linearly as long as sufficient bandwidth is available. Ideal speedup is, however, difficult to achieve when dense intercommunication between the cores or complex memory access patterns is required. This is caused by expensive synchronization and thread switching, and insufficient latency toleration. These facts guide programmers away from straight-forward parallel processing patterns toward complex and error-prone programming techniques. To address these problems, we have introduced the Thick control flow (TCF) Processor Architecture. TCF is an abstraction of parallel computation that combines self-similar threads into computational entities. In this paper, we compare the performance and programmability of an entry-level TCF processor and two Intel Skylake multicore CPUs on commonly used parallel kernels to find out how well our architecture solves these issues that greatly reduce the productivity of parallel software development. Code examples are given and programming experiences recorded.

Keywords Parallel computing · Multiprocessors · Thick control flow · Performance comparison · Programmability

1 Introduction

Multicore *Central Processing Units* (CPUs) are the workhorses of modern general purpose computing devices, such as workstations, tablets and smartphones. They were taken into commercial use almost 20 years ago when it became evident that the clock speeds of single core CPUs could not be increased any more due to power

✉ Martti Forsell
Martti.Forsell@VTT.Fi

Extended author information available on the last page of the article

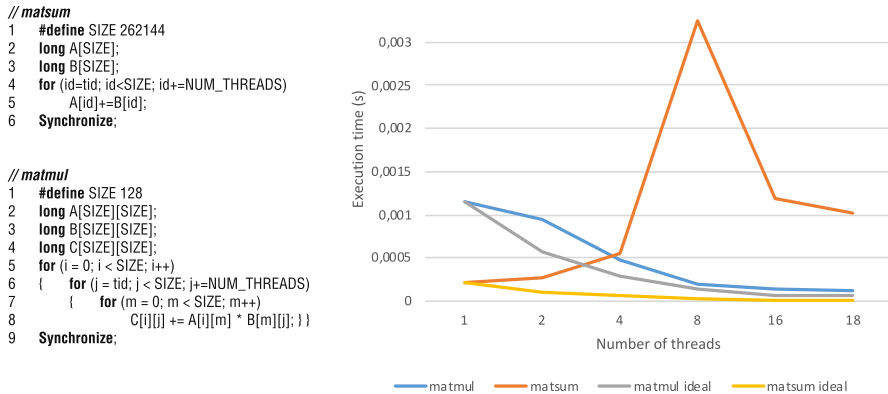


Fig. 1 Left The *matsum* and *matmul* Pthreads programs (*tid* = thread identifier, *NUM_THREADS* = number of threads, *Synchronize* = barrier synchronization). Right The execution time of the *matmul* and *matsum* test programs in a 18-core Intel Skylake Xeon W CPU as a function of the number of parallel threads. Additional curves illustrating ideal linear speedup behavior are shown

density issues [1]. The main idea of multicore CPUs was to integrate multiple processor cores on a single chip and use those cores for concurrent computation such that a P -core chip would execute P times more instructions within the same time [2]. At best the original application would run P times faster than in a single core processor. A precondition for the linear increase in performance is that the application can be expressed as a *parallel program* that can be compiled to an executable controlling the cores participating in computation. Multicore CPUs seemed a promising way to continue increasing the performance of processors since Moore's Law, predicting doubling of the number of transistors per silicon area unit in a fixed time period (18–24 months, e.g.), was valid at that time [1].

Multicore CPUs have indeed turned out to improve the performance over single core processors for independent parallel tasks nearly linearly as long as the provided memory bandwidth is sufficient. Ideal speedup is, however, not typically possible even if this precondition is fulfilled since running multiple cores on a single chip generates more heat than a single core leading to decreased clock frequency to avoid overheating. More serious performance scalability obstacles are caused by frequent inter-communication between the cores and non-trivial memory access patterns. Figure 1 shows the source code and execution time of two simple C/Pthreads programs—*matmul* and *matsum*—in an 18-core Intel Skylake Xeon W CPU as a function of the number of threads (and thus processor cores utilized in execution). For comparison, we show also ideal scaling behavior curves based on the execution time with a single thread. These programs adapt their execution according to parallelism specified by the programmer (*NUM_THREADS*), so one might expect to see execution time scaling consistently with the specified number of threads as long as this does not exceed the number of physical cores. In the case of *matmul*, the performance scales relatively well with the number of concurrent threads: A speedup of 9.23 relative to single core is achieved with 18 threads which is 51.3% of the linear speedup. On the contrary, as the number of parallel threads for *matsum* increases,

the performance actually decreases giving a slowdown of a factor of 4.72 with 18 cores. This is an 85.0 times weaker result than that predicted by linear speedup. The catastrophically poor performance behavior of *matsum* is caused by its interleaved access pattern that interferes with the on-chip cache distribution and line scheme of Xeon W CPU. Namely, the cache memory is distributed to core-wise blocks of 64-byte cache lines holding consecutive addresses. As a result, eight consecutive matrix elements that are processed in different cores should be placed into a single cache block and a single line in it. Depending on the size of the data set, this is causing congestion, intensive inter-block communication and extensive invalidation of private Level 2 and shared Level 3 cache line elements. It is noteworthy that the same problem does not occur in *matmul* although it has in principle more complex access pattern. It is possible to alleviate this scalability problem in the case of *matsum* by changing the mapping of data to the threads; but in more complex cases slowdowns due to demanding access patterns are hard to avoid. In addition to the performance problems, the productivity of software development (or programmability) for multicore CPUs is lacking behind. For performance reasons, programmers need deal with mapping and partitioning problems as seen in the *matsum* case. Generally speaking, programmers often cannot employ natural, straight-forward parallel processing patterns; but have to replace them with more complex and error-prone structures [3] as will be confirmed by our experiments. This can be seen as extra code lines compared to textbook counterparts of *matmul* and *matsum* [4, 5] which reduce the number of active code lines (4–6 and 5–9, respectively) in both cases to a single code line containing just a parallel statement with no for-loops and explicit synchronization.

Our analysis indicates that the performance and programmability problems of multicore CPUs are caused by inoptimality of current architectures for certain types of computing patterns rather than inefficient use of the methodology [6–9]. Particularly, the reasons for these problems include high costs of synchronizing and switching between threads, weakly scalable latency tolerance mechanisms and lack of support for key patterns of parallel computation. In order to solve these problems, we have introduced the *Thick Control Flow* (TCF) concept and outlined the *Thick Control Flow Processor Architecture* (TPA) for executing programs employing TCFs natively [10, 11]. TPA belongs to our REPLICA multiprocessor framework and utilizes efficient shared memory emulation [12]. A TCF is an abstraction of parallel computation that merges self-similar threads into a single computational entity that is independent of the number of threads. Self-similarity refers here to properties of flowing through the same control path and having homogeneous operations. We call the component threads of a TCF *fibers* to distinguish them from ordinary threads having their own control. The fibers within a TCF are executed synchronously with respect to each other in order to simplify parallel programming.

1.1 Related work

While there are already a number of performance comparisons between TPA and its predecessors showing its potential [11, 13–15], it is not known how well TPA

solves the performance and programmability issues of multicore CPUs and more specifically, how TPA can perform against current commercial multicore processors employing industry standard programming models.

The first [11] work outlines our TCF architecture TPA compares its performance to dual-mode REPLICAs *Configurable Emulated Shared Memory* (CESM) architecture that has quite similar shared memory system but lacks support for the TCF model. It shows that the TCF architecture has better performance than CESM in all tests included in the study. The investigation [15] summarizes the architectural techniques supporting concurrent memory access in TCF architectures. The TPA armed with the proposed techniques is compared to the baseline TPA without concurrent memory access support and speedup factors up to $\log N$ for problems of N elements are detected. An architectural technique support flexible fibering for TCF architectures is introduced in [14]. It is shown that a TPA with flexible fibering can be up to twice as fast as the baseline TPA with standard threading. Finally, the work [13] discusses techniques for supporting multioperations in TCF processors. Again, the TPA armed with the proposed techniques is compared to the baseline TPA and shown up to $\log N$ times faster.

The performance of REPLICAs CESM architecture, sharing almost the same shared memory emulation support techniques as TPA, was compared with a six-core Intel Xeon X5660 CPU and 448-core Nvidia Tesla M2050 *Graphics Processing Unit* (GPU) as well as other architectures supporting shared memory emulation such as XMT and SB-PRAM [16]. An early programmability comparison recording the used code lines between parallel REPLICAs CESM, sequential DLX and parallel pthreads Intel Core i7 programs performing the same functionality is published in [17].

Earlier attempts to realize an idealized shared memory include the NYU Ultracomputers [18], Fluent Parallel Machine [19], SB-PRAM [6], Eclipse [20], XMT [21] and REPLICAs CESM [17]. The Ultracomputers featured strong computational model but the implementation was based on a weakly-scalable network and processor architecture was primitive. The Fluent Parallel Machine was backed up with a proven theory but had many inefficiencies in its shared memory emulation algorithm and still employed a weakly-scalable network. The SB-PRAM solved some of these shared memory emulation inefficiencies but the architecture of the processor was again very simple ignoring, e.g., possibilities for exploiting low-level parallelism. The ECLIPSE architecture solved both the processor architecture and low-level parallelism issues as well as the network scalability issues but featured only exclusive shared memory access and poor thread utilization in low-parallelism cases. The REPLICAs CESM solved the utilization and shared memory access problems with its dual-mode architecture and concurrent memory access and multioperation support. However, all these architecture provided a fixed threading scheme that forced a programmer to use loops to match the software and hardware parallelism. The XMT architecture solved the threading limitations but lost synchronicity between the threads. None of these attempts was able to provide flexible threading along along with strictly synchronous operations on a top of scalable network like TPA does.

The programming and programmability aspects of these approaches are discussed in [4, 6, 22, 23] while the related models of computation are reviewed in [8, 24]. The Valiant's multicore *Bulk Synchronous Parallel* (BSP) model bridges the BSP model to practical multicore processors and hints the possible architectural approach leading the efficient parallel execution. The extended *Parallel Random Access Machine - NonUniform Memory Access* (PRAM-NUMA) model links together these emulated shared memory architectures, vector execution model, Vishkin's flexible threading model and the TCF programming model.

Besides these works, there exists a large base of benchmark studies that compare the performance of Intel multicore processors to other multicore processors and discuss the Intel multicore processors and their details.

