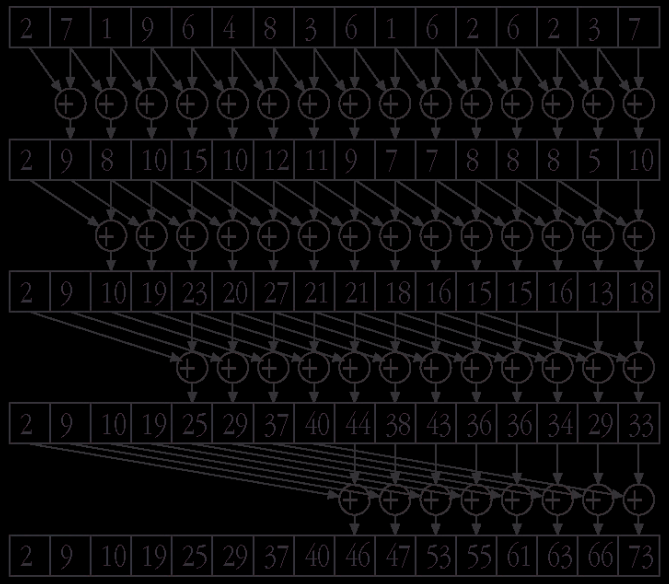
VTT

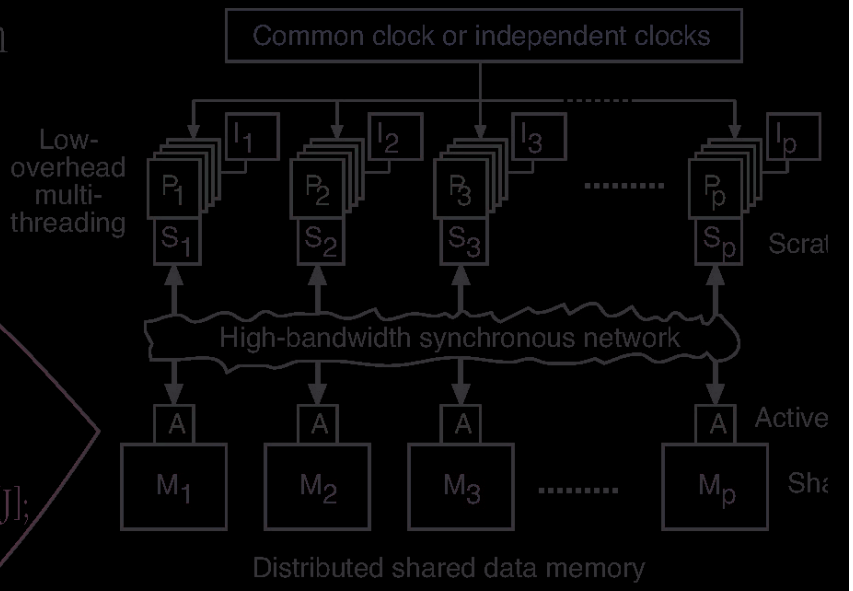
Parallelism, programmability and architectural support for them on multi-core machines

Martti Forsell

Platform Architectures Team
VTT, Oulu
Martti.Forsell@VTT.Fi




```
FOR K:=0 TO ⌈log N⌉-1 DO
  FOR J:=2^K+1 TO N PARDO
    Table[J]:=Table[J-2^K]+Table[J];
```



Artist Summer School in Europe 2011

September 4-9, 2011
Aix-Les-Bains, France

Parallelism, programmability and architectural support for them on multi-core machines

ABSTRACT: The advent of multi-core systems (CMPs, MP-SOCs, NOCs) has raised an old but very challenging problem to the limelights of embedded and general purpose system design—How to program these parallel systems so that application development would be as simple as for single core systems and still get high utilization and close to linearly improving performance with respect to sequential solutions out of them for all kinds of computational problems? Namely, according to our measurements and also wide-spread consensus among computer architects and software developers, current solutions making use of asynchronous shared memory and message passing do not solve neither the above programmability nor performance requirements unless the set of applicable computational problems is severely limited. Since virtually all current solutions define the same computability as the strongest theoretical models, i.e. can simulate each others at some, not necessarily linearly slowed, execution rate, the problem is mainly architectural—current multi-core machines are not efficient enough in executing certain important patterns of parallel processing.

In this lecture we will take a look at the nature of parallelism in computation and methods to capture intrinsic parallelism ignoring the most architectural implementation dependent details. To give a more practical insight, we will explain the key problems related to current multi-core solutions and the effect of them on performance and programmability of parallel computational problems both at methodological and architectural level using qualitative examples and our quantitative performance models. We will also take a look at architectures that could be used to avoid these problems and therefore provide significantly simpler parallel programmability without sacrificing the performance for a very wide set of computational problems. Simplified application examples are given.

About the author...

Martti Forsell is a Chief Research Scientist of Computer Architecture and Parallel Computing at VTT, Oulu, Finland, as well as an Adjunct Professor in the Department of Electrical and Information Engineering at the University of Oulu. He received M.Sc., Ph.Lic., and Ph.D. degrees in computer science from the University of Joensuu, Finland in 1991, 1994, and 1997 respectively. Prior to joining VTT, he has acted as a lecturer, researcher, and acting professor in the Department of Computer Science, University of Joensuu. Dr. Forsell has a long background in parallel and sequential computer architecture and parallel computing research. He is the inventor of the first scalable high-performance CMP architecture armed with an easy-to-use general-purpose parallel application development scheme (consisting of a computational model, programming language, experimental optimizing compiler, and simulation tools) exploiting the PRAM-model, as well as a number of other TLP and ILP architectures, architectural techniques and development methodologies and tools for general purpose computing. At application-specific front, he has acted as the main architect of the Silicon Hive CSP 2500 processor and programming methodology aimed for low-power digital front-end radio signal processing. He is a co-organizer of the Highly Parallel Processing on a Chip (HPPC) workshop series. His current research interests are processor and computer architectures, chip multi-processors, networks on chip, models of parallel computing, functionality mapping techniques, parallel languages, compilers, simulators, and performance, silicon area and power consumption modeling. He has published 85 scientific publications, holds one patent on processor architectures and programming methodology, and has participated to various research and development projects in cooperation with academia and industry. Recently has has been named as the leader of a large VTT funded project, REPLICA, aiming to remove the performance and programmability limitations of chip multiprocessor architectures with a help of a strong PRAM model of computation.

Contents

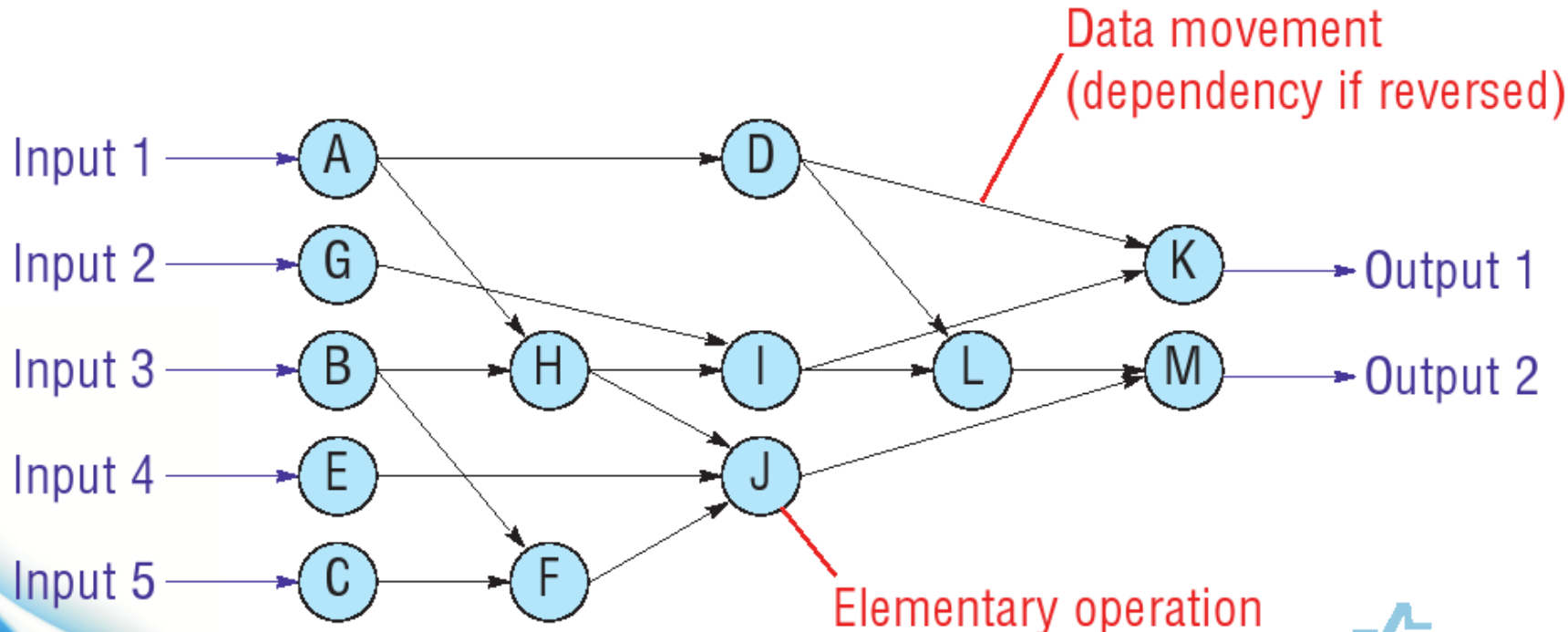
Part 1. Parallelism

Computation and its presentation
Executing computation—adding time and machinery dimensions
Reusing execution units (renamed as processors)
Alternative ways to execute, concept of work
Mapping and scheduling
Real world challenges
Existence of parallelism

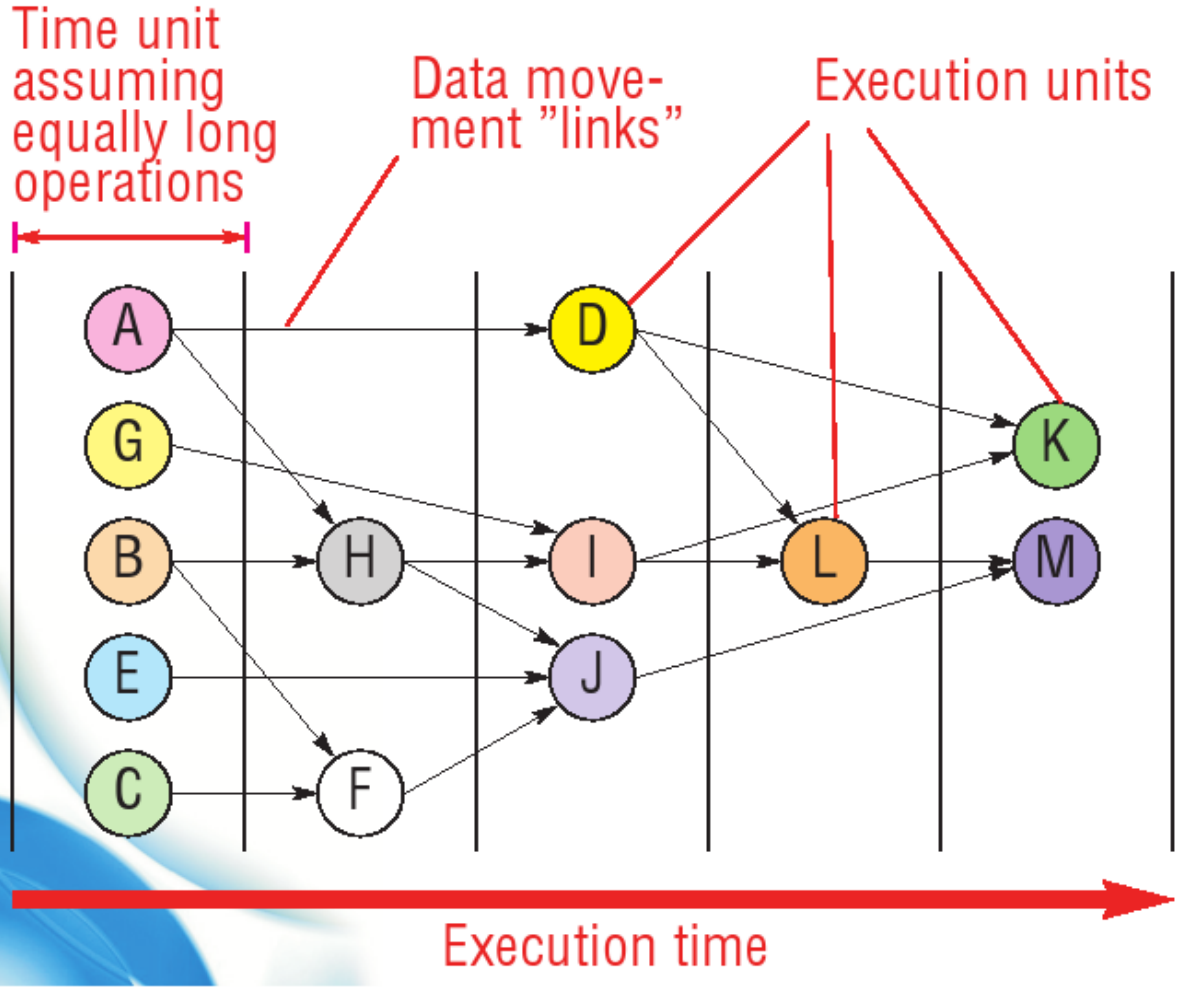
Computation and its presentation

Computation = apply some operations to input data to get output data

Let's present computation as a graph:



Executing computation—adding time and machinery dimensions

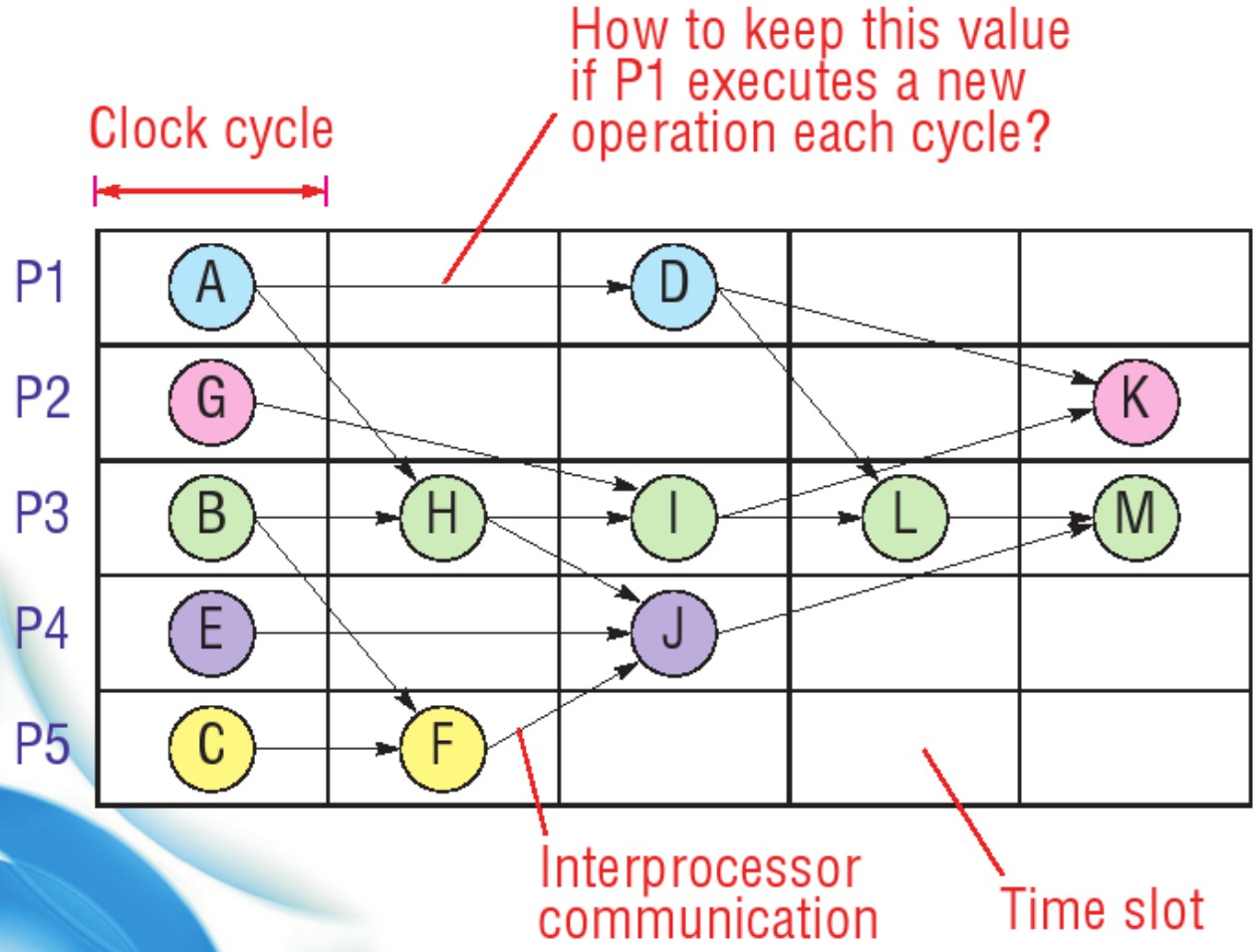


Dedicated logic -style solution

Problems:

- throughput
  (5 time units between executions)
- high Fan-In, Fan-Out
  (buffers may be needed)
- high number of execution units
  (13 potentially different execution units)
- programmability
  (HDL, new implementation for each functionality)

