How to Extend Single-Processor Approach to Explicitly Many-Processor Approach

János Végh

1 Introduction

Physical implementations of a computer processor in the 70-year old computing paradigm have several limitations [1]. As the time passes, more and more issues come to light, but development of processor, the *central* element of a computer, could keep pace with the growing demand on computing till some point. Around 2005 it became evident that the price paid for keeping Single Processor Approach (SPA) paradigm [2], (as Amdahl coined the wording), became too high. "The *implicit hardware/software contract*, that increases transistor count and power dissipation, was OK as long as architects maintained the existing sequential programming model. This contract led to innovations that were inefficient in transistors and power—such as multiple instruction issue, deep pipelines, out-of-order execution, speculative execution, and prefetching—but which increased performance while preserving the sequential programming model" [3]. The conclusion was that "*new ways of exploiting the silicon real estate need to be explored*" [4].

"*Future growth in computing performance must come from parallelism*" [5] is the common point of view. However, "*when we start talking about parallelism and ease of use of truly parallel computers, we're talking about a problem that's as hard as any that computer science has faced*" [3]. Mainly because of this, parallel utilization of computers could not replace the energy-wasting solutions introduced

Projects no. 125547 and 135812 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the K funding scheme. Also the support by project 101005020 from ERC is acknowledged.

J. Végh (✉)
Kalimános BT, Debrecen, Hungary

H. R. Arabnia et al. (eds.), *Advances in Software Engineering, Education, and e-Learning*, Transactions on Computational Science and Computational Intelligence, https://doi.org/10.1007/978-3-030-70873-3_31

to the formerly favorited single-thread processors. They remained in the Multi-Core and/or Many-Core (MC) processors, greatly contributing to their dissipation and, through this, to the overall crisis of computing [6].

Computing paradigm itself, the *implicit hardware/software contract*, was suspected even more explicitly: "*Processor and network architectures are making rapid progress with more and more cores being integrated into single processors and more and more machines getting connected with increasing bandwidth. Processors become heterogeneous and reconfigurable …No current programming model is able to cope with this development, though, as they essentially still follow the classical van Neumann model*" [7]. On one side, when thinking about "advances beyond 2020", the solution was expected from the "*more efficient implementation of the von Neumann architecture*" [8]. On the other side, there are statements such as "*The von Neumann architecture* is fundamentally inefficient and non-scalable for representing massively interconnected neural networks" [9].

In our other works [10–13] we have pointed out that one of the major reasons is neglecting the temporal behavior of computing components. The other major reason is, that the architecture developed for that classic paradigm is development-unaware, and cannot be equally good for the present needs and the modern paradigm. These two reasons together represent the major bottleneck—among others—to build supercomputers having reasonable efficiency in solving real-life tasks and biology-mimicking systems with the required size and efficiency, such as Artificial Intelligence (AI)s [14, 15] and brain simulators [16]. The interplay of these two reasons is that conventional processors do not have autonomous communication. The classic paradigm is about a segregated processor and, because of this, its communication is implemented using Input/Output (I/O) instructions and needs help of the operating system (OS). Both of these features increase non-payload (and sequential!) portion of the code and so they degrade efficiency, especially in excessive systems.

It is worth, therefore, to scrutinize that *implicit hardware/software contract*, whether the processor architecture could be adapted in a better way to the changes that occurred in the past seven decades in technology and utilization of computing. *Implicitly*, both hardware (HW) and software (SW) solutions advantageously use multi-processing. The paper shows that using a less rigid interpretation of terms that that contract is based upon, one can extend the single-thread paradigm to use several processors *explicitly* (enabling direct core-to-core interaction), without violating the 'contract', the 70-year old HW/SW interface.

Section 2 shortly summarizes some of the major challenges, modern computing is expected to cope with and sketches the principles that enable it to give a proper reply. The way to implement those uncommon principles proposed here is discussed in Sect. 3. Because of the limited space, only a few of the advantages are demonstrated in Sect. 4.

2 The General Principles of EMPA

During the past two decades, computing developed in direction to conquer also some extremes: the ‘ubiquitous computing’ led to billions of connected and interacting processors [17], the always higher need for more/finer details, more data and shorter processing times led to building computers comprising millions of processors to target challenging tasks [18], different cooperative solutions [19] attempt to handle the demand of dynamically varying computing in the present, more and more mobile, computing. *Using computing under those extreme conditions led to shocking and counter-intuitive experiences* that can be comprehended and accepted using parallels with modern science [10].

Developing a new computing paradigm being able to provide a theoretical basis for the state of the art of computing cannot be postponed anymore. Based on that, one must develop different types of processors. As was admitted following the failure of supercomputer Aurora’18: “*Knights Hill* was canceled and instead be replaced by a “new platform and new microarchitecture specifically designed for exascale”” [20]. Similarly, we expect shortly to admit that building large-scale AI systems is simply not possible based on the old paradigm and architectural principles [14, 15, 21]. The new architectures, however, require a new computing paradigm, that can give a proper reply to power consumption and performance issues of our present-day computing.

2.1 Overview of the Modern Paradigm

The new paradigm proposed here is based on fine distinctions in some points, present also in the old paradigm. Those points, however, must be scrutinized individually, whether and how long omissions can be made. These points are:

- consider that *not only one processor* (aka Central Processing Unit) exists, i.e.
  - processing capability is *one of the resources* rather than a central singleton
  - not necessarily *the same processing unit* is used to solve all parts of the problem
  - a kind of redundancy (an easy method of replacing a flawed processing unit) through using virtual processing units is provided (mainly to *increase the mean time between technical errors*)
  - instruction stream can be transferred to another processing unit [22, 23]
  - *different processors can and must cooperate* in solving a task, including direct data and control exchange between cores, communicating with each other, being able to set up ad-hoc assemblies for more efficient processing in a flexible way
  - the large number of processors can be used for *replacing memory operations with using more processors*
  - a core can outsource the received task

- misconception of segregated computer components is reinterpreted
  - *efficacy of using a vast number of processors is increased* by using multi-port memories (similar to [24])
  - a "memory only" concept (somewhat similar to that in [25]) is introduced (as opposed to the "registers only" concept), using *purpose-oriented, optionally distributed, partly local, memory banks*
  - principle of locality is introduced at hardware level, through introducing hierarchic buses
- misconception of "sequential only" execution [26] is reinterpreted
  - von Neumann required only "proper sequencing" for a single processing unit; this concept is *extended* to several processing units
  - tasks are broken into reasonably sized and logically interconnected fragments
  - the "one-processor-one process" principle remains valid for task fragments, but not necessarily for the complete task
  - fragments can be executed (at least partly) simultaneously if both data dependence and hardware availability enables it (another kind of asynchronous computing [27])
- a closer hardware/software cooperation is elaborated
  - hardware and software only exist together: the programmer works with virtual processors, in the same sense as [28] uses this term, and lets computing system to adapt itself to its task at run-time, through mapping virtual processors to physical cores
  - when a hardware has no duty, it can sleep ("does not exist", does not take power)
  - the overwhelming part of the duties such as synchronization, scheduling of the OS are taken over by the hardware
  - the compiler helps work of the processor with compile-time information and the processor can adapt (configure) itself to its task depending on the actual hardware availability
  - strong support for multi-threading, resource sharing and low real-time latency is provided, at HW level
  - the internal latency of large-scale systems is much reduced, while their performance is considerably enhanced
  - task fragments shall be able to return control voluntarily without the intervention of OS, enabling to implement more effective and more simple operating systems
  - the processor becomes "green": only working cores take power

