Survey of the sequential and parallel models of computation : Technical report LUSY-2012/02

Matevž Jekovec, Andrej Brodnik
University of Ljubljana, Faculty of Computer and Information Science,
Tržaška 25, SI-1000 Ljubljana, Slovenia
{matevz.jekovec,andrej.brodnik}@fri.uni-lj.si

Abstract

In this survey we review the most significant sequential and their counterpart parallel models of computation. The sequential models are Comparison model, Sequential register machine (Counter, Pointer, RAM and RASP machines), Cell-Probe models (Trans-Dichotomous and Word RAM), Hierarchical Memory (Hierarchical Memory Model, HMM with Block transfer, External Memory and Universal Memory Hiererchy model) and a Cache-Oblivious model. Reviewed parallel models of computation models are the PRAM-based (MPC, LPRAM, BPRAM, Asynchronous versions of PRAM — APRAM, Asynchronous PRAM and XPRAM; and Queud Read Queued Write PRAM), Bulk Synchronous Process, LogP and Multi-level BSP models, Parallel Memory Hierarchy models (Parallel External Memory and Multicore Cache model) and the Parallel Cache-Oblivious model along with the space-bounded scheduler. Finally we describe a Vector RAM model and the K-model which captures specifics of the vector stream multiprocessors and the memory organisation found in the modern GPUs.

Keywords: theoretical computing, models of computation, parallel computation, high performance computing

1 Introduction

Modelling is used to capture important characteristics of complex phenomena. When defining a new model we always choose between the clarity and simplicity on one hand and the degree of accuracy and realism on the other.

There are three types of models used in computer science and engineering [39]: a model of computation, a programming model and a hardware model. The *model of computation* defines an

- execution engine powerful enough to produce a solution to the relevant class of problems and
- captures computing characteristics of practical computing platforms.

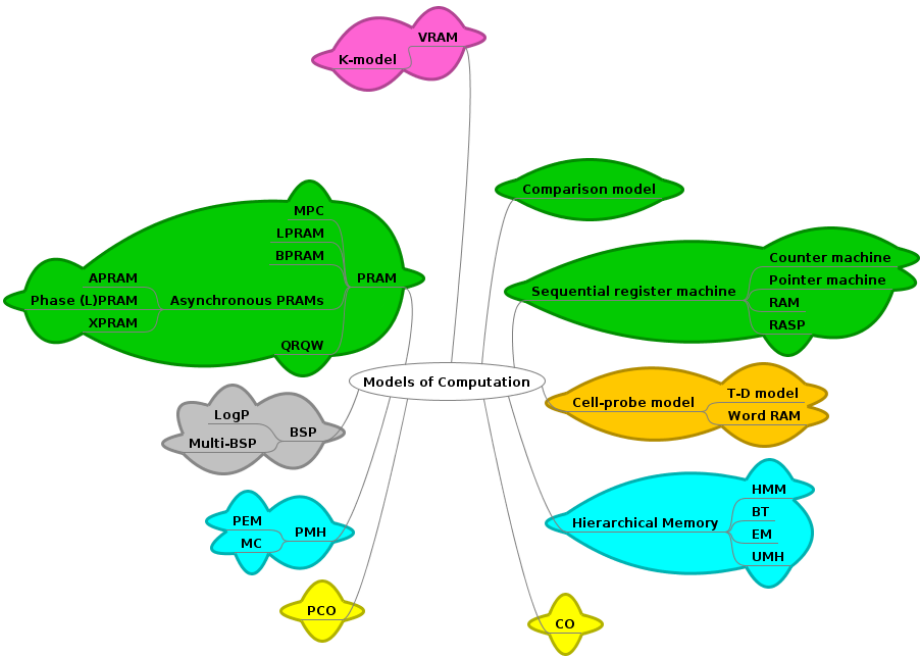

Figure 1: The map of models of computations covered by this paper. Sequential models are on the right and parallel models on the left.

The model of computation can alternatively be called an *abstract machine model*, a *cost model* or a *performance model*.

The *programming model* provides the semantic of programming abstraction which serves as a basis for the programming language. A primary goal is to allow reasoning about program meaning and correctness.

Architectural or hardware model is used for specific language implementation and most notably for the high performance machine design purpose.

The goal of this survey is to cover the most significant models of computation to date. Two effects influenced on the development of the models of computation through time. Computer engineers designed new architectures to solve industry problems of that time which also needed a new or adapted model of computation. Or a new model of computation was designed to effectively solve a problem from the theoretical point of view. Engineers designed a real architecture afterwards, if the solution turned out to be feasible. Nevertheless the aim of both approaches was to optimize or design new algorithms, data structures and underlying architectures to find solutions to yet unsolved problems or to optimize existing solutions.

Figure 1 illustrates the models of computation covered by this survey. On the right hand side are the sequential and on the left hand the parallel models of computation. Both sequential and their parallel counterparts taking similar

characteristics into account are painted the same color. The most abstract ones are located on top and the most realistic ones on the bottom. Models are dated similarly — the older ones are on top and the newer ones on the bottom. Basically there are three generations of models of computation: the first generation developed since 1978 consists of the *shared memory models*, the second generation developed since 1984 consists of the *distributed memory models* and finally, the third generation consists of the *hierarchical memory parallel computational model*.

This paper's structure resembles the figure 1. We first describe the sequential models of computation from the most abstract to the most realistic ones. We begin with the comparison model which does not take into account neither the CPU instructions nor the memory bottlenecks. The second family — the sequential register machine — is a Turing-equivalent family of models and assume a predefined set of instructions but doesn't know anything about how the data is stored. The cell-probe family of models assume data to be stored in a memory of addressable words. This allows calculation of memory addresses according to the contents. Slow memory access times were always the bottleneck for computer engineers. The next family models the multi-level memory hierarchy present in any modern computer architecture. The final model is the cache-oblivious model which builds on assumption that, if there is an optimal cache complexity behaviour between two levels of memory hierarchy, there is an optimal cache complexity throughout all the hierarchy. The model assures optimal usage of memory cache independent of the parameters like the cache size, block size or the hierarchy depth.

The second part of the document describes the parallel models. We begin with the Parallel RAM and describe its variations like splitting memory into modules, introducing local private memory per processing element, using block transfers, asynchronous communication for larger systems and queued reading and writing. We then introduce a family of models based on the communication complexity and message passing — the Bulk Synchronous Process. In comparison to PRAM, these models also consider the latency, network bandwidth and synchronisation period between processing units. Afterwards follows the family of models taking the memory hierarchy when multiple processing elements into account, each with its own private memory on the first level. Finally the most realistic model to date — the parallel cache-oblivious model — is introduced. This model also requires the introduction of the space-bounded schedulers because the general recursive assumption of the optimal cache usage throughout the memory hierarchy does not hold anymore as was the case for the sequential cache-oblivious model.

Finally we describe the Vector RAM model which evaluates the behaviour of both SISD or SIMD vector computers and the K-model for designing and evaluating algorithms running on modern Graphic Processing Units (GPUs). Vector computers execute instructions which operate on a vector of memory words in comparison to only operating with scalar values. *Stream multiprocessors* on modern GPUs are a kind of vector computers. Furthermore, a two-level memory hierarchy and a separate global memory is also an integral part of

these architectures. K-model captures these specifics. GPUs with their support for executing arbitrary programs (also known as the General Purpose GPU – GPGPU) are important in high performance computing because they offer much higher memory throughput than traditional multi-core systems.

2 Sequential models of computation

Sequential models of computation assume a single processing element (PE) executing the algorithm sequentially in steps. Depending on the model, data can be stored in the unbounded number of registers (also called the *memory*) or implicitly inside graph nodes. We denote the problem size as n.

2.1 Comparison model

The *comparison model* is an abstract model of computation which doesn't define the CPU instruction set neither does it assume how the data are stored and their access time.

The comparison model originates from the comparison-based sorting algorithms where they measured how many times the numbers were "compared" to each other. The binary search is a technique for finding the correct item inside an ordered array questioning "Is the required item larger or smaller than the element I'm currently pointing at?". The following step — searching the right or the left hand side of the array from the pointing element — depends on the answer to that question. This technique needs $\Omega(\lg n)$ steps to find the desired element among n elements, if all the elements are unique. The binary search defines the lower bound for all comparison-based sorting algorithms: In order to find the correct permutation vector of numbers among $n!$ permutations we need $\Omega(\lg n!) = \Omega(n \lg n)$ comparisons using a binary search.

The comparison model measures the time complexity of the algorithm. The overall running time is the number of executed steps or lines of code. Each step i is executed in a constant time t_i. To measure the total running time of the algorithm we need to *unfold* the loops and count the number of steps in total $t = \sum_i t_i$. The only parameter is the problem size n and the resulting time complexity is a function of n.

2.2 Sequential register machine

Sequential register machine is formally defined as a machine having a finite number of uniquely addressable registers $r_1, r_2 ...$ and a single processing unit. Each register can hold a single integer of a finite size resembling of register sizes of physical computers. We should stress out that having register sizes small is an important assumption. If we assume that register sizes are arbitrarily big, the entire problem could be encoded into a single register which results in a computer able to solve PSPACE-complete problems in polynomial time [12].

From the computing power point of view, the pointer machine, RAM and RASP are equivalent to the Turing machine. However, at least two counter machines are equivalent to the Turing machine. The sequential register machine, in comparison to other Turing equivalents like the lambda calculus or a universal Turing machine, better resembles the nowadays computer architectures.

2.2.1 Counter Machine

A general counter machine consists of a set of registers or accumulators and of the increase and conditional jump instructions. Depending on the counter machine implementation [41, 44, 36] it assumes a subset of the following instruction set:

CLR(r) Clear register r. (Set r to zero.)

INC(r) Increment the contents of register r.

DEC(r) Decrement the contents of register r.

CPY(r_j, r_k) Copy the contents of register r_j to register r_k leaving the contents of r_j intact.

JZ(r,z) If register r contains Zero THEN Jump to instruction z else continue in sequence.

JE(r_j, r_k, z) If the contents of register r_j Equals the contents of register r_k then Jump to instruction z else continue in sequence.

HALT Halts the machine. Computation is finished.

At least two counter machines are needed to simulate the Turing machine. Counter machine does not support executing a stored program from the memory, but behaves as a finite state machine.

Time complexity is measured by counting the number of instructions executed during the program execution. Execution of any instruction lasts for the same period of time. The number of used registers determines the space complexity which is always constant regardless of the program input.

2.2.2 Pointer Machine

We will present the pointer machine model [11] based on the Schönhage's Storage Modification Machine directed-graph representation [43]. Every node represents a stored word (register) and every edge a pointer. The pointer machine offers three operations:

NEW(w) creates a new node representing word w by following necessary edges.

SET(w,v) moves the last edge of the word w to the last edge of v.

IF $v = w$ **THEN** z if word w equals v then jump to word node z. Else continue.

The pointer machine is mostly interesting from the theoretical point of view because it cannot do any arithmetic but only reading, modifying and doing various tests on its storage.

Time complexity is measured by counting the number of hops during the program execution. Uniform access to any node is assumed. Space complexity is measured by counting the maximum number of nodes in the graph. All the nodes are of equal size. The pointer machine does not support indirect addressing — making decisions based on the followed path.

2.2.3 Random Access Machine — RAM

The Random Access Machine model or RAM [22] is a multiple-register counting machine with added support for indirect addressing. This is achieved by adding so called indirect instructions specifying the register's address not only as a direct address in the instruction itself but indirectly as the address stored in another register. Consequently a fixed program is able to access an unbounded number of registers depending on the program input. A scheme of the RAM can be seen in figure 2.