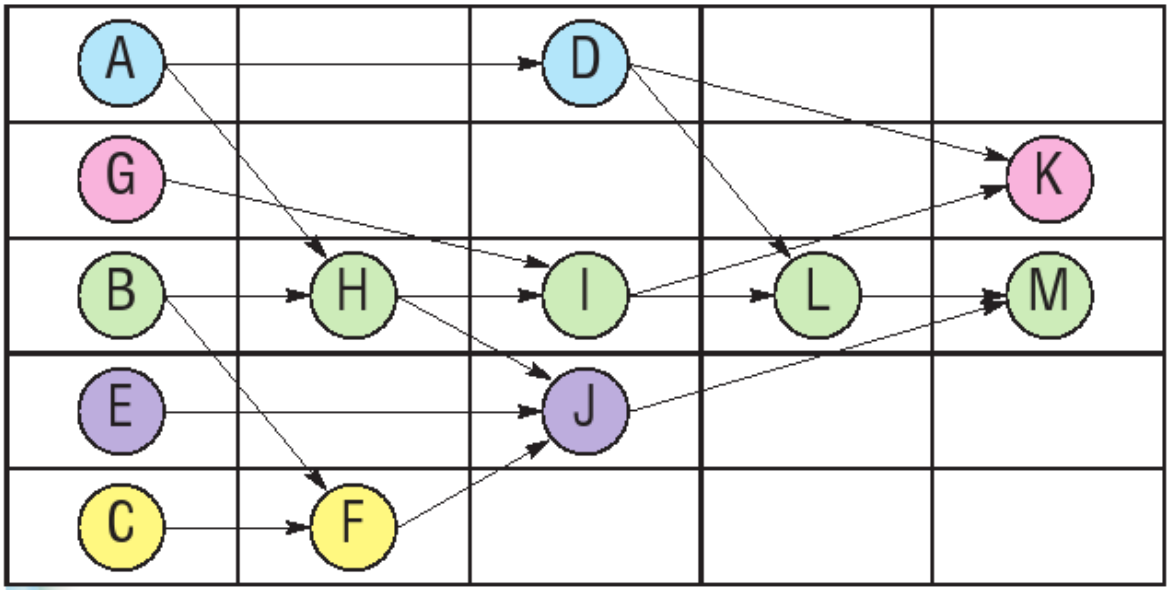
Reusing execution units (renamed as processors)



Challenges:

- synchronization/balancing OPs (max of OP latencies ≤ min clock cycle => overhead)
- high Fan-In, Fan-Out (buffers/queuing needed)
- multipurpose units (program & sequencer HW needed for each processor => overhead)
- programmability (program for each functionality => concept of control needed)
- storaging (long arcs require extra units that just repeat data => memory needed)

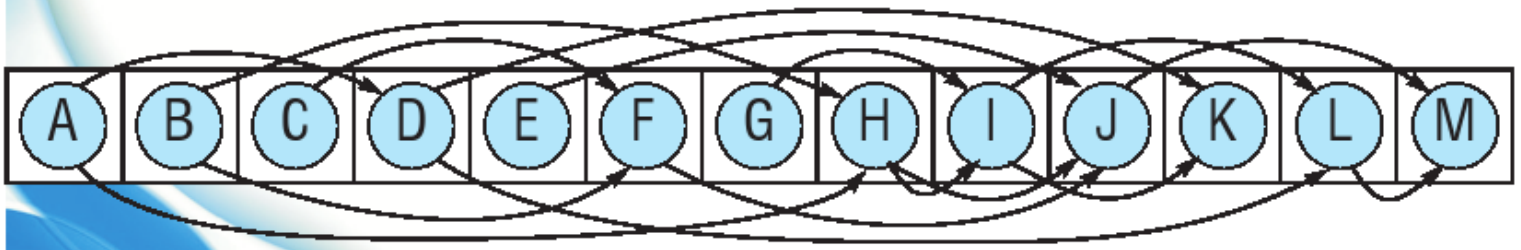
Alternative ways to execute, concept of work



Fastest parallel execution

- T=5, #P=5, W=25 (limited by dependencies, not by slots)

Work = execution time T x number of processors P



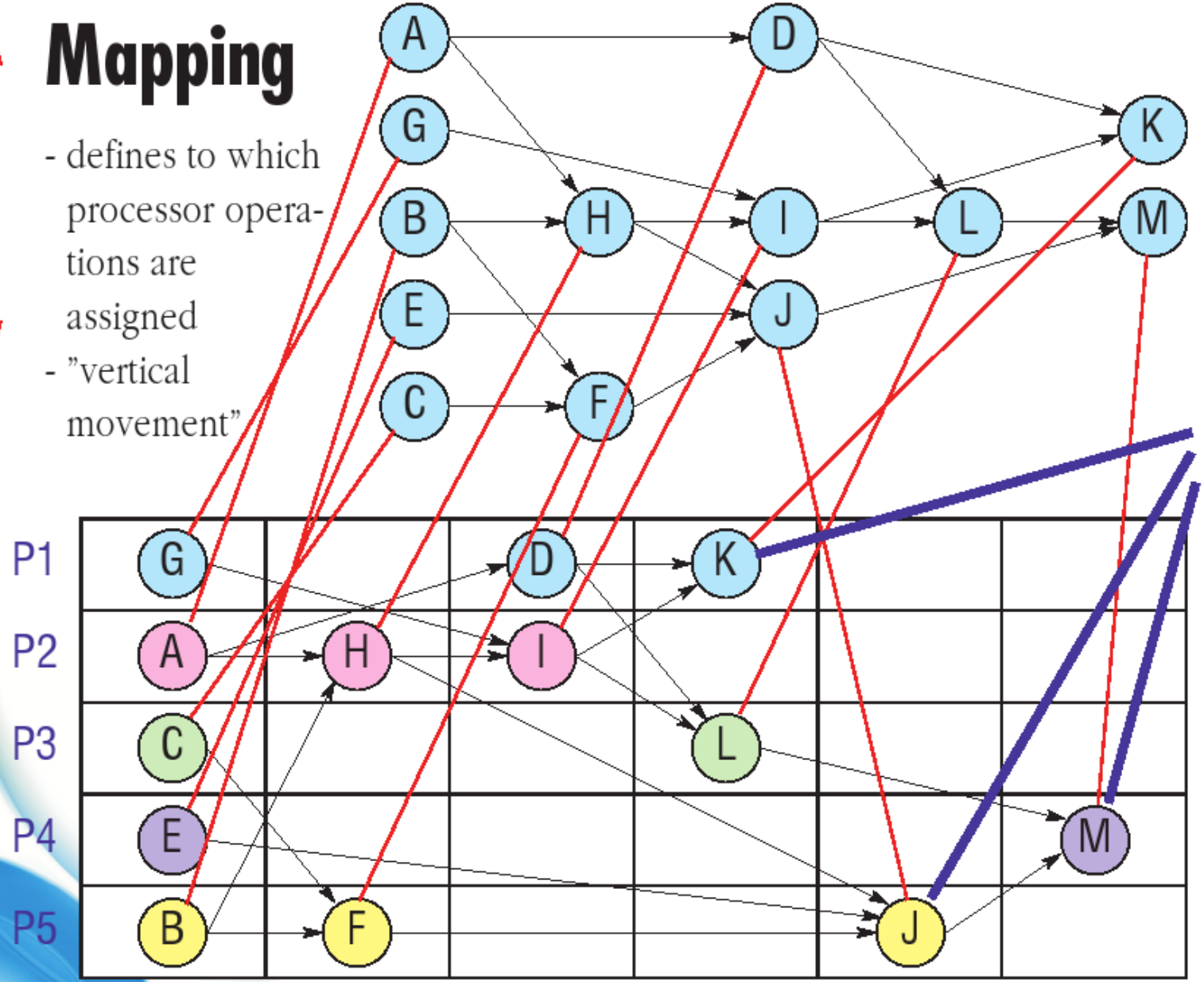
Fastest sequential execution

- T=13, #P=1,W=13

Work optimal = Work is (asymptotically) close to that of the fastest sequential execution

Mapping

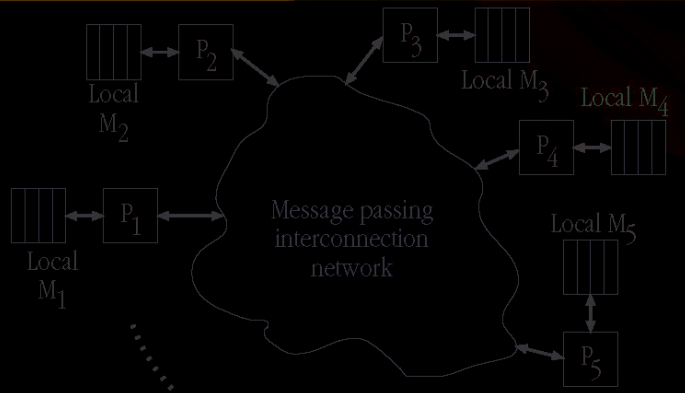
- defines to which processor operations are assigned
- "vertical movement"



Scheduling

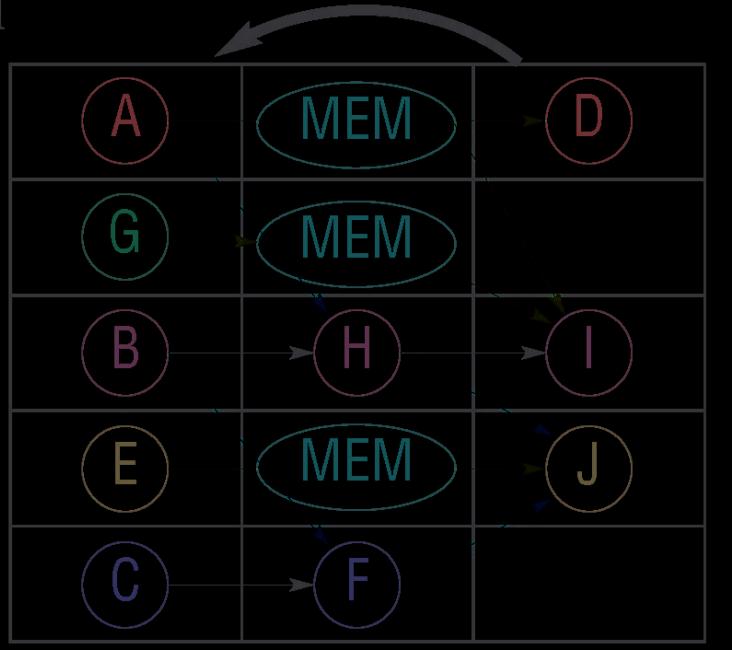
- defines in which order and at which moment operations are executed
- "horizontal movement"

Real world challenges



Executing a functionality is not as simple as above:

- **Memory** is needed to store data while it is not processed
- There can be **multiple references** to the memory **in parallel**
- Intercommunication network introduces **latency**, throughput, **asynchronicity**/non-determinism, **congestions**, dead locks
- Inclusion of **control operations** makes it possible to express a number of functionalities with a single code but makes optimization much more difficult
- Sequential execution provides often **insufficient execution time**, thus we need exploit parallelism



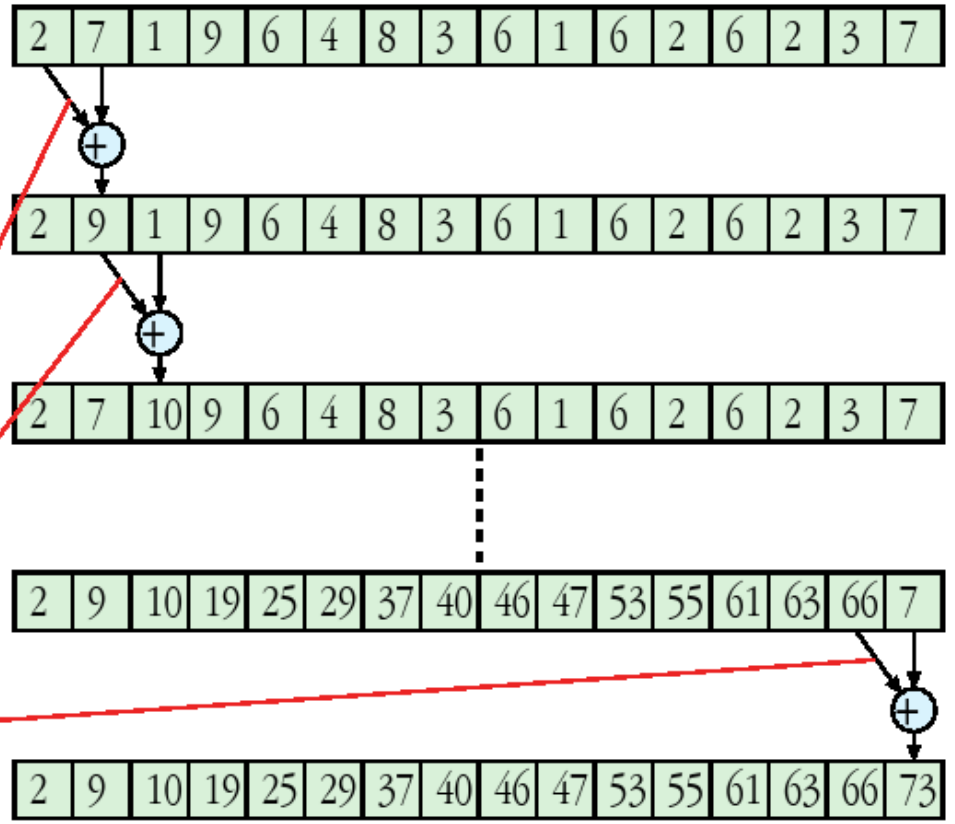
Existence of parallelism

Vast majority of computational problems contains **a lot of parallelism**

(even though some of them **may seem strictly sequential** at the first glance)

Often the parallelism is **fine-grained**, i.e. requires frequent intercommunication, in general purpose computation

Prefix sum computation (N-element vector)

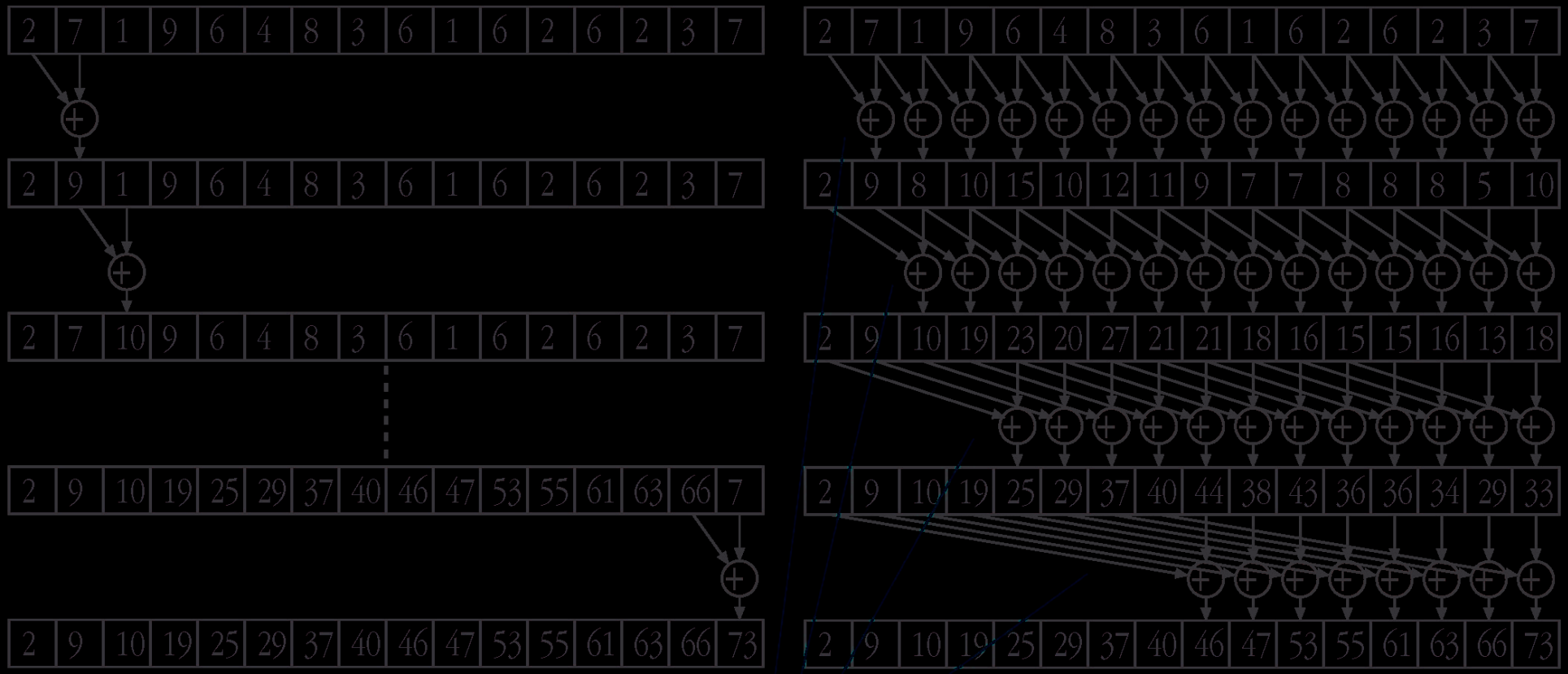


Every iteration is dependent on the previous one

Prefix sum computation—some variants

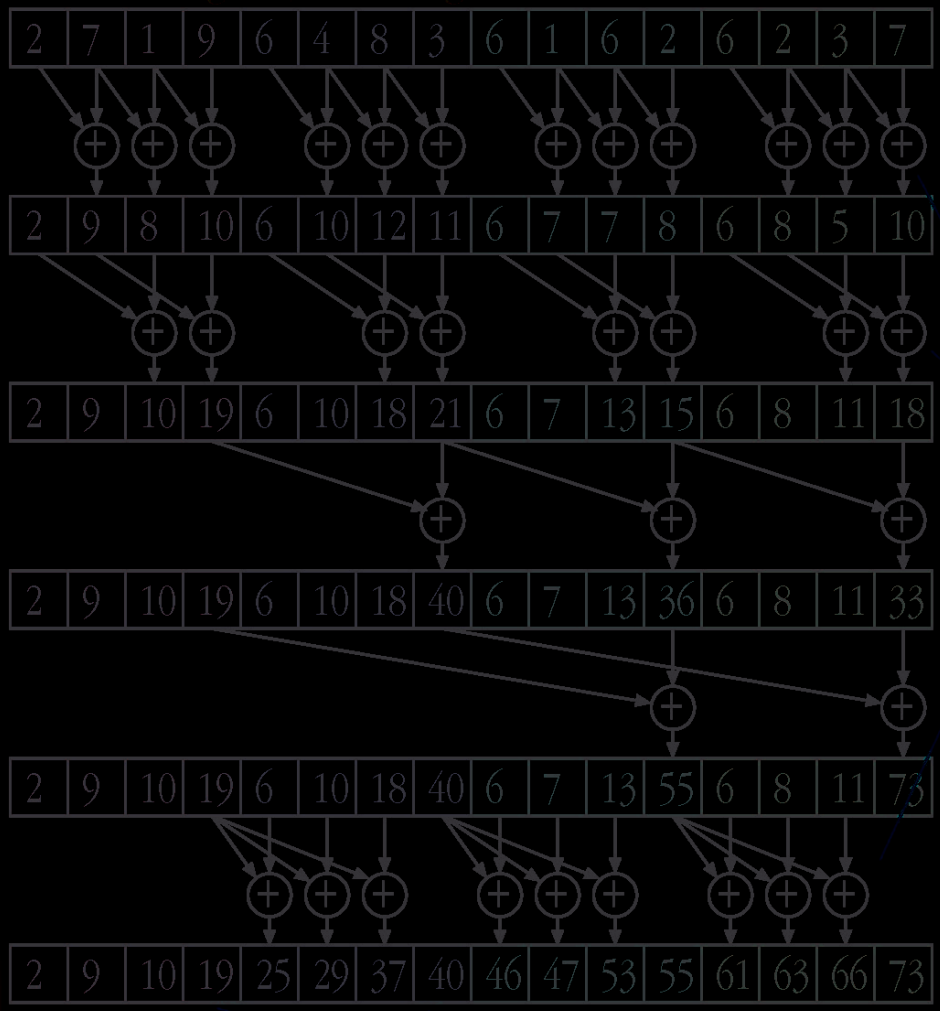
Sequentiquential: P=1, T=Θ(N), W=Θ(N), S=0