2.2 Details of the Concept

We propose to work at programming level with *virtual processors* and to map them to physical cores at run-time, i.e., to *let the computing system to adapt itself to its task*. A major idea of *EMPA* is to use *quasi-thread (QT)* as atomic unit of processing, that comprises both *HW* (the physical core) and the *SW* (the code fragment running on the core). Its idea was derived with having in mind the best features of both *HW* core and *SW* thread. *QT*s have "*dual nature*" [10]: in the HW world of "classic computing" they are represented as a 'core', in SW world as a 'thread'. However, they are the same entity in the sense of 'modern computing'. We borrow the terms 'core' and 'thread' from conventional computing, but in 'modern computing', they can actually exist only together in a time-limited way.[1] *EMPA is a new computing paradigm* (for an early version see [29]) which needs a new underlying architecture, *rather than a new kind of parallel processing* running on a conventional architecture, so *it can be reasonably compared to terms and ideas used in conventional computing only in a minimal way*; although the new approach adapts many of its ideas and solutions, furthermore borrows its terms, from 'classic computing'.

One can break the executable task into reasonably sized and loosely dependent Quasi-Thread (QT)s. (The QTs can optionally be nesteded, akin to subroutines.) In *EMPA*, *for every new QT a new independent Processing Unit (PU) is also implied, the internals (PC and part of registers) are set up properly*, and they execute their task independently[2] (but under the supervision of the processor comprising the cores).

In other words: *we consider processing capacity as a computing resource* in the same sense as memory is considered as a storage resource. This approach enables programmers to work with virtual processors (mapped to physical *PU*s by the computer at run-time) and they can utilizequick resource *PU*s to replace utilizing slow resource memory (say, renting a quick processor from a core pool can be competitive with saving and restoring registers in slow memory, for example when making a subroutine call). The third primary idea is that *PUs can cooperate* in various ways, including data and control synchronization, as well as *outsourcing part of the received job (received as an embedded QT)* to a helper core. An obvious example is to outsource housekeeping activity to a helper core: counting, addressing, comparing, can be done by a helper core, while the main calculation remains to the originally delegated core. As mapping to physical cores occurs at run-time (a

[1]Akin to dynamic variables on the stack: their lifetime is limited to the period when the HW and SW are appropriately connected. The physical memory is always there, but it is "stack memory" only when handled adequately by the HW/SW components.

[2]Although the idea of executing the single-thread task "in pieces" may look strange for the first moment, the same happens when the OS schedules/blocks a task. The key differences are that in EMPA *not the same* processor is used, the Explicitly Many-Processor Approach (EMPA) cuts the task into fragments in a reasonable way (preventing issues like priority inversion [30]). The QTs can be processed *at the same time* as long as their mathematical dependence and the actual HW resource availability enable it.

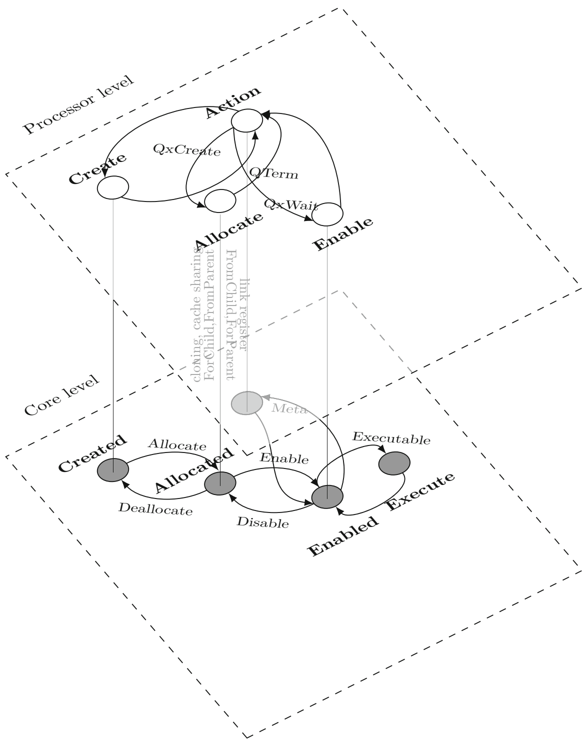

Fig. 1 EMPA processors have a two-layer processing: configuring resources and computing, respectively

function of actual HW availability), an EMPA processor can avoid using (maybe temporarily) denied cores as well as to adapt the resource need (requested by the compiler) of their task to actual computing resource availability.

Processor has an additional control layer for organizing joint work of its cores, see Fig. 1. Cores have just a few extra communication signals and can execute both conventional and so-called meta-instructions (for configuring their internal architecture) in the two layers. A core executes a meta-instruction in a co-processor style: when finding a meta-instruction, the core notifies its processor which suspends conventional operation of the core, then controls executing the meta-instruction (utilizing resources of the core, providing helper cores and handling connections between cores as requested), then resumes core operation.

The processor needs to find the needed *PUs* (cores), and its processing ability has to accommodate to the received task. Also, inside the processor, quickly, flexibly, effectively, and inexpensively. *A kind of 'On demand' computing that works 'As-a-Service'*. This task is not only for the processor: the complete computing system must participate, and for that goal, the complete computing stack must be rebuilt.

Behind former attempts to optimize code execution inside processor, there was no established theory, and they had only marginal effect, because processor is working in real-time, it has not enough resources, knowledge and time do discover those options entirely [31]. In contrary, compiler can find out anything about enhancing performance but has no information about actual run-time HW availability. Furthermore, it has no way to tell its findings to the processor. Processor has HW availability information but has to "reinvent the wheel" to enhance its performance; in real-time. In EMPA, compiler puts its findings in the executable code in form of meta-instructions ("configware"), and the actual core executes them with the assistance of a new control layer of the processor. The processor can choose from those options, considering actual HW availability, in a style '**if** NeededNumberOfResourcesAvalable **then** Method1 **else** Method2', maybe nested one into another.

2.3 Some Advantages of EMPA

The approach results in several considerable advantages, but the page limit enables us to mention just a few of them.