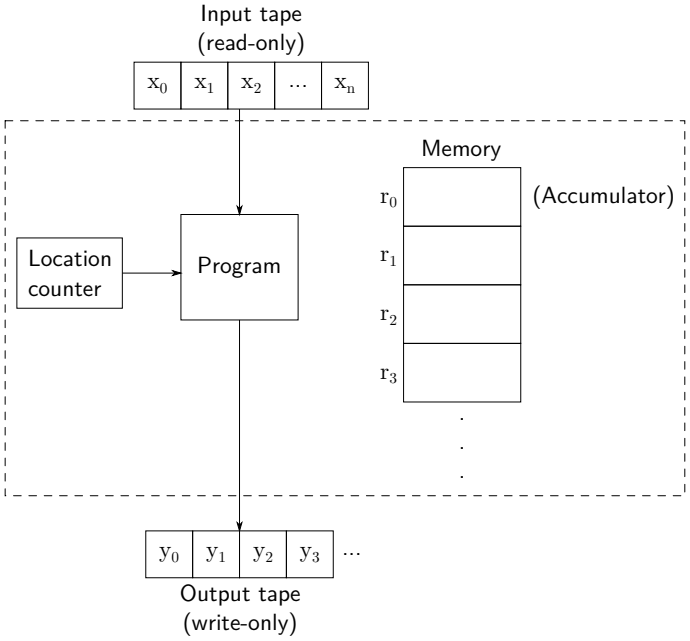

Figure 2: A random access machine.

RAM supports the same instructions as the counter machine, but adds the memory access ones:

Memory access: LOAD, STORE

Arithmetic operations: ADD, SUB, MULT, DIV

I/O operations: READ, WRITE

Flow control: JUMP, JGTZ, JZERO, HALT

The program is stored inside a machine in a finite-state-automate manner and cannot be changed during the execution.

The time and space complexity are measured using the *uniform cost criterion* or the *logarithmic* one [6]. The uniform cost assumes registers large enough to fit any integer result in the program. This results in constant time and space complexity for a single instruction.

The logarithmic cost assumes registers to be $\lg i$ big, where i is the operand. Logarithmic time complexity is based on a crude assumption that the cost performing an instruction is proportional to the length of the operands of the instructions. Time needed to execute a single instruction is therefore $\lg i$ where i is the operand. Similarly the space needed to store the result i of a single instruction is $\lg i$.

In comparison to counter machines, the uniform time complexity in RAM is the same as the time complexity of the algorithm in counter machines. We simply count the number of executed instructions. The space complexity on the other hand was constant in counter machines because the set of registers was predefined. In RAM the space complexity depends on the program input. Logarithmic cost criterion was not used in counter machines.

2.2.4 Random Access Stored Program machine — RASP

The Random Access Stored Program machine or RASP [32] is similar to the RAM model but has its program in its registers along with its input. This results in ability for the program to change itself during its execution. Instructions in RASP are the same as in RAM except for indirect addressing ones which are not needed any more.

From the complexity analysis point of view ability for the program to change its execution code allows transformation of indirect register addresses to direct ones thus increasing the instruction execution speed. It was shown that RAM and RASP models are equivalent within a constant factor.

On a side note, RASP might resemble more to the von Neumann architecture because of the exclusive instruction/data memory accesses. RAM on the other hand resembles to the Harvard architecture accessing both instructions and data memory simultaneously.

2.3 Cell-probe model

The cell-probe model of computation was introduced by Yao [51]. The author investigated the lower time bounds of `contains()` operation, i.e. if an element with the given key is in a table or not, yet using a minimal storage space. We know that the binary search operation on a sorted array has a lower bound of

$\lceil \lg n \rceil$ comparisons in worst case when using a comparison-based model. Yao managed to overcome this lower bound by exposing the full power of address-computation on RAM using different encodings.

Cell-probe model besides the problem size n also defines a word (a cell) of length w-bits and a universal set of possible element keys $U = 0, 1..., 2^{w-1}$. By carefully designing an off-line algorithm for storing the elements to a table based on their content and not solely on the relation to other elements, Yao achieved much less probes than the original binary search on a sorted table. For extreme cases where $|U| \approx n$ and $|U| \geq 2^{16n^2}$ he achieved constant two probes in worst case. More generally his results show that using binary search method for large $|U|$ is optimal in comparison-model only, but far from optimal on RAM.

Cell-probe findings are closely related to the integer sorting algorithms like the counting or radix sort. Similarly, these algorithms resemble to the work done in post offices where they sort n envelopes by not comparing them to each other, but putting them directly into U buckets, one for each district, all in linear time.

Time complexity analysis of algorithms in cell-probe model measures the number of probes to the memory words (reads or writes). It is a function of n and $|U|$. Space complexity is measured the same way as in RAM. Because the model assumes arbitrary word size, it is mostly used for determining the theoretical lower bounds of the problem.

2.3.1 Transdichotomous model

Transdichotomous model [27] bridges the real computer world comprising of registers of the given size with the problem domain of size n. It assumes that in order to address the input data, we need at least $\lg n$ bits. On the other hand, if we want to do address-computation as in the cell-probe model, we want to store data addresses into a single word. Therefore, the word size w should be:

$$w \geq \lg n$$

which is a realistic assumption for majority of the real world problems.

Authors presented a data structure for predecessor queries called *fusion tree*. By compressing the actual element keys, they managed to reduce the required key size to fit into a single word of a feasible size. Taking the word size into account results in a much more realistic model of computation, however operations in comparison to RAM are not limited to a single word but can operate on multiple words simultaneously.

Time and space complexity measures are similar as in the cell-probe model. Besides n and $|U|$, the word size w is introduced.

2.3.2 Word RAM

Transdichotomous model allows you to operate on a finite number of words in constant time. Word RAM [31] on the other hand supports the C-like operations

on a single word in constant time and is much closer to the real computer architectures:

Integer arithmetic: $+, -, *, /, \%, <, >$;

Bitwise operations: AND, OR, XOR, NEG;

Bit shifting: $<<, >>$;

Dereferencing (indirect addressing): [ ].

Word RAM has become a standard RAM model when designing and assessing algorithms when not taking memory hierarchy and concurrency into account.

2.4 Hierarchical Memory

While the Sequential register machine variants consider execution time of instructions in CPU as the basis for analysing and designing algorithms, on real computers the memory access time represents the bottleneck. This was a consequence of the rapid development of CPU and memory chips on one hand and slower increases in bus throughputs. Memory hierarchy is a well known technique in computer engineering to overcome this issue by introducing additional memory layers between the CPU registers and the main memory. Access times to these layers are smaller than of the main memory, but their size is also smaller.

Modelling the multi-level memory hierarchy is essential for describing modern computer characteristics. Such models instead of having a uniform memory access times for any memory location, they catch characteristics of the *temporal locality* of memory access times using the least recently used (LRU) or FIFO strategy. Since it has been shown in [45] that LRU and FIFO exchange strategies are at most for a constant factor slower than the optimal replacement strategy for twice the cache size, these are viable choices.

2.4.1 Hierarchical Memory Model — HMM

HMM [1] describes a set of registers $R_1, R_2..., R_i, ...$ with nonuniform access time $\lceil \log i \rceil$ (see). The memory hierarchy layers can be viewed as increasing exponentially in size:

- 1 location taking 0 time units to access (CPU registers),
- 1 location taking 1 time units to access (level 0),
- 2 locations taking 2 time units to access (level 1),
- 4 locations taking 3 time units to access (level 2),
- ...

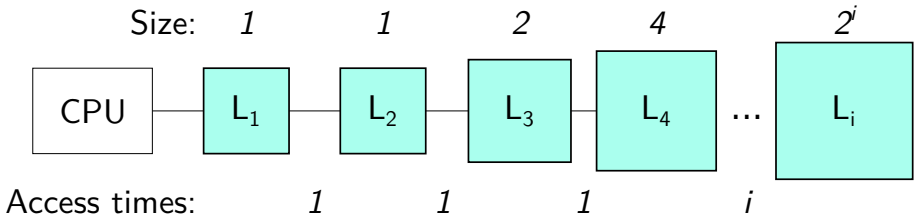

Figure 3: Hierarchical Memory Model. Memory size is growing exponentially, access times between each level is constant.

- 2^i locations taking $i + 1$ time units to access (level i).

From the design and analysis point of view, the model supports the same instructions as RAM but counts RAM accesses in the nonuniform way. Authors analysed existing algorithms for semiring matrix multiplication, FFT, sorting, binary search and achieved slowdowns from $\Theta(1)$ to $\Theta(\log n)$ in worst case.

2.4.2 HMM with Block Transfer — BT

Hierarchical Memory model with Block Transfer [2] outlines the memory the same way as HMM with the following additions:

- Cost for memory access x is arbitrary $f(x)$.
- Support for copying interval of memory words. The cost for copying $x - \delta...x$ to $y - \delta...y$ is $f(\max(x, y)) + \delta$.

Authors analysed the same algorithms as in HMM and obtained slowdowns of $O(f(x))$ in general. Having algorithm complexity dependent on $f(x)$ is the main concern of using BT for algorithm analysis and design in practice.

2.4.3 External Memory model — EM

External Memory model [5] (also called I/O model or Disk Access Model) combines two older memory models: Floyd's idealized two-level storage and the Red-Blue Pebble Game model. The Floyd's model [25] assumed infinitely big memory being split to blocks of size B (see figure 4). He showed that the lower bound to access any word in the memory of size n words is $\Omega(\frac{n}{B} \log B)$ assuming the *tall disk form* (the number of blocks is larger than the block size: $n/B > B$).

The Red-Blue Pebble Game model [34, 35] takes algorithm operations being presented as nodes on a Directed Acyclic Graph and edges representing operation dependencies (see figure 5). They define two levels of memories: the infinitely large one but slow (blue pebbles) and an infinitely fast but small of size M (red pebbles). The pebble game starts with blue pebbles on the input nodes. The goal is to proceed towards the output nodes by using at most M red pebbles at any time and the following moves:

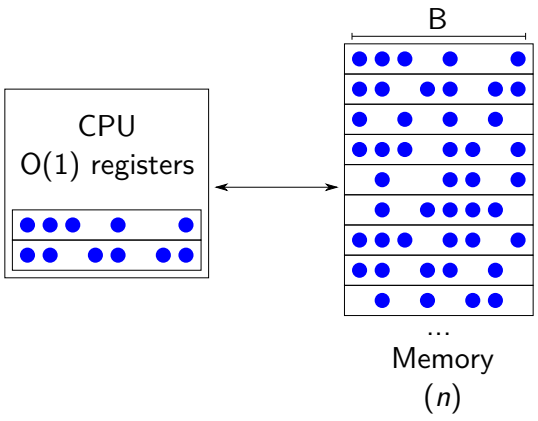

Figure 4: Floyd's idealized two-level storage. With a fixed number of registers in CPU and the main memory of size n being split to blocks of size B.

- Place a red pebble on a node, if all predecessors have red pebble.
- Remove pebble from node (forget information).
- Writing: Color red pebble to blue (memory $\rightarrow$ disk).
- Reading: Color blue pebble to red (disk $\rightarrow$ memory).

The game ends when all the outputs are covered by blue pebbles. Multiple paths to the solution exist in the Red-Blue pebble game. We are interested in the one minimizing the colorings from one color to another one (communication between the two levels of memory — I/O complexity). Authors examined DAGs of existing algorithms (FFT, matrix multiplication, scanning) and obtained speedups from $\Theta(\log M)$ to $\Theta(M)$. This was possible because only memory transfers are taken into account (cache accesses are infinitely fast according to the model).

Finally, External Memory model defines two levels of memory the same way as the red-blue pebble game model: The second level which is slow but infinitely big, and a first level which is infinitely fast but limited in size of M. Both levels of memory are split into blocks of size B. Memory transfers are now done in blocks.