1.2 Contribution

In this paper⁰, we compare quantitatively the performance and programmability of TPA and Intel Skylake client and server multicore CPUs with a number of kernels that are widely used in general purpose computing to find out how well our architecture solves the aforementioned issues that greatly reduce the performance of multicore CPUs and productivity of parallel software development in general. In particular we¹:

- Introduce the TPA as well as Skylake client and server processors architecture/hardware with focus on details relevant to performance. This includes explaining the key functional blocks and operation of the processors;
- Describe the differences in programming approaches. Code examples are given and experiences discussing the goodness of the included approaches are listed;
- Evaluate the performance and length of key functionality of a number of kernel benchmarks on Skylake CPUs and TPA. The kernels are written in Pthreads, OpenMP and a baseline TCF programming language featuring the same level of explicit control on computation as Pthreads but including TCF-specific primitives. The performance is compared by executing the benchmarks in computers featuring Skylake client and Skylake server processors and recording the execution time. The performance of TPA is determined by executing the TCF version of the benchmark in a clock accurate simulator (modeling the TPA microarchitecture down to details of registers and logic between them) and recording the execution time. The comparison is done based on assumption that the processors would have the same clock frequency. The programmability comparison is done by comparing the number of active code lines in benchmarks written for Skylake and TPA;
- Study the effect of access patterns, threading, synchronization and latency hiding to performance in both TPA and Skylake CPUs. These factors are also compared

¹ This paper is an extended version of the paper [25] with more detailed description of the compared processors and programming schemes, additional benchmarks, OpenMP experiments showing the effect of access patterns and the fraction of bandwidth achieved, and an extended discussion of the results.

to those in ideal machines to determine the optimality of resource usage in the processors;

- Record experiences on programmability and performance aspects of the Skylake and TPA systems from a skilled programmer with a lot of experience in parallel computing and a person not familiar with neither the theories of parallel computing nor the used programming methodologies.
- Identify methods to avoid performance pitfalls in programming multicore CPUs employing the industry standard architecture, such as Skylake.

The rest of this paper consists of Sect. 2 describing the hardware architectures of the compared processors, Sect. 3 explaining shortly the programming methodologies used for producing the test programs, Sect. 4 performing the actual comparison with discussion on results, and Sect. 5 giving our conclusions.

2 Hardware architectures

Hardware architectures of our interest targeted for general purpose parallel computing include TCF architectures [11], *Emulated Shared Memory* (ESM) architectures [6, 20, 21, 26], their development versions and current commercial multicore architectures from Intel, Apple, AMD and IBM. In this paper, we study TPA, Intel Skylake client Core i7 and Skylake server Xeon W and utilize them in the comparison of Sect. 4.

2.1 TPA

The *Thick Control Flow Processor Architecture* (TPA) is a scalable multiprocessor architecture that can be configured at design time for various constellations [11]. It belongs to our REPLICA multiprocessor framework that aims at addressing the performance and programmability issues of current general purpose multicore architectures [12]. The framework defines principles how to build efficient ESM processors; how to make them flexible and expandable with accelerators; how to achieve backwards compatibility with existing commercial product lines; includes our processor, interconnect and memory system architectures and designs needed for that purpose and outlines the methodology to develop program so that our ambitious key performance indicators—performance, programmability, productivity, silicon area, power consumption and security are met. In ESM, the latency of the memory system is hidden via multithreading and sufficient bandwidth, the synchronization cost is virtually eliminated using wave synchronization and low-level parallelism exploitation is optionally improved by chaining of *Functional Units* (FU) assuming there is enough parallelism in the functionality at hands [6, 20, 26]. Instead of multithreading, TPA uses a similar technique for fibers, called interleaved multifiber. This lets a fiber to execute other fibers while it is making a memory reference. If the executed program contains enough fibers the latency of the shared memory system can be completely hidden. Sufficient interconnection bandwidth is provided by using

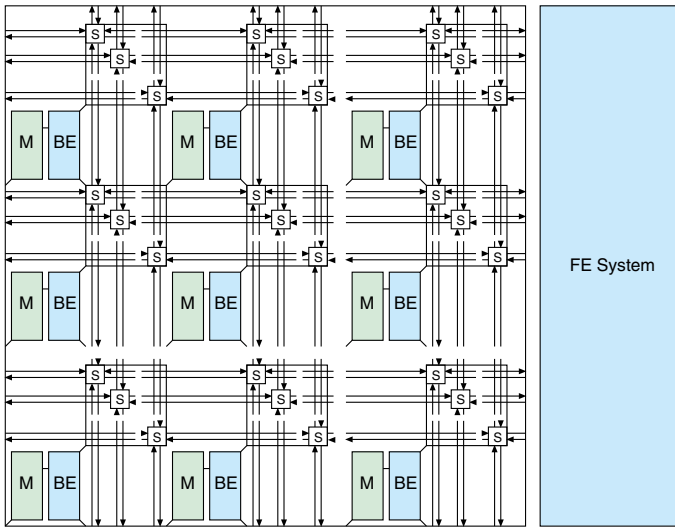


Fig. 2 Simplified block diagram of TPA (FE = frontend processing unit system, BE = backend processing unit, M = memory module (or first level cache), s = intercommunication switch). External memory system and spreading/return channel networks between FEs and BEs are not shown

an M -way network, e.g., multimesh. The synchronization wave is used to separate memory references belonging to consecutive steps of execution by issuing all fibers followed by a synchronization message. Synchronization messages are routed in the network through all possible paths so that when a synchronization message arrives to a router, it blocks the message (and related paths) until a synchronization message can be found in all inputs. Then the router fetches all the incoming messages and sends out a synchronization message via its outputs. Low-level parallelism is supported in TPA by organizing FUs as a sequential chain rather than in parallel so that consecutive instructions can be executed regardless of possible interdependencies within a step.

A TPA multiprocessor consists of F Frontend (FE) processing units and B Backend (BE) processing units, intercommunication networks and a memory system (see Fig. 2). FEs take care of fetching instructions from the memory and executing the common parts of TCFs, such as control of the flow and base address computation. In turn, BEs handle execution of individual fibers. The memory system consists of two parts: Depending on the FE architecture of our choice, FEs are connected to either a traditionally organized *Symmetric MultiProcessor* (SMP) or *NonUniform Memory Access* (NUMA) memory system and BEs are attached to an ESM system employing a multimesh interconnect. The latter supports synchronous operations and parallel-computing specific access patterns such as concurrent reads and writes, reductions and multiprefix computations as well as powerful atomic compute-update operations. The FE and BE parts of the memory system are also connected together.

A TPA FE system resembles an ordinary multicore CPU and in fact the REPLIC framework allows a processor designer to use a variant of existing commercial CPU core as FE. In this paper, however, we use the REPLIC default,

For each active FE do

- F1.** Select the next TCF from the TCF buffer if requested by the previous instruction or TCF management logic.
- F2.** Fetch a VLIW instruction pointed by the PC of the current TCF from the FE memory system.
- F3.** Execute the subinstructions specified by the instruction in the FUs. Memory subinstructions are typically targeted to the FE memory system. If the instruction contains a BE part, select the operands and send them along with the part to the BEs assigned to the FE via the work spreading network. Store the data of current TCF to the TCF buffer and switch to the next TCF if requested by the corresponding subinstruction or TCF management.

For each BE do

- B1.** If the BE is not executing the previous instruction any more, fetch the next instruction from the spreading network and determine the fibers to be executed in the BE. Otherwise continue executing the previous instruction.
- B2.** Generate the fibers of the TCF to be pipelined according to the assignment determined in B1.
- B3. For each fiber bf do:**
 - B3.1** Select the operands from the received FE data and BE register block.
 - B3.2** Execute the BE subinstructions. Memory subinstructions are targeted to the shared memory system.
 - B3.3** Write back the BE register block and send the optional reply data back to the FE via the return channel built into the spreading network.

After all active TCFs from the FE have been in execution for a single instruction, TPA issues a special synchronization TCF of thickness zero per BE that only sends and receives a synchronization to/from the BE shared memory system.

Fig. 3 Phases of TPA execution

our *Minimal Pipeline Architecture (MPA) Very Long Instruction Word (VLIW)* processor [27] as the FE architecture. MPA features a number of FUs commanded by dedicated subinstruction fields in a single (compound) instruction word. The original version of MPA features a special minimal pipeline with only two stages—fetch and execute with no pipeline delays in the case of control transfer. We made some modifications to the pipeline to allow multi-TCF operation but were able retain pipeline delay-free operation also for control operations.

A BE is a special processing unit resembling a *MultiBunched/Threaded Architecture with Chaining (MBTAC)* ESM processor core [17] and containing logic for operand selection, chain of FUs, latency compensation unit and write back logic. However, a TPA BE does not include an instruction fetch unit and sequencer. In TPA these belong to the FE system.

Execution of an instruction in TPA happens in three FE phases and three BE phases (see Fig. 3). Note that the FEs and BEs are integral parts of TPA and work together to execute instructions efficiently.

The architecture and methodology of TPA and REPLICA framework in general are reviewed in details in our white paper [12].

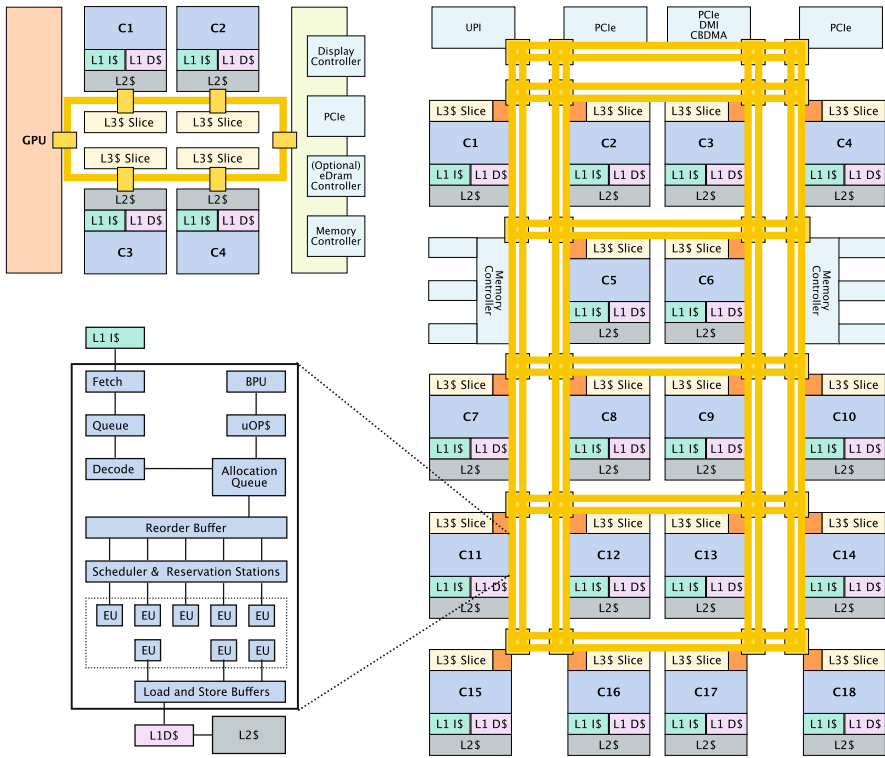


Fig. 4 *Top left* Block diagram of a 4-core Intel Skylake client Core i7 (Cn = processor core n, L1 IS = level 1 instruction cache, L1 DS = level 1 data cache, L2\$ = level 2 unified cache, L3\$ Slice = shared level 3 unified cache slice, GPU=graphics processing unit). *Bottom left* Block diagram of a Skylake processor core (BPU = branch prediction unit, EU = execution unit, Intel’s name of a FU). *Right* Block diagram of a 18-core Intel Skylake server Xeon W