- as a new *QT* receives a new PU, there is no need to save/restore registers and return address (less memory utilization and less instruction cycles)
- OS can receive its PU, initialized in kernel mode and can promptly (i.e., without the need of context change) service the requests from the requestor core
- for resource sharing, a PU can be temporarily delegated to protect the critical section; the next call to run the code fragment with the same offset shall be delayed (by the processor) until processing by the first PU terminates
- processor can natively accommodate to the variable need of parallelization
- out-of-use cores are waiting in low energy consumption mode
- hierarchic core-to-core communication greatly increases memory throughput
- asynchronous-style computing [32] largely reduces loss stemming from the gap [33] between speeds of processor and memory
- *principle of locality can be applied inside the processor*: direct core-to-core connection (more dynamic than in [34]) greatly enhances efficacy in large systems [35]
- the communication/computation ratio, defining decisively efficiency [11, 15, 36], is reduced considerably
- QTs thread-like feature akin to $fork()$ and hierarchic buses change the dependence of the time of creating many threads on the number of cores from linear to logarithmic (enables to build exascale supercomputers)
- inter-core communication can be organized in some sense similar to Local Area Network (LAN)s of computer networking. For cooperating, cores can prefer cores in their topological proximity
- as the processor itself can candle scheduling signals in HW and in most cases the number of runnable tasks does not exceed the number of available computing resources, the conventional scheduling iin multi-tasking systems can be reduced considerably

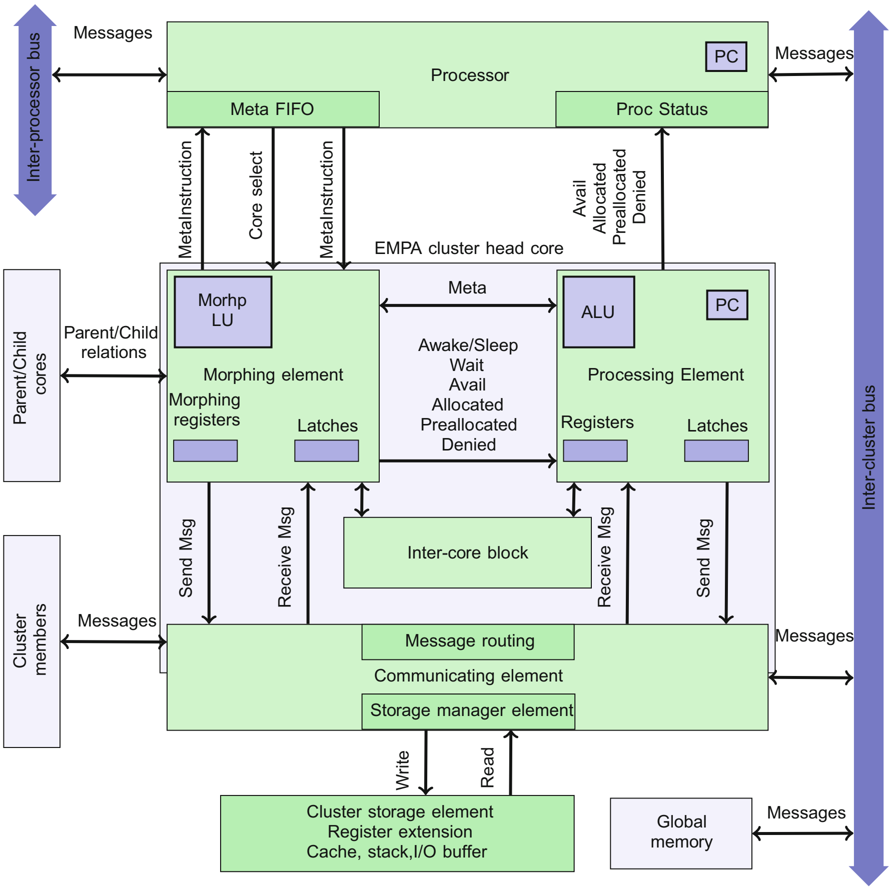

Fig. 2 The logical overview of the EMPA-based computing.

3 How to Implement EMPA

The best starting point to understand implementation of EMPA principles is conventional many-core processors. Present electronic technology made kilo-core processors available [37, 38], in a very inexpensive way and in immediate proximity of each other, in this way making the computing elements a “free resource” [39]. Principles of SPA, however, enable us to use them in a rather ineffective way [40]. Their temporal behavior [12] not only makes general-purpose cores ineffective [41], but their mode of utilization (mainly: their interconnection) leads to very low efficiency in performing real-life tasks [16, 42].

Given that true parallelism cannot be achieved (working with components anyhow needs time and synchronization via signals and/or messages, the question is only time resolution), *EMPA targets an enhanced and synchronized paral-*

lelized sequential processing based on using many cooperating processors. The implementation uses variable granularity and as much truly parallel portions as possible. However, *focus is on the optimization of the operation of the system, rather than providing some new kind of parallelization*. Ideas of cooperation comprise job outsourcing, sharing different resources and providing specialized many-core computing primitives in addition to single-processor instructions; as well as explicitly introducing different types of memory.

In this way *EMPA is an extension of SPA*: conventional computing is considered consisting of a single non-granulated thread, where (mostly) SW imitates the required illusion of granulating and synchronizing code fragments. Mainly because of this, many of components have a name and/or functionality familiar from conventional computing. Furthermore, we consider the *computing process as a whole* to be the subject of optimization rather than segregated components individually.

In SPA, there is only one active element, the Central Processing Unit (CPU). The rest of components of the system serves requests from CPU in a passive way. As EMPA wants to *extend* conventional computing, rather than to *replace* it, its operating principle is somewhat similar to the conventional one, with important differences in some key points. Figure 2 provides an overview of operating principle and major components of EMPA. We follow hints by Amdahl: "*general purpose computers with a generalized interconnection of memories or as specialized computers with geometrically related memory interconnections and controlled by one or more instruction streams*" [2].

3.1 The Core

An EMPA core of course comprises an EMPA Processing Element (EPE). Furthermore, it addresses two key deficiencies of conventional computing: inflexibility of computing architecture by EMPA Morphing Element (EME), and lack of autonomous communication by EMPA Communicating Element (ECE). Notice the important difference to conventional computing: *the next instruction can be taken either from memory pointed out by the instruction pointer (conventional instruction) or from the Meta FIFO (morphing instruction).*

The Processing Element

The EPE receives an address, fetches the instruction (if needed, also its operands). If the fetched instruction is a meta-instruction, EPE sets its 'Meta' signal (changes to 'Morphing' regime) for the EME and waits (suspends processing instructions) until the EME clears that signal.

The Morphing Element

When EPE sets ‘Meta’ signal, EME comes into play. Since the instruction and its operands are available, it attempts to process the received meta-instruction. However, the meta-instruction refers to resources handled by the processor. At processor level, order of execution of meta-instructions depends on their priority. Meta-instructions, however, may handle the ‘Wait’ of the core signal correspondingly. Notice that the idea is different from configurable spatial accelerator [43, 44]: the needed configuration is assembled ad-hoc, rather than chosen from a list of preconfigured assemblies.

Unlike in SPA, communication is a native feature of EMPA cores and it is implemented by ECE. Core assemble message content (including addresses), then after setting a signal, the message is routed to its destination, without involving a computing element and without any respect to where destination is. Message finds its path to its destination autonomously, using EMPA’s hierarchic bus system and ECEs of the fellow cores, taking the shortest (in terms of transfer time) path. Sending messages is transparent for both programmer and EPE.

The Storage Management Element

EMPA Storage Manager Element (ESME) is implemented only in cluster head cores, and its task is to manage storage-related messages passing through ECE. It has the functionality (among others) similar to that of memory management unit and cache controller in conventional computing.