Parallel, logarithmic: P=N, T=Θ(log N), W=Θ(N log N), S=Θ(log N)

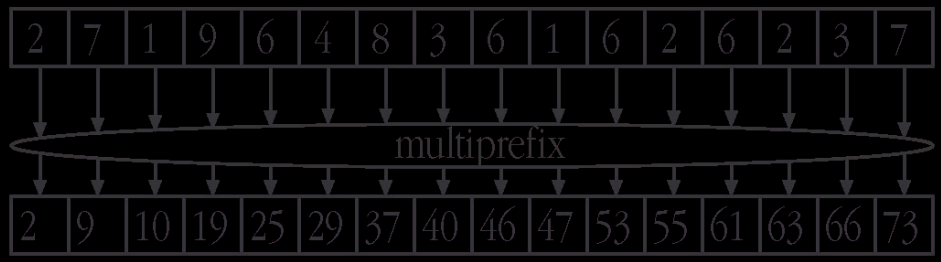


Operations at each iteration must be executed synchronously!

Parallel, blocking P=N, T=O(log N), W=O(N log N), S=O(log N)



Parallel, constant time, P=N, T=O(1), W=O(N)



If blockwise computations are executed sequentially in a single processor, we get:

Parallel, blocking P=B≤N, T=O(N/P + log P), W≈O(N), S=O(log P)

<Home work>

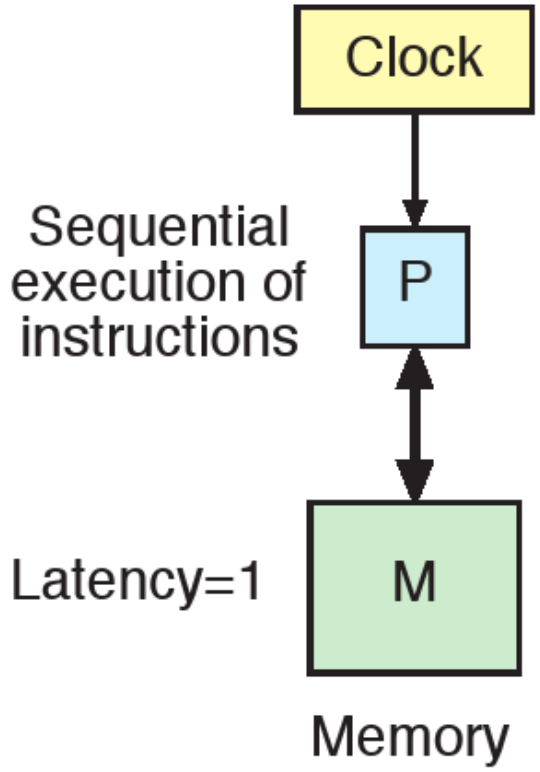
Part 2. Programmability

Why sequential computing has been a tremendous success?
Why current CMPs are hard to program for GP applications?
Programmability vs. current trends
Our receipt to avoid most of these problems
PRAM
A true horror story—how the first attempt can lead to a disaster
Application-specific approach (trading generality for optimality)
Available architectural approaches for general purpose CMPs, MP-SOCs, NOCs
Evaluating approaches with a parametric benchmark
Code size
Is the problem with architectures or programming paradigms/tools?

Why sequential computing has been a tremendous success? —Right type of an abstraction: RAM

- Captures the **essential properties** of the underlying machine
- **Good portability** of programs between machines with substantially different properties
- **Idealized properties** of the computational model, e.g.
  - access time = 1 cycle
  - sequential operation

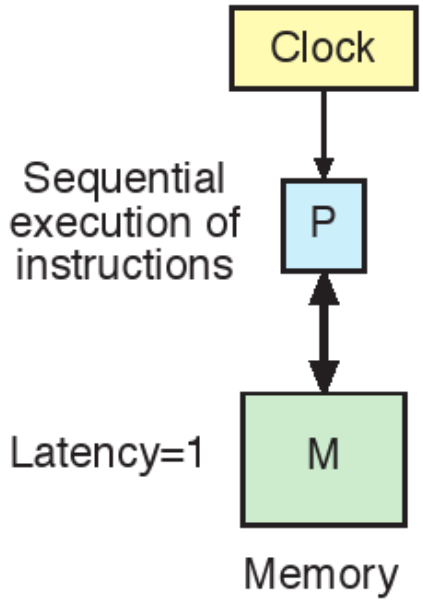
  can be **emulated well** enough even with a speculative superscalar architecture with virtual memory
- **Easy to** learn and **use**
- **Theory of algorithms exists** and helps programming & analysis
- **Efficient teaching** (all universities and schools teach it)



Why sequential computing has been a tremendous success? —Synchronization, techniques, locality

Synchronization

- **Subtasks** of originally parallel computational problems are **executed synchronously** due to deterministically sequential execution at the program level
- Looping is used to **process multiple** data **elements**
- Multitasking is used to **emulate parallelism** (problems are introduced, atomic operations, lockings etc. needed)



Trivial concept of locality that is in line with memory hierarchies

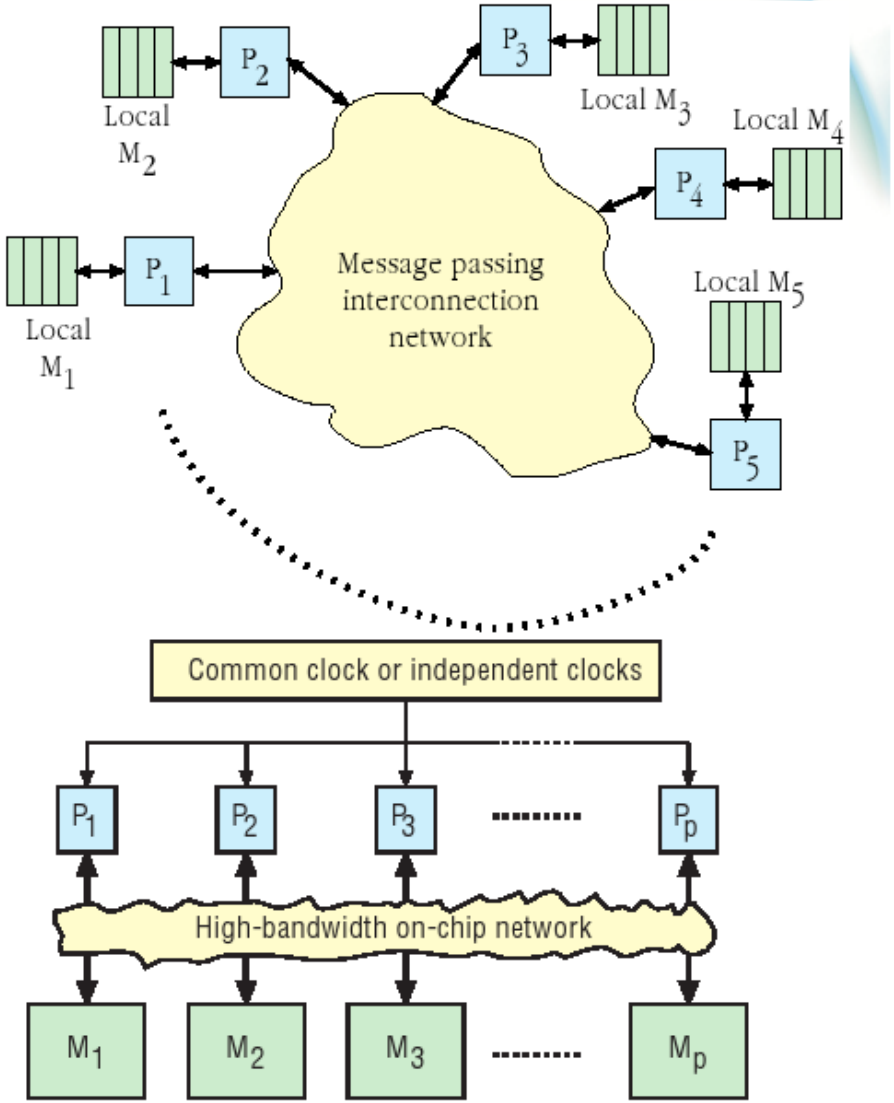
Performance enhancement techniques

- Static or dynamic **superscalar execution**
- Various types of **speculations**
- Memory hierarchies relying on **caching**

Why current CMPs are hard to program for GP applications—Abstraction

Inefficient, way too low and **inapropriate abstraction** of the underlying machine

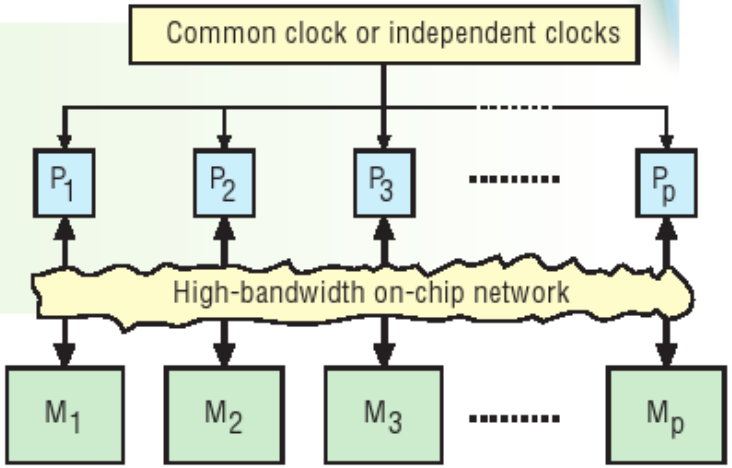
- A **programmer** needs to **take care of** low-level communication, mapping, partitioning, and synchronization
- **Portability** between machines with different properties is often **limited** (rewriting of software may be needed if machine changes/is updated)
- **Difficult to** learn and **use, error-prone**
- **No** general **theory of** parallel **algorithms**
- **Teaching** virtually nonexistent (some advanced level courses exist in universities)



Why current CMPs are hard to program for GP applications? —Synchronization and techniques

Synchronization

- **Asynchronous** operation (missing low-level synchronicity)
- High-level **synchronization very expensive** (barriers take typically hundreds of cycles)
- This **limits** greatly the **applicability** and rules out fine-grained parallel algorithms



Performance enhancement techniques (from sequential computing)

- Static or dynamic **superscalar execution**
- Various types of **speculations**
- Memory hierarchies relying on **coherent caching**
- **Inefficient implementation** or missing support for important **primitives** of parallel computing, e.g. exploitation of ILP, concurrent memory access, multioperations

Why current CMPs are hard to program for GP applications? —Locality

Most experts agree that **locality** is needed **to perform** parallel computation **efficiently**.

Locality = area/neighborhood in which certain special technique is possible

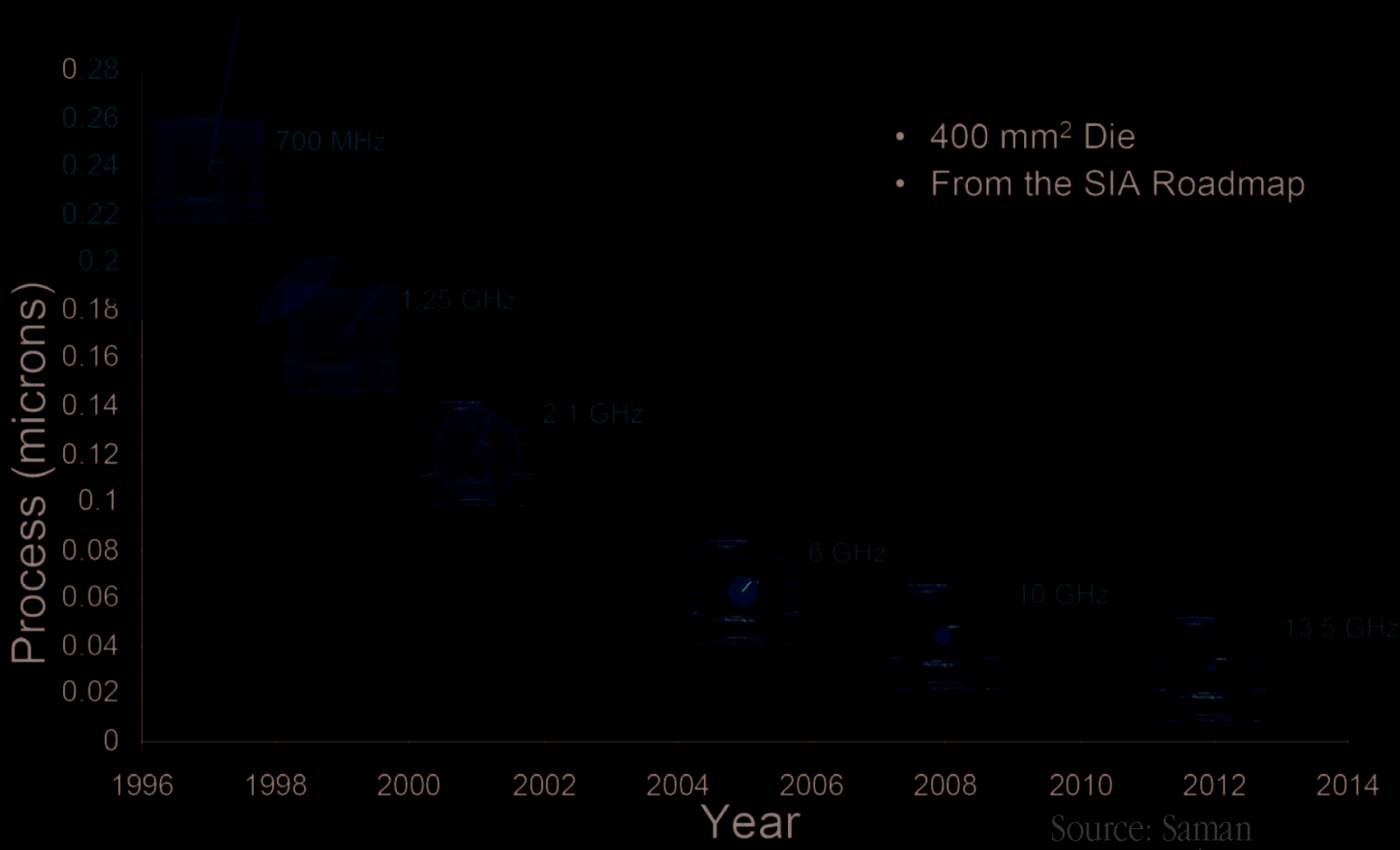
Problems: - The concept of locality is too narrowly understood as a locality to a single processor
- There is no way to maximize data locality with respect to processor cores in general case

In fact, there are different kinds of (or hierarchy of) localities including e.g.:

- VLSI locality (range of a wire in one clock cycle)
- Synchronous area in VLSI design (beyond which one must use the GALS paradigm)
- Chip locality (everything could be put onto a single chip)
- Processor core locality (locality to a processor core)
- Synchronous shared memory locality (beyond which one must use asynchronous SM)
- Machine locality (everything could be put onto a single machine)
- Grid locality (everything fits into a grid)