2.2 Core i7

Intel Core i7 6820HQ is a 64-bit 2.7 GHz (3.2 GHz turbo boost with all cores on) 4-core, 8-FU/core Skylake client microarchitecture CPU [28] (see Fig. 4). Skylake is Intel’s high-performance out-of-order microarchitecture. Intel has used Skylake derivate architectures with minor changes from 2015 to 2021 in their 14 nm silicon process based processors. The memory system of 6820HQ features a 32 KB *Level 1* (L1) instruction and data cache per core, a 256 KB *Level 2* (L2) cache per core and a 2 MB shared *Level 3* (L3) cache per core. The bandwidth from core to L1 instruction cache is 16 bytes, i.e., two 64-bit words per clock cycle. The L1 data cache is connected to two load and one store unit in the core, each having 32 bytes, i.e., four 64-bit words per clock cycle bandwidth. Both instruction and data L1 caches have own 64 bytes per clock connection to common L2 cache. The L2 cache is connected to the L3 cache via a 32 bytes per clock cycle connection. The microchip includes also a 9th generation Intel HD Graphics 530. The cores, GPU and memory

controllers are connected with a single bi-directional ring bus. 6820HQ has a single memory controller and two memory channels with maximum memory bandwidth of 31.79 MB/s. The processor was tested on an Apple MacBook Pro (late 2016) laptop computer with 16 GB of onboard 2133 MHz *Low-Power Double Data Rate 3* (LPDDR3) *Synchronous Dynamic Random-Access Memory* (SDRAM).

Compared to TPA, Core i7 provides the programmer with eight asynchronously operating hardware threads. The mechanism of tolerating the latency of memory accesses is a hierarchy of caches with a coherence mechanism. The memory system is chosen to provide enough bandwidth for its 256-bit *Advanced Vector eXtension 2* (AVX2) *Single Instruction Multiple Data* (SIMD) units access down to L3 caches. Core i7 is an out-of-order dynamic superscalar processor that fetches a number of instructions from the level 1 instruction cache, decodes them into *micro-Operations* (uOP) and places them into the reservation station. Each clock cycle, its internal scheduler tries to move at most eight microoperations, whose operands are available, for execution in the FUs. Branch prediction is used to decrease the control delays caused by its long pipeline. This happens by letting the processor execute the branch predicted by the *Branch Prediction Unit* (BPU) and placing the uOPs also in the reorder buffer and completing execution of them only after the predictions are verified to be correct. Mispredictions are solved by flushing all incorrectly speculated uOPs from the reorder buffer and re-executing instructions and related uOPs from the correct path. Thus, the Skylake core solves the scheduling problem of instructions (and their uOPS) due to interdependencies between instructions by increasing the length of the pipeline with dynamic superscalar execution. Since this happens with the cost of substantially longer control transfer delays, Skylake compensates that with branch prediction. This is just the opposite way compared to TPA's MPA, which uses static superscalar execution and minimizes the length of the pipeline to eliminate all pipeline delays.

2.3 Xeon W

Intel Xeon W-2191B is a 64-bit 2.3 GHz (3.2 GHz turbo boost with all cores on) 18-core, 8-FU/core Skylake server microarchitecture CPU [29] (Fig. 4). Its memory system features 32 KB level 1 instruction and data caches per core, 1 MB level 2 cache per core, 1.375 MB level 3 cache per core and 2 memory controllers and 4 memory channels with maximum memory bandwidth of 79.47 MB/s. The L1 and L2 caches and their interconnection in a core are similar in Xeon W as in 6820HQ but the on-die interconnect is mesh type. Compared to a single ring that would connect all cores and IO, the mesh provides faster access to data from IO and L3 cache slice located far from the core requiring that data. The connections between the core and L1 data cache as well as L2 and L3 caches are 512-bit, twice that of Core i7. The sheer size of the die makes it impossible to have similar short average latency as in 6820HQ for high core count processor like this. The effect of longer latencies can be seen in tests where L3 cache lines are shared with multiple cores or data are otherwise scattered across the mesh. The processor was tested with an Apple iMac

Pro (late 2017) workstation with 128 GB of onboard 2666 MHz *Error Correction Code* (ECC) DDR4 SDRAM.

Xeon W works in the same way as Core i7 but there are more processor cores and the memory system as well as interconnects are designed for higher server workload. The memory system is chosen to provide enough bandwidth for its 512-bit AVX-512 SIMD units access down to L3 caches.

3 Programming methodologies

The main idea of parallel computation is to decompose or divide the computational problem at hands into subproblems that can be solved in parallel and to compose the solution of the original problem from the results of the subproblems [6, 21, 22, 30–37]. This may naturally happen hierarchically, recursively and/or in consecutive parts. Solving the subproblems in parallel introduces the need for communication between the parallel parts, which in turn may create dependencies that require synchronization between the parallel parts. Finally, to get actual results, these parallel parts need to be executed in physical processors which raises a need to define the relationship of execution units and parts executed in parallel, known as mapping. A related concept, partitioning, refers to the way how data are placed in the distributed shared memory computer like TPA and Skylake CPUs. A processor can be said to have good programmability if the functionalities can be expressed compactly and naturally without unnecessary architecture-dependent constructs. An important factor of programmability is also portability among a group of processors using the same paradigm/approach but different hardware implementation parameters.

There are a huge number of parallel programming models/languages taking different approaches [33]. Some of them, such as *Message Passing Interface* (MPI) [38], arrange intercommunication via messages send to/received from other threads, while most of them utilize some kind of a shared memory for exchanging data between threads [39, 40]. There are differences in how this shared memory is presented to threads and how exclusive and concurrent data access handles, how synchronizations are organized and how race conditions and deadlocks are avoided [5, 6, 23, 41]. In this section, we focus on the TCF programming scheme as well as on highly popular multicore programming solutions, Pthreads and OpenMP, that will be utilized in TPA and Skylake CPUs, respectively.

3.1 Thick control flows

The *Thick Control Flow* (TCF) concept is an abstraction of parallel computation that merges self-similar threads (called *fibers*) flowing through the same control path into computational entities (called *TCFs*) independently of the number of threads [10]. The number of fibers in a TCF is called the *thickness*. A programmer can dynamically change the thickness of a TCF during execution. The fibers within a TCF are executed synchronously with respect to each other to enable simple parallel programmability. The concept shares many properties of the idealized *Parallel Random*

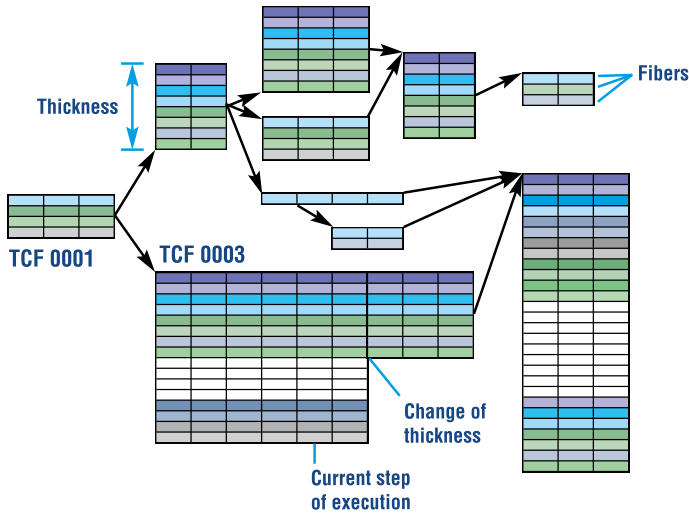


Fig. 5 A program consisting of 10 TCFs while the fifth instruction of TCF 0003 featuring thickness 16 is being executed

Access Machine (PRAM) model of computation [31]. A PRAM consists of a number of processors running under a single clock connected to a synchronous shared memory with idealized latency properties. There exists a well-developed theory of parallel algorithms for PRAM that can be useful many ways in understanding the intrinsic parallelism and complexity of the computational problem at hands [4, 6].

TCFs have a single control but they can process multiple data elements in parallel. When a TCF with thickness T calls a subroutine, the subroutine is not called separately by each T fibers, but the TCF calls it only once with thickness T . A call stack is not related to each fiber but to each of the TCFs, since fibers do not have program counters. This implies that stack variables many times have thickness T reflecting the fact that there is a data element for each fiber. Multiple TCFs can be executed in parallel in multiple FEs and in overlapped way in a single FE. Figure 5 shows an example of a program consisting of 10 TCFs with a relatively complex structure with interdependencies. The TCF 0003 is in execution and features a change in thickness from 16 to 8 after one more step.

In a TCF system, such as our TPA processor, the fibers of the currently executed TCF are evenly distributed to the backend execution units. The execution units operate in parallel and maintain synchronous behavior between consecutive instructions so that the TCF concept executes as the programmer expects.

3.2 POSIX threads

POSIX threads (Pthreads) is a set of C language interfaces (functions, header files) for threaded programming [39]. It allows a program to control multiple different threads of computational work that overlap in time. Each thread executes its instructions independently. All threads in a process share the memory space, functions,

data and files. In addition, thread-specific data can also be defined. Any thread can create new threads and threads can have different kinds of relations with each other, such as master-slave or producer-consumer. Synchronization is needed to ensure that multiple threads are collaborating correctly, to avoid problems like deadlocks and race conditions. Synchronization can be done via mutual exclusion locks, semaphores, join functions and barriers, which can be implemented by a set of corresponding provided functions. Threads in different processes can be synchronized via synchronization variables in the shared memory [39]. In a typical SMP system, such as Apple MacBook Pro and Apple iMac Pro running Apple Mac OS operating system utilized in this paper, threads are periodically assigned to processor cores with least amount of work. Therefore, a parallel program written in pthreads is often actually executed in parallel in SMP systems.

3.3 OpenMP

Open Multi-Processing (OpenMP) is a portable, shared-memory thread programming interface for programming languages, such as Fortran, C, and C++ [40]. It supports both data and task-level parallelism and designed for coarse-grained parallelism. The parallel computing related mechanisms are implemented mainly as pre-processor pragmas but there are some runtime routines and environment variables too. An ambitious goal of OpenMP is to be able to use a sequential program and just add directives for parallelism so that if a program is compiled ignoring the pragmas, it will behave like a standard sequential program while taking them into account would lead to an efficient parallel implementation.

4 Comparison

In order to compare TPA against Skylake multicore processors, we performed a series of quantitative performance and programmability tests, studied the effect of access patterns, threading and synchronization to performance and resource-efficiency as well as collected experiences on programmability. In addition, program code examples are given.