EM model counts the number of blocks transferred between the two levels of memories. The lower bound for the memory transfers is limited by $\Omega(\frac{\#cellprobes}{B})$ and the upper bound to the RAM running times.

In comparison to Hierarchical models described in previous sections, EM model focuses on the slowest memory levels in the memory hierarchy only — the communication between the last and the pre-last one. In practice, these are usually the disk and the main memory. This made the model among the

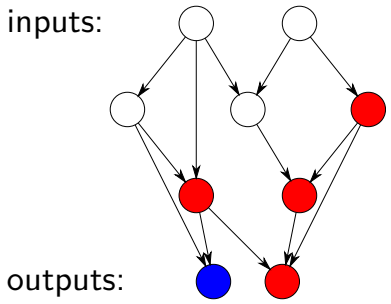

Figure 5: The Red-Blue pebble game on a DAG nearing the end. $M = 4$

cache-oblivious model being the most favourable when designing algorithms and data structures and keeping memory hierarchy in mind.

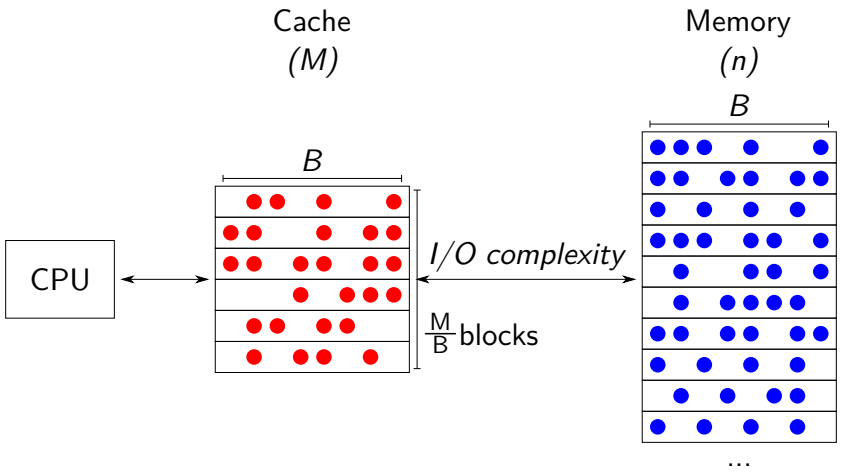

Figure 6: External Memory model with two levels of memory: the second level slow memory on the right of size n and the infinitely fast first level memory on the left of size M. Memory transfers are done in blocks of size B.

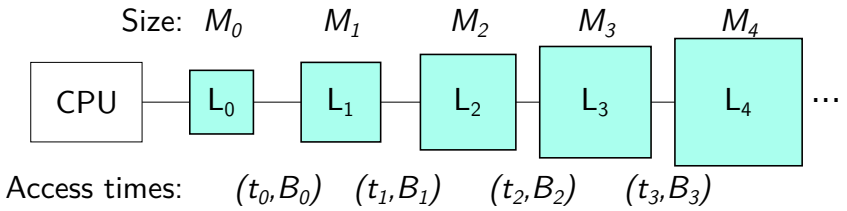

Figure 7: The Memory Hierarchy models cache sizes, block sizes and access times of all the memory levels in the hierarchy.

2.4.4 Uniform Memory Hierarchy model — UMH

The Memory Hierarchy model (MH) is a multilevel version of the EM model and a most realistic model for describing the memory hierarchy. For each memory level L_i, it defines the cache size (M_i), block size (B_i) and the time needed to transfer a block from the higher level memory (t_i), see figure 7. Authors also assume simultaneous memory transfers between different memory levels which is feasible for real architectures.

The Uniform Memory Hierarchy Model [7] abstracts all the l levels of the MH model to only two parameters:

- aspect ratio $\alpha = \frac{M/B}{B}$,
- block growth $\beta = \frac{B_{i+1}}{B_i}$.

The following abstractions are done:

$$B_i = \beta^i$$

$$\frac{M_i}{B_i} = \alpha\beta^i$$

$$t_i = \beta^i f(i)$$

where $f(x)$ denotes the cost function accessing a memory cell x.

In comparison to HMM with block transfer, the UMH model captures the block size the same way as it is done on real computers and not using arbitrary block intervals. The drawback of the UMH model is the remaining cost function $f(x)$ which, the same as in HMM, makes the algorithm design and analysis both difficult and unrealistic in practice.

2.5 Cache-Oblivious model — CO

Cache oblivious model [28] assumes the external memory model but requires the algorithm to run optimally regardless of the block size B and the cache size M. Authors proved that optimal algorithms in CO model also run optimally for memory hierarchies of arbitrary number of levels. They argued that hierarchy

can always be split into two parts — the slower and the faster one. If memory transfers between the two parts are optimal, then, in general, memory transfers are also optimal for any other splitting.

CO model's obliviousness to B and M is particularly useful in real world cases because these parameters vary even during a single program execution (different disk track lengths, cache sharing with another process).

CO model counts the I/O cache transfers Q (the *cache complexity*) the same way as the EM model does. Authors also show that the algorithm taking Q transfers using the ideal cache replacement strategy can efficiently be implemented using LRU or FIFO strategies taking Q^* transfers. Because obliviousness to B, the design of data structures in CO model usually makes use of the recursive splitting of data until the chunk size of $\leq B$ is reached. For M usually the tall cache assumption is made ($M = \Omega(B^{1+\epsilon})$). For example $M = 4B$ is the most common in order for two blocks of data (split across 4 blocks) communicate to each other efficiently.

3 Parallel models of computation

The goal of using parallel systems to solve a problem is to obtain a p-fold speedup when p processors are available. *Instruction level parallelism* (ILP) [33] is the parallelism achieved in a single core using complex pipelines consisting of multiple processing elements (PEs). *Flynn's bottleneck* describes the fact that as long as the CPU issues one instruction per cycle, it cannot exceed IPC (executed Instructions Per Cycle) > 1. To overcome this limit, superscalar computers issue more than 1 instruction per cycle, but the following dependencies between instructions occur which reduce the theoretically upper-bound speedup:

Data dependencies or RAW hazards These occur when the result of the instruction, needed by another instruction, is not calculated yet. This can be solved by using *dynamic scheduling*.

Name dependencies The same register is used by different instructions, no data is passed between them though. This can effectively be solved by *register renaming*.

Control dependencies Because of the pipeline, instructions including the branches are not evaluated instantly. Meanwhile, other instructions already entered the pipeline which should not be executed. Control dependencies can be partially solved using the speculation (predictors). However, control dependencies are still the main reason why Flynn's bottleneck appear in practice.

Increasing IPC is a difficult task. Modern CPUs achieve IPC of around $0.8 - 1.5$ (http://software.intel.com/en-us/articles/intel-hyper-threading-technology-analysis-of-the-ht-effects-on-a-server-transactional-workload). In order to achieve better parallelism, multi-core or even many-core (e.g. when $p > 1000$) architectures are used today. In comparison to ILP, a whole core represents a single

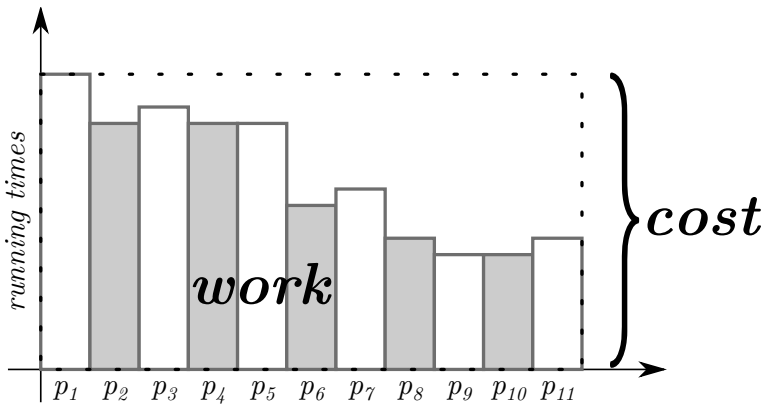

Figure 8: Comparison of the work and cost in practice on 11 processing elements.

processing element now and not separate processing units inside the pipeline. Multi-core architectures are comprised of a number of cores on a single chip (known as *chiplevel multi-processor* or CMP). Multiple chips (or processors) can be present inside a single computer. Technically cores communicate with each other using the *shared memory*. Multiple computers are connected using a network and communicate with each other using message passing. This kind of system organisation is called *distributed memory*. While parallel systems quickly achieve optimal speedup from the computing point of view, the communication between PEs represents the real issue in a parallel algorithm design and analysis [4]. Low memory bandwidth, latency and network congestion appear soon when p is increased. This is because the shared memory and the network became a shared point not only where data is stored, but also where synchronisation and other PE communication takes place.

Measures

We will present basic parallel measures as in [37]. Parallel models of computation assume a computer consisting of p processing elements. Function $T_{seq}(n)$ denotes the best running time of the algorithm on a problem size n in the sequential model. $T_{par}(p, n)$ denotes the running time of the algorithm on a problem of size n using p processors. We will abbreviate $T_{seq} = T(1)$ and $T_{par} = T(p)$.

Work $W(n)$ is defined as the sum of actual calculations (steps) done by each processor. *Cost* is defined as $C(n) = T(p) * p$. Figure 8 illustrates the difference.

Speedup is defined as $S_p(n) = \frac{T(1)}{T(p)}$ and *efficiency* as $E_p(n) = \frac{S_p(n)}{p} = \frac{T(1)}{C_p(n)}$. Obviously, $S_p(n) \leq p$ and $E_p(n) \leq 1$. If $S_p(n) = p$, the algorithm is optimal. If $S_p(n) = \Theta(p)$, the algorithm is efficient. Note that the speedup for optimal and efficient algorithms is independent of the problem size n.

Redundancy is defined as $\frac{W(p)}{W(1)}$, *utilization* as $\frac{W(p)}{pT(p)}$ and *quality* as $\frac{T^3(1)}{pT^2(p)W(p)}$.

Bounds and complexity classes

Amdahl's law [9] defines the maximum speedup of the algorithm as $S_p \leq \frac{1}{(1-P)+\frac{P}{p}}$ where P denotes the portion of parallel code in the algorithm and p the number of processing elements. This means even if infinitely number of processing elements are used for processing, actual speedup is always upper bounded by the amount of sequential code $1/1 - P$.

Before looking at concrete parallel models of computation, we introduce the simplest practically feasible circuit called AC^0 and a circuit complexity class NC often used for parallel algorithms. Circuit AC^0 consists of a polynomial number $O(n^\epsilon)$ of circuit elements of depth $O(1)$ and unlimited fan-in gates where n denotes the problem size.

An algorithm solving the problem of size n is in *Nick's class* (NC), if it can be solved in polylogarithmic time $O(\log^* n)$ using polynomial number of processing elements $O(n^\epsilon)$ with constant $O(1)$ fan-in gates. NC introduces its own *NC hierarchy*: NC^i is the i-th level in the NC hierarchy where i denotes the depth of processing elements and consequently the running time of the algorithm $O(\log^i n)$.

Integer addition, subtraction and multiplication was shown to be in NC^1 using the Wallace tree [50]. However, only addition and subtraction are in AC^0. Multiplication requires at least logarithmic circuit depth. Intuitively, NC is in a way similar to class P for sequential algorithms running in polynomial time — only such algorithms are practically feasible.

Parallel environment characteristics

The main question when designing a parallel model of computation is which characteristics of the parallel system should we capture. Maggs, Matheson and Tarjan [39] stress out the following ones:

Memory access How fast is the concurrent memory access to the same memory word or block?