3.2 Executing the Code

The Quasi-Threads

Code (here it means a reasonably sized sequence of instructions) execution begins with ‘hiring’ a core: the cores by default are in a ‘core pool’, in low energy consumption mode. The ‘hiring core’ asks for a helper core from its processor. If no cores are available at that moment, the processor sets the ‘Wait’ signal for the requester core and keeps its request pending. At a later time, processor can serve this pending request with a ‘reprocessed’ core.

Notice that the idea is quite different from the idea of e**X**plicit **M**ulti**T**hreading [45, 46]. Although they share some ideas such as the need for fine-grained multi-threaded programming model and architectural support for concurrently executing multiple contexts on-chip, unlike XMTs, QTs embody not simply mapping the idea of multi-threading to HW level. QTs are based on a completely unconventional computing paradigm; they can be nested.

This operating principle also means that code fragment and active core exist only together, and this combination (called Quasi-Thread) has a lifetime. Principle of the implementation is akin to that of the 'dynamic variable'. EMPA hires a core for executing a well-defined code fragment, and only for the period between creating and terminating a QT. In two different executions, the same code fraction may run on different physical cores.

Process of Code Execution

When a new task fragment appears, an EMPA processor must provide a new computing resource for that task fragment (a new register file is available). Since an executing core is 'hired' only for the period of executing a specific code fragment, it must be returned to core pool when execution of the task fragment terminates. The 'hired' PU is working on behalf of the 'hiring' core, so it must have the essential information needed for performing the delegated task. Core-to-core register messages provide a way to transfer register contents from a parent core to a child core.

Beginning execution of an instruction sets signal 'Meta', i.e. selects either EPE or EME for the execution, and that element executes the requested action. The acting core repeats the process until it finds and 'end of code fragment' code. Notice the difference to conventional computing: processing of the task does not terminate; only the core is put back into 'core pool' as at the moment it is not anymore needed.

When 'hired' core becomes available, processing continues with fetching an instruction by the 'hired' core. For this, the core sends a message with the address of the location of the instruction. The requested memory content arrives at the core in a reply message logically from the addressed memory, but the ESME typically intercepts the action. The process is similar to the one in conventional computing. However, here memory triggers sending a reply to the request when it finds the requested contents, rather than keeping the bus busy. Different local memories, such as addressable cache, can also be handled. Notice also that the system uses complete messages (rather than simple signals with the address); this makes possible accessing some content independently from its location, although it needs location-dependent time.

Of course, 'hiring' core wants to get back some results from the 'hired' core. When starting a new QT, 'hiring' core also defines, with sending a mask, which register contents the hired core shall send back. In this case, synchronization is a serious issue: parent core utilizes its registers for its task, so it is not allowed to overwrite any of its registers without an explicit request from parent. Because of this, when a child terminates, it writes the expected register contents to a latch storage of the parent, then it may go back to 'core pool'. When parent core reaches the point where it needs register contents received from its child, it explicitly asks to clone the required contents from latches to its corresponding register(s). It is the parent's responsibility to issue this command at such a time when no accidental register overwriting can take place.

Notice that beginning execution of a new code fragment needs more resources, while terminating it frees some resources. Because of this, terminating a QT has a higher priority than creating one. This policy, combined with that cores are able to wait until their processor can provide the requested amount of resources, prevents “eating up” computing resources when the task (comprising virtually an infinite number of QTs) execution begins.

Compatibility with Conventional Computing

Conventional code shall run on an EMPA processor (as an implicitly created QT). However, that code can only use a single core, since it contains no meta-instructions to create more QTs. This feature enables us to mix EMPA-aware code with conventional code, and (among others) to use the plethora of standard libraries without rewriting that code.

Synchronizing the Cooperation

The cores execute their instruction sequences independently, but their operation must be synchronized at several points. Their initial synchronization is trivial: processing begins when the ‘hired’ core received all its required operands (including instruction pointer, core state, initial register contents, mask of registers the contents of which the hiring core requests to return). The final synchronization on the side of ‘hired’ core is simple: the core simply sends contents of the registers as was requested at the beginning of executing the code fragment.

On the side of a ‘hiring’ core, the case is much more complex. The ‘hiring’ core may wait for the termination of the code fragment running on the ‘hired’ core, or maybe it is in the middle of its processing. In the former case, a simple waiting until the message arrives is sufficient, but in the latter case, receiving some new register contents in some inopportune time would destroy its processing. Because of this, register contents from the ‘hired’ core are copied to the corresponding registers only when the ‘hiring’ core requests so explicitly. Figure 3 attempts to illustrate the complex cooperation between EMPA components.

3.3 Organizing ‘ad hoc’ Structures

EME can ‘morph’ nternal architecture of the EMPA processor, as required by the actual task (fragment). EMPA uses principle of creating ‘parent-child’ (rather than ’Master-Slave’) relation between its cores. The ‘hiring’ core becomes parent, and the ‘hired’ core becomes child. A child has only one parent, but parents can have any number of children. Children can become parents in some next phase of execution; in this way, several ‘generations’ can cooperate. This principle provides

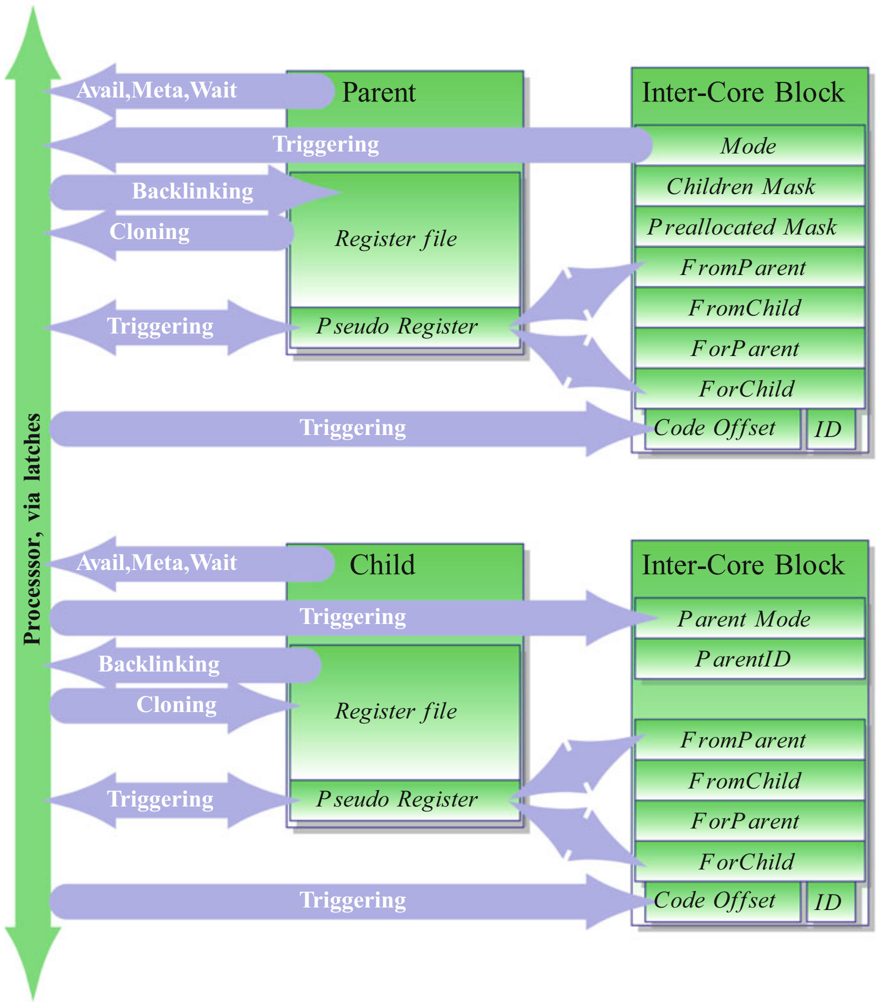

Fig. 3 Implementing the parent-child relationships: registers and operations of the EICB