4.1 Quantitative measurements

We measured the execution time and counted the number of active code lines for ten short program kernels (Table 1) that represent widely used functionalities of parallel computing on three multiprocessor systems representing TPA and Intel Skylake CPUs introduced in Sect. 2 (Table 2). Each program, except *bprefix* and *sync*, was implemented as a single TCF version for TPA and three Pthreads versions for Skylake CPUs—straight-forward, matched parallelism, and blocked versions. The TCF versions are written as a single TCF, maximally parallel, synchronous programs utilizing available primitives of parallel computing where relevant. There is no need for explicit synchronizations in the tested TCF algorithms. The *straight-forward*

Table 1 Benchmark programs

block	Copies an array of 1024/262,144 64-bit integers to another location in the memory. (<i>Tests memory access with a simple pattern</i>)
lprefix	Calculates the prefix sum of an array of 1024/65,536 64-bit integers using the logarithmic prefix sum algorithm. (<i>Tests demanding memory access pattern, dense intercommunication and change in thickness</i>)
bprefix	Calculates the prefix sum of an array of 1,048,576 64-bit integers using the blocking prefix sum algorithm. (<i>Tests blocking-style prefix computation—includes only one version since there is no straight-forward variant of this other than lprefix</i>)
madd	Adds an array of 1024/262,144 64-bit integers to another similar array. (<i>Tests simple exclusive memory access pattern and basic matrix operation</i>)
mcarlo	Calculates pi using the Monte Carlo circle algorithm with 1024/262,144 64-bit integers. (<i>Tests computation-intensive processing</i>)
mmul	Multiplies two arrays of 128×128 64-bit integers to a third similar array. (<i>Tests exclusive and concurrent memory access and basic matrix operation—includes only one version due to built-in LLVM optimizations making head-to-head comparisons difficult</i>)
permute	Writes thread or fiber identifier to a random location in an array of 1024/262,144 64-bit integers, reads data from a second random location and writes data to a third random location. (<i>Tests dense exclusive memory access with a complex and randomized access pattern</i>)
spread	Spreads the first element to rest of the elements in an array of 1024/262,144 64-bit integers. (<i>Tests a simple concurrent memory access pattern</i>)
threshold	Applies a threshold filter to an array of 1024/262,144 64-bit integers. (<i>Tests compute-update operations</i>)
sync	Performs a barrier synchronization of threads. (<i>Tests synchronization cost. Uses internally dense intercommunication</i>)

Table 2 Tested multiprocessors (*FE* frontend, *BE* backend, *MU* memory unit, *HT* Intel's 2-way simultaneous hyperthreading, *TCF* thick control flow scheme, *BW* bandwidth)

	Core i7	Xeon W	TPA
Number of cores	4	18	1 FE/16 BE
Clock (sustained/peak all/peak 1)	2.7/3.2 GHz	2.3/3.2/4.2 GHz	3.2 GHz
Number of FUs (FE/BE)	8	8	4/10
Number of MUs	3	3	1
Threading scheme	HT	HT	TCF
Core-L1/L2/L3 BW (64-bit words)	12/32/16	54/144/144	1 + 16
SIMD-L1/L2/L3 BW (64-bit words)	48/32/16	432/144/144	–

Pthreads versions are written similarly as the TCF versions except that synchronizations are added to the end to be able to determine when all the threads have completed their tasks and wherever they are needed to guarantee the correct execution order of operations assigned to different cores. The *matched parallelism* versions limit the number of threads to a given maximum, which in our case is the number of processor cores P . The matching is done by employing loops that process at most P elements (and threads) at the time. We expect matched parallel versions to be substantially faster than straight-forward ones. This is because matching eliminates

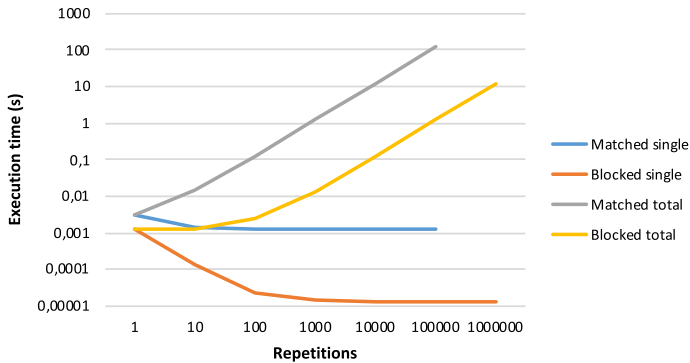


Fig. 6 Average execution time of matched parallel and blocked Pthreads versions of the *block* benchmark as the function of repetitions in Intel Xeon W multicore CPU. The total test execution times are also shown. All tests are also averages of five independent runs. Note that the scale is logarithmic

interference between time slots defined by the operating system scheduler and actual computation as well as the thread management overhead, especially in the case of fine-grained parallel functionality. The *blocked* versions divide the processed data elements to blocks that are executed in the processor cores in parallel. This kind of mapping and implied partitioning should also improve the performance over the matched parallelism versions due to increased locality and reduced inter-processor communication.

The TPA configuration used in the comparison (TPA-16) was selected so that it represents a typical entry-level constellation and its resources would not exceed those of Intel Skylake client by a large margin. TPA's integer-only silicon area, estimated to be 19 mm^2 at an artificial 11 nm process [13], is clearly smaller than that of the Skylake client without its on-chip GPU (50.354 mm^2 at Intel's 14 nm process). The memory bandwidth of Core i7, 12 64-bit words from the actual cores and 48 words from the SIMD units to the L1 caches matches roughly that of a single 64-bit word from the FE and 16 words from BEs to the on-chip memory modules. The Skylake server chip (485 mm^2 at Intel's 14 nm process) is included for comparison purposes to get an idea how Skylake designs scale to high-end versions and how a scaled-up version performs with respect to an entry-level TPA. The bandwidth of Xeon W, 54 64-bit words from cores and 432 words from the SIMD units to the L1 caches are roughly 3.4 and 27 times higher than in TPA. The clock frequency of TPA was set to 3.2 GHz. This matches the maximum clock frequency of the both Skylake CPUs with all cores active.

The multicore CPU computers were running Apple MacOS Mojave 10.14.6. The test programs were compiled with the Apple LLVM compiler (cc) with the -O3 optimization, except for *lprefix* matched parallelism and blocking versions that were compiled with the -O2 optimization. Each run executed the programs repeatedly at least 10 seconds up to million times so that slow startup time, effects of *Dynamic Voltage Frequency Scaling* (DVFS) and cooling were minimized in the most cases and the frequency was kept close to the frequency of 3.2 GHz (see Fig. 6). In tests not employing all Skylake cores, the frequency was higher giving Skylake some

advantage because, we assume that TPA would always run at 3.2 GHz. Besides having multiple iterations for each run, we repeated each test five times and calculated the average time of the runs as the measurement result for the comparison. Since our TPA simulations do not utilize the outermost part of the memory system, we selected the problem sizes included in the comparison so that the data set fits to the shared L3 caches of the Skylake CPUs that roughly corresponds to the TPA on-chip shared memory modules. Other selection criteria include that the used MacOS operating system is able to create the threads for the execution in both Skylake CPUs and the simulation time in TPA simulator stays reasonable.

We selected to mostly focus on comparing TPA with Skylake CPUs cores without SIMD units because:

- SSE/AVX-style SIMD units are not included in the BEs of TPA-16 but there is option to do so. The selection and details of that are, however, beyond the scope of this paper.
- Performance of Skylake cores only (without SIMD units) is a better-defined comparison point than that with different versions AVX2 and AVX-512 with different number of execution units and higher bandwidth than processor cores.
- Inclusion of SIMD units defines yet another dimension of optimization techniques and question of applicability to computational problems/algorithms at hand potentially increasing the complexity of programming and optimization.
- The included Pthreads programs compile without SIMD-instructions even with `-O3` optimization on.

Since the SIMD units are important parts of Skylake CPUs, we performed, however, a few tests where we ensured that AVX instructions are actually utilized.

The TPA programs were compiled by hand to TPA assembler and executed in our in-house TPASim simulation software that provides a clock-accurate model of the TPA configuration used in the tests. TPASim models accurately the TPA micro-architecture down to registers and logic between them. It is actually used as a reference design against which we compare our on-going FPGA implementation work. We do not consider usage of hand-written assembler a big optimization advantage for TPA since we have experimented with compilation in the previous version of REPLICA processor including the similar chaining technique as in TPA with good results [16]. In addition, we checked the code generated by LLVM and gcc compilers for Skylake Pthreads and OpenMP tests to look efficient. There was no need to make multiple runs with TPA since by the exclusion of DVFS and outer memory system effects, the simulator always gives the same results.

4.1.1 Overall tests

The straight-forward version results of our measurements are shown as relative performance and relative length of the active program code lines in top rows of Figs. 7 and 8. The relative performance was determined by dividing the average execution time of a test program in 3.2 GHz clock cycles in the Core i7 processor, which acts as a comparison baseline, by the average execution time of the test program in

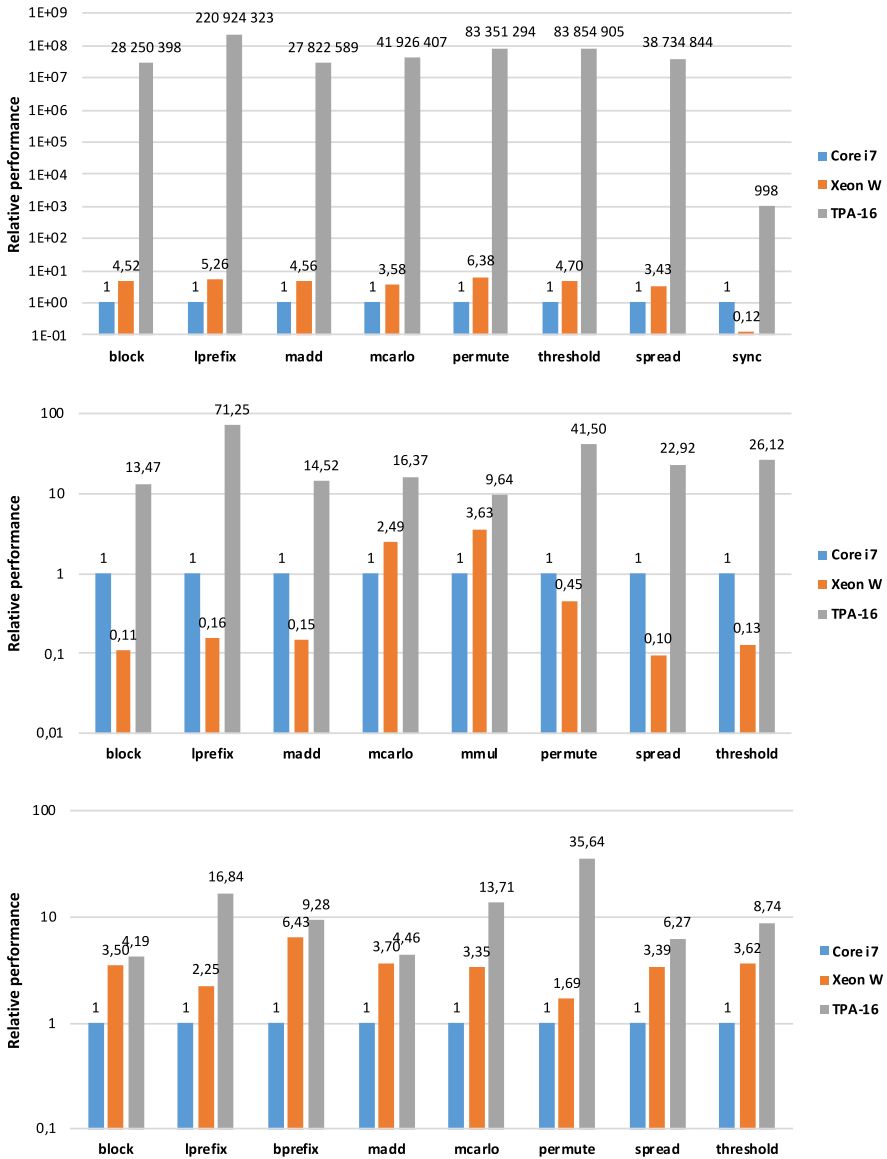


Fig. 7 Relative performance of the straight-forward, matched parallelism and blocking test programs in TPA-16 and Intel Skylake multicore CPUs (Core i7 = 1.0). Note that the scale is logarithmic for all performance charts