Synchronization Are all processing elements synchronized using a global clock or asynchronous?

Latency What's the latency to access arbitrary word for the first time? What about the second time?

Bandwidth Is the bandwidth between the levels of the memory hierarchy unlimited?

Primitives Does the system has any unit-cost built-in primitives like the *scan* operation?

Good models capture as much characteristics as possible while still maintaining simplicity in algorithm design and analysis.

The network

The goal of an efficient network connecting p nodes representing the PEs is to have a small degree d, for example $O(\log p)$, and a small diameter $\max_{u,v}$ which allows for a message to be sent fast between any two PEs. In theory, the *Moore graph* is a graph with a minimal diameter for the given p and d. However, such graphs are usually hard to implement.

In practice PEs communicate using two families of network topologies: the *hyper-cube* and the *expander graph*. We will closely look at the former one. First, we define two operators: the brackets operator $\langle x \rangle$ denotes a set of elements $\{0, 1, 2, ..., x\}$ and the index i in word w_i denotes the word w with flipped i-th bit. For example $w = 5, w_0 = 4, w_1 = 7, w_2 = 1$.

The following graphs are commonly used in the hyper-cube family:

N-butterfly

$$V = \langle N \rangle \times \langle n + 1 \rangle$$

$$E = \{([w, t], [w, t + 1]), ([w, t], [w_t, t + 1]) | w \in \langle N \rangle, t \in \langle n \rangle\}$$

N-shuffle-exchange

$$V = \langle N \rangle$$

$$E = \{[w, w{+}1] | w \in V, w\ even\} \cup \{[w, 2w \mod (N{-}1)] | w \in < N{-}1 >\} \cup \{[N{-}1, N{-}1]\}$$

N-shuffle-exchange introduces two kind of edges: the *exchange* edges connect each node w to w_0 — the node differing in the least significant bit. The *shuffle* edges connect w to w' by cyclically shifting the binary representation of w. eg. $N = 8, Nnodes, w = 5, d = 3, diam = 2 \log N$ incident w edges are $(5, 4), (5, 6), (3, 5)$. The first one is the exchange edge and the other ones the shuffle edges.

N-cube (also called the *binary n-dimensional Hyper cube*)

$$V = \langle N \rangle$$

$$E = \{(w, w_i) | w \in V, i \in \langle n \rangle\}$$

N-cube-connected-cycles (also abbreviated as N-CCC) is an N-butterfly network except that for each $w \in \langle N \rangle$, nodes $[w, 0]$ and $[w, n]$ are the same node. The butterfly network is therefore transformed into a ring of n nodes.

In general, the hyper-cube family of networks can be used to efficiently execute the majority of algorithms like the FFT, odd-even merge etc. sketched in algorithm 1. For N-CCC and N-butterfly, we need $p = N \log N$ PEs and for N-butterfly and N-cube $p = N$.

Algorithm 1: The sketch of the family of algorithms which are efficiently executable on the hyper-cube family of networks.

```
for t = 0..n − 1 do
    for j ∈ ⟨N⟩ do in parallel
        if t^th bit of j = 0 then
            A[j], A[j + 2^t] ← OPER_{t,j}(A[j], A[j + 2^t]) ;
        end
    end
end
```

3.1 Parallel RAM — PRAM

PRAM [26] is a parallel model of computing consisting of p processing elements (PEs) and a uniform shared memory. Each processing element consists of a local private memory, accumulator and a program counter register. PEs are of MIMD type, but every processing element executes the instruction in the following synchronous steps where interprocessor communication and synchronisation is free:

1. Fetch an operand from the shared memory;
2. Perform some computation on local memory (registers);
3. Store a value back into the shared memory.

PRAM uses the same instruction set as RAM does plus the *FORK(label)* instruction. This instruction executed by PE P_i selects the first inactive processor P_j, clears its local memory, copies P_i's accumulator into P_j's and starts P_j running at *label*.

There are 3 models of PRAM based on their concurrent access to the same memory cell:

Concurrent Read Concurrent Write — CRCW Unbounded number of PEs can both read and write the same cell in a single step. The strongest model from the computing complexity point of view.

Concurrent Read Exclusive Write — CREW This was the initial PRAM model presented in 1978. An unbounded number of PEs can read the same cell and only one PE can write to a cell in a single step. If multiple PEs try to write to the same cell, PRAM halts. This PRAM variant is a good compromise between the ease of algorithm design and realism.

Exclusive Read Exclusive Write — EREW The memory cell is "locked" for reading and writing to a single PE per step. This is the weakest but the most realistic model.

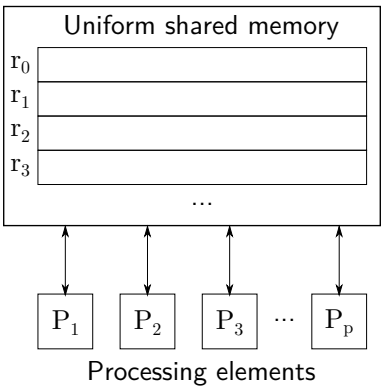

Figure 9: The PRAM model consisting of p processing elements and a uniform shared memory. Processors' local private memory, accumulator and program counter register are omitted.

Simulation

We can simulate every p processor CRCW PRAM on a EREW PRAM with a slowdown factor of $\Theta(\log p)$ (so called *separation factor*). The proof is based on ability of the EREW PRAM to sort, find or compute the result of p processors for every instruction in a program using the reduction which takes $O(\log p)$ time for p elements. It is important to know that this simulation is *not work-preserving* resulting in algorithms not exhibiting linear or near-linear speedups when increasing the number of PEs.

We can also reduce the number of processing elements p in the system. *Brent's theorem* [17] defines the maximum slowdown for the same PRAM model using different number of PEs: If an algorithm takes $T(1)$ time on a PRAM with an unlimited number of PEs executing m operations, a PRAM of the same model with p processing elements take $O(\frac{m}{p} + T(1))$ time. Using the Brent's theorem we can also show that any simulated algorithm in NC stays in NC no matter how many PEs we use.

CRCW hierarchies

CRCW model introduces the following submodels when concurrent writes to the same memory cell occur [42]:

Undefined — CRCW-U The value written is undefined.

Detecting — CRCW-D A special symbol representing detected collision is written.

Common or Consistent — CRCW-C Concurrent writes allowed only, if all store the same value. Also called the *weak CRCW* model [24].

Random or Arbitrary — CRCW-R The value of a random concurrent PE is written.

Priority — CRCW-P The processor's value with the lowest priority (index) is written.

Max/Min — CRCW-M The largest/smallest value is written. Also called the *strong CRCW* model.

Reduction or Fusion The arithmetic sum (CRCW-S), logical AND (CRCW-A), logical XOR (CRCW-X), or some other combination of the multiple values taking $O(1)$ time is written.

Above described submodels are not equally strong. A PRAM submodel is less powerful than the another submodel, if such problems exist for which the first model requires an order of magnitude more computational steps than the second one. For example, if we want to store the maximum value of all concurrent writes, CRCW-D requires at least $\Omega(\log n)$ steps, whereas CRCW-M writes the maximum value in $O(1)$. In general, the following relationships are known for some submodels:

$$EREW < CREW < CRCW\text{–}D < CRCW\text{–}C < CRCW\text{–}R < CRCW\text{–}P$$

Reader interested in strength comparison and simulation of different CRCW submodels should consult [19].

Algorithm design and analysis in PRAM model determines the parallel execution time $T(n)$ of the algorithm and the number of needed processing elements p using a specific PRAM EREW, CREW or CRCW model. PRAM model is the most abstract model of parallel computing and can serve as a reference for a more detailed analysis. Practically feasible algorithms are those, running in logarithmic parallel time using a linear number of processing elements running on EREW PRAM.

3.1.1 Module Parallel Computer — MPC

The Module Parallel Computer model [40] takes the slowdown of concurrently accessing the same memory block into account — also known as the problem of the *memory granularity*. This is a feasible assumption since the same data cannot be served to an unbounded number of PEs in constant time.

The MPC model splits the memory of size n into m modules allowing only a limited number of accesses per time unit. If $m = n$, then we get EREW PRAM. Let R_j denote the number of concurrent accesses to module $0 < j < m$ and $R_{max} = \max_j R_j$, the goal is to design algorithms in a way to minimize R_{max} over all steps. Obviously $R_{max} = p$ in worst case when $mp \leq n$. There are two approaches used to minimize R_{max}:

Randomization The key idea is to utilize universal hashing in the simulating machine. In general, using a uniform memory access distribution results in minimal memory contention.

Copies We keep several copies of each logical address in distinct memory modules. When memory contention happens even though randomization is used, the concurrent accesses to the same logical word are *load-balanced* to multiple physical memory modules.

The following relations between the EREW PRAM and MPC were shown:

- Let $m = p^3$. Then a $T(n)$-time bounded PRAM with p PEs and n memory cells can be simulated by a randomized MPC with p PEs and m memory modules and total memory of size $n + 2p$ in time $O(T(n))$.
- Let $m = p^2$. Then a $T(n)$-time bounded PRAM with p PEs and n memory cells can be simulated by a randomized MPC with p PEs and m memory modules and total memory of size $n + 4p$ in time $O(T(n) \log n)$.
- Let $m = p$. Then a $T(n)$-time bounded PRAM with p PEs and n memory cells can be simulated by a randomized MPC with p PEs and m memory modules and total memory of size $(n + p) \log p$ in time $O(T(n) \log p)$.
- Let $m = p^{1+\epsilon}$ for $\epsilon > 0$. Then a $T(n)$-time bounded PRAM with p PEs and n memory cells can be simulated by a randomized MPC with p PEs and m memory modules and total memory of size $(1+2/\epsilon)(N+p)$ in time $O(T(n))$.

Authors showed that there is a trade-off between the number of modules and the amount of additional memory we are willing to spend in order to achieve the desired speedup. Algorithms running in EREW PRAM run transparently in MPC though, taking the upper bounded time as shown above. The model is more interesting from the theoretical point of view as from the algorithm design and analysis.

3.1.2 Local-memory PRAM — LPRAM

Local-memory PRAM model [4] focuses on the communication complexity between the processing elements because the communication latency can represent the majority of the running time.

LPRAM is a PRAM model with unbounded both local and global memory. Each algorithm step can be of two types:

Communication step A processor can write, and then read a word from global memory;

Computation step A processor can perform an operation on at most two words that are present in its local memory.

The machine is taken to be a concurrent-read, exclusive-write PRAM. The input data are initially in the global memory and the final outputs must also be stored there.

LPRAM model measures the computation time (computation steps) and the number of communication steps per processing element separately. Only

the problem size n and the number of PEs p are taken into account. Measuring two complexities is useful because the latency is hidden implicitly inside the communication steps and can vary, depending on the architecture. Parallel computation time is measured the same as in the original PRAM without the global store and load operations. Parallel communication steps (or *communication delay*) are calculated by representing the algorithm as DAG and solving operation dependencies, similar to the Red-Blue pebble game in the EM model (see figure 5). Usually, a lower bound for p depending on n is provided for the algorithm to run efficiently. Authors showed that matrix multiplication two $n \times n$ matrices can be done in $O(n^3/p)$ computation time and $O(n^2/p^{2/3})$ communication steps.

3.1.3 Block PRAM — BPRAM

BPRAM model [3] exposes *spatial locality* to enhance efficiency. This is feasible because locality is offered by memory caches in the real memory hierarchy.

BPRAM has, similar to LPRAM model, unbounded local and global memory. Each PE can access local memory in a unit-time and transfer a block of words from the global memory to the local memory in $l+b$ time where l denotes the latency to find a block and b the block size. Any number of PEs can access global memory simultaneously (unlimited bandwidth). Access to the same word in the global memory is not allowed (EREW). Initially, data are located in the global memory and in the end it needs to be written back to the global memory.