a dynamic processing capacity for different tasks (in different phases of execution). The 'parent-child' relations simply mean storing addressing information, in the case of children combined with concluding the address from 'hot' bits of a mask.

As 'parents are responsible for their children', parents cannot terminate their code execution until all their children returned result of the code fragment that their parent delegated for them. This method enables parents also to trust in their children: when they delegate some fragment of their code to their children, they can assume that that code fragment is (logically) executed. It is the task of compiler to provide the required dependence information, how those code fragments can be synchronized.

This fundamental cooperation method enables the purest form of delegating code to existing (and available) cores. In this way, all available processing capacity can be used, while only the actually used cores need energy supply (and dissipate). *Despite its simplicity, this feature enables us to make subroutine calls without needing to save/restore contents through memory and to implement mutexes working thousands of times quicker than in conventional computing*.

3.4 Processor

Processor comprises many physical EMPA cores. An EMPA processor appears in role of a 'manager' rather than a number-crunching unit, it only manages its resources.

Although individual cores initiate meta-instructions, their synchronized operation requires the assistance of their processor. Meta-instructions received by EMPA cores are written first (without authorization) in a priority-ordered queue (Meta FIFO) in the processor, so the processor can always read and execute only the highest priority meta-instruction (a core can have at most one active meta-instruction).

3.5 Clustering the Cores

The idea of arranging EMPA cores to form clusters is somewhat similar to that of CNNs [47]. In computing technology, one of the most severe limitations is defined by internal wiring, both for internal signal propagation time and area occupied on the chip [1]. In conventional architectures, cores are physically arranged to form a 2-dimensional rectangular grid matrix. Because of SPA, there should not be any connection between segregated cores, so the inter-core area is only used by some kind of internal interconnection networks or another wiring.

In EMPA processors, even-numbered columns in the grid are shifted up by a half grid position. In this way cores are arranged in a way that they have common boundaries with cores in their neighboring columns. In addition to these neighboring cores, cores have (up to two) neighbors in their column, with altogether up to six immediate neighbors, with common boundaries. This method of positioning also means that cores, logically, can be arranged to form a hexagonal grid, as shown in Fig. 4. Cores *physically* have a rectangular shape with joint boundaries with their neighbors, but *logically* they form a hexagonal grid. This positioning enables to form "clusters" of cores, forming a "flower": an orange *ovary* (the cluster head) and six *petal*s (the leaf cores of cluster, the members).

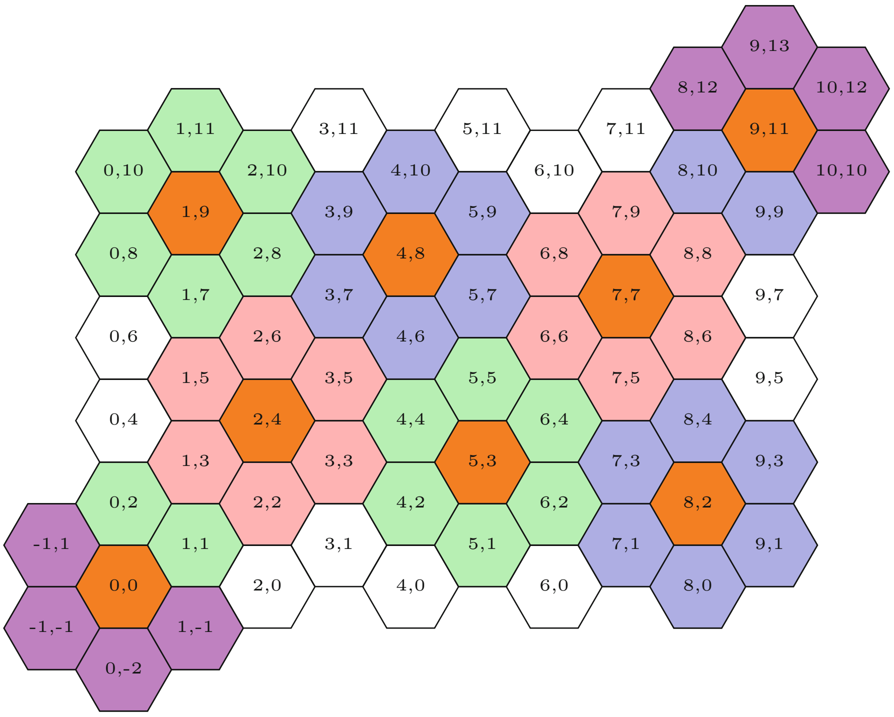

Fig. 4 The (logically) hexagonal arrangement (internal clustering) of EMPA cores in the EMPA processor

Between cores arranged in this way also neighborhood size can be interpreted similarly to the case of cellular computing. Based on neighborhood of size $r = 1$ (that means that cores have precisely one common boundary), a cluster comprising up to six cores (cluster members) can be formed, with the orange cell (of size $r = 0$, the cluster head) in the middle. Cluster members have shared boundaries with their immediate neighbors, including their cluster head. These cores define the external boundary of the cluster (the “flower”). Cores within this external boundary are “ordinary members” of the cluster, and the one in the central position is *head of the cluster*.

There are also “corresponding members” (of size $r = 2$): cores having at least one common boundary with one of the “ordinary members” of the cluster. “Corresponding members” may or may not have their cluster head, but have a common boundary with one of the “ordinary members”. White cells in the figure represent “external members” (also of size $r = 2$): they have at least one common boundary with an “ordinary member”, like the “corresponding members”, but unlike the “corresponding members” they do not have their cluster head. Also, there are some “phantom members” (see the violet petals in the figure) around the square edges in the figure: they have a cluster head and the corresponding cluster address,

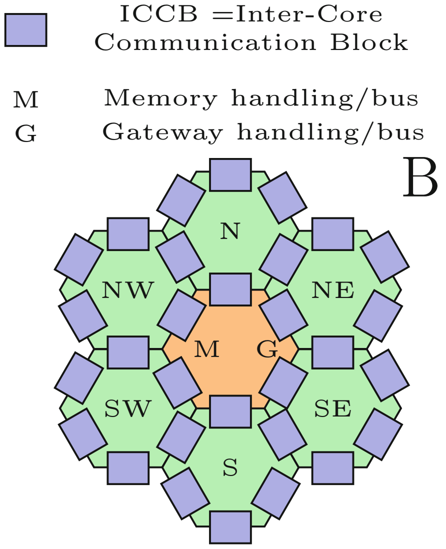

Fig. 5 Implementation of the zeroth-level communication bus in EMPA

but (as they are physically not implemented in the square grid of cores during the manufacturing process) they do not exist physically.