3.2 GHz clock cycles in the target processor. The average speedup was calculated by taking the geometric mean of individual speedups. The relative *High-Level Language* (HLL) code length was determined by dividing the number of active code lines in the Skylake processors, which acts as a comparison baseline and use the same code, by the number of code lines in the target processor. The average code

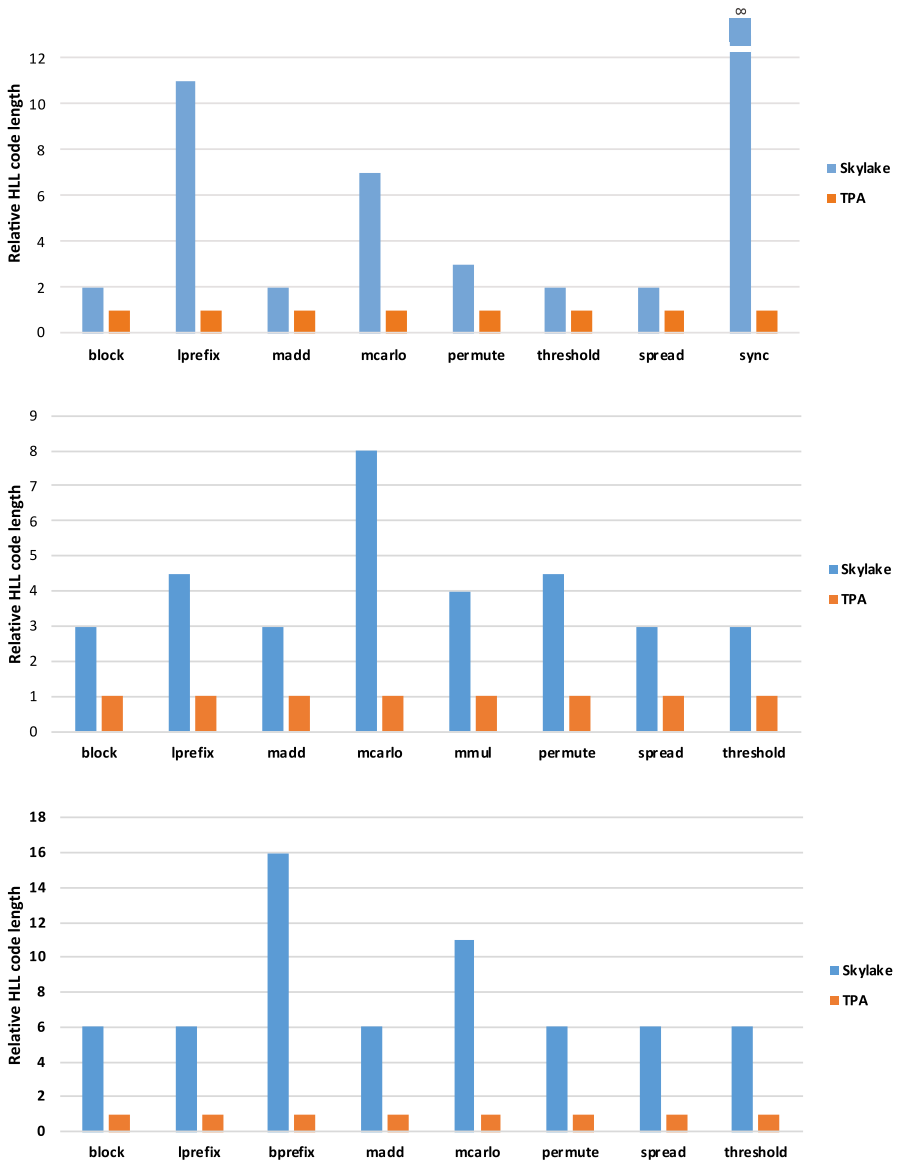


Fig. 8 Relative source code length of the straight-forward, matched parallelism and blocking test programs for TPA and Intel Skylake multicore CPUs (TPA = 1.0)

length improvement was calculated by taking the geometric mean of individual relative code length values. From these measurements we can make the following observations:

- TPA-16 provides vastly higher performance than the Skylake CPUs. In average, it executed the straight-forward test programs (*block*, *lprefix*, *madd*, *mcarlo*, *per-*

mute, *threshold* and *spread*) 57.1 million times faster than Core i7. The boost of switching Core i7 to Xeon W is 4.54-fold, reducing the average speedup of TPA-16 over Xeon W to still very high 12.6 million. The extremely slow execution speed of Skylake processors is caused mainly by high synchronization and thread switching costs. Namely, the operating system scheduler assigns frequently a relatively long standard execution time slice for a large number of waiting threads causing unnecessarily long delays. Other slowdown factors include cache misses and missing support for low-level parallelism between the threads.

- The barrier synchronization program *sync* in TPA-16 executes 998 times faster than in Core i7 and 8154 times faster than in Xeon W. Core i7 executes this faster since there are 4.5 times fewer processor cores to synchronize and the interconnect is simpler.
- The Skylake test programs are longer than those for TPA. The number of active code lines in the test programs in Skylake processors doing actual computation (*block*, *lprefix*, *madd*, *mcarlo*, *permute*, *spread*, *threshold*) is 3.23 times higher than that in TPA. The extra lines come from synchronization and thread management.
- Synchronization does not need any extra effort in TPA due to synchronous nature of TCFs, while in Skylake CPUs, one needs to perform synchronizations explicitly. As a result, the relative code length of a synchronization is infinitely shorter in TPA. Naturally, we did not count that in while determining the average.

It is clear that no programmer wants to write programs this way for Skylake CPUs if it can be avoided. Instead, one will select to use either better performing but more complex parallel implementation, such as a matched parallel or blocked realization, or just a sequential algorithm that performs a way better than the straight-forward implementations. Unfortunately, in current CPUs there are no radically faster ways to perform a barrier synchronization needed in parallel programs including inter-thread dependencies.

The matched parallelism version results are shown as relative performance and relative length of the active program code lines in middle rows of Figs. 7 and 8. Considering the achieved results, we can summarize that:

- The average performance of TPA-16 is 21.99 times higher than that of Core i7 and 67.29 times higher than that in Xeon W in executing matched parallel versions of the test programs.
- Despite its larger number of processor cores, Xeon W performs 3.06 times worse than Core i7 in these tests in average due to the higher intercommunication network latency and congestion. TPA-16 does not share this problem although its network [42] is almost as large as that in Xeon W in terms of the number of nodes.
- The *mmul* and *mcarlo* test programs perform differently than the other programs since the LLVM compiler recognized matrix multiplication and replaced the written baseline algorithm with a more optimized one. The *mcarlo* test applies a computation intensive algorithm giving Xeon W advantage over Core i7 despite a challenging access pattern.

- With respect to TPA the active code line count overhead in Skylake jumps from at least 2 to at least 3. This is caused by addition of looping to match the software parallelism with the hardware one in Skylake.

Also the performance of this kind of algorithms is disappointing in Skylake since it gives barely any speedup over sequential algorithms. Thus, programmers are tempted to turn to a sequential solution or they need to invent better parallel algorithm for these functionalities. Speedup possibilities can, e.g., be found from the blocking technique but again by increasing the complexity of the implementations.

The blocked version measurement results are shown as relative performance and relative length of the active program code lines in bottom rows of Figs. 7 and 8. Regarding these results, we can make the following observations:

- Introduction of blocking decreases the average relative performance advantage of TPA-16 versus Core i7 down to 9.70x. The highest speedup is achieved with the *lprefix* and *permute* test program due to inter-thread dependencies and change in thickness as well as demanding access patterns.
- Xeon W performs now close to TPA-16 in the *block* and *madd* test programs, while the average performance advantage of TPA-16 versus Xeon W drops down to 2.97x.
- The blocked Skylake versions increase the code line count overhead with respect to TPA from at least 3 to at least 6. In average, the blocked test programs were 7.3 times longer in Skylake than in TPA.

These results look much better for Skylake CPUs with the cost of clearly more complex programs. However, they are still left a way behind TPA. A deeper analysis of the factors behind the performance and programmability results is given in the next subsections.

4.1.2 OpenMP and sequential notation

Pthreads is sometimes considered as a primitive way to program parallel functionalities. For comparison purposes we implemented the *block*, *prefix*, *madd*, *mmul* and *threshold* functionalities also with OpenMP that has more compact notation than Pthreads. Our design principle was to retain the sequential algorithm lookout as much as possible and use *omp parallel* and *omp for* pragmas to request compiler to take care of the parallelization. We used gcc with the `-O2` optimization since LLVM compiler and gcc `-O3` failed to compile without SIMD instructions. The results are shown in Fig. 9. We can make the following observations from the results:

- The OpenMP versions are in average 11.53 times slower in Core i7 and 20.27 times slower in Xeon W than TCF programs in TPA.
- Xeon W achieves only 56.9% of the performance of Core i7 even though it has 4.5 times more processor cores. This is due to scalability problems of OpenMP programs. It is possible to get better performance for Xeon W but it requires expertise in OpenMP and makes programs more complex.