Latency is theoretically of size $l = O(\log p)$ if machines are connected using a hyper-cube. However, the difficulty of wiring and packaging machines with large number of processors might make $l = p^\alpha$ where $\alpha > 0$ a more realistic measure.

In comparison to other models, if we set latency $l = 1$, we obtain the original EREW PRAM. If we do not allow block transfers ($b = 0$), we obtain the LPRAM model.

BPRAM measures the parallel time complexity and work taking the number of PEs p, problem size n and latency l into account. Usually, two complexities are given: one for a latency-processor product $lp \leq n$ and another for $lp > n$. Authors showed that the matrix multiplication of size $n \times n$ takes $\Theta(n^3)$ parallel running time on $p = n$-processor PRAM using a conservative memory latency $l = n$, which is not any better than running the algorithm on a uniprocessor RAM. Parallel running time on BPRAM can be reduced to $\Theta(n^2)$. Similarly, running times of matrix transposition can be reduced from $\Theta(n^2)$ taking latency $l = n$ into account to $\Theta(n \log n)$ using BPRAM and exposing spatial locality.

3.1.4 Asynchronous PRAMs

Original PRAM is a synchronous model of computation meaning there is a global clock for all p processing elements doing first reading from memory, then computing and finally writing back to the memory at each step of the algorithm. While processing elements inside a single machine might run synchronously,

many argue that massively parallel machines must be asynchronous because there is always an operating system and other programs not related to the algorithm running at the same time. Therefore, algorithms should be designed for an asynchronous model. We will introduce three asynchronous models: APRAM, Phase (L)PRAM and XPRAM.

APRAM APRAM [21] is a PRAM model consisting of p PEs and m global memory cells. Every algorithm step is divided into at most 3 substeps, called *events*: reading, computing and writing. Algorithm execution is now observed from the events point of view, so concurrent reads and write events from different algorithm steps can be executed simultaneously. To determine the correct execution order, we need to draw a DAG of events which describes their data dependencies. The synchronisation cost is hidden implicitly inside the algorithm's graph.

For synchronisation, a *virtual clock* is introduced. This clock ticks when at least one of the events is being processed. Every tick marks the end of a *round*. The Brent's theorem can directly be applied to APRAM: A t round, p PEs algorithm can be simulated using $p' \leq p$ PEs in $O(tp/p')$ rounds.

APRAM measures the minimum number of rounds needed to complete the computation depending on the problem size n and the number of PEs p. The goal in the APRAM model is to redesign algorithms so that PEs synchronise in constant-size sets only. When this is achieved, it leads to algorithms with the same time complexity as their PRAM counterparts.

Phase (L)PRAM Phase (L)PRAM [29] is a PRAM model consisting of p PEs, each with its own local, private memory and a global, shared memory. The model defines 4 types of instructions, completed after unbounded, but finite time:

Global reads Reading the contents of a shared memory location into a local memory location,

Local operations Computing done accessing the local memory only,

Global writes Writing the contents of a local memory back to the global memory,

Synchronization steps Logical points in a computation where each processor waits for the other processors to arrive before continuing in its local program.

The local program consists of a series of *phases* in which the PE runs independently, separated by synchronisation steps. In Phase (L)PRAM, these steps synchronise all PEs, assigned to the program. Synchronisation takes B time overall.

Processors can be of either exclusive or concurrent asynchronous read and write type to the shared memory. However, no processor may read the same

memory location that another one writes unless there is a synchronisation step involving both PEs between the two accesses. This eliminates race conditions between PEs.

The difference between the Phase PRAM and LPRAM is the PRAM charges a single unit time for global reads and writes, whereas the LPRAM charges d for global writes and $2d$ for global reads. This gives the following relation: Any EREW PRAM algorithm running in time t using p PEs can be simulated by and EREW Phase (L)PRAM running in time $O(Bt)$ with p/B processor. In worst case, PEs are synchronised after each step. The goal in designing efficient Phase (L)PRAM algorithms is to overcome these immediate time and processor bounds.

Brent's theorem can be used for Phase PRAM: A Phase PRAM program using p PEs, s synchronisation steps, a total of x work and running time $t+Bps$ can be simulated by a Phase PRAM using $p' < p$ PEs in time $O((x/p') + t + Bp's)$.

Phase (L)PRAM measures the parallel running time of the algorithm. Phase PRAM parameters are the problem size n, number of PEs p and synchronisation time B. Phase LPRAM adds the global memory writing time parameter d (reading time is $2d$). When analysing the algorithm complexity, a sum of critical paths between synchronisation steps are taken into account plus the total time needed for synchronisation steps.

XPRAM XPRAM model [48] assumes globally synchronised p sequential PEs, each with its own unbounded local memory, connected by a sparse message-passing communication network. PEs execute operations in supersteps.

In each superstep, every PE i performs a_i operations using only the access to the local memory, sends b_i and receives c_i messages. Each global operation takes $g \geq 1$ time. Every processor i thus needs

$$r_i = g^{-1}a_i + b_i + c_i$$

time to finish the step. The overall parallel running time of a superstep is

$$t = \max_{i \in \langle p \rangle}(r_i)$$

where $\langle p \rangle$ is a set of PEs.

XPRAM is bulk-synchronous model which means PEs should be barrier synchronised at regular time intervals. More precisely, we define the period of the superstep L meaning that every L steps, a superstep will be called. We round the runtime of a superstep to $(\lceil t/L \rceil)L$ standard global operations or time $(\lceil t/L \rceil)Lg$. The period L is usually dependent on p, the authors assumed $L = \log p$.

The memory of size m is split into modules. The i^{th} PE contains all words $j \in \langle m \rangle$ such that

$$j \mod p = i$$

. Then we choose a hash function h randomly from a class of universal hash functions H. Each h maps memory words $\langle m \rangle$ to themselves. The purpose of h is to spread out the memory accesses as uniformly as possible. *Good hash functions* are those having the expected number of references to the memory module

$$\hat{R} = O(\log p)$$

for the memory size $p \log p$. *Fast hash functions* take $O(g)$ time where g denotes the access time for the global memory.

Authors defined the E-PRAM as EREW PRAM and C-PRAM as CRCW PRAM with non-unit time access to the main memory g. In comparison to XPRAM they showed the properties of the following simulations:

- XPRAM with p PEs with fast good hash function can simulate an E-PRAM with v PEs with expected optimal efficiency if $v > p \log p$.
- XPRAM with p PEs with fast good hash function can simulate a C-PRAM with v PEs with expected optimal efficiency if $v \geq p^{1+\epsilon}$ for some $\epsilon > 0$.
- A p-cube or p-directed butterfly can simulate an XPRAM with p PEs with expected optimal efficiency.

The goal of the XPRAM model is to easily simulate more general PRAM algorithms and still having a model which can easily be implemented on networks. The model measures the parallel running times taking the number of PEs p, superstep period L and the access time to the global memory g into account.

3.1.5 Queued Read Queued Write PRAM — QRQW

Queued Read Queued Write PRAM [30] introduces a queued access to the RAM. The model focuses on *memory contention*, that is the number of PEs accessing the same memory cell in one PRAM step. Empirical results on commercial machines show that the slowdown in parallel environments is linear to the memory contention. In comparison to CRCW PRAM, the contention is completely ignored and the memory access takes a single time-unit for an unbounded number of PEs. On the other hand, the EREW memory access stalls PEs until the memory contention is solved, which appears too strict in practice. The majority of algorithms written in EREW model run faster on real architectures indeed.

Queued PRAM allows each location to be read or written by any number of processors in each step. Concurrent reads or writes to a location are serviced one-at-a-time. Queued reads and writes can be combined with PRAM's concurrent or exclusive reads and writes thus obtaining hybrid CRQW, QREW, QRCW or ERQW models. Authors empirically show that among concurrent, exclusive or queued PRAM models, the queued model (QRQW or CRQW) best describes the behaviour of existing super-computers.

QRQW PRAM is strictly more powerful than EREW PRAM and less powerful than CRCW for a separation factor of $\sqrt{\lg n}$ time and preserving work. Unlike CRCW, CREW or EREW PRAM which ignore network topology, QRQW

uses the more realistic hypercube-type non-combining networks. This type of network can practically be implemented and it's the reason for the linear slow-down when executing CRCW algorithms on many cores in practice.

Consider a single step of a PRAM, consisting of a read substep, compute substep and a write substep. The *maximum contention* (denoted κ) of the step which is the maximum, over all locations x, of the number of processors reading x or the number of processors writing x. A step with no reads or writes is defined to have maximum contention 1.

Two QRQW variants exist where both are synchronous in terms of the PRAM substeps (reading, computing, writing):

SIMD-QRQW PRAM model is a PRAM in which the time cost for a step with maximum contention κ is κ. If there are multiple writers to the same location, arbitrary PE succeeds.

QRQW PRAM is a PRAM consisting of a number of processors, each with its own private memory, communicating by reading and writing locations in a shared memory. The time cost for a step with the maximum contention κ, number of reads r_i, computations c_i and writes w_i per processor i is $\max_i(r_i, c_i, w_i, \kappa)$. This model, in comparison to SIMD-QRQW, CRCW or EREW, allows multiple reads and writes per PE at a time and adding read/write requests to a location queue.

The primary advantage from the algorithm complexity point of view of the QRQW PRAM model over the SIMD- is that the QRQW permits PEs each to perform a series of reads and writes in a step while incurring only a single penalty for the contention of these reads and writes. In SIMD-QRQW, a penalty is charged after each read or write in the series. Both models have the same order of complexity though: A p-processor QRQW PRAM algorithm running in time t can be emulated on a pt-processor SIMD-QRQW PRAM in time $O(t)$.

We adopt the Brent's theorem for the QRQW model: Any algorithm in the QRQW work-time presentation with x operations and time t, where t is the sum of the maximum contention at each step, runs in at most $\frac{x}{p} + t$ time on p PEs QRQW PRAM. We assume the processor allocation to be free.

QRQW PRAM measures, similar to the PRAM, the parallel time-cost per step, total parallel time and work, also taking memory contention into account. Empirically, QRQW model better describes the running times on commercial machines than the EREW or CRCW models without introducing any new variables in time complexity functions. The model is thus being more appropriate for high-level algorithm design.

3.2 Bulk Synchronous Process model — BSP

BSP [49] bridges in a performance-faithful manner what the hardware executes and what is in the mind of the software writer. The model assumes that:

- computation is divided into *supersteps*;

- PEs behave as asynchronous MIMD. Synchronisation is done after each superstep;
- PEs can arbitrarily send and receive messages to and from other PEs in a non-blocking manner;
- messages are delivered at the end of each superstep.

BSP is a message-passing model. Model parameters include speed of each processor s, the number of PE steps to synchronize processors or to build paths between them l (latency) and the number of processor steps per word to deliver a message g (gap) using an already built path. These parameters are unavoidable physical constraints. Beside the number of physical PEs p the model also introduces a number of virtual processors v. BSP assumes sufficiently large number of virtual processors ($v \geq p \log p$). Virtual processors can be looked as a processes or tasks. This eases algorithm design and reduces overall communication latency because context-switch latency is much lower than communication between the physical PEs.

BSP is a realistic model for measuring the communication complexity between PEs (the number of transferred messages). Algorithm analysis usually provides two complexities: one for small latency l and another one for the larger latency.

3.2.1 LogP machine model

LogP model [23] keeps all variables from BSP and extends it by adding additional overhead time o for preparing the message for transmission. The intuition behind is that if o is larger than l, BSP becomes unrealistic.