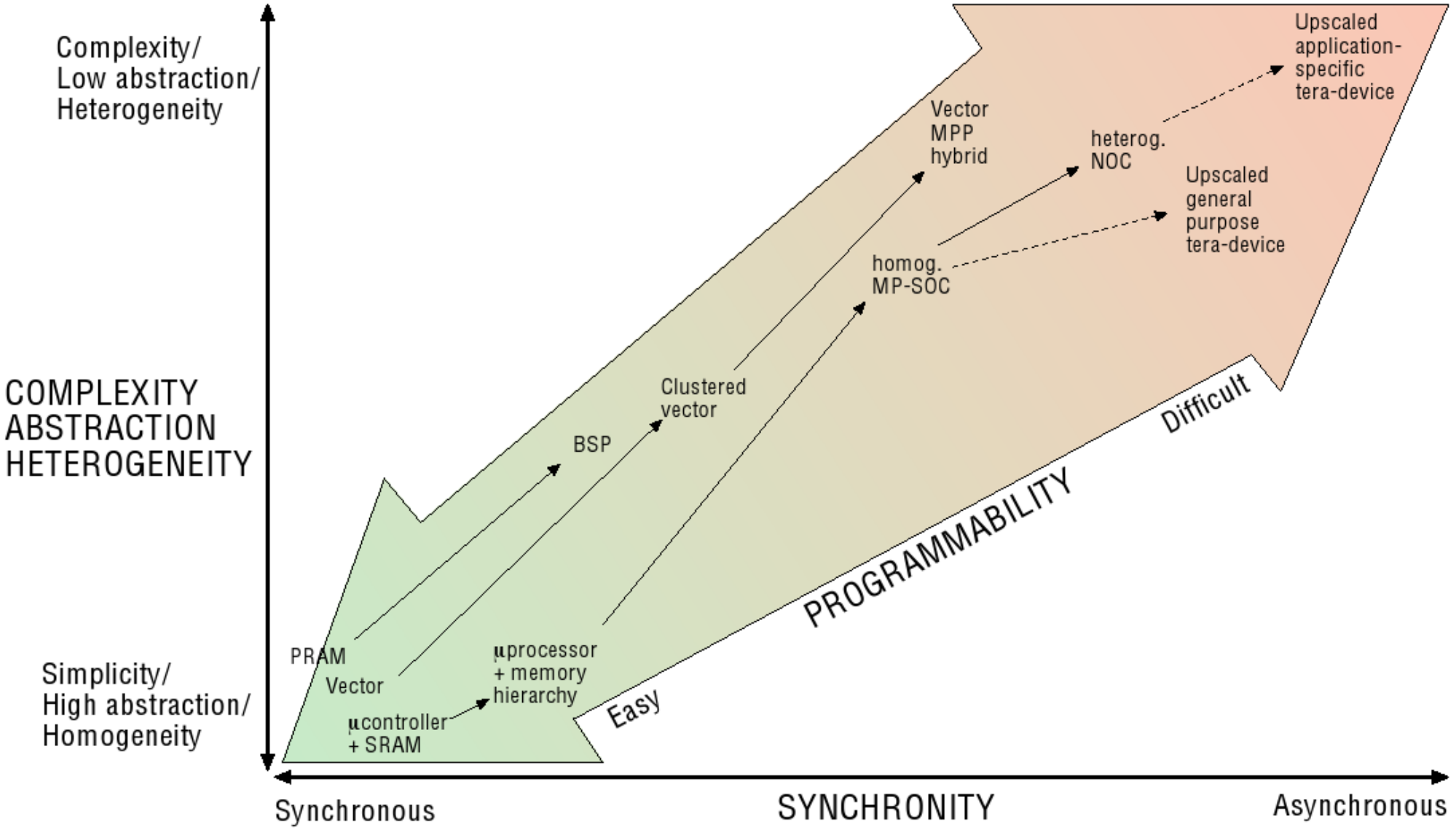
VLSI locality vs. chip locality

Interestingly, these **localities also tend to change** both absolutely and with respect to each others (sometimes quite dramatically) as time goes on.



Source: Saman Amarasinghe, MIT, 2007

Programmability vs. current trends



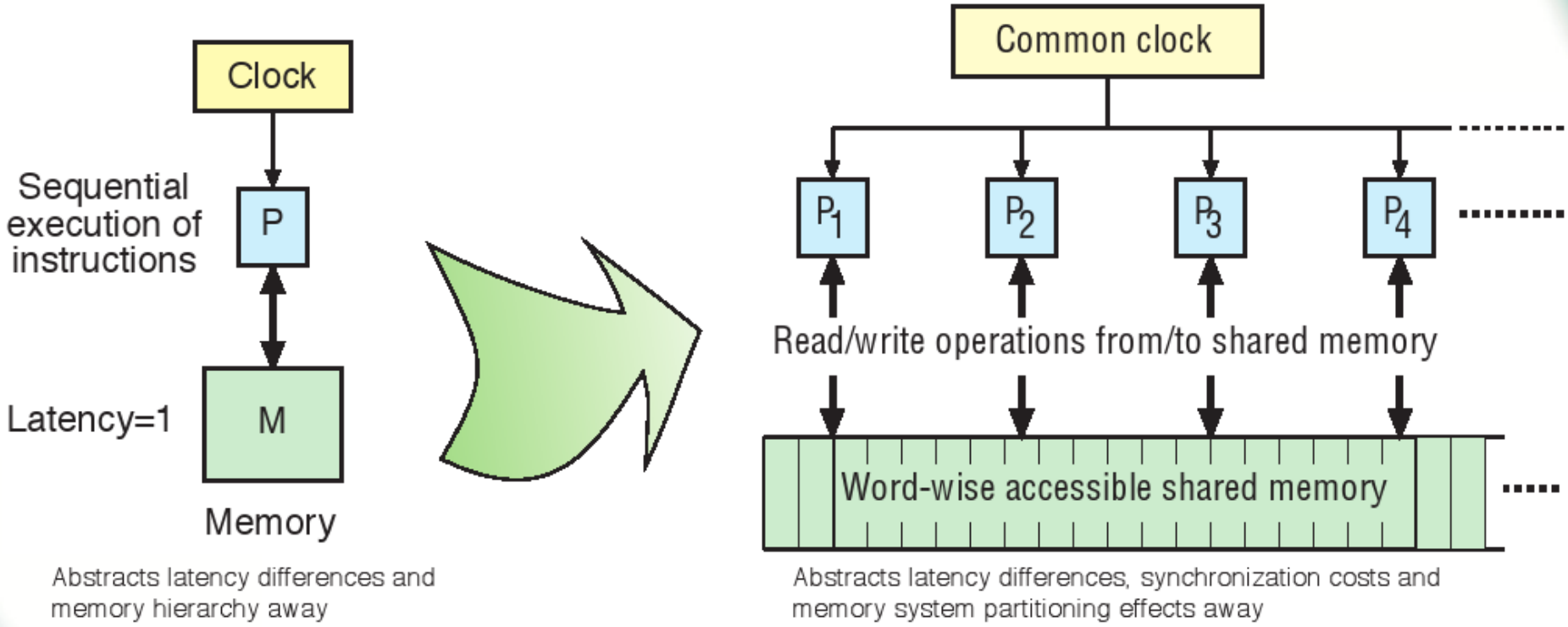
Our receipt to avoid most of these problems

High-communication-bandwidth HW architecture supporting

- Proper TLP **synchronization** (exploit the **synchronous shared memory locality**)
- Correct **abstraction** (synchronous shared memory machine)
- New kind of **performance enhancement techniques** (not those borrowed from sequential computing), e.g. chaining, concurrent memory access, multioperations
- **Algorithmics** for fine-grained synchronous TLP (exists already [Jaja92, Keller01]!)
- Teaching, further research etc...

Parallel Random Access Machine (PRAM)

—Natural extension of the model of sequential computation

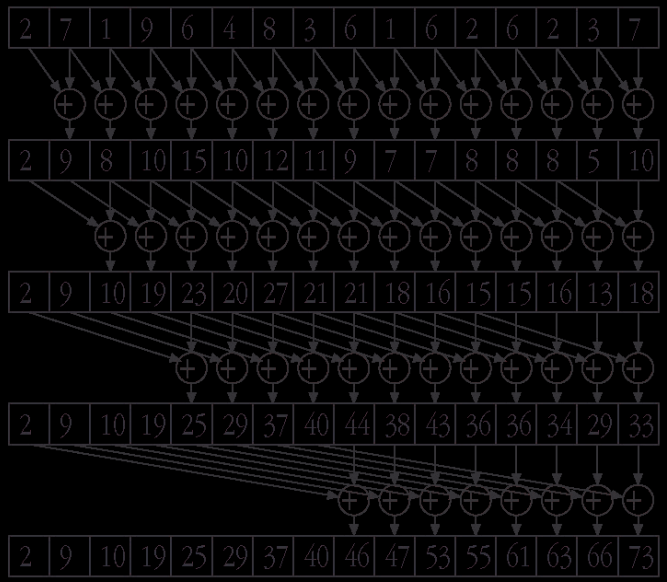


The first model of computation that comes into the mind of a programmer as he starts to think how to solve a computational problem in parallel

— abstracts away latency, synchronization cost and data partitioning effects (like its conterpart)

Does it work? A true horror story—how the first attempt can lead to a disaster

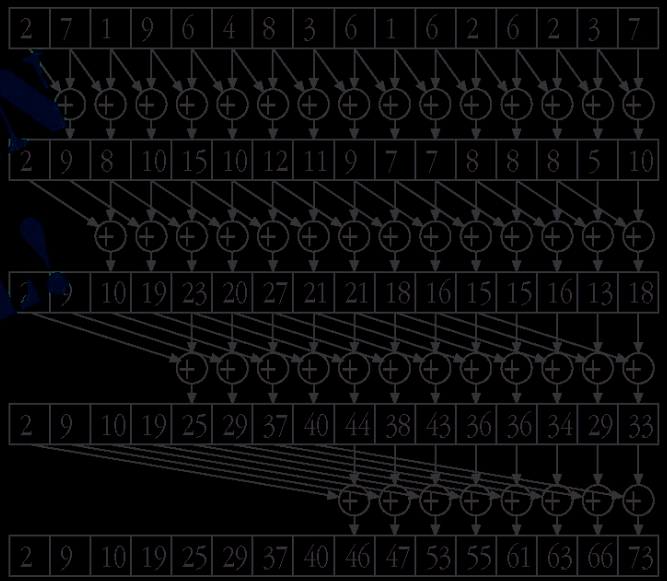
Algorithm: Standard text-book logarithmic prefix sum $P=N$, $T = O(\log N)$
Data set: 16-element array ($N = 16$)
Processor: Intel Core2Duo
Tool: PThreads
Number of threads: 16
Synchronization: Explicit barriers where necessary



Any guesses how we performed?

Does it work? A true horror story—how the first attempt can lead to a disaster

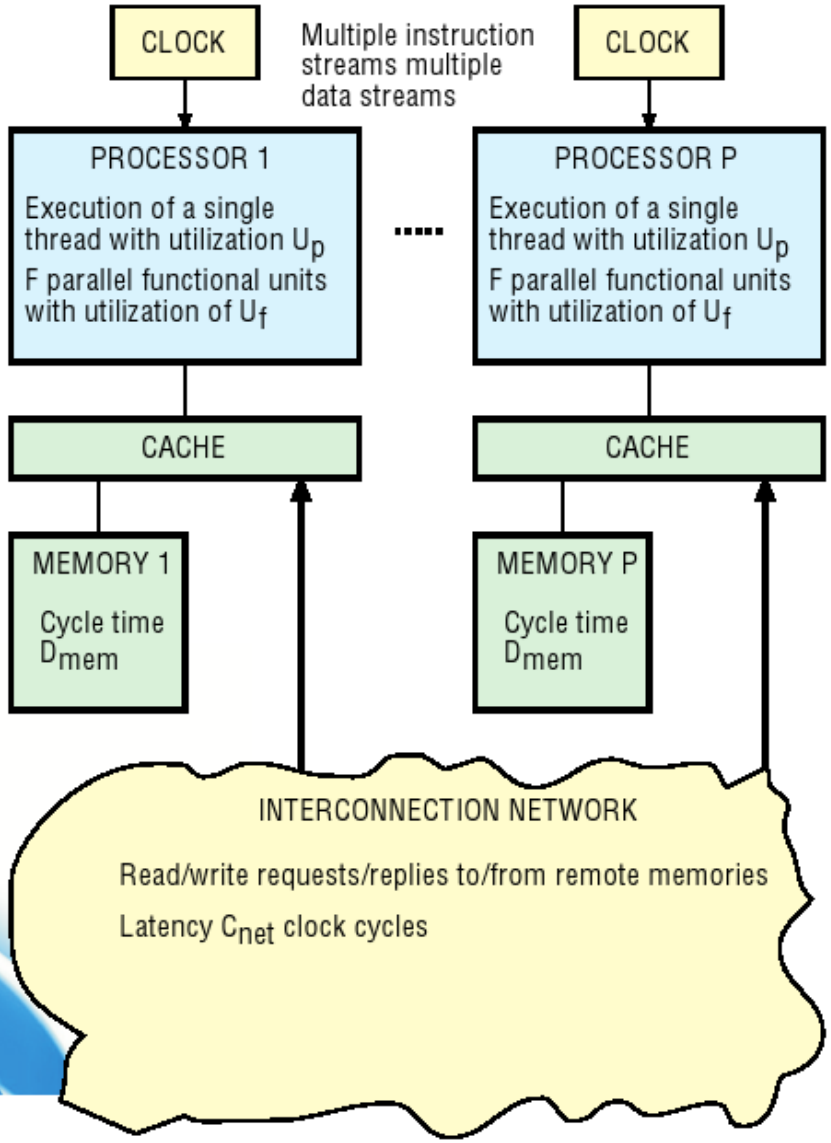
Algorithm: Standard text-book logarithmic prefix sum $P=N$, $T = O(\log N)$
Data set: 16-element array ($N = 16$)
Processor: Intel Core2Duo
Tool: PThreads
Number of threads: 16
Synchronization: Explicit barriers where necessary



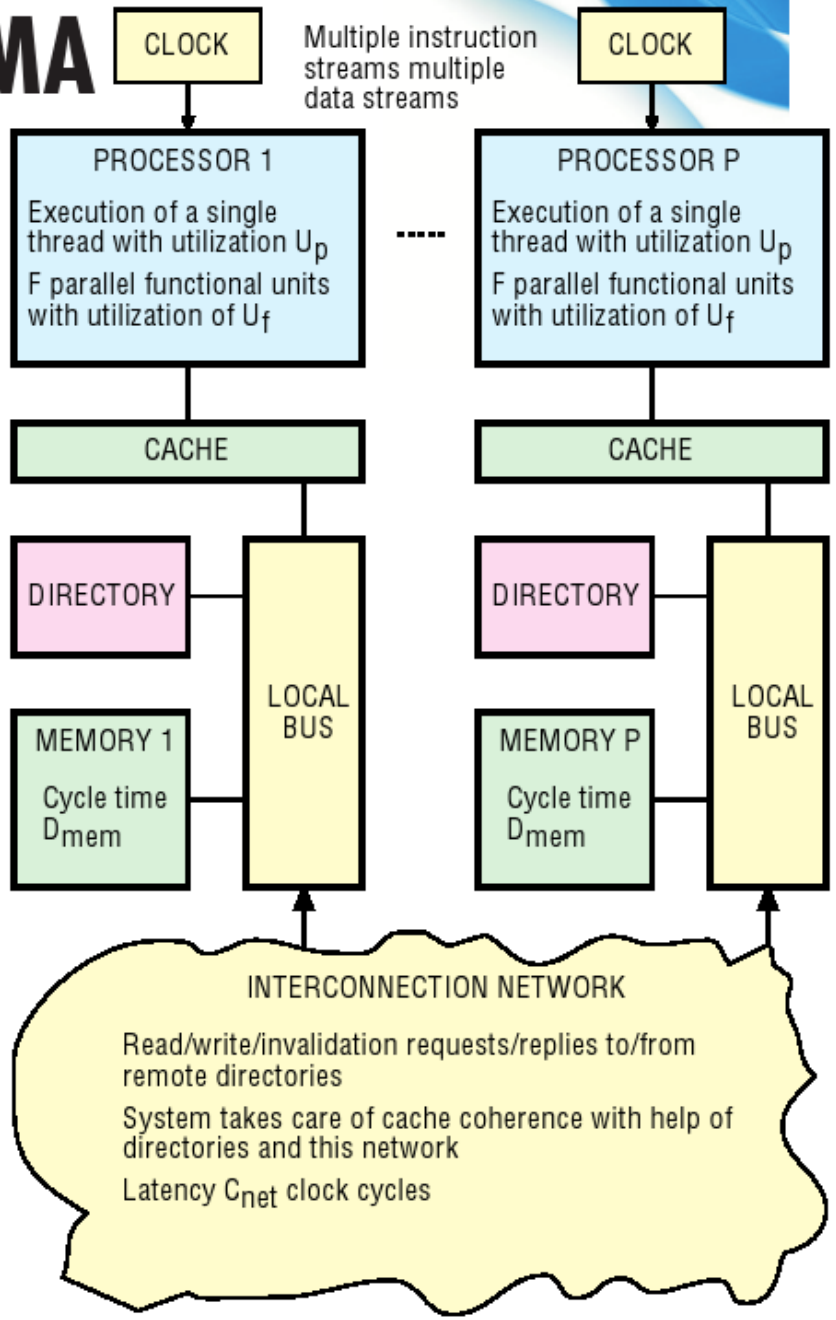
Let's see where is the problem: Available architectural approaches for GP CMPs, MP-SOCs, NOCs

SMP	Symmetric multiprocessor
NUMA	Non-uniform memory access
CC-NUMA	Cache coherent non-uniform memory access
MP	Message passing machine
VC	Vector computer