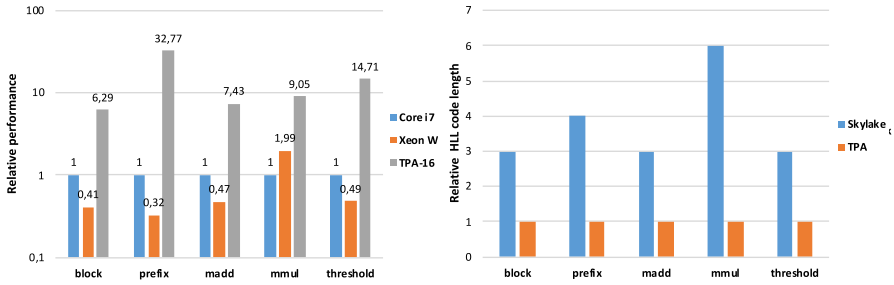


Fig. 9 *Left* Relative performance of the OpenMP versions in Intel Skylake multicore CPUs and TCF versions in TPA (Core i7 = 1.0). *Right* Relative source code length of the the OpenMP programs for Intel Skylake multicore CPUs and TCF programs for TPA (TPA = 1.0). Note that the scale is logarithmic for the left hand side performance chart

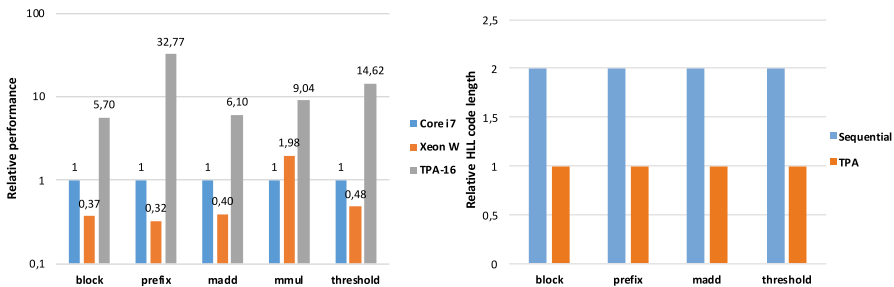


Fig. 10 *Left* Relative performance of the OpenMP -O3 optimized versions (including SIMD unit instructions) in Intel Skylake multicore CPUs and TCF versions in TPA (Core i7 = 1.0). Note that the scale is logarithmic. *Right* Relative source code length of the sequential and TPA test programs (TPA = 1.0)

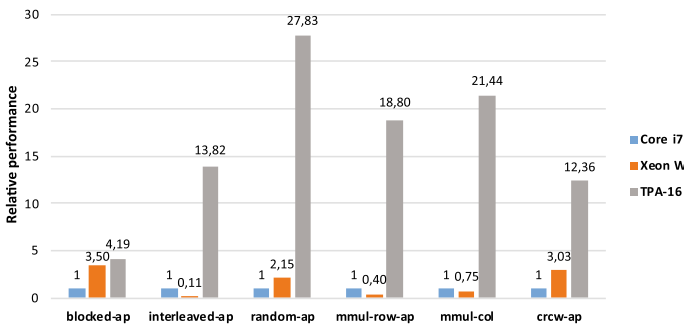
- While OpenMP versions of the programs are shorter than those written with Pthreads (except in the straight-forward case), they still increase the code line count by a factor of 3.65 with respect to TCF programs.

The performance slowdown when moving to a larger machine even when the parallel implementation was done by OpenMP compiler is disappointing. This is an indication of a more general scalability problem caused by increased latency and decreasing relative bandwidth of a memory system per module/bank and even bisection of the machine.

In order to determine the effect of compiler optimization and utilization of SIMD units, we compiled the programs with -O3 optimization and ensured that the resulting code indeed employs SIMD instructions. The results are shown in Fig. 10. Our immediate observation is that the speedup over -O2 optimized versions is only 6.24% in Core i7 and 0.5% in Xeon W. We tested also -mavx2 and -mavx512f options but without better success. This problem is apparently caused

Table 3 Access patterns

blocked-ap	Reads elements from an array of 262,144 64-bit integers and writes them to another location in the memory using the blocked algorithm
interleaved-ap	Reads elements from an array of 262,144 64-bit integers and writes them to another location in the memory using the matched parallel algorithm
random-ap	Reads elements from an array of 262,144 64-bit integers and writes them to another location in the memory using the permute algorithm, where read and write happen from/to random address within the array
mmul-row-ap	Reads elements from two arrays of 262,144 64-bit integers and writes them to similar arrays in the memory using the mmul style row-parallel pattern
mmul-col-ap	Reads elements from two arrays of 262,144 64-bit integers and writes them to similar arrays in the memory using the mmul style column-parallel pattern
crw-ap	Replicates a 64-bit integer to 262,144 instances and sums the values into a single location

**Fig. 11** Relative performance of *blocked-ap*, *interleaved-ap*, *random-ap*, *mmul-row-ap*, *mmul-col-ap* and *crw-ap* read-write access patterns in Intel Skylake multicore CPUs and TPA (Core i7 = 1.0)

by inability of gcc to parallelize OpenMP code in a scalable way with a simple set of gcc optimization flags.

The TPA test programs implemented for this study are very compact. We compared them to sequential programs solving the same computational problems. It turned out that TPA versions are even shorter than their sequential counterparts (Fig. 10). This is because loops for going through data elements of arrays are not needed if TCFs are used for programming.

4.1.3 The effect of access patterns

The used memory access pattern have effect on the achieved performance in a distributed memory system including both multicore CPUs and TCF processors. Since the principles of the memory systems are different in Skylake and TPA we measured the execution time of programs employing six read-write access patterns (Table 3). The results as relative performance are shown in Fig. 11. We observe that TPA provides a much higher performance in these tests except for the *blocked-ap* access

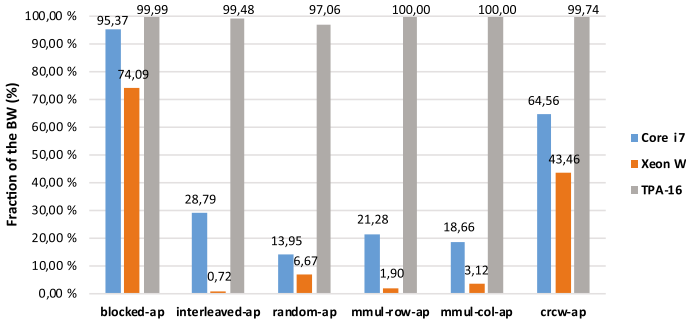


Fig. 12 Fraction of the utilized bandwidth from the maximum for *blocked-ap*, *interleaved-ap*, *random-ap*, *mmul-row-ap*, *mmul-col-ap* and *crw-ap* read-write access patterns in Intel Skylake multicore CPUs and TPA

pattern. The fractions of utilized memory bandwidths from the maximum bandwidth are shown in Fig. 12 with respect to situation in which only one memory operation per core would define the maximum bandwidth of memory access in Skylake processors. If we take into account that Skylake cores actually have three memory units per processor core, Core i7 and Xeon W fractions should be displayed as three times lower. The main observation from these measurements is that TPA utilizes the available bandwidth very close to maximum, while in Skylake side, good utilization is achieved only for the *blocked-ap* access pattern (utilized in the blocked Pthreads tests *block*, *bprefix*, *madd* and *threshold* of Sect. 4.1.1) in a single memory unit perspective per core only. This behavior is much more intensive in Xeon W where the number of cores is higher and the interconnect is more complex. This indicates serious scalability challenges in Skylake interconnect/memory architecture as predicted by our studies on architectural approaches and utilized interconnect networks for general purpose computing [7, 43].

The poor Skylake CPU results in straight-forward parallel tests even though *block*, *madd*, *threshold* use the blocked access pattern are caused by inefficient threading and synchronization that will be discussed in the next subsection.

4.1.4 Resource efficiency, threading and synchronization

The above favorable results for TCF processors are mostly because they can use the available resources efficiently and Skylake processors fail in that in certain demanding cases. TPA gains additional boost due to special techniques, such as support for concurrent memory access, multioperations and compute-update operations. There are also techniques/resources that favor Skylake processors. They, e.g., have three times more memory units than TPA and SIMD units that can process four times more data words per cycle in Core i7 and eight times more in Xeon W. We let the compiler determine whether multiple *Memory Unit* (MU) or SIMD instructions can be used but did not explicitly add them. According to our test with OpenMP, the SIMD units did not provide fourfold and eightfold speedups, accordingly.

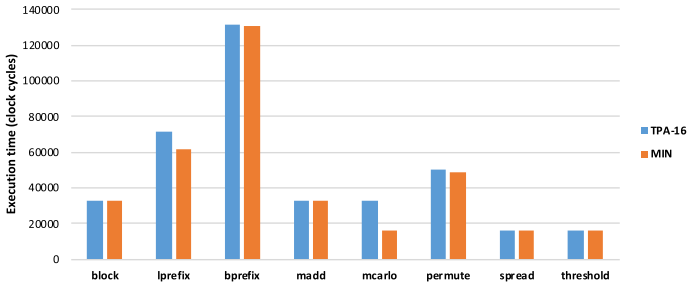


Fig. 13 Measured TPA execution time and the lower bound of execution time estimate based on obligatory memory references and available memory resources (MIN)

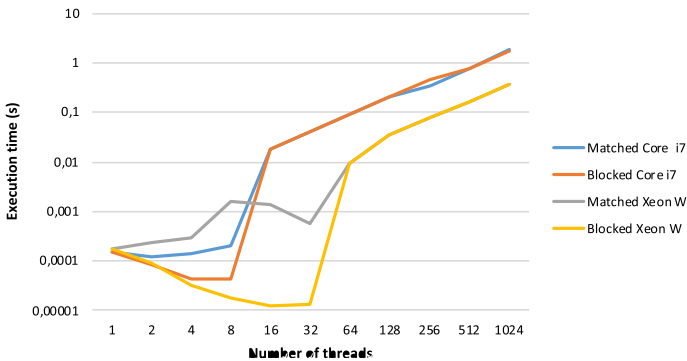


Fig. 14 Execution time of matched parallel and blocked Pthreads versions of the *block* benchmark as a function of the number of threads in Intel Skylake multicore CPUs. Note that the scale is logarithmic

In order to understand how well TPA utilizes the available memory system hardware resources, we compare the number of transferred data words evenly distributed among BEs to actual execution time. The results for TPA benchmarks are shown in Fig. 13. We observe that the average overhead of TPA execution is 11.56% corresponding to the utilization of HW is 89.64%. The benchmarks showing noticeably higher execution time than predicted by the memory traffic are *lprefix* and *mcario*. The former slowed down by quantification effects and the latter is computation-intensive algorithm that could be made fully utilized by adding a few functional units per BE.