By introducing o, LogP bridges the shared-memory model with the distributed-model through an implicit exchange of messages [38] taking at most $O(l+o+g)$ time.

LogP is used for analysing the algorithm's communication complexity between PEs. Two results are usually provided depending on the small or large o.

3.2.2 Multi-level BSP model — Multi-BSP

Multi-BSP model [47] is a hierarchical model, with an arbitrary number of memory levels d (depth). In comparison to the basic BSP, no direct horizontal communication is allowed between PEs in Multi-BSP. Such communication would need to be simulated via memory at a higher level. This behaviour is typical for multi-core systems.

Multi-BSP uses $4d$ parameters

$$(p_1, g_1, L_1, m_1)(p_2, g_2, L_2, m_2)(p_3, g_3, L_3, m_3)...(p_d, g_d, L_d, m_d)$$

, where:

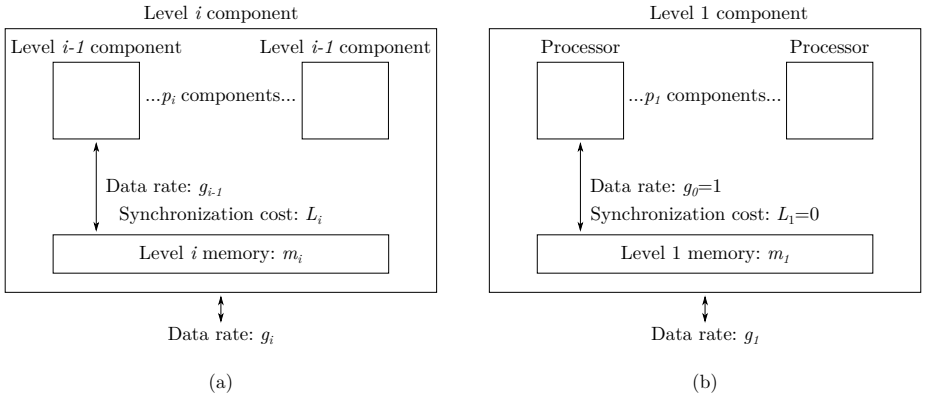

Figure 10: a) Schematic diagram of the Multi-BSP model component structure for the level i. b) Level 1 component.

p_i is the number of components on level $i-1$. p_1 denotes the number of raw processors, which can be regarded as a level 0 components.

g_i is the communication bandwidth parameter. It is the ratio of the number of operations that a processor can do in a second versus the number of words that can be transmitted in a second between the memories of a component at level i.

L_i is the cost charged for this barrier synchronization for a level i superstep. $L_1 = 0$ since the subcomponents of a level 1 have no memories and directly read from and write to the memory level 1.

m_i is the number of words of memory inside each of the i^{th} level component.

Figure 10 illustrates the model. In comparison to multi-core models like PMH (see section 3.3) or PCO (see section 3.4), it doesn't take the cache block size into account but the issue of synchronization.

Multi-BSP model can easily be adopted to describe similar models:

$d = 1, (p_1 = 1, g_1 = \infty, L_1 = 0, m_1)$ gives the von Neumann model,

$d = 1, (p_1 \geq 1, g_1 = \infty, L_1 = 0, m_1)$ gives the general PRAM,

$d = 2, (p_1 = 1, g_1 = g, L_1 = 0, m_1 = m)(p_2 = p, g_2 = \infty, L_2 = L, m_2)$ gives the general BSP model with constraint that communication between components on a the same level is forbidden, but requires passing the message to a level higher. This concrete model is called the BSPRAM model [46].

An *optimal Multi-BSP algorithm* with respect to the given algorithm in original BSP A is the following:

- It is optimal in *parallel computation steps* to the constant factors and in *total computation steps* to within additive lower order terms;
- It is optimal in *parallel communication costs* to constant factors among other Multi-BSP algorithms;
- It is optimal in *synchronization costs* to constant factors among other Multi-BSP algorithms.

With respect to the real architectures, Multi-BSP has two requirements. First, the barrier synchronisation needs to be supported efficiently. The second, the model controls the storage explicitly. This might be problematic because there are various cache protocols used in practice which are not necessarily revealed.

Multi-BSP measures the communication and synchronisation costs separately. Both are calculated by summing costs over all d levels. Each level cost depends on the $4d$ parameters described above. Multi-BSP doesn't define the usage of concurrent or exclusive reading or writing. Usually algorithms are analysed in EREW manner and problem lower bounds in CRCW.

3.3 Parallel Memory Hierarchy — PMH

PMH [8] consists of a height-h tree of memory units, called *caches*. The tree-of-caches has processing elements for leaves and the root represents an infinitely large main memory. For every node its cache is *shared* among all its descendants. Processing elements posses *private memory* because they don't have any siblings sharing their parents' memory.

The model assumes *non-inclusive cache*. This means that a memory block is stored only once in the whole tree without being stored at all ancestor caches. On writing a weak consistency is assumed meaning that cache lines are merged on writing back to memory thus avoiding "false sharing" issues.

Each level i in the tree is parametrised by four parameters:

- Cache size (M_i);
- Line or block size (B_i);
- Cost of a level-i cache miss (C_i);
- Fanout — the number of level-$(i-1)$ caches below a single level-i cache (f_i).

Figure 11 illustrates parallel memory hierarchy.

We will examine two specific variants of the PMH model: the two-level Parallel External Memory model and a three-level multicore cache model. Described PMH models ignore synchronization cost between PEs. Communication and latency are measured implicitly using the memory transfers between memory levels.

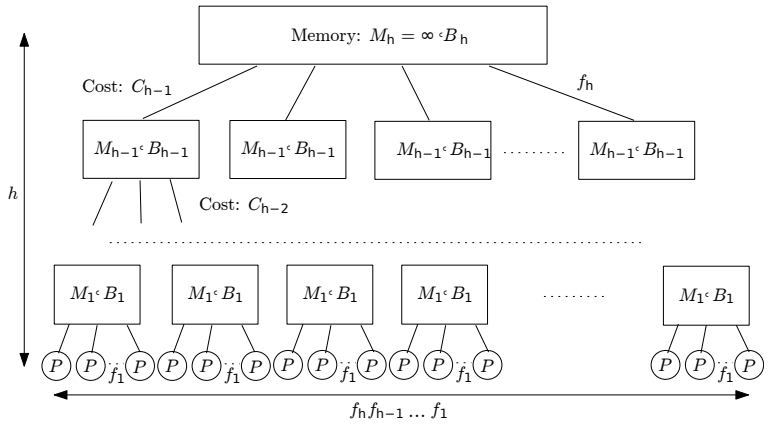

Figure 11: The Parallel Memory Hierarchy model.

3.3.1 Parallel External Memory - PEM

PEM model [10] offers a viable tradeoff between the Parallel RAM (PRAM) and External memory model (EM). The model is comprised of p processors and a two-level memory hierarchy consisting of the external memory shared by all processors and p internal memories (caches). Each cache is of size M and partitioned in blocks of size B. Caches are private meaning processor cannot access other processors' caches. To perform any operation on the data, a processor must have the data in its cache. The data is transferred between the main memory and the caches in blocks of size B. Figure 12 illustrates the model.

If concurrent writes to the main memory occur, any of the CRCW, CREW and EREW models from the PRAM can be assumed. For simulating CRCW on EREW, CRCW-P is used by sequentially writing the same block to the main memory. This can be done in $O(\log p)$ block transfers by combining resulting blocks of pairs, fourths and so on processors into a single block in $\log p$ rounds.

PEM measures the parallel I/O complexity meaning the number of *parallel block transfers*. For example, if an algorithm with each of the p processors simultaneously read one (different or the same) block from the main memory we would obtain $O(1)$ complexity and not $O(p)$ transfers. If we set the cache sizes to be constant or non-existant, the PEM algorithms turn into corresponding PRAM algorithms. On the other hand, if we only use a single processor, the algorithms turn into solutions for EM.

3.3.2 Multicore cache model — MC

The multicore cache model [14] reflects the real chip multiprocessor (CMP) architectures in a way that both the private (L1) and shared among other cores (L2) caches are on the same chip.

There are $p > 1$ processors in the MC model along with private L_1 caches

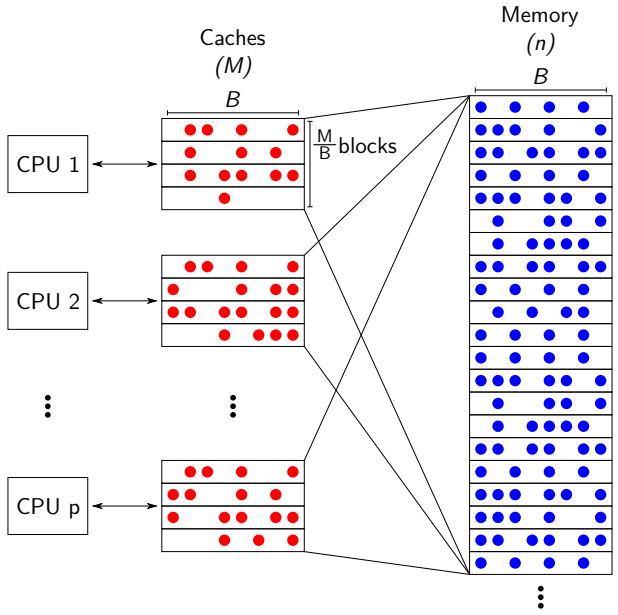

Figure 12: The Parallel External Memory model.

each of size C_1, a shared L_2 cache of size $C_2 \geq pC_1$ and the main memory of unbounded size. Each memory level is partitioned into data blocks of constant size B. Initially, all caches are empty and the input resides in the main memory. Each processor can only read/write into L_1 cache resulting in moving blocks from one level to another one to obtain this. There is only a single copy of the block in the caches or memory (the non-inclusive caches as in PMH). All other copies are "invalidated" as in the real CMPs. A block can also be "evicted" from a cache to make room for another block. The model assumes LRU block replacement strategy. The MC model is illustrated in figure 13.

The multi-level cache complexity largely depends on the selected task scheduler. This main issue happens when multiple CPUs want to write the same memory block at the same time. A key new dimension appears: *Scheduling tasks* so that the writes of multiple PEs to the same memory block are far apart in time. This is also called *constructive sharing* of a largely overlapping working set in contrast to the *destructive competition* for the limited on-chip cache.