That means: a cluster has one core as "cluster head"; up to six "ordinary members", and up to twelve "corresponding members"; i.e., an "extended cluster" can also be formed, comprising up to 1+6+12 members. Notice that around the edge of the square grid "external members" can be in the position of the "corresponding members", but the upper limit of the total number of members in an extended cluster does not change. Interpreting members of size $r >= 2$ has no practical importance. *The cores with* $r <= 2$ *have a direct communication mechanism* (Fig. 5).

3.6 Communication in EMPA

As discussed in [14], communication strongly degrades computing performance, even in relatively small-size computing systems [15, 21]. Its basic reason is the shared medium (whether it is physically Ethernet-like or serial connection), so EMPA introduces network-like addressing scheme, organizes traffic and introduces hierarchic bus system, to implement principle of locality at HW level.

Addressing and transport systems must provide support for all transport modes. Cluster addressing is of central importance because of the topology of cores: *cores having common boundary surely do not need a bus between the neighboring cores*. In this sense, the native, cross-boundary data transfer represents a zeroth-level communication bus (actually several, parallelly working "buses"), with no contention. This feature, combined with the "small world" nature of most computing tasks (especially the biology mimicking ones) and that nearby cores can share

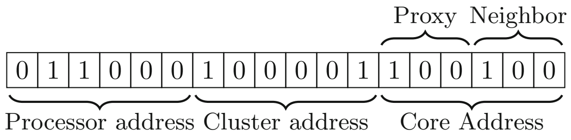

Fig. 6 Implementing the hierarchical cluster-based addressing bit fields of the cores of EMPA processors. A cluster address is globally unique.

memory contents available through the cluster head core) results in a serious performance boost. These architectural changes shall be useful for future neuromorphic architectures/applications, as AIs represent a very specific workload.[3]

Addressing (see also Fig. 6) must support the goal to keep messages at the lowest level of communication buses. Messages from/to outside the cluster are received/sent by cluster head. The rest of the messages are sent directly (through the corresponding Inter-Core Communication Block (ICCB)) or with using a proxy to their final destination. To implement that goal, EMPA processors use the addressing scheme shown in Fig. 6. Notice that the proposed addressing system a network logical address can be directly (and transparently) mapped to the ID and vice versa.

In EMPA, cluster addressing carries also topological information, partly relies on relative topological positions, and enables to introduce different classes of relationship between cells. As mentioned, cluster head cores have a physically distinct role (In this sense, they can also be a “fat” core) and enables us to introduce cluster addressing for members of the extended clusters. Only cluster head cores have an immediate global memory access, see Fig. 5 (considerably reducing the need for wiring). The cores being in neighborhood of size $r = 1$ can access memory through their cluster head. These cores can also be used as a proxy for cores in neighborhood of size $r = 2$. The latter feature also enables to replace a denied cluster head core.

In SPA, the grid and linear addressing are purely logical ones, which use absolute addresses known at compile time. Similarly to computer networks, EMPA cores have (closely related through the cluster architecture) both logical and physical addresses, enabling autonomous (computing-unrelated) communication and virtual addressing.

3.7 The Compiler

Compiler plays a significant role in EMPA. It should discover all possibilities of cooperation, especially the ones that become newly available with the philosophy

[3]https://www.nextplatform.com/2019/10/30/cray-revamps-clusterstor-for-the-exascale-era/: *artificial intelligence, ... it's the most disruptive workload from an I/O pattern perspective.*

that *appearance of a new task is attached with appearance of new computing resource, with a new register file*. Because at the time of compilation actual HW availability cannot be known, code for different scenarios must be prepared and put in the object code.

The philosophy of coding must be drastically changed. Given that, with outsourcing, a new computing facility appears, and processor assures proper synchronization and data transfer, there is no need to store/restore return address and save/restore data in the registers, leading to less memory traffic, and quicker execution time.

Object code is essentially unchanged, except that some fragments (the QTs) are bracketed by meta-instructions. The QTs can be nested (i.e., meta-instructions are inserted into conventional code). One can consider that QTs represent a kind of atomic macros which have some input and output register contents but do not need processing capacity from the actual core.

4 New Features EMPA Offers

Although EMPA does not want to address *all* challenges of computing, it addresses many of them (and leaves the door open for addressing further challenges). Due to lack of space, code examples, comparisons, and evaluations, based on the loosely-timed SystemC simulation [48], are left for simulator documentation and the early published version [49].

4.1 Architectural Aspects

Notice that ad hoc assemblies consider both current state of cores, and also their 'Denied' signal. That is, the flawed (or just temporarily overheated) cores are not used, significantly increasing mean time between machine failures. Also, notice that this approach enables using 'hot swap' cores, in this way providing dynamic, connected systems (the addressing is universal, and the information is delivered by messages; it takes time, but possible), as well as *to deliver the code to the data*: the physical cores can be located in the proximity of the 'big data' storage, instruction is delivered to the place, and only processed, needed result is to be transported back.

Virtualization at HW Level

In EMPA no absolute processor addresses are utilized: virtual processors seen by the programmer are mapped 'on the fly' to physical core by the EMPA processor. Physical cores have a 'denied' state that can be set permanently (like fabrication yield) or temporarily (like overheating), in which case the core will not be used to map a virtual core to it. When combined with a proper self-diagnostic system, this feature prevents extensive systems to fail because of a failing core. Processor has the right and possibility to replace a physical core with another one at any time.

Redundancy

Huge masses (literally millions/billions) of silicon-based elements are deployed in all systems. As a consequence, components showing a tolerable error rate in "normal" systems, but (purely due to the high number of components) need special care in the case of large-scale systems [50].

The usual engineering practice is to rely on the high reliability of components. Fault-tolerant systems require particular technologies, typically majority voting, but they are also based on the same type of single high-reliability components.

Reduced Power Consumption

The operating principle of a processor is based on the assumption that processors are working continuously, executing instructions one after the other, as their control unit defines the required sequencing. Because of this principle, in the OS an 'idle' task is needed. In EMPA, cores can return control voluntarily, enabling most of the cores to stay in a 'ilow power' state.

Also, as discussed in [12], a major contribution of power consumption comes from moving data unnecessarily. Given that EMPA reduces memory usage in many ways (and, that according to [51], about 80% of consumed energy is used for moving data), it shall have a significant effect also of power efficiency of computing.

4.2 Attacking Memory Wall

The 'memory wall' is known as the 'von Neumann' bottleneck of computing, especially after that memory access time became hundreds of times slower than processing time. Although in SPA systems 'register only' processing and cache memories can seriously mitigate its effect, in the case of large systems the 'sparse' calculations that poorly use the cache, show up orders of magnitude worse computing efficacy, i.e., further improvement in using the memory is of utmost importance.