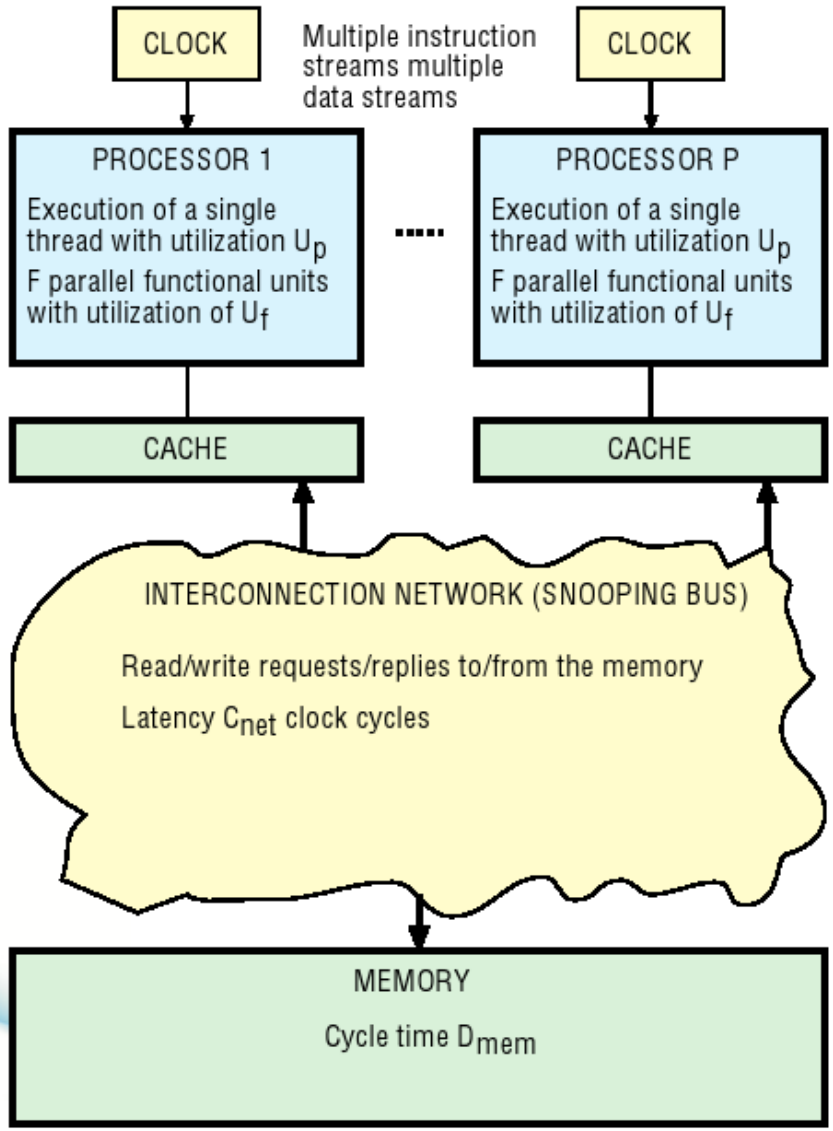
NUMA


CLOCK
Multiple instruction streams multiple data streams
CLOCK
PROCESSOR 1
Execution of a single thread with utilization U_p
F parallel functional units with utilization of U_f
PROCESSOR P
Execution of a single thread with utilization U_p
F parallel functional units with utilization of U_f
CACHE
CACHE
MEMORY 1
Cycle time D_{mem}
MEMORY P
Cycle time D_{mem}
INTERCONNECTION NETWORK
Read/write requests/replies to/from remote memories
Latency C_{net} clock cycles

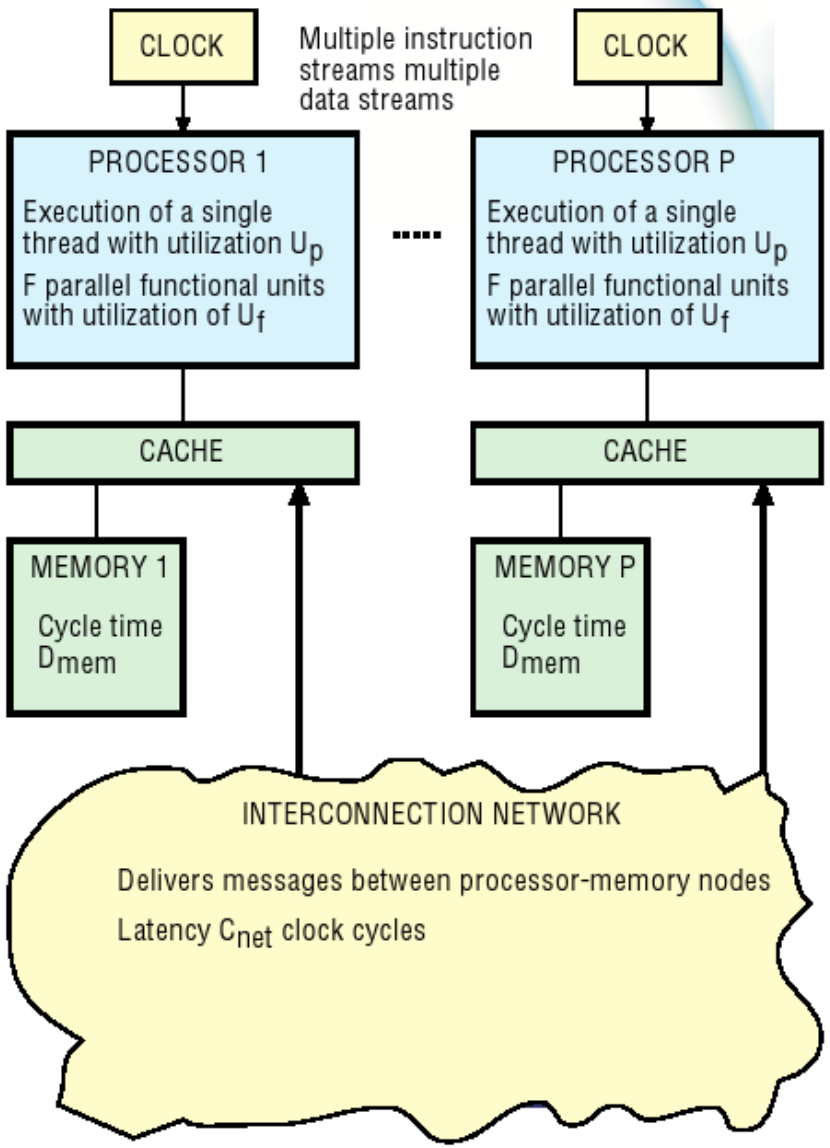
CC-NUMA


CLOCK
Multiple instruction streams multiple data streams
CLOCK
PROCESSOR 1
Execution of a single thread with utilization U_p
F parallel functional units with utilization of U_f
PROCESSOR P
Execution of a single thread with utilization U_p
F parallel functional units with utilization of U_f
CACHE
CACHE
DIRECTORY
DIRECTORY
LOCAL BUS
LOCAL BUS
MEMORY 1
Cycle time D_{mem}
MEMORY P
Cycle time D_{mem}
INTERCONNECTION NETWORK
Read/write/invalidation requests/replies to/from remote directories
System takes care of cache coherence with help of directories and this network
Latency C_{net} clock cycles

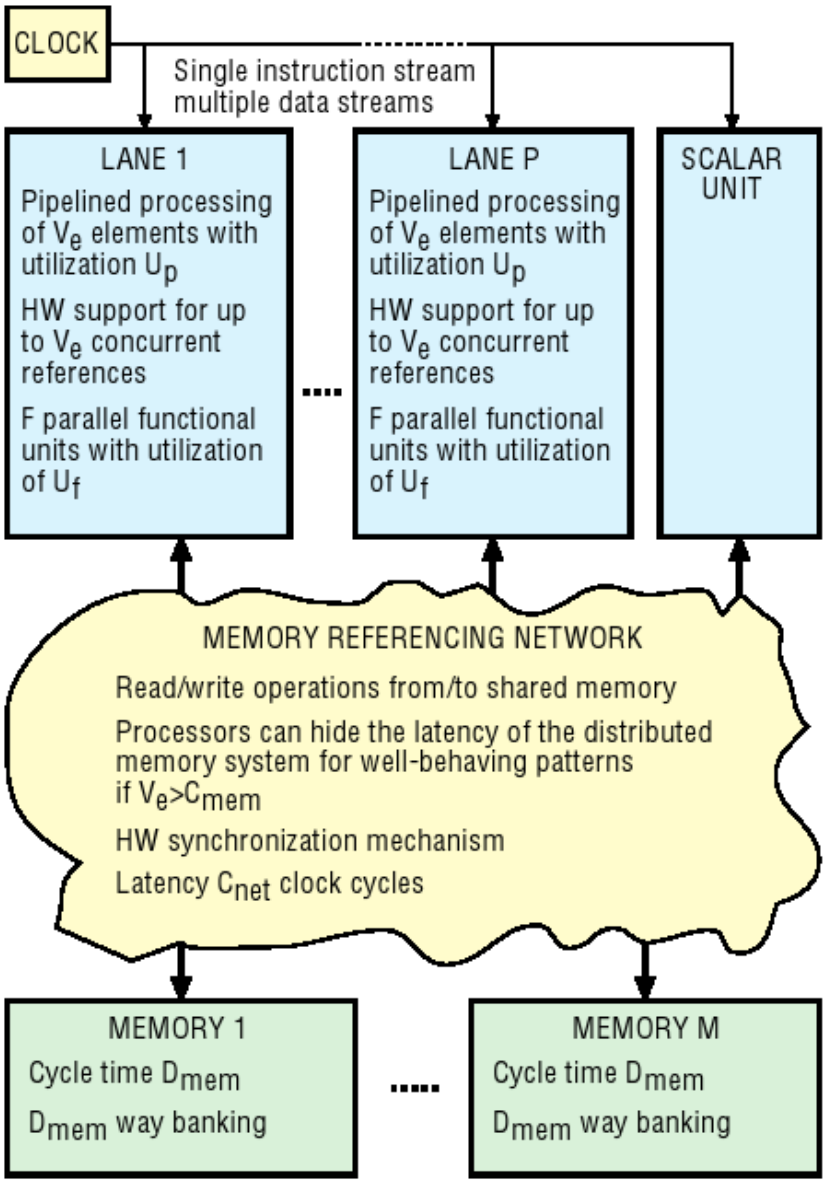
SMP


CLOCK
Multiple instruction streams multiple data streams
CLOCK
PROCESSOR 1
Execution of a single thread with utilization U_p
F parallel functional units with utilization of U_f
PROCESSOR P
Execution of a single thread with utilization U_p
F parallel functional units with utilization of U_f
CACHE
CACHE
INTERCONNECTION NETWORK (SNOOPING BUS)
Read/write requests/replies to/from the memory
Latency C_{net} clock cycles
MEMORY
Cycle time D_{mem}

MP


CLOCK
Multiple instruction streams multiple data streams
CLOCK
PROCESSOR 1
Execution of a single thread with utilization U_p
F parallel functional units with utilization of U_f
PROCESSOR P
Execution of a single thread with utilization U_p
F parallel functional units with utilization of U_f
CACHE
CACHE
MEMORY 1
Cycle time D_{mem}
MEMORY P
Cycle time D_{mem}
INTERCONNECTION NETWORK
Delivers messages between processor-memory nodes
Latency C_{net} clock cycles

VC


CLOCK
Single instruction stream
multiple data streams
LANE 1
Pipelined processing of V_e elements with utilization U_p
HW support for up to V_e concurrent references
F parallel functional units with utilization of U_f
LANE P
Pipelined processing of V_e elements with utilization U_p
HW support for up to V_e concurrent references
F parallel functional units with utilization of U_f
SCALAR UNIT
MEMORY REFERENCING NETWORK
Read/write operations from/to shared memory
Processors can hide the latency of the distributed memory system for well-behaving patterns if $V_e > C_{mem}$
HW synchronization mechanism
Latency C_{net} clock cycles
MEMORY 1
Cycle time D_{mem}
D_{mem} way banking
MEMORY M
Cycle time D_{mem}
D_{mem} way banking

Evaluated programming patterns

TLP programming patterns:

- Sequential part of a program
- Independent parallel part of a program
- Dependent parallel part of a program
- Splitting between sequential and parallel parts
- Joining between parallel and sequential parts
- Exclusively accessible non-local shared data
- Concurrently accessible non-local shared data
- Functionally dependent data placement (i.e. locality)
- Barrier synchronization

Evaluated parameters (parametric application, parametric architectures)

Determine execution time of a Bench making use of these patterns as a function of

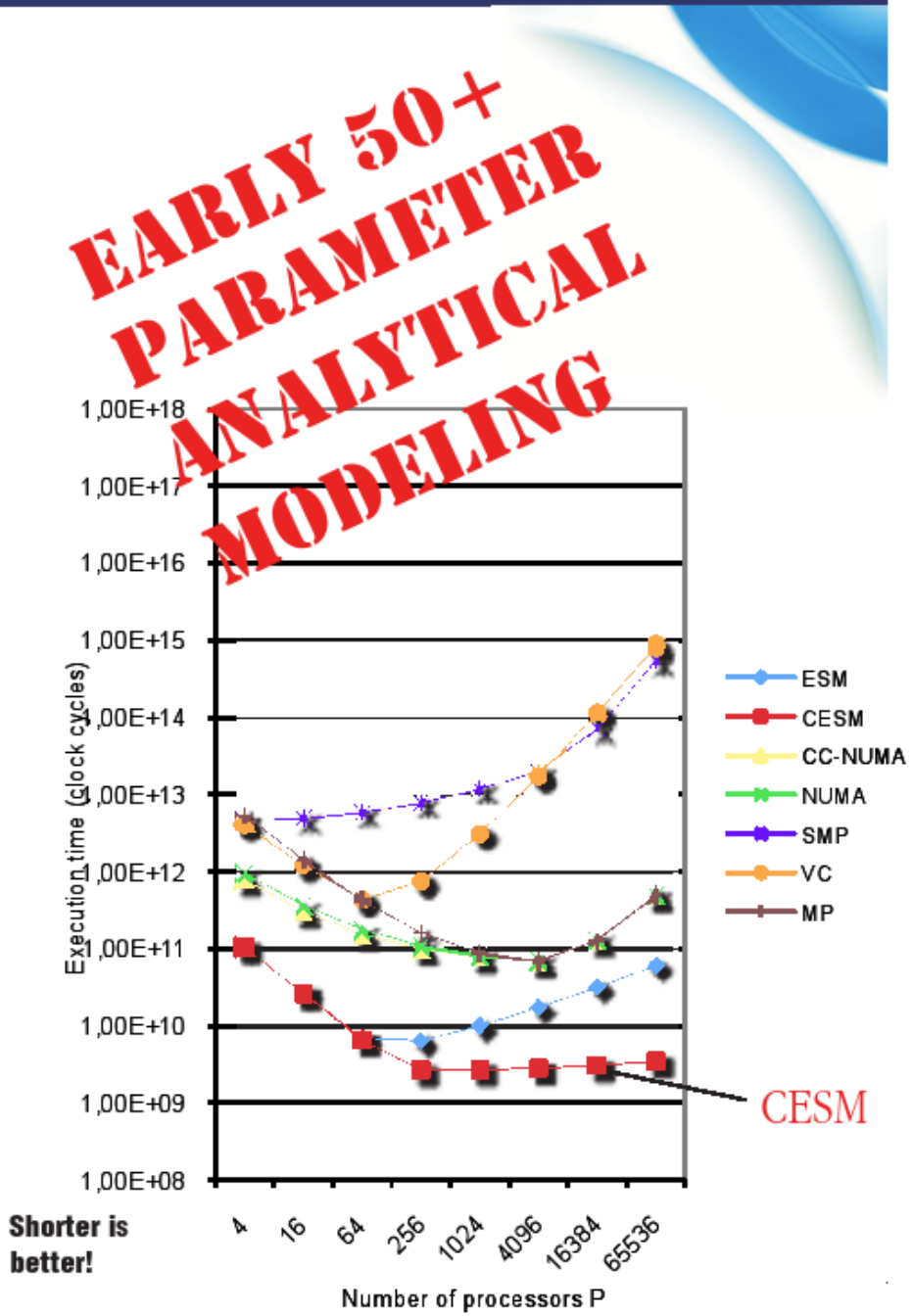
- Fraction of sequential portions F_s
- Fraction of dependent parallel portions F_{dp}
- Number of processors P
- Number of functional units F
- Number of threads in Bench T_b
- Probability that a memory reference is targeted to the global memory P_g
- Probability of a cache miss P_{cm}
- Fraction of non-vectorizable code F_{nv}
- Probability that a memory reference in a dependent parallel portion is targeted to concurrently accessible shared data P_{csd}

Evaluating approaches with a parametric benchmark 1

Execution time in the case of

- 10% of the code (of parallel version) is sequential
- 90% of the code is parallel of which
  - 50% independent
  - 50% fully dependent

- Number of processors 4 .. 65536
- Number of SW threads 8192
- High-bandwidth 2D-multi-mesh interconnect