To solve this issue, we need to represent the algorithm as DAG and analyse its cache accesses. In general, two families of schedulers exist. The *parallel depth-first* (PDF) scheduler schedules the task optimally according to their priority in the algorithm's DAG. The *work-stealing* (WS) scheduler forkfs tasks when a loop occurs and places them on a local processor's work queue. Processors take tasks from the bottom of the local queue and "steal" from other queues when their local queue gets empty. Both PDF and WS schedulers suffer from

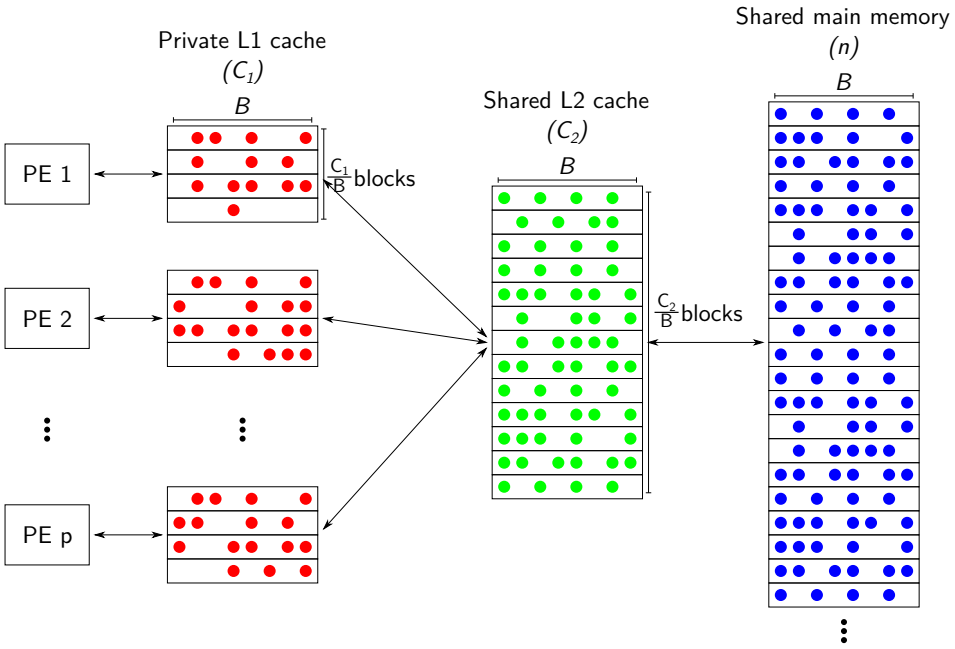

Figure 13: The Multicore cache model.

excessive shared and private cache misses respectively for pathological problems. Figure 14 illustrates the difference between the work-stealing (WS) schedulers and the space-bounded parallel depth-first schedulers (PDF) for the merge sort from the cache hit/miss ratio point of view.

The MC model considers three performance metrics: the parallel time complexity, and both the $L_1 \leftrightarrow L_2$ and $L_2 \leftrightarrow$ main memory block transfers. Complexity functions use the problems size n, number processing elements p, block size B and cache sizes C_1 and C_2. The goal when designing an algorithm in MC model is that cache misses match the sequential cache complexity while maintaining full parallel speed-up.

3.4 Parallel Cache-Oblivious model — PCO

PCO is a parallel version of the cache oblivious model. The main issue when exposing the CO model to more processing elements is that caches are not fully shared anymore. This eliminates the CO theorem which proves, if cache complexity is ideal for two levels of the memory hierarchy it is ideal for all levels of the memory hierarchy. PCO model overcomes this issue by introducing a modified *space-bounded* scheduler.

PCO model assumes algorithms with *nested parallelism* which allows arbitrary dynamic nesting of parallel loops (different depths for different iterations).

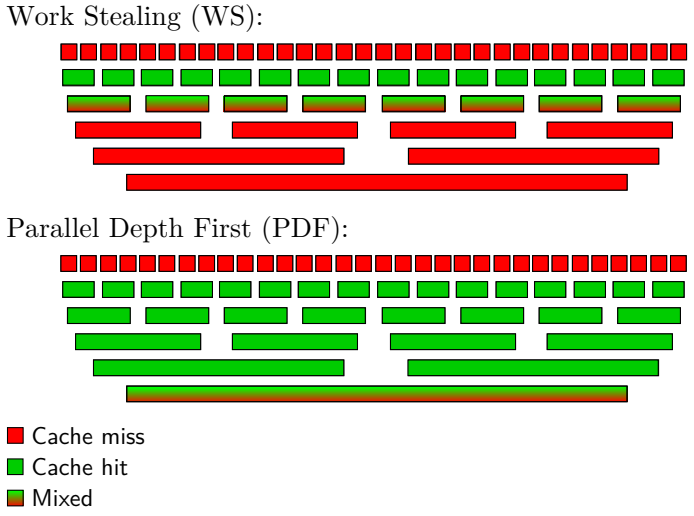

Figure 14: Parallel Merge Sort: WS vs. PDF schedulers cache miss ratio using 8 cores where the shared cache is of size n. PDF uses sequential execution order of tasks for task priorities.

Synchronisation is done solely on forks and joins. Majority of algorithms can be presented using the nested parallelism.

PCO model [15] uses the Parallel memory hierarchy (PMH) architecture having tree-of-caches. PCO consists of three components: the parallel cache cost model, the new cost metric dependent on the level of parallelism and a modified space-bounded scheduler. The *PCO cache cost model* originates from the CO model's cache complexity $Q^*(n; M, B)$. The model for sequential strands remains the same as in CO. When a task t forks a set of child tasks though, one of the following rules are considered:

- If t fits in M, all children tasks start with the cache state of the parent task.
- If t doesn't fit in M, cache state is emptied at the fork and join points.

PCO assumes data reuse among PEs only when there is a serial relationship between instructions accessing the same data. Formally, the cache complexity for a strand s, parallel block b and a task t is defined as:

$$Q^*(s; M, B; \kappa) = Q_{CO}(s; M, B; \kappa)$$

For $b = t_1||t_2||...||t_k$:

$$Q^*(b; M, B; \kappa) = \sum_{i=1}^{k} Q^*(t_i; M, B; \kappa)$$

And for $t = c_1; c_2; ...; c_k$:

$$Q^*(t; M, B; \kappa) = \sum_{i=1}^{k} Q^*(c_i; M, B; \kappa_{i-1})$$

where $\kappa_i = \emptyset$, if $S(t; B) > M$ and $\kappa_i = \kappa \cup_{j=1}^{i} loc(c_j; B)$, if $S(t; B) \leq M$. $loc(t; B)$ denotes the set of distinct cache lines accessed by task t and κ the cache state at the given point in time.

Work complexity is obtained by setting $M = 0$ and $B = 1$ obtaining $Q^*(t; 0, 1; \kappa)$.

The second component — the new cost metric — penalizes large imbalance in ratio of space to parallelism in subtasks. If the level of parallelism α is low, this consequently leads to unused processing elements along with their caches resulting in lower cache sizes overall. Authors define the *effective cache complexity* $\hat{Q}_\alpha(c)$ taking this parameter into account. It was also shown in [16] that with sufficient available parallelism, for p processors only a slightly larger shared cache is needed than for a single processor (usually adds $O(p \log^* n)$).

The third component — the *modified space-bounded scheduler* — is an extension to the greedy space-bounded scheduler [20]. A space-bounded scheduler accepts dynamically parallel programs that have been annotated with space requirements for each recursive subcomputation called a "task". Authors show that any space-bounded scheduler assures that the number of misses across all caches at each level i of the machine's hierarchy is at most $Q^*(n; M_i, B_i)$. In comparison to WS and PDF schedulers, the modified space-bounded scheduler provably bounds the number of cache misses.

PCO measures the running time on parallel machines and the cache complexity of the algorithm the same way as CO does, but taking PMH into account. The cache complexity function includes the problem size n, temporal locality (M), spatial locality (B) and the level of parallelism (α). Parallel running times also include the number of processing elements p. Optimal PCO algorithms have the same cache complexity as in the CO model and the same running times divided by p.

4 Vector RAM — VRAM

Vector RAM [13] (VRAM) model is used to describe vector machines. Vector processor in comparison to the scalar one (as seen in (P)RAM model) uses the vector as a primitive data structure. Scalar PEs on the other hand only use scalar atomic values for operands.

VRAM model beside the memory as in PRAM defines another memory as an unbounded number of addressable memory cells each containing arbitrarily large *simple vector* of words. In addition to the vector memory it also contains a parallel vector processor and vector I/O ports. Figure 15 illustrates a pure vector architecture.

VRAM instruction set includes the following primitive instruction set:

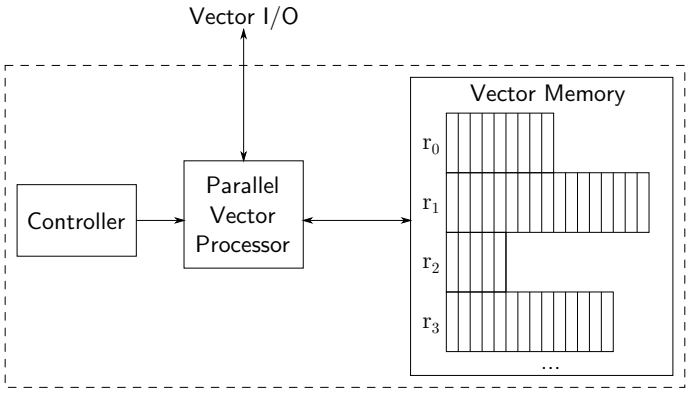

Figure 15: Vector RAM architecture consisting of the Controller, Vector CPU and Vector memory consisting of arbitrarily large vector words. The scalar memory and scalar CPU can also be present separately, but are omitted in the figure.

Scalar instructions Basically instructions of a standard RAM like the arithmetic and logical operations, a conditional-jump instruction and indirect-addressing instructions.

Elementwise instructions They operate on equal-length vectors producing a result vector of the same length. These are arithmetic and logical operations such as $+$, $-$, $*$, OR and NOT.

Permute instructions These take two vector arguments: a data vector and an index vector. The result vector includes the elements from the data vector permuted for locations in the index vector. Resulting vector can be of same size as the data vector is or less, if the index vector doesn't include all the element indices.

Scan instructions These operations execute a scan operation on a vector. These include the sum over all the elements in the vector, finding a minimum or maximum element.

Vector-Scalar instructions These instructions are *extract*, *insert*, *distribute* and *length* and take both scalar and vector arguments.

Each instruction is executed in a single unit-step. Using a combination of these instructions one can implement a rich set of operations. Authors provide implementations of the *index*, *reduce*, *distribute*, *append*, *pack*, *split*, *flag-merge*, *inverse-permute*, *enumerate* and *max-index* using $O(1)$ steps.

From the programming point of view the main difference between VRAM and PRAM is that in PRAM the parallelism comes from serial primitives running in parallel. Programming requires both serial (program running on each

PE) and parallel control (synchronisation). In VRAM, the parallelism comes from parallel primitives themselves while strictly having a serial control. This is also the reason why VRAM is not classified as a parallel model of computation — it doesn't provide a parallel control.

VRAM measures the *step complexity* and the *element complexity*. The step complexity is the number of calls to the primitive instructions and the element complexity is the vector length (source and destination) per primitive instruction call, summed over the number of calls. Step complexity can be thought of as the parallel complexity of the algorithm for $p = \infty$ in the PRAM model and the element complexity for $p = 1$. Similarly one could think the step complexity as the number of layers in the boolean circuit and the element complexity as a sum over the elements over all the layers. VRAM model can describe both the SIMD architecture, if taking the algorithm step complexity into account, or SISD architecture, if taking the algorithm element complexity into account. Both complexities depend only on the problem size n and not the number of PEs. Algorithm step complexity in VRAM take $O(\lg n)$ less than the parallel complexity of algorithms in EREW PRAM and for some algorithms in CRCW PRAM as well.

4.1 K-model

K-model [18] is a hybrid model of computation capturing characteristics of modern Graphical Processing Units (GPUs). These consist of a number of *stream multiprocessors* (SM). Each SM contains many cores executing the same instruction over different data (SIMD).

A *stream program* organizes data as streams and expresses all computations as *kernels*. A stream is a sequence of elements of the same type and in comparison to array does not allow random access to arbitrary location. A kernel takes a set of input streams, performs a computation and produces output streams. Each stream can be manipulated independently. Streams passing among multiple computation kernels form a stream program.

Figure 16 illustrates the K-model architecture. K-model consists of k scalar execution units $E[k]$ connected to a single *instruction unit* where our stream program is stored. Variable k therefore denotes the number of scalar execution units in a single SIMD processing element in contrast to variable p found in parallel models of computation which denotes fully fledged MIMD machines. Behaviour of these execution units closely resembles the VRAM model for its serial control. K-model introduces two types of memories:

Internal memory Also called the *shared memory* located on a SM chip in the GPU world. Internal memory is of unbounded size divided into k modules or *banks*. For every bank, the Queued-Read-Queued-Write model is assumed (see chapter 3.1.5).