Register-to-Register Transfer

The idea of immediate register-to-register transfer [34] seriously can increase performance of real-life tasks [35]. In EMPA, the idea is used in combination with the flexibility of using virtual cores, multiple register arrays via children.

Subroutine Call Without Stack

In SPA, a subroutine call requires to save/restore return address and (at least part of) register file; unfortunately, one can use only main memory for that temporary storage. In EMPA, for executing subroutine code, another PU is provided. Because of this solution, HW can remember (in a nested way) the return address. Furthermore, working area is provided by the register file of the 'hired' core. Given that a register-to-register transfer is provided, code execution can be hundreds of times quicker. With proper organization, hiring and hired cores can also run partly parallel.

Interrupt and Systems Calls Without Context Switching

Given that interrupts and OS service calls can be considered as special service calls, where also context switching is needed, using a prepared (waiting in kernel mode) core can service a request thousands of times quicker. Event, interrupts can be serviced *without interrupting* the running process.

Resource Sharing Without Scheduling

For multitasking, only the OS can provide exclusive access to some resource (as in SPA, no other processor/task exists). EMPA offers a simple, elegant, and quick solution: it can delegate a QT for the task of guarding a critical section, and all tasks issue a conditional subroutine call to the code guarded by that QT. All but the first requester QT must wait (but are scheduled automatically by the processor), and after servicing all requests, the delegated core is put back to the pool. Since compiler creates reasonably sized code fragments, cases leading to priority inversion [30] cannot happen, so no specialized protocols are needed in the OS: the orchestrated work in EMPA prevents those issues.

4.3 Attacking the Communication Wall

In SPA, communication is not natively present (no other processor exists); it must be performed and synchronized using I/O instructions and OS operations, in payload processing time; resulting in performing a severe amount of non-payload instructions.

Decreasing the Internal Latency

When using interconnected cores, ECE can take over most of the non-payload duties, enabling to decrease the sequential-only portions of the task that decisively define communication/computation ratio [36]; a significant point when developing large scale computing systems [11] or using AI-type workloads [15]. As discussed in [11], the housekeeping (the FP_0) contribution is a considerable limitation when running High Performance Linpack (HPL) benchmark. Borrowing nearby cores and using their physical proximity enables us to achieve higher HPL maximum performance values.

Hierarchic (Local) Communication

Using temporally or spatially local memory accesses can increase efficiency dozens of times. Similarly, providing 'interconnection cache' for EMPA processor can result in considerable improvement in final efficiency of the system. As computing tasks change their state between 'computing bound' and 'communication bound' dynamically, this solution mitigates both limiting factors as much as possible.

Fully Asynchronous Operation

As von Neumann only required a 'proper sequencing' of instructions, and having less 'idle' times during core operation appears as performance increase, asynchronous operation (i.e., turning all components to active) can considerably contribute to more effective (i.e., comprising fewer losses) operation.

5 Summary

In computing, incremental development methods face more and more difficulties, because of the drastic changes both in technology and utilization. The final reason, as has been suspected by many researchers, is the computing paradigm reflecting a 70-year old state of the art. Computing needs renewal [49] and rebooting. Firstly, the ever smaller components driven by ever quicker clock signal, because of scientific reasons, show a temporal behavior [12], and suppressing their natural behavior causes severe computing performance loss (and enormously increased power consumption). Secondly, many technical implementations and architectural solutions, inherited from the past decades, become outdated. It was presented that *it is not a necessary condition that the same computer* solves all the tasks: von Neumann only required a "proper sequencing" in executing machine instructions. This requirement can be satisfied in a much better way via using the presently available many "free" processors. That way requires an entirely different thinking (and component base) and offers real advantages. We can implement the introduced new paradigm by putting the presently available technology solutions along with different principles that approach offers considerable advantages.

References

1. I. Markov, Limits on fundamental limits to computation. Nature **512**(7513), 147–154 (2014)
2. G.M. Amdahl, Validity of the single processor approach to achieving large-scale computing capabilities, in *AFIPS Conference Proceedings*, vol. 30, pp. 483–485 (1967)
3. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009)
4. J.A. Chandy, J. Singaraju, Hardware parallelism vs. software parallelism, in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism, ser.* HotPar '09 (USENIX Association, Berkeley, CA, USA, 2009), pp. 2-2
5. S.H. Fuller, L.I. Millett, Computing performance: Game over or next level? Computer **44**, 31–38 (2011)
6. US National Research Council, The Future of Computing Performance: Game Over or Next Level? (2011). [Online]. Available: http://science.energy.gov//media/ascr/ascac/pdf/meetings/mar11/Yelick.pdf
7. S(o)OS Project, Resource-independent execution support on exa-scale systems (2010). http://www.soos-project.eu/index.php/related-initiatives
8. Machine Intelligence Research Institute, Erik DeBenedictis on supercomputing (2014). [Online]. Available: https://intelligence.org/2014/04/03/erik-debenedictis/
9. J. Sawada et al., TrueNorth ecosystem for brain-inspired computing: Scalable systems, software, and applications, in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 130–141 (2016)
10. J. Végh, A. Tisan, The need for modern computing paradigm: Science applied to computing, in *Computational Science and Computational Intelligence CSCI The 25th Int'l Conf on Parallel and Distributed Processing Techniques and Applications* (IEEE, 2019), pp. 1523–1532. [Online]. Available: http://arxiv.org/abs/1908.02651
11. J. Végh, Finally, how many efficiencies the supercomputers have? J. Supercomput. **76**(12), 9430–9455 (2020). [Online]. Available: http://link.springer.com/article/10.1007/s11227-020-03210-4
12. J. Végh, Introducing temporal behavior to computing science, in *2020 CSCE, Fundamentals of Computing Science* (IEEE, 2020). Accepted FCS2930, in print. [Online]. Available: https://arxiv.org/abs/2006.01128
13. J. Végh, A.J. Berki, Do we know the operating principles of our computers better than those of our brain? (2020). [Online]. Available: https://arxiv.org/abs/2005.05061
14. J. Végh, Which scaling rule applies to Artificial Neural Networks, in *Computational Intelligence (CSCE) The 22nd Int'l Conf on Artificial Intelligence (ICAI'20)* (IEEE, 2020). Accepted ICA2246, in print; in review in Neurocomputing. [Online]. Available: http://arxiv.org/abs/2005.08942
15. J. Végh, How deep machine learning can be, ser. *A Closer Look at Convolutional Neural Networks* (Nova, In press, 2020), pp. 141–169. [Online]. Available: https://arxiv.org/abs/2005.00872
16. J. Végh, How Amdahl's Law limits performance of large artificial neural networks. Brain Informatics **6**, 1–11 (2019). [Online]. Available: https://braininformatics.springeropen.com/articles/10.1186/s40708-019-0097-2/metrics