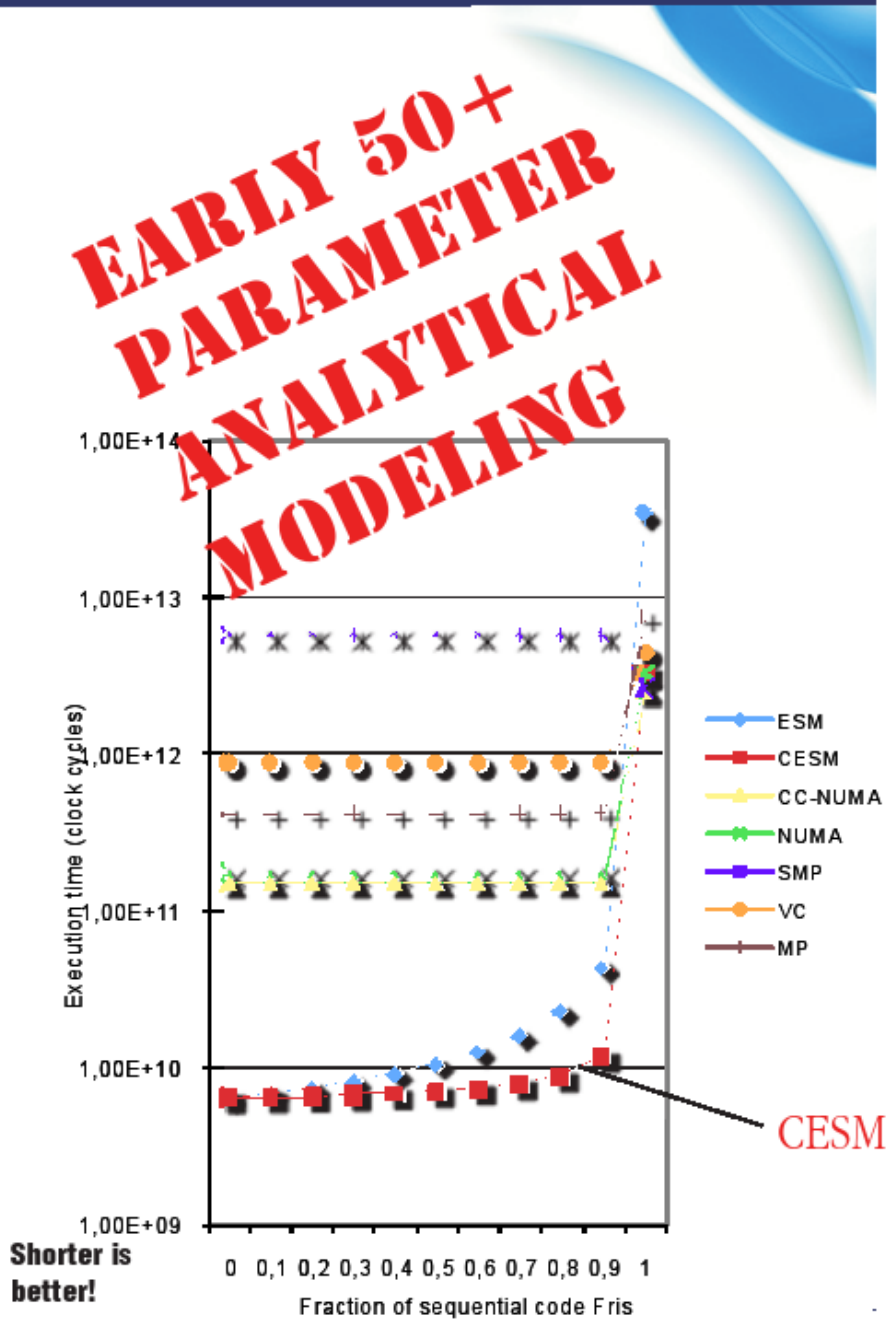


Evaluating approaches with a parametric benchmark 2

Execution time in the case of

- 0-100% of the code (of parallel version) is sequential
- Rest of the code is parallel of which
  - 50% independent
  - 50% fully dependent

- Number of SW threads 8192
- High-bandwidth 2D-multi-mesh interconnect

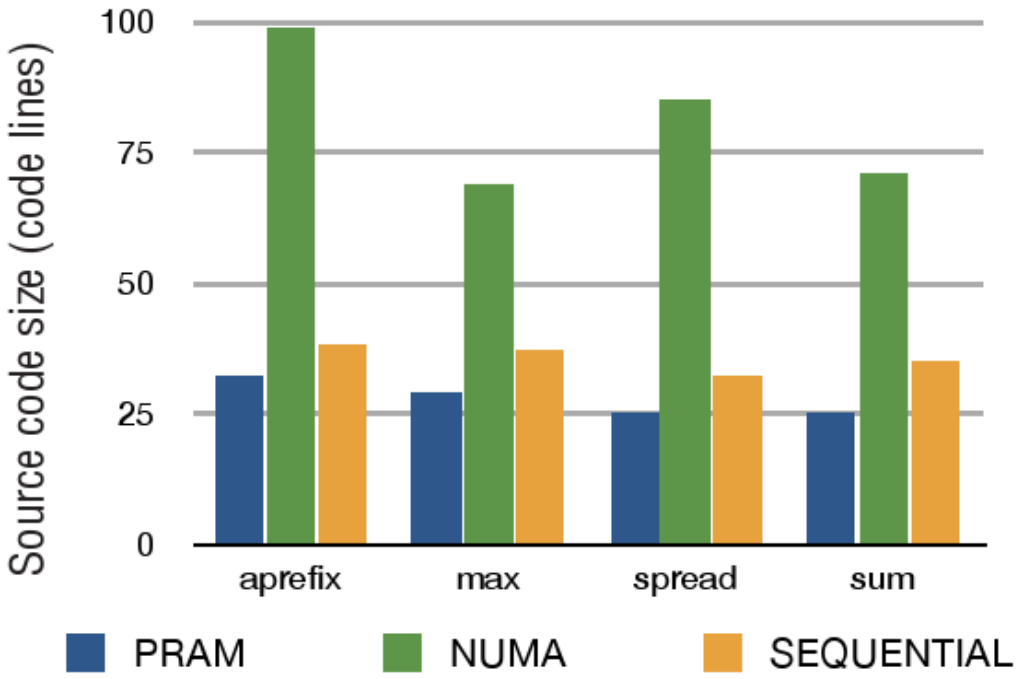


Code size [Forsell11b]

Size of the source code reflects the relative **complexity of programming** (abstracted shared memory machine aka PRAM vs. current NUMA vs. SEQUENTIAL)

Extra code lines are needed due to

- explicit synchronizations
- processing as blocks to maximize locality and minimize the need for costly synchronizations



Languages: C, e

Is the problem with architectures or programming paradigms/tools?

Virtually all current solutions define **the same computability** as the strongest theoretical models (can simulate each others at some, not necessarily linearly slowed, execution rate)

=> the problem (with parallel computing) is mainly architectural

Implications:

- Current multi-core machines are not efficient enough in executing certain important patterns of parallel processing
- Using the best practicies and most efficient compiling tools does not help much since architectures are weak

Part 3. Architectural support for parallelism on multi-core machines

Application-specific vs. general purpose approach
Hiding the latency: throughput computing
Fast orchestration of threads: wave based synchronization
Avoiding pipeline hazards
Changing the organization of FUs
Additional architectural techniques
Concurrent memory access
Multioperations

Special themes:
Why distributed shared memory rather than unified?
Balancing the speed difference between processors and memories
Congestion avoidance with hashing

Application-specific vs. general purpose approach (trading optimality for generality)

+ Performance/area, **performance/power**
+ **Natural partitioning**

→ **Matching applications**

- **Partitioning problems**
- **Inflexibility**, waste of area (and power)

→ **Non-matching applications**

- **Methodological complexity**

  $APP_1+APP_2+...+APP_N$ = *N* **x** methodology!

- *N* **x** risk of **software rewrite** due to architectural changes
- Reuse-based design complexity reduction may **not** be **as efficient** as replication-based

General purpose
Homogeneity
Programmability
Flexitility
Replication

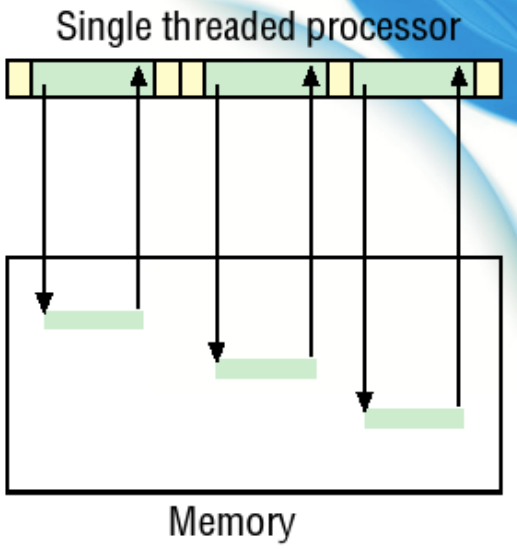
Application-specific
Heterogeneity
Optimality for fixed applications
Reuse of existing blocks

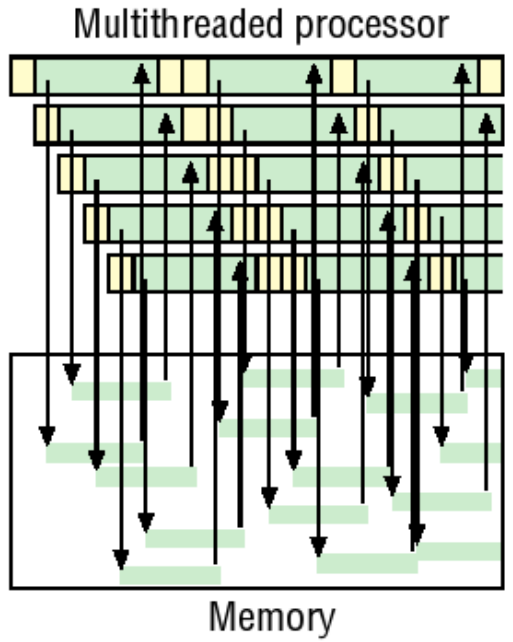
Hiding the latency: throughput computing

A special scheme of parallel computing focusing in

- Increasing the throughput or **utilization of the parallel execution** machinery rather than trying to maximize the clock frequency
- Hiding the latency of the memory system with **slack of parallel applications** rather than by employing more complex cache coherence mechanisms

Currently being **utilized** e.g. **in** Sun Microsystems **Ultra Sparc Tx series** of processors combining chip multiprocessors (CMP) and chip multithreading (CMT).

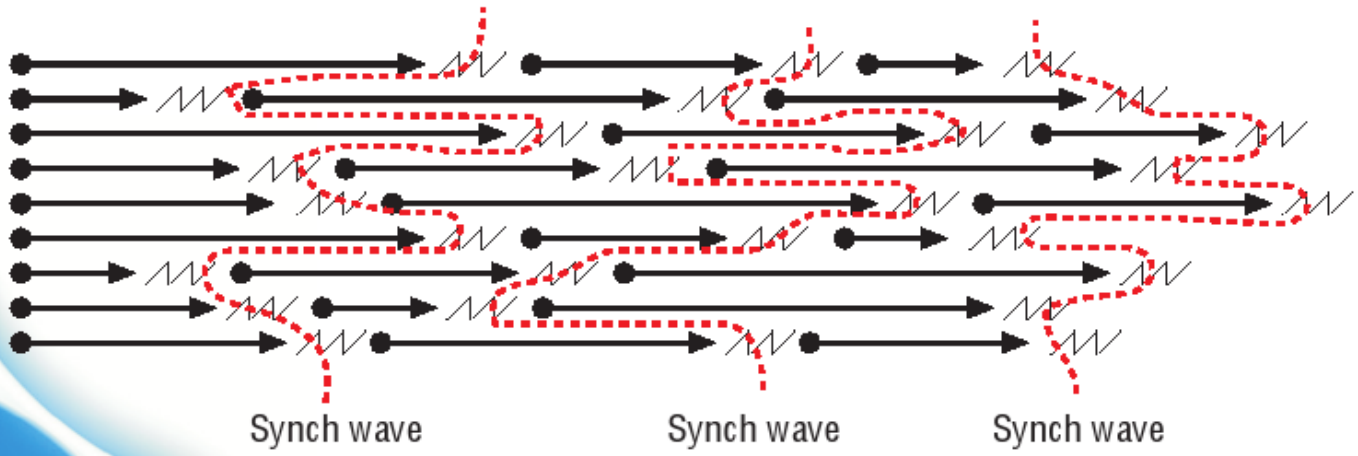




Fast orchestration of threads: wave based synchronization

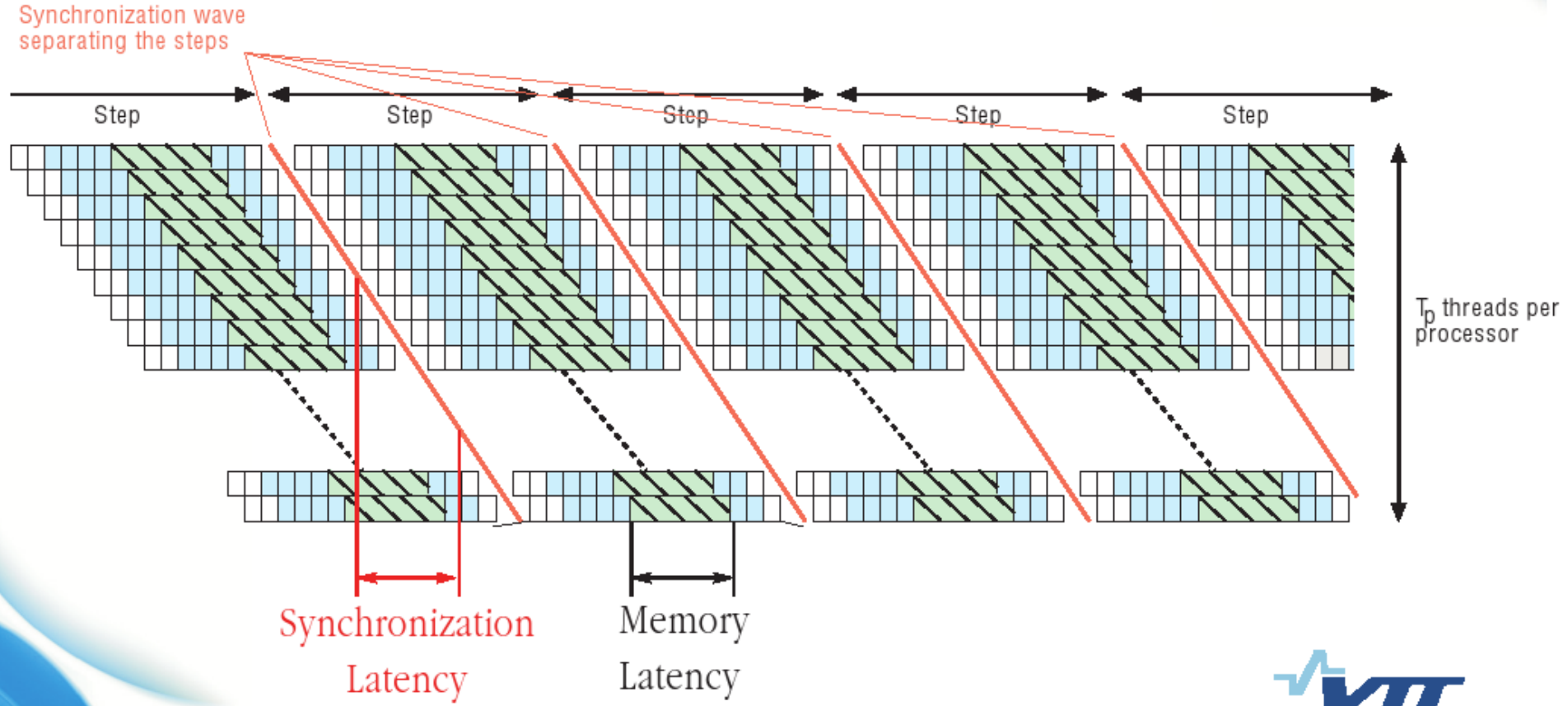
A low-cost architectural technique for maintaining synchrony:

- An elastic wave-like front of synchronization messages separating memory references belonging to consecutive multithreaded steps

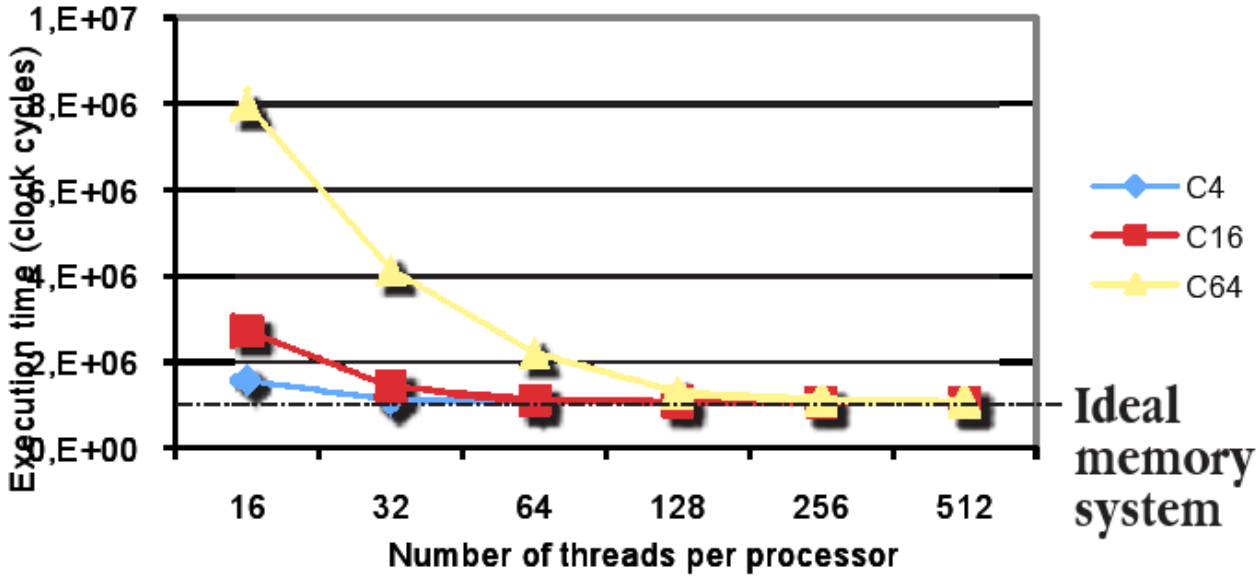


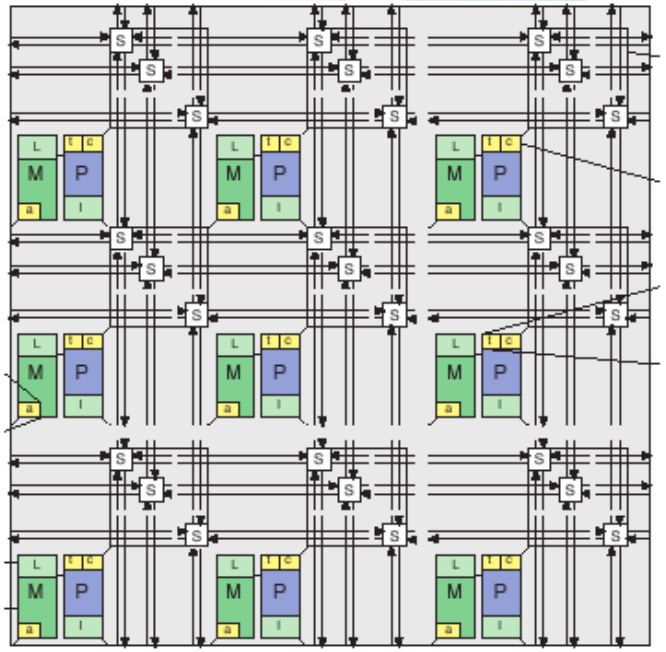
Synchronization wave + throughput computing

By combining wave synchronization with throughput computing we can **drop the cost** of synchronization **from 100 down to 1/100** (more formally speaking to $1/T_p$).