As demonstrated with the straight-forward benchmarks above, the threading scheme used in multicore CPUs has a big effect on the performance. To study the further impact of the number of threads in them, we measured the execution time of matched parallel and blocked Pthreads versions of the *block* benchmark as a function of number of threads in Skylake processors (Fig. 14). We observe that lowest execution time is achieved when there is one thread per processor core or all the physical threads supported by Intel's 2-way multithreading are used. Increasing the number of threads beyond that decreases the performance radically. This is because thread switching time is 100 times higher than a typical

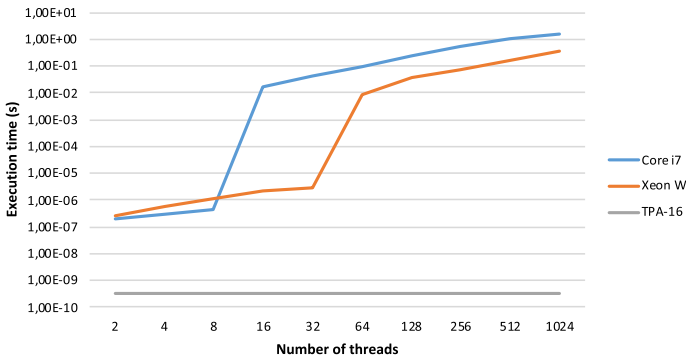


Fig. 15 Execution time of a barrier synchronization (*sync* benchmark) as a function of the number of threads in Intel Skylake multicore CPUs. Note that the scale is logarithmic

instruction execution latency and each thread does not utilize many instructions per an operating system time slot. The Skylake architecture's asynchronous nature plays a big role in this since unlike in TPA, the synchronization cost increases superlinearly as the number of threads increases (Fig. 15). Another observation is that with matched parallelism versions, there are no real speedup compared to the single thread case.

4.1.5 Factors of efficient programming

Based on the lessons learnt from the above overall tests and deeper analysis/measurements of memory access pattern, threading and synchronization, we can outline a multi-step guideline for avoiding the hurdles of multicore programming. Starting from the straight-forward parallel implementation, a programmer need to do the following five consecutive transformations:

- (i) *Primitive operations* Implementation of parallel primitives, such as concurrent memory access, multioperations and compute-update instructions, with available instructions
- (ii) *Synchronization* Identification of points where synchronization is needed to ensure the correctness of the parallel implementation
- (iii) *Synchronization minimization* Minimization of the number of synchronizations to avoid slowdown due to cost of synchronization
- (iv) *Threading* Matching the hardware and software parallelism to avoid inefficiency and scheduling problems observed in the straight-forward tests
- (v) *Locality maximization* Maximization of the locality of memory accesses to move the program away from straight-forward and matched parallelism parallel hurdles employing interleaved access and latency/contention effects.

Note that all these transformations (i)–(v) were applied to blocking test programs, while the straight-forward tests did only transformations (i)–(iii) and matched parallelism programs did transformations (i)–(iv). The cost of this is unfortunately

```

// Straight-forward Skylake version
1  if (tid >= thi)
2    pthread_exit(&status);
3  if (tid==0)
4  { NUM_THREADS-=1;
5    Sync_Thickness-=1; }
6  Synchronize;
7  thi=NUM_THREADS;
8  for (i=1; i<SIZE; i<<=1)
9  { tmp[tid]=A[tid];
10   Synchronize;
11   tmp[tid]+=A[tid+i];
12   Synchronize;
13   A[tid+i]=tmp[tid];
14   Synchronize;
15   if (thi->i0)
16     if (tid >= thi-i) {
17       pthread_exit(&status);
18     }
19     if (tid==0)
20     { NUM_THREADS-=i;
21       Sync_Thickness-=i; }
22     Synchronize;
23     thi=NUM_THREADS; } }

// Matched parallelism Skylake version
1  for (i=1; i<SIZE; i<<=1)
2  { for (id=tid; id<SIZE; id+=NUM_THREADS)
3    if (id->=0)
4      tmp[id]=A[id]+A[id-i];
5    else tmp[id]=A[id];
6    Synchronize;
7    for (id=tid; id<SIZE; id+=NUM_THREADS)
8      A[id]=tmp[id];
9    Synchronize; }

// Blocked Skylake version
1  blocksize=SIZE/NUM_THREADS;
2  start = tid * blocksize;
3  stop = start + blocksize;
4  for (i=1; i<SIZE; i<<=1)
5  { for (id=start; id<stop; id+=gap)
6    if (id->=0)
7      tmp[id]=A[id]+A[id-i];
8    else tmp[id]=A[id];
9    Synchronize;
10   for (id=start; id<stop; id+=gap)
11     A[id]=tmp[id];
12   Synchronize; }

// TPA version
1  for (i=1, #N-1; i<N; #-=i, i<<=1)
2    A_[$+i] += A_[$];

```

Fig. 16 The high-level language versions of the *lprefix* test program for TPA and Skylake. (# = thickness of the TCF, \$ = fiber identifier, tid = thread identifier, NUM_THREADS = number of threads in pthreads system, thi = thread private number of threads, Sync_Thickness = thread count information needed by barrier synchronization, SIZE = N = problem size). Note that for simplicity, the matched parallelism and blocked versions are not using exactly the same algorithm as the TCF and straight-forward versions

increasingly complex parallel programs as indicated by the active code line measurements. Actually, we see a clear trade–Off situation between performance and programmability in Skylake CPUs.

TCF processors do not need these transformations but the version designed for the idealized model are directly executable in them. Instead, a programmer can select straight-forward threading, convenient mapping of the functionality and suitable placement/partitioning of data.

4.2 Program examples

In order to illustrate the practical programming differences between TPA and Skylake CPUs, let us have a look at the *lprefix* test program that computes the prefix sum of an array A of N integers using the logarithmic algorithm. It is the most complex test program utilizing constantly altering inter-core communication pattern and reducing thickness during the execution. Figure 16 shows the TPA version of *lprefix* written in C-style parallel TCF language as well as straight-forward, matched parallelism and blocked Skylake versions in C/pthreads. These are the same program versions as used in the tests of Sect. 4.1.

The TPA version consists just of two active code lines and corresponds to a typical parallel computing textbook version of the algorithm [4, 6]. The functionality is implemented with a single TCF that executes the for loop while its thickness decreases iteratively by exponentially increasing steps starting from one.

The straight-forward Skylake version does the same thing as the TPA versions, but needs explicit synchronizations due to asynchronous execution of threads in Skylake. For the same reason, a temporary variable *tmp* is needed since the actual data array *A* changes its content along with the execution introducing risk of reading old data. Explicit *pthread_exit()* functions are needed to adjust the number of threads along the execution. As a result, the number of active code lines is 11 times higher than in TPA.

The straight-forward Skylake versions have catastrophically poor performance due to very slow switching and synchronization of threads (Fig. 15). Switching a thread in Skylake processors takes more than 100 clock cycles. A longer delay is caused by the operating system scheduler that typically allocates a much longer time slot for a thread than that needed for simple computation, such as *lprefix* iteration or synchronization variable checking repeatedly executed in the barrier synchronization. The programmer may therefore want to limit the number of threads. The matched parallelism Skylake version drops the number of parallel threads from *N* down to 4 in Core i7 and 18 in Xeon W. This happens by turning the parallel statements into loops that process just 4 or 18 elements in parallel. We dropped the altering thickness scheme of the algorithm since it would have been quite complex to decrease the number of threads with the loops so that the resulting behavior matches exactly to the straight-forward version. As a result, the number of active code lines drops to 9 but the algorithm is not exactly the same any more. If the altering thickness scheme would have been included the number of code lines would have been higher than in the straight-forward version.

The matched parallelism versions have also unnecessary low performance, especially for Xeon W. This problem can be partially be solved by minimizing the inter-core communication. The blocked Skylake test programs partition the data to blocks of *N/P* elements, where *P* is the number of cores, to maximize the locality of references and to avoid inter-core traffic as much as possible. The blocking increases the number of active program lines to 12 but again without the altering thickness scheme.

4.3 Programming experiences

The relative source code length tests of Sect. 4.1 indicate that TPA takes much fewer active code lines to express a kernel program functionality than Skylake, especially if the goal is to have good performance for the latter. The existing literature does not discuss how programmers experience this simplified coding in practise and whether the software development productivity is improved accordingly. Therefore, to give the first idea, we collected our experiences on (performance) programming from an expert programmer (a lot of experience) and an inexperienced programmer (not familiar with the theories of parallel computing

nor either of the used programming methodologies) points of view knowing that these are not statistically meaningful nor objective opinions. The set of programs the expert and inexperienced programmer created included the programs used in our comparison. The beginner's experiences related to programming include:

- “Trying out TPA assembler (for hand compiling) in parallel program implementation seemed quite impossible at the first glance but turned out easier than expected. There was no need to try out X86 assembler for Skylake but getting a parallel program working that way would have been difficult due to low-level complexity of Pthreads.”
- “Pthreads feels like an understandable way to describe parallel implementation. It allows a beginner to briefly sketch how the parallel program works, and first after that make decisions regarding block sizes and number of threads, which makes it easier for a beginner to start with. In the case of unexpected results, that happen often with Pthreads, the reason behind the problems is hard to discover and solve. With TPA, thickness and memory addresses must be thought about from the beginning. This makes it harder to get started with but can help to avoid some primitive mistakes when dividing data between threads.”
- “While programming (lprefix or any other program) with Pthreads, it was troublesome to decide how many threads should be utilized. After a brief introduction to parallel programming, it seemed like adding more threads always would increase the performance, but it turned out to be more complicated than this. With TPA, thickness must also be noticed, but its effects on the performance are easier to predict. Therefore, programming TPA feels in principle simpler than that for Skylake, but to be successful TPA definitely needs better tools than current academic quality ones.”
- “Sometimes synchronizing threads in Skylake programs felt more troublesome than writing the program itself. For a beginner it was challenging to decide when synchronization clauses are needed, and it felt tempting to use more of them than was needed, just to make sure that the program was working correctly. With TPA version, there was no need for synchronization clauses, which was a relief.”
- “OpenMP was useful tool while writing simple programs, which were straightforward to parallelize. However, in case of complex programs, OpenMP often did not behave exactly as expected. Sometimes it was difficult to find out if the unexpected results were due to a programming error or parallelization made by OpenMP.”

The expert experiences include notes on higher-level concepts behind the actual tools for programming and the resulting performance:

- “Getting decent performance out of parallel functionality targeted for Skylake multicore CPUs is much more difficult than that for TPA. The main issues with the Skylake are asynchronous nature of thread execution, high cost of synchronizations, threading model that is in practice bounded above to the number of HW threads if the granularity of interthread communication is not coarse.”

- “The factors that make parallel programming easy for TPA are the simplicity of allocating the right amount of parallelism with the TCF model, synchrony of execution that makes scheduling the operations to smoothly advancing consecutive steps easy, and insensitivity to partitioning and mapping choices that makes code portable.”
- “Even though TPA uses a VLIW instruction set with chaining of FUs, writing programs in assembler makes some sense for critical inner loop compilation.”
- “Performance portability and scalability are quite hard to achieve with Skylake CPUs while with TPA it is easy since TPA’s programming concept directs a programmer automatically to simple and adaptive code.”
- “While the idea of being able to retain the sequential algorithms in OpenMP programming may sound a good idea, there are a number of problems related to it. It prevents a programmer from applying proper explicit control leaving it many times unclear how the parallel version is actually implemented. It is also clear that in many cases the optimal parallel algorithm cannot be generated from a sequential algorithm by the compiler, leading to unoptimal results.”
- “For an experienced parallel programmer it was actually easier to achieve good performance and scalability with Pthreads rather than OpenMP. But if the architecture supports TCF execution, parallel computing becomes a lot simpler than with either of them.”