17. J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of Things (IoT): A vision, architectural elements, and future directions. Future Gener. Comput. Syst. **29**, 1645–1660 (2013)
18. R.F. Service, Design for U.S. exascale computer takes shape. Science **359**, 617–618 (2018)
19. J. Du, L. Zhao, J. Feng, X. Chu, Computation offloading and resource allocation in mixed fog/cloud computing systems with min-max fairness guarantee. IEEE Trans. Commun. **66**, 1594–1608 (2018)
20. www.top500.org, Intel dumps knights hill, future of xeon phi product line uncertain (2017). https://www.top500.org/news/intel-dumps-knights-hillfuture-of-xeon-phi-product-line-uncertain///
21. J. Keuper, F.-J. Preundt, Distributed training of deep neural networks: theoretical and practical limits of parallel scalability, in *2nd Workshop on Machine Learning in HPC Environments (MLHPC)* (IEEE, 2016), pp. 1469–1476. [Online]. Available: https://www.researchgate.net/publication/308457837
22. ARM, big.LITTLE technology (2011). [Online]. Available: https://developer.arm.com/technologies/big-little
23. J. Congy, et al., Accelerating sequential applications on CMPs using core spilling. Parallel Distribut. Syst. **18**, 1094–1107 (2007)
24. Cypress, CY7C026A: 16K x 16 Dual-Port Static RAM (2015). http://www.cypress.com/documentation/datasheets/cy7c026a-16k-x-16-dual-port-static-ram
25. R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, P. Marwedel, Scratchpad memory: Design alternative for cache on-chip memory in embedded systems, in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, ser. CODES '02 (ACM, New York, NY, USA, 2002), pp. 73–78. [Online]. Available: http://doi.acm.org/10.1145/774789.774805
26. J. Backus, Can programming languages Be liberated from the von Neumann style? A functional style and its algebra of programs. Commun. ACM **21**, 613–641 (1978)
27. P. Gohil, J. Horn, J. He, A. Papageorgiou, C. Poole, IBM CICS Asynchronous API: Concurrent Processing Made Simple (2017). http://www.redbooks.ibm.com/redbooks/pdfs/sg248411.pdf
28. R.H. Arpaci-Dusseau, A.C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 0th edn. (Arpaci-Dusseau Books, 2015)
29. J. Végh, A new kind of parallelism and its programming in the explicitly many-processor approach. ArXiv e-prints (Aug. 2016). [Online]. Available: http://adsabs.harvard.edu/abs/2016arXiv160807155V
30. O. Babaoglu, K. Marzullo, F.B. Schneider, A formalization of priority inversion. Real Time Syst. **5**(4), 285–303 (1993). [Online]. Available: https://doi.org/10.1007/BF01088832
31. D.W. Wall, Limits of instruction-level parallelism, New York, NY, USA, pp. 176–188 (Apr. 1991). [Online]. Available: http://doi.acm.org/10.1145/106974.106991
32. S. Kumar, et al., Acceleration of an asynchronous message driven programming paradigm on ibm blue gene/q, in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing* (IEEE, Boston, 2013). [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6569854
33. N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, P. Dubey, Can traditional programming bridge the ninja performance gap for parallel computing applications? Commun. ACM **58**(5), 77–86 (2015). [Online]. Available: http://doi.acm.org/10.1145/2742910
34. F. Zheng, H.-L. Li, H. Lv, F. Guo, X.-H. Xu, X.-H. Xie, Cooperative computing techniques for a deeply fused and heterogeneous many-core processor architecture. J. Comput. Sci. Technol. **30**(1), 145–162 (2015)
35. Y. Ao, C. Yang, F. Liu, W. Yin, L. Jiang, Q. Sun, Performance optimization of the HPCG benchmark on the sunway TaihuLight dupercomputer. ACM Trans. Archit. Code Optim. **15**(1), 11:1–11:20 (2018)
36. J.P. Singh, J.L. Hennessy, A. Gupta, Scaling parallel programs for multiprocessors: Methodology and examples. Computer **26**(7), 42–50 (1993)

37. B. Bohnenstiehl, A. Stillmaker, J.J. Pimentel, T. Andreas, B. Liu, A.T. Tran, E. Adeagbo, B.M. Baas, KiloCore: A 32-nm 1000-processor computational array. IEEE J. Solid State Circuits **52**(4), 891–902 (2017)
38. PEZY, 2048 core chip (2017). https://www.top500.org/green500/lists/2017/11/
39. S.B. Furber, D.R. Lester, L.A. Plana, J.D. Garside, E. Painkras, S. Temple, A.D. Brown, Overview of the SpiNNaker system architecture. IEEE Trans. Comput. **62**(12), 2454–2467 (2013)
40. M.D. Hill, M.R. Marty, Amdahl's law in the multicore era. IEEE Computer **41**(7), 33–38 (2008)
41. R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B.C. Lee, S. Richardson, C. Kozyrakis, M. Horowitz, Understanding sources of inefficiency in general-purpose chips, in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10 (ACM, New York, NY, USA, 2010), pp. 37–47. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815968
42. J. Végh, J. Vásárhelyi, D. Drótos, The performance wall of large parallel computing systems, in *Lecture Notes in Networks and Systems*, vol. 68 (Springer, 2019), pp. 224–237. [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-3-030-12450-221
43. K.E. Fleming Jr., K.D. Glossop, S.C. Steely Jr., J. Tang, A.G. Gara, Processors, methods, and systems with a configurable spatial accelerator, no. 20180189231 (July 2018). [Online]. Available: http://www.freepatentsonline.com/y2018/0189231.html
44. Intel, Processors, methods and systems with a configurable spatial accelerator (2018). http://www.freepatentsonline.com/y2018/0189231.html
45. U. Vishkin, Explicit multi-threading (XMT): A PRAM-on-chip vision – A desktop supercomputer (2007). Last accessed Dec. 12, 2015 [Online]. http://www.umiacs.umd.edu/users/vishkin/XMT/index.shtml
46. U.Y. Vishkin, Spawn-join instruction set architecture for providing explicit multithreading (1998). https://patents.google.com/patent/US6463527B1/en
47. V. Cimagalli, M. Balsi, Cellular neural networks: A review, in *Proc. 6th Italian Workshop on Parallel Architectures and Neural Networks, Vietri sul Mare, Italy* (World Scientific, 1993), pp. 12–14. iSBN: 9789814534604
48. J. Végh, EMPAthY86: A cycle accurate simulator for explicitly many-processor approach (EMPA) computer (Jul 2016). [Online]. Available: https://github.com/jvegh/EMPAthY86
49. J. Végh, *Renewing Computing Paradigms for More Efficient Parallelization of Single-Threads*, ser. Advances in Parallel Computing, vol. 29, ch. 13 (IOS Press, 2018), pp. 305–330. [Online]. Available: https://arxiv.org/abs/1803.04784
50. C. Wrired, Cosmic Ray Showers Crash Supercomputers. Here's What to Do About It (2018). https://www.wired.com/story/cosmic-ray-showers-crashsupercomputers-heres-what-to-do-about-it/
51. H. Simon, Why we need Exascale and why we won't get there by 2020, in *Exascale Radioastronomy Meeting*, ser. AASCTS2, 2014. [Online]. Available: https://www.researchgate.net/publication/261879110 Why we need Exascale and why we won't get there by 2020