External memory Also called the *global memory* located on a device in the GPU world. External memory is of unbounded size divided into blocks of size k. A single block is transferred to PEs in a transaction so the

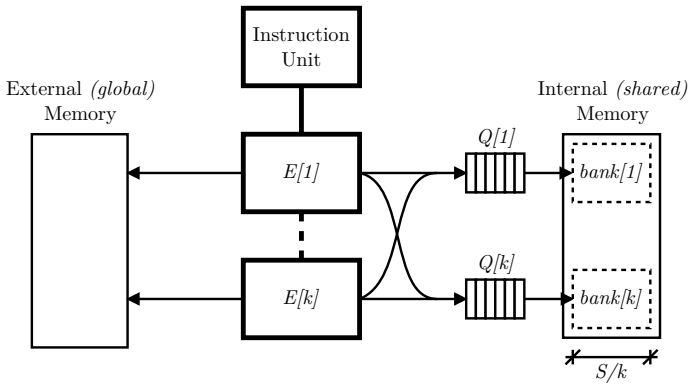

Figure 16: The K-model architecture consisting of k PEs, an instruction unit, queued internal memory and the external memory.

coalesced access pattern is the optimal one. This memory is similar to the PEM for $B = k$ and $M = k$.

Scalar execution units can communicate with both internal and external memory depending on the stream program. Initially, all the input data is stored in the external memory and eventually all the output data need to be stored there.

K-model measures three complexities: the parallel time or *latency* (T), the work or *length* (W) and external memory transactions (G). The parallel time is the latency for each instruction summed over all executed instruction on k execution units. According to the QRQW model, the instruction latency is increased for the number of simultaneous accesses to the internal memory. The work describes the execution time for a single scalar execution unit. Ideally, as in the parallel models, the optimal speedup is achieved when $\frac{W}{kT} = 1$. Function G measures the number of transferred blocks from and to the external memory. Algorithm designers often pick a trade-off between the external and the shared memory usage. All the complexities depend solely on two parameters — the problem size n and the number of scalar processing elements k.

5 Conclusion

In this survey we investigated the most significant sequential models of computations through time along with their parallel model counterparts. Finally we described a Vector RAM model which is in-between the sequential and parallel models of computation. A good model of computation gives the algorithm designer formal environment to design and analyse a new or improve the existing algorithm assuring good performance on real architectures.

To design an efficient algorithm we need to consider the architectural bottle-

necks. The most critical one in sequential algorithms is the time needed for data to become available for the processing element to operate with them. In the parallel environment, we aim to ideally achieve a p-fold speedup using p processing elements. Basically, parallel environments can be classified as the multi-core or many-core environments which offer both serial and parallel control over the processing elements, and the vector stream processing environments which offer only serial control.

When dealing with many-core environments (very high number of processing elements) the time needed for PEs for synchronisation and communication becomes considerable. Modern parallel models of computation such as the PCO assume we can bound the needed time for synchronisation and communication, if we localise the algorithm execution by focusing on efficient usage of the memory cache. This cache complexity bound also reflects to the communication time in the practically feasible hyper-cube network topology. Therefore, it is crucial to design parallel algorithms with good parallel cache complexity. This is generally done in two steps: First, we design a cache-oblivious algorithm that is cache efficient under sequential execution and the execution graph has low depth (e.g. $O(\log^* n)$). Second, we choose an appropriate scheduler that converts good sequential cache complexity to a good parallel one.

Programming for vector stream processors is different in a way that primitive instructions can manipulate on a vector of data and not a single memory word. This forces algorithm designer to focus on parallel implementation from the beginning because the only way to write parallel programs for such machines is to use separate vector instructions in a serial control manner. It requires organising data in a way that vector instructions can effectively manipulate them. While the serial control in the program seems easier than the approach described for the multi-core scalar computers, the memory hierarchy and the limited vector size in practice make designing the algorithms for such architectures as difficult as for the multi-core ones.

References

[1] Aggarwal, A., Alpern, B., Chandra, A., and Snir, M. A model for hierarchical memory. In *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC '87* (New York, New York, USA, Jan. 1987), ACM Press, pp. 305–314.

[2] Aggarwal, A., Chandra, A. K., and Snir, M. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)* (Oct. 1987), IEEE, pp. 204–216.

[3] Aggarwal, A., Chandra, A. K., and Snir, M. On communication latency in PRAM computations. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures - SPAA '89* (New York, New York, USA, Mar. 1989), ACM Press, pp. 11–21.

[4] Aggarwal, A., Chandra, A. K., and Snir, M. Communication complexity of PRAMs. *Theoretical Computer Science 71*, 1 (Mar. 1990), 3–28.

[5] Aggarwal, A., and Vitter, Jeffrey, S. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9 (Aug. 1988), 1116–1127.

[6] Aho, A. V., and Hopcroft, J. E. The Design and Analysis of Computer Algorithms.

[7] Alpern, B., Carter, L., and Feig, E. Uniform memory hierarchies. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science* (Oct. 1990), IEEE Comput. Soc. Press, pp. 600–608.

[8] Alpern, B., Carter, L., and Ferrante, J. Modeling Parallel Computers as Memory Hierarchies (Extended Abstract). Tech. rep., Jan. 1993.

[9] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)* (New York, New York, USA, Apr. 1967), ACM Press, p. 483.

[10] Arge, L., Goodrich, M. T., Nelson, M., and Sitchinava, N. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures - SPAA '08* (New York, New York, USA, June 2008), ACM Press, p. 197.

[11] Ben-Amram, A. M. What is a "pointer machine"? *ACM SIGACT News 26*, 2 (1995), 88–95.

[12] Bertoni, A., Mauri, G., and Sabadini, N. A characterization of the class of functions computable in polynomial time on Random Access Machines. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81* (New York, New York, USA, May 1981), ACM Press, pp. 168–176.

[13] Blelloch, G. E. Vector models for data-parallel computing.

[14] Blelloch, G. E., Chowdhury, R. A., Gibbons, P. B., Ramachandran, V., Chen, S., and Kozuch, M. Provably good multicore cache performance for divide-and-conquer algorithms. 501–510.

[15] Blelloch, G. E., Fineman, J. T., Gibbons, P. B., and Simhadri, H. V. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures - SPAA '11* (New York, New York, USA, June 2011), ACM Press, p. 355.

[16] Blelloch, G. E., Gibbons, P. B., and Matias, Y. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM 46*, 2 (Mar. 1999), 281–321.

[17] Brent, R. P. The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM 21*, 2 (Apr. 1974), 201–206.

[18] Capannini, G., Silvestri, F., and Baraglia, R. K-Model: A New Computational Model for Stream Processors. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on* (2010), pp. 239–246.

[19] Chlebus, B. S., Diks, K., Hagerup, T., and Radzik, T. Efficient Simulations Between Concurrent-Read Concurrent-Write PRAM Models. 231–239.

[20] Chowdhury, R. A., Silvestri, F., Blakeley, B., and Ramachandran, V. Oblivious algorithms for multicores and network of processors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (Apr. 2010), pp. 1–12.

[21] Cole, R., and Zajicek, O. The APRAM: incorporating asynchrony into the PRAM model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures - SPAA '89* (New York, New York, USA, Mar. 1989), ACM Press, pp. 169–178.

[22] Cook, S. A., and Reckhow, R. A. Time-bounded random access machines. In *Proceedings of the fourth annual ACM symposium on Theory of computing - STOC '72* (New York, New York, USA, May 1972), ACM Press, pp. 73–80.

[23] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. LogP: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPOPP '93* (New York, New York, USA, Aug. 1993), vol. 28, ACM Press, pp. 1–12.

[24] Eppstein, D., and Galil, Z. Parallel algorithmic techniques for combinatorial computation. *Annual Review Of Computer Science 3* (1988), 233–283.

[25] Floyd, R. W. Permuting information in idealized two-level storage. *Complexity of Computer Calculations*, 105 (1972).

[26] Fortune, S., and Wyllie, J. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing - STOC '78* (New York, New York, USA, May 1978), ACM Press, pp. 114–118.

[27] Fredman, M. L., and Willard, D. E. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci. 47*, 3 (Dec. 1993), 424–436.

[28] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. Cache-Oblivious Algorithms. In *Foundations of Computer Science, IEEE Annual Symposium on* (Oct. 1999), IEEE Computer Society, p. 285.

[29] Gibbons, P. B. A more practical PRAM model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures - SPAA '89* (New York, New York, USA, Mar. 1989), ACM Press, pp. 158–168.

[30] Gibbons, P. B., Matias, Y., and Ramachandran, V. The Queue-Read Queue-Write PRAM Model: Accounting for Contention in ParallelAlgorithms. Tech. rep., June 1996.

[31] Hagerup, T. Sorting and Searching on the Word RAM. 366–398.

[32] Hartmanis, J. Computational complexity of random access stored program machines. *Theory of Computing Systems 5*, 3 (1971), 232–245.

[33] Hennessy, J. L., and Patterson, D. A. Computer Architecture, Fourth Edition: A Quantitative Approach.

[34] Hopcroft, J., Paul, W., and Valiant, L. On Time Versus Space. *Journal of the ACM 24*, 2 (Apr. 1977), 332–337.

[35] Jia-Wei, H., and Kung, H. T. I/O complexity. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81* (New York, New York, USA, May 1981), ACM Press, pp. 326–333.

[36] Lambek, J. How to program an infinite abacus. *Canad. Math. Bull. 4* (1961), 295–302.

[37] Lee, R. B. Empirical results on the speed, efficiency, redundancy and quality of parallel computations. *International Conference Parallel Processing* (1980), 91–100.

[38] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. The Stanford Dash multiprocessor. *Computer 25*, 3 (Mar. 1992), 63–79.

[39] Maggs, B. M., Matheson, L. R., and Tarjan, R. E. Models of parallel computation: a survey and synthesis. *Proceedings of the TwentyEighth Annual Hawaii International Conference on System Sciences 2* (1995), 61–70.

[40] Mehlhorn, K., and Vishkin, U. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica 21*, 4 (Nov. 1984), 339–374.

[41] Minsky, M. L. *Computation: finite and infinite machines*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, Jan. 1967.

[42] Parhami, B. *Introduction to Parallel Processing: Algorithms and Architectures*. Springer, Jan. 1999.

[43] Schönhage, A. Storage Modification Machines. *SIAM Journal on Computing 9*, 3 (1980), 490–508.

[44] Shepherdson, J. C., and Sturgis, H. E. Computability of Recursive Functions. *Journal of the ACM 10*, 2 (Apr. 1963), 217–255.

[45] Sleator, D. D., and Tarjan, R. E. Amortized efficiency of list update and paging rules. *Communications of the ACM 28*, 2 (Feb. 1985), 202–208.

[46] Tiskin, A. The bulk-synchronous parallel random access machine. *Theoretical Computer Science 196*, 1-2 (Apr. 1998), 109–130.

[47] Valiant, L. A Bridging Model for Multi-core Computing. In *Algorithms - ESA 2008*, D. Halperin and K. Mehlhorn, Eds., vol. 5193 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 13–28.

[48] Valiant, L. G. *General purpose parallel architectures*. MIT Press Cambridge, MA, USA, Jan. 1989.

[49] Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (Aug. 1990), 103–111.

[50] Wallace, C. S. A Suggestion for a Fast Multiplier. *Electronic Computers, IEEE Transactions on EC-13*, 1 (1964), 14–17.

[51] Yao, A. C.-C. Should Tables Be Sorted? *Journal of the ACM 28*, 3 (July 1981), 615–628.