Evaluation results—Latency hiding sparse mesh interconnect





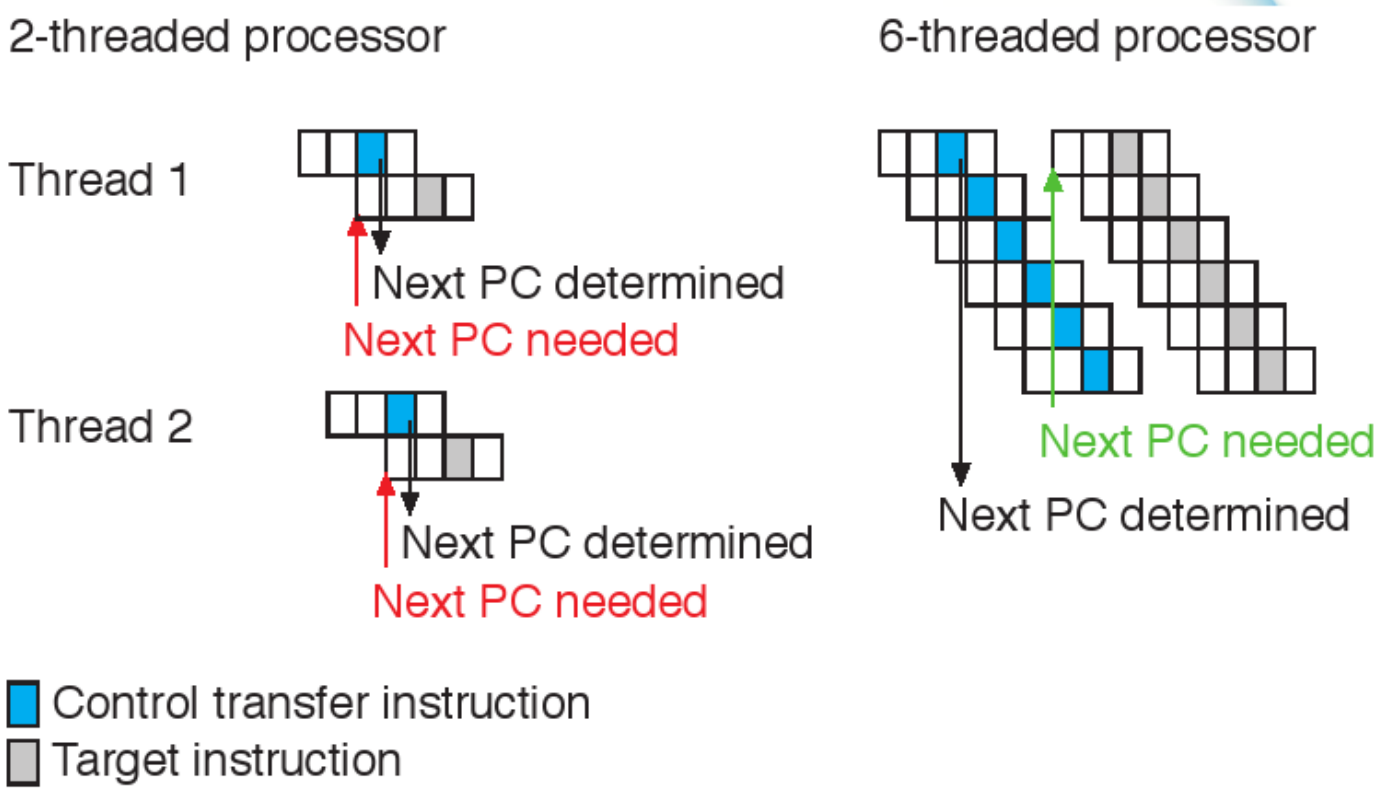
Execution time vs. the number of threads

-- Combination of dense (reference per every clock cycle) exclusive and concurrent memory access patterns randomly selected from SPEC CPU 2000 traces.

The amount of executed references is constant per thread, thus C64 has 4 times more data to process than C16 and 16 times more than C4.

Avoiding pipeline hazards

If the number of threads per processor is large enough, one can **eliminate pipeline hazards** (and the need for complex branch prediction mechanisms).



This is possible if

- The number of threads in the application ≥ the number of threads in architecture
- The number of threads in architecture ≥ ordinal of sequencer stage

Further performance increase?

Can we use

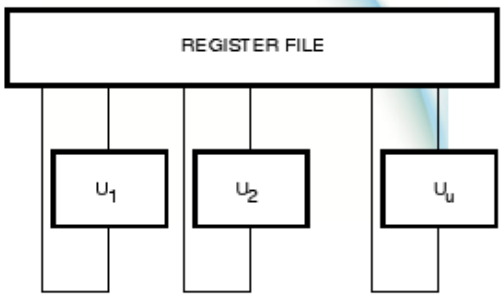
- parallel **slackness**
- **throughput computing** scheme

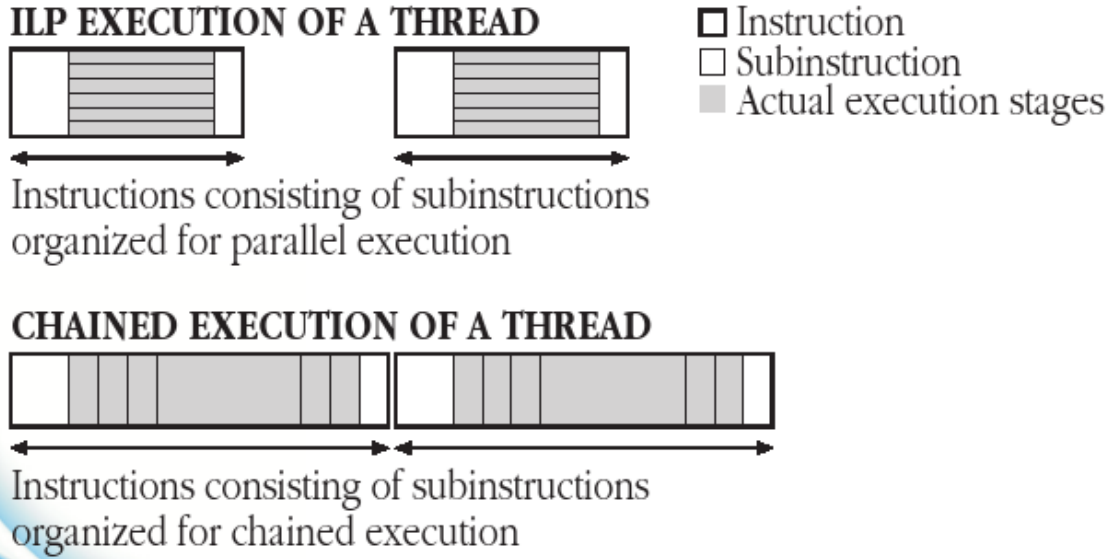
also to **improve the utilization of ILP**?

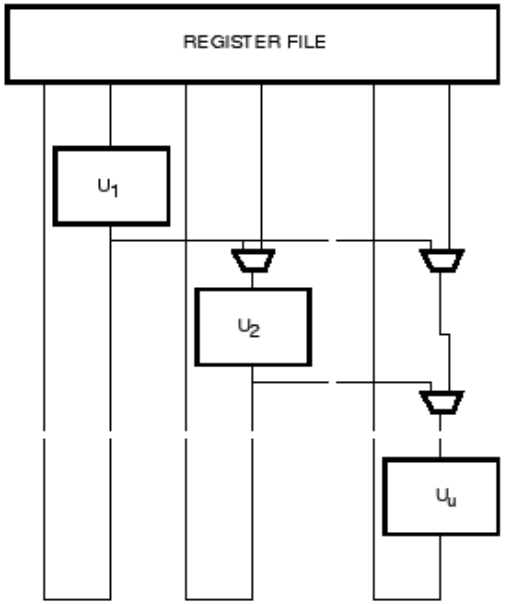
Yes, we can!

Changing the organization of FUs

Reorganize the computation of subinstructions in VLIW (or simultaneously executed instructions in superscalar processor) to happen **sequentially in a chain** rather than in parallel!







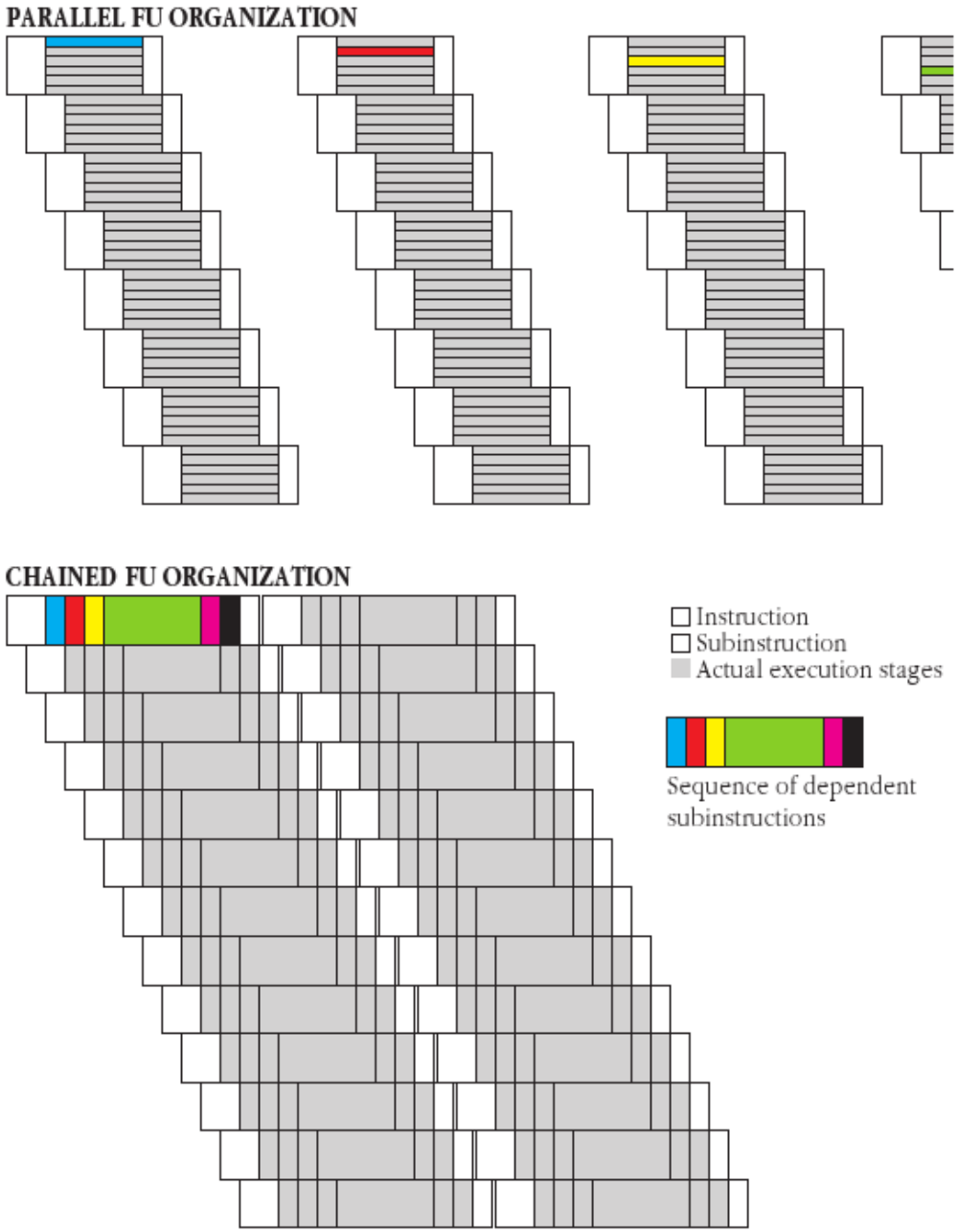
Ordering of FUs

The **ordering of** the chained **FUs** should be selected so that it reflects the ordering of a **typical basic block** of code:

- 2/3 of ALUs (includes address calculation)
- Memory unit
- 1/3 of ALUs
- Compare unit (often in the end)
- Sequencer (always in the end)

Baseline	A, M, C, S
T5	A→M→C→S
T7	A→A→M→A→C→S
T11	A→A→A→A→A→M→A→A→C→S

It is **possible to extend** this technique to **multi-MU machines** per processor but then some of the advantages are lost.



Evaluation results—Chaining [Forsell03]

Does chaining **work in practise**?
What kind of **speedups** can be **expected**?
How to compile for it?
Here some evaluation results [Forsell03]:

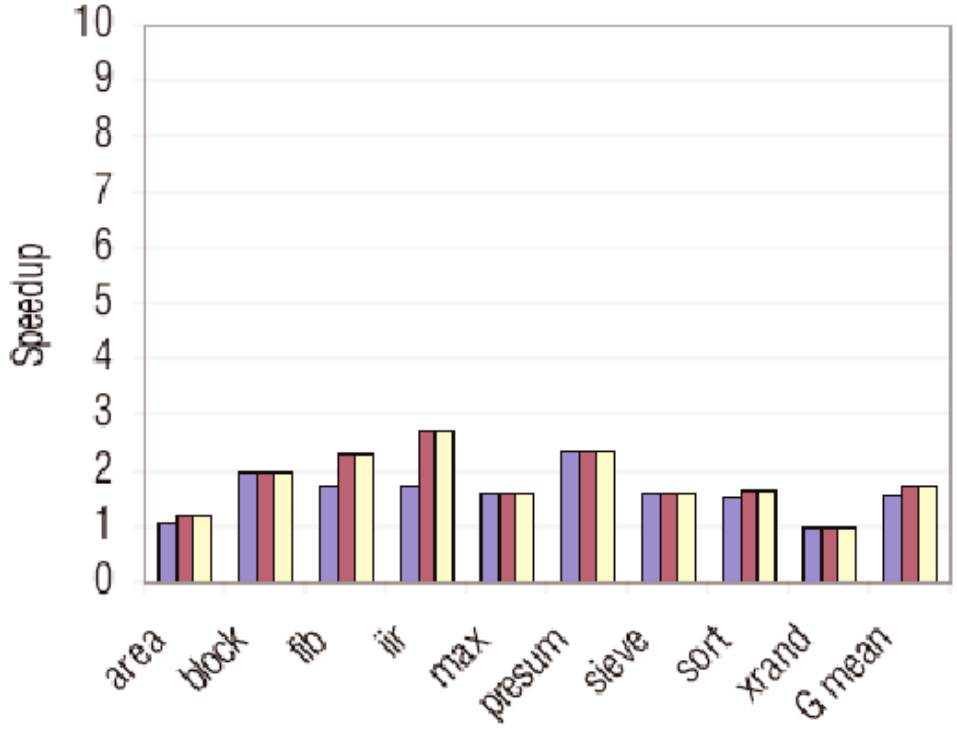
Evaluated configurations:

Baseline	A, M, C, S (comparable to DLX, SPARC)
T5	A→M→C→S
T7	A→A→M→A→C→S
T11	A→A→A→A→A→M→A→A→C→S

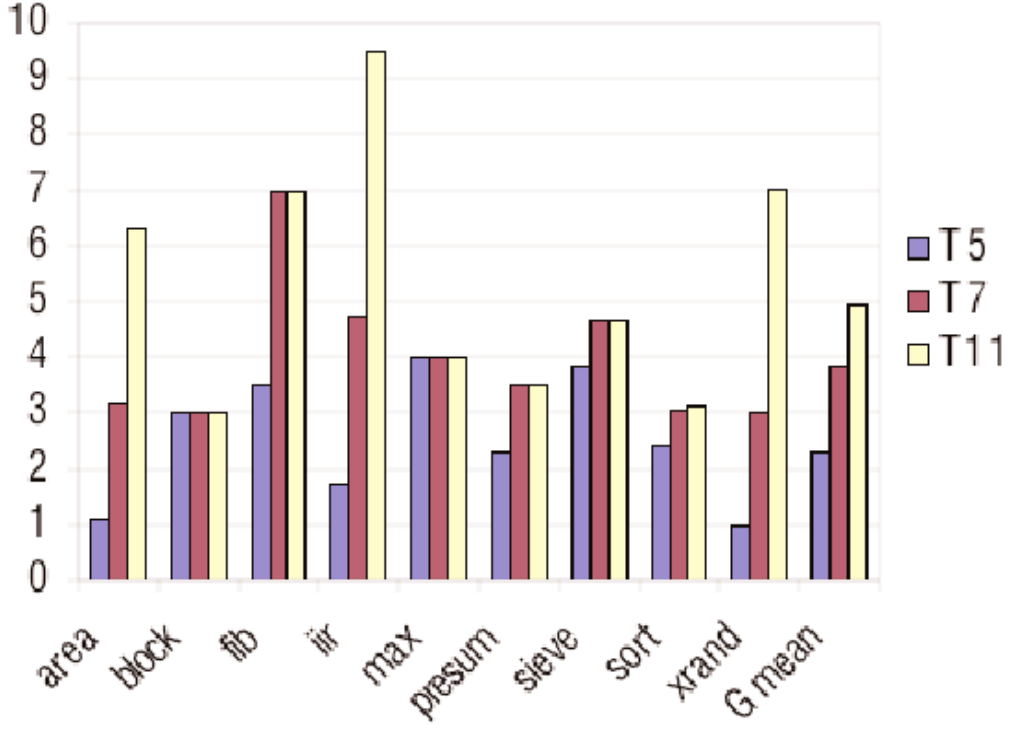
Test programs consisted of randomly selected SPECInt92 basic blocks and the following:

area	A program that calculates an integral of a polynomial
block	A program that moves a block of integers from a location to another
fib	An iterative program that calculates the Fibonacci number of given parameter
iir	A program that applies infinite response filter to a table of given integers
max	A program that finds the maximum of given table of integers
presum	A program that calculates the prefix sums for given table of integers
sieve	A program that finds prime numbers using the sieve of Eratosthenes
sort	A program that sorts given table of integers using recursive quicksort algorithm
xrand	A strictly sequential program that calculates a pseudo random number