These experiences are not in contradiction with measurements that programmability of TPA for parallel functionalities is in a way better shape than that of Skylake multicore CPUs even though the latter have very strong collection of available tools as well as wide support of a large processor manufacturer, the software community and schools teaching programming and parallelism. One remarkable indicator of the scale of issues in the current approach is that the paradigm of software engineering is still mostly sequential although multicore CPUs have been exclusively used for almost 20 years.

5 Conclusions

We have made a performance and programmability comparison between TCF processors and current multicore CPUs. We focused on an entry-level TCF architecture TPA-16 against Intel Skylake client and server multicore CPUs Core i7 and Xeon W. The comparison was implemented by writing similar parallel programs for all processors with popular programming solutions (Pthreads, OpemMP and baseline TCF language), measuring the execution time with a clock accurate simulator (TPA) and actual computers (Skylake CPUs) and counting the active code lines of programs. According to our measurements, TPA-16 can perform significantly better than Intel Skylake Core i7 in the both categories: The included TCF programs execute 57.1 million times faster in TPA than their straight-forward parallel Pthreads counterparts in Core i7, while the latter needs twice the active code lines. If the parallelism of programs is matched to that of Core i7 hardware, the TPA-16 performance advantage still remains almost 22, while the Skylake code line count overhead increases

to three. By maximizing the locality using partitioning of data and functionality to blocks, the Core i7 is able to shrink the TPA-16 speedup down to 9.70 but the cost of this is at least six times longer programs. We compared TPA-16 also to high-end Intel Skylake Xeon W processor with more cores but over ten times the estimated silicon area and 3.4/27 times the memory bandwidth of TPA-16. The corresponding TPA-16 speedup factors are 12.6 million, 67.3 and 3.0, respectively. We studied also the reasons causing these slowdowns and scalability issues in Skylake and found substantial inefficiencies in support for different access patterns, synchronization cost and resource efficiency. Switching Pthreads to OpenMP can save a bit in code length while the performance and, especially scalability turned out to be quite weak. The recorded notes from a beginner and experienced programmer are not in contradiction with the measurements that TPA-16 is simpler to program than a Skylake CPU although TPA programming tools were primitive. Likewise, getting the performance predicted by the theory of parallel algorithms is easier with TPA-16. All in all, TPA-16 is able to achieve execution times that are very close to the theoretical minima with compact straight-forward parallel code, whereas current multi-core CPUs, such as Intel Skylake processors do not succeed so well at least with the included test programs and programming tools.

Our future work plans include comparing the performance and programmability of multicore CPUs thoroughly to different TPA configurations including SIMD-specific optimizations. For that, additional benchmarks, programming languages and other processors are considered. We are also looking possibility to implement TPA on silicon to be able to run real-life benchmarks and to extend comparisons also to energy efficiency.

Funding Open access funding provided by Technical Research Centre of Finland (VTT).

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References


1. International Technology Roadmap for Semiconductors, Semiconductor Industry Association, year 2003 edition. <http://www.itrs.net/>
2. Research at Intel From a Few Cores to Many (2006): A Tera-scale Computing Research Overview, White Paper, Intel,
3. Patterson D (2010) The trouble with multicore. *IEEE Spectr* 47(7):28–32
4. Jaja J (1992) Introduction to parallel algorithms. Addison-Wesley, Reading

5. Mäkelä J-M, Forsell M, Leppänen V (2017) Towards a language framework for thick control flows. In: Proc. of the High Level Programming Models and Supporting Environments (HIPS'17), May 29, 2017, Orlando, FL, USA
6. Keller J, Kessler C, Träff JL (2001) Practical PRAM programming. Wiley, New York
7. Forsell M (2002) Architectural differences of efficient sequential and parallel computers. *J Syst Architect* 47(13):1017–1041
8. Forsell M, Leppänen V (2013) An extended PRAM-NUMA model of computation for TCF programming. *Int J Netw Comput* 3(1):98–115
9. Forsell M (2018) Accelerating general purpose parallel computing with the TPA architecture. In: ScalPerf 2018, September 23–28, 2018, Bertinoro, Italy
10. Leppänen V, Forsell M, Makela J-M (2011), Thick control flows: introduction and prospects. In: Proc. PDPTA 2011, July 18–21, Las Vegas, USA, pp 540–546
11. Forsell M, Roivainen J, Leppänen V (2016) Outline of a thick control flow architecture. In: Proc. MPP 2016, SBAC-PAD 2016, October 26–28, 2016, Marina del Rey Marriott, Los Angeles, USA
12. REPLICa Multiprocessor Framework, White Paper, VTT (2020)
13. Forsell M, Roivainen J, Leppänen V, Träff JL (2018) Implementation of multioperations in thick control flow processors. In: Proc. APDCM'18, May 21–25, 2018, Vancouver, Canada
14. Forsell M (2018) Flexible fibering scheme for thick control flow processors. In: Proc. PDPTA'18, July 30–August 2, 2018, Las Vegas, USA, pp 16–20
15. Forsell M, Roivainen J, Leppänen V, Träff JL (2018) Supporting concurrent memory access in TCF processor architectures. *Microprocess Microsyst* 63:226–236
16. Hansson E, Alnervik E, Kessler C, Forsell M (2014) A quantitative comparison of PRAM based emulated shared memory architectures to current multicore CPUs and GPUs. In: Proceedings of the 11th Workshop on Parallel Systems and Algorithms (PASA'14) in Conjunction with the 27th International Conference on Architecture of Computing Systems (ARCS'14), February 25–26, Luebeck, Germany, pp 1–7
17. Forsell M, Roivainen J, Leppänen V (2018) REPLICa MBTAC—multithreaded dual mode processor. *J Supercomput* 74(5):1911–1933
18. Schwartz J (1980) Ultracomputers. *ACM Trans Program Lang Syst* 2(4):484–521
19. Ranade A, Bhatt S, Johnsson S (1988) The fluent abstract machine. In: Proc. Fifth MIT Conference on Advanced Research in VLSI, March 1988, pp 71–94; TR-573. Department of Computer Science, Yale University
20. Forsell M (2002) A scalable high-performance computing solution for network on chips. *IEEE Micro* 22(5):46–55
21. Vishkin U (2011) Using simple abstraction to reinvent computing for parallelism. *Commun ACM* 54(1):75–85
22. Vishkin U (2014) Is multicore hardware for general-purpose parallel processing broken? *Commun ACM* 57(4):35–39
23. Ghanim F, Vishkin U, Barua R (2018) Easy PRAM-Based high-performance parallel programming with ICE. *IEEE Trans Parallel Distrib Syst* 29(2):377–390
24. Valiant L (2011) A bridging model for multi-core computing. *J Comput Syst Sci* 77(1):154–166
25. Forsell M, Nikula S, Roivainen J (2021) Preliminary performance and programmability comparison of the thick control flow architecture and current multicore CPUs. In: Arabnia H, Deligiannidis L, Grimaila M, Hodson D, Joe K, Sekijima M, Tinetti F (eds) *Advances in Parallel & Distributed Processing, and Applications: Proceedings from PDPTA'20, CSC'20, MSV'20, and GCC'20* (July 27–30, 2020, Las Vegas, Nevada, USA). Springer
26. Ranade A (1991) How to emulate shared memory. *J Comput Syst Sci* 42:307–326
27. Forsell M (1996) Minimal pipeline architecture—an alternative to superscalar architecture. *Microprocess Microsyst* 20(5):277–284
28. Skylake (client)—Microarchitectures—Intel, Wiki Chip www document at address: [https://en.wikipedia.org/wiki/intel/microarchitectures/skylake_\(client\)](https://en.wikipedia.org/wiki/intel/microarchitectures/skylake_(client)). Accessed March 14, 2021
29. Skylake (server)—Microarchitectures—Intel, Wiki Chip www document available at [http://en.wikipedia.org/wiki/intel/microarchitectures/skylake_\(server\)](http://en.wikipedia.org/wiki/intel/microarchitectures/skylake_(server)). Accessed March 14, 2021
30. Flynn M (1972) Some computer organizations and their effectiveness. *IEEE Trans Comput* 21(9):948–960
31. Fortune S, Wyllie J (1978) Parallelism in random access machines. In: Proceedings of 10th ACM STOC, Association for Computing Machinery, New York, pp 114–118
32. Almasi G, Gottlieb A (1994) Highly parallel computing. Benjamin/Cummings, Redwood City

33. Skillicorn D, Talia D (1998) Models and languages for parallel computation. *ACM Comput Surv* 30(2):123–169
34. Culler D, Singh J (1999) *Parallel computer architecture—a hardware/ software approach*. Morgan Kaufmann Publishers Inc., San Fransisco
35. Rajasekaran S, Reif J (eds) (2008) *Chapman Handbook of parallel computing—models algorithms and applications*. Hall/CRC
36. Kirk D, Hwu W-M (2010) *Programming massively parallel processors—a hands-on approach*. Morgan Kaufmann
37. Pacheco P (2011) *An introduction to parallel programming*. Morgan Kaufmann
38. The MPI Forum, CORPORATE (November 15–19, 1993), MPI: a message passing interface. In: *Proc. 1993 ACM/IEEE Conference on Supercomputing*
39. Lewis B, Berg D (1996) *PThreads primer: a guide to multithreaded programming*. Sunsoft Press
40. Chandra R, Menon R, Dagum L, Kohr D, Maydan D, McDonald J (2001) *Parallel programming in OpenMP*. 1st edn. Morgan Kaufmann Publishers
41. Carlson W, Draper J, Culler D, Yelick K, Brooks E, Warren K, Livermore L (1999) Introduction to UPC and language specification. In: *CCS-TR-99-157*, IDA Center for Computing Sciences
42. Forsell M, Roivainen J, Leppänen V (2016) The REPLICA on-chip network. In: *NORCAS 2016*, November 1–2, 2016, Copenhagen, Denmark
43. Forsell M, Leppänen V, Penttonen M (2015) Cost of bandwidth-optimized sparse mesh layouts. In: *Proceedings of 13th International Conference on Parallel Computing Technologies (PaCT'15)*, Lecture Notes in Computer Science (LNCS), vol 9251, August 31–September 4, 2015, Petrozavodsk, Russia, pp 375–389

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Martti Forsell¹  · Sara Nikula¹ · Jussi Roivainen¹ · Ville Leppänen² · Jesper Larsson Träff³

Sara Nikula
Sara.Nikula@VTT.Fi

Jussi Roivainen
Jussi.Roivainen@VTT.Fi

Ville Leppänen
Ville.Leppanen@utu.fi

Jesper Larsson Träff
traff@par.tuwien.ac.at

¹ VTT, Box 1100, FI-90571 Oulu, Finland

² Department of Computing, University of Turku, FI-20014 Turku, Finland

³ Faculty of Informatics, Vienna University of Technology, Vienna, Austria