Parallel organization:



Chained organization:



We implemented a basic block compression-based optimization algorithm for our compiler [Forsell03]:

Benchmark suite:	Chained	230%, 386% and 496% (within 1% of the theoretical maximum)
	Parallel	157%, 175% and 175% (for 4,6, and 10 FU CMPs)
SPECInt92:	Chained	248%, 401% and 453% (for 4,6, and 10 FU CMPs)

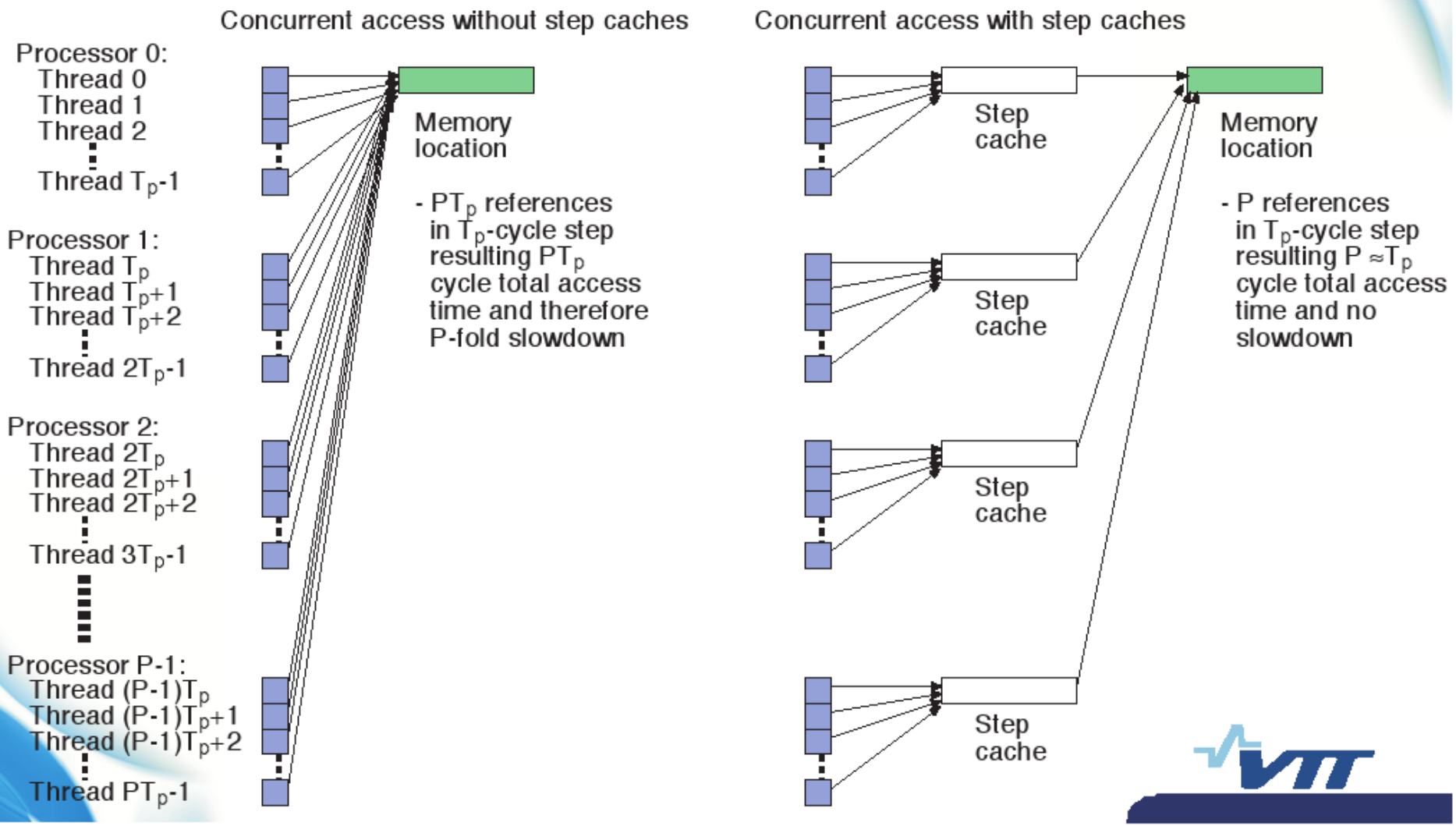
Additional architectural techniques

This kind of architecture makes it possible to use some new architectural techniques that are not familiar from sequential computer architecture:

- Concurrent memory access
- Multioperations

These techniques can be used to speed up execution of many computational problems by a logarithmic factor [Jaja92].

Concurrent memory access



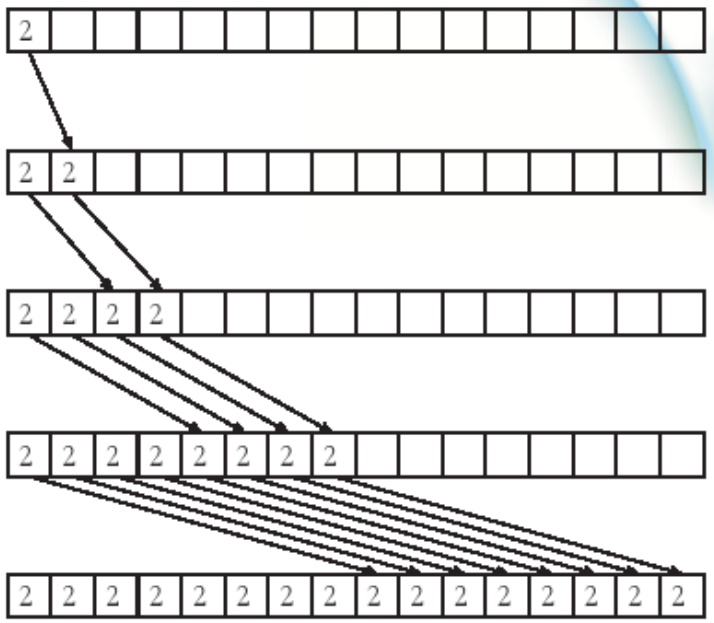
Support for concurrent access (CRCW)

SPREAD SCALAR a_ TO ARRAY b_:

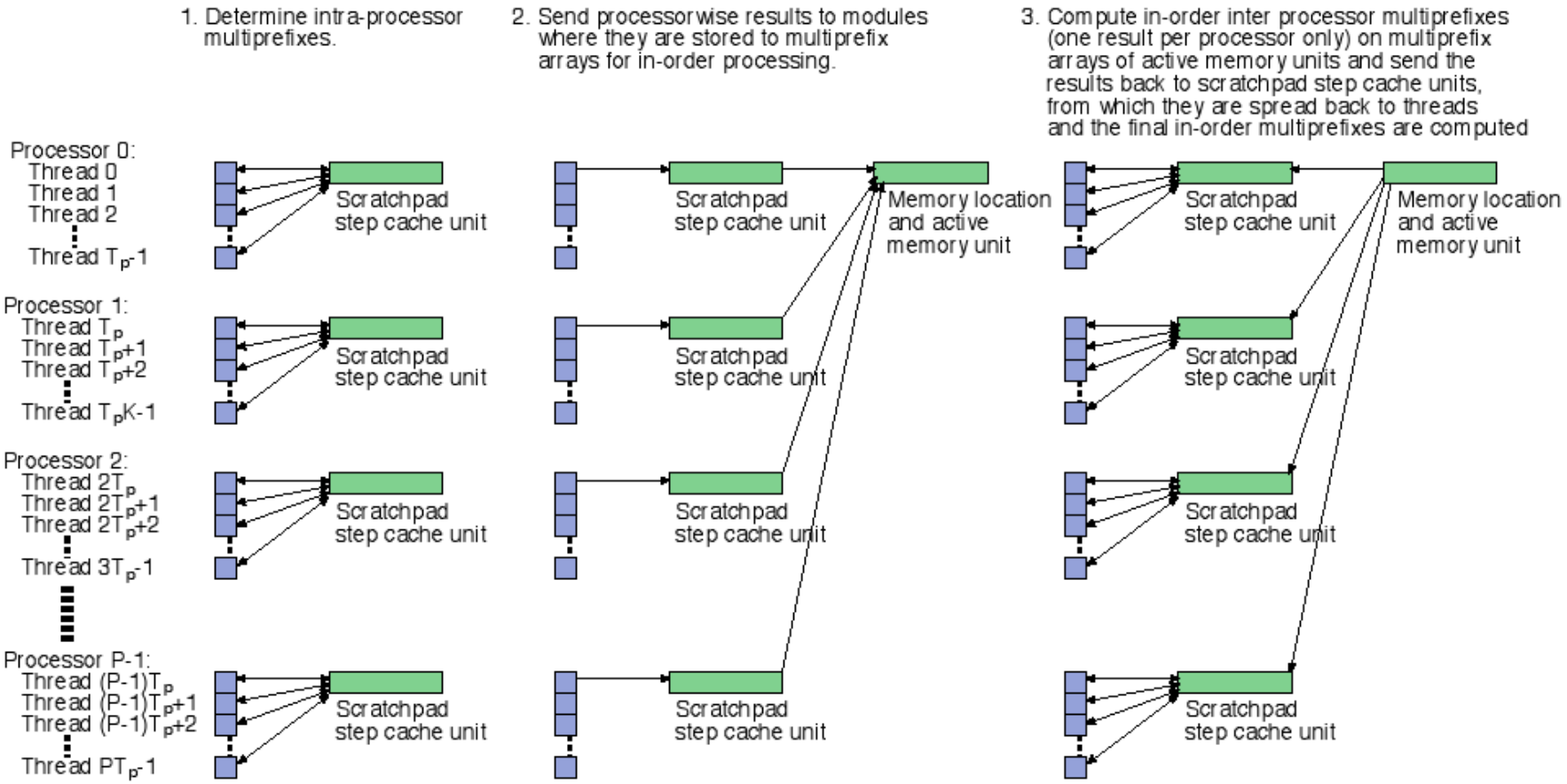
```
int a_;              // A shared variable
int b_[size];        // A shared array of integers
int tmp_[size];      // Thread-wise copies of a_ EREW
```

```
// EREW O(log n) version:
int i;
// Spread a_ to tmp_ with a logarithmic algorithm
if_ ( _thread_id==0 , tmp_[0]=a_; );
for (i=1; i<_number_of_threads; i<<=1)
  if_ ( _thread_id-i>=0 ,
    tmp_[_thread_id]=tmp_[_thread_id-i]; );
b_[_thread_id]+=tmp_[_thread_id]
```

```
// CRCW O(1) version:
b_[_thread_id]+=a_; // Concurrent read
```



Multioperations



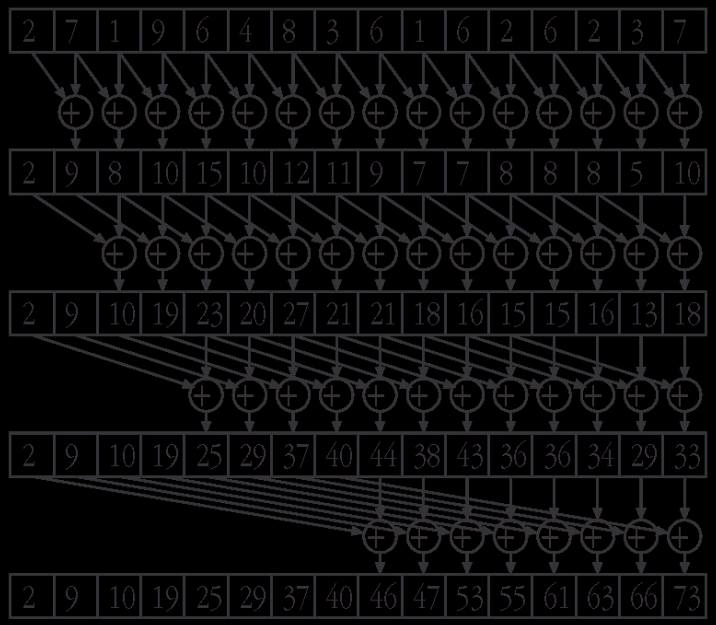
BMPxx instruction

SMPxx instruction
- first reference triggers an external memory reference
- ordering is preserved here by storing in non-deterministic order arriving references according to IDs of sending processors

OMPxx instruction
- the processor-wise offset is computed to thread-wise results
- threads that have already used their execution slots will be updated in the end of the memory reply pipeline segment

Support for multioperations

```
COMPUTE A SUM OF ARRAY a_ to sum_:

int sum_;          // A shared variable
int a_[size];      // A shared array of integers

// EREW O(log n) version—logarithmic algorithm for sum

for_ ( i=1 , i<_number_of_threads , i<<=1 ,
  if (_thread_id-i>=0)
    a_[_thread_id] += a_[_thread_id-i]; );
sum_=a_[_number_of_threads-1]

// Multioperation version
// —just call the constant time sum primitive

multi(MADD,&sum_,a_[_thread_id]);
```



multi(OP,m,c) Perform a multioperation *OP* for components *c* in memory location *m*.

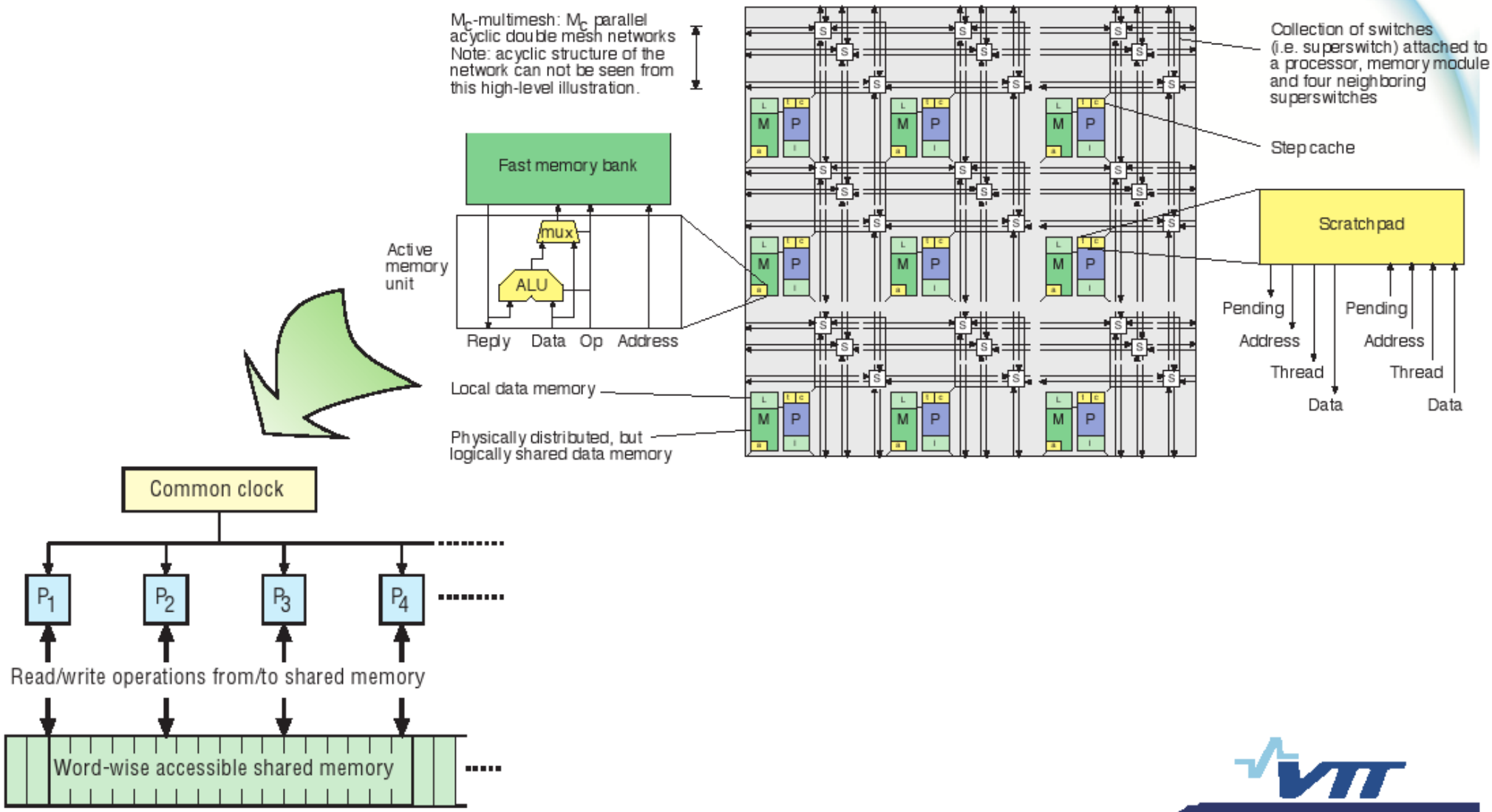
Results—concurrent access, multiop-erations [Forsell09b]

	E4	C4	C4+	E16	C16	C16+	E64	C64	C64+
aprefix	1,00	1,41	11,99	3,18	4,83	47,27	9,49	14,63	184,07
max	1,00	1,38	9,85	3,16	4,74	39,05	12,75	16,31	151,90
search	1,00	12,09	12,08	3,26	48,12	44,83	12,37	189,07	188,90
spread	1,00	72,00	71,93	3,18	287,43	287,17	11,16	1136,46	1135,45
sum	1,00	1,41	14,39	3,18	4,83	56,02	11,13	14,63	221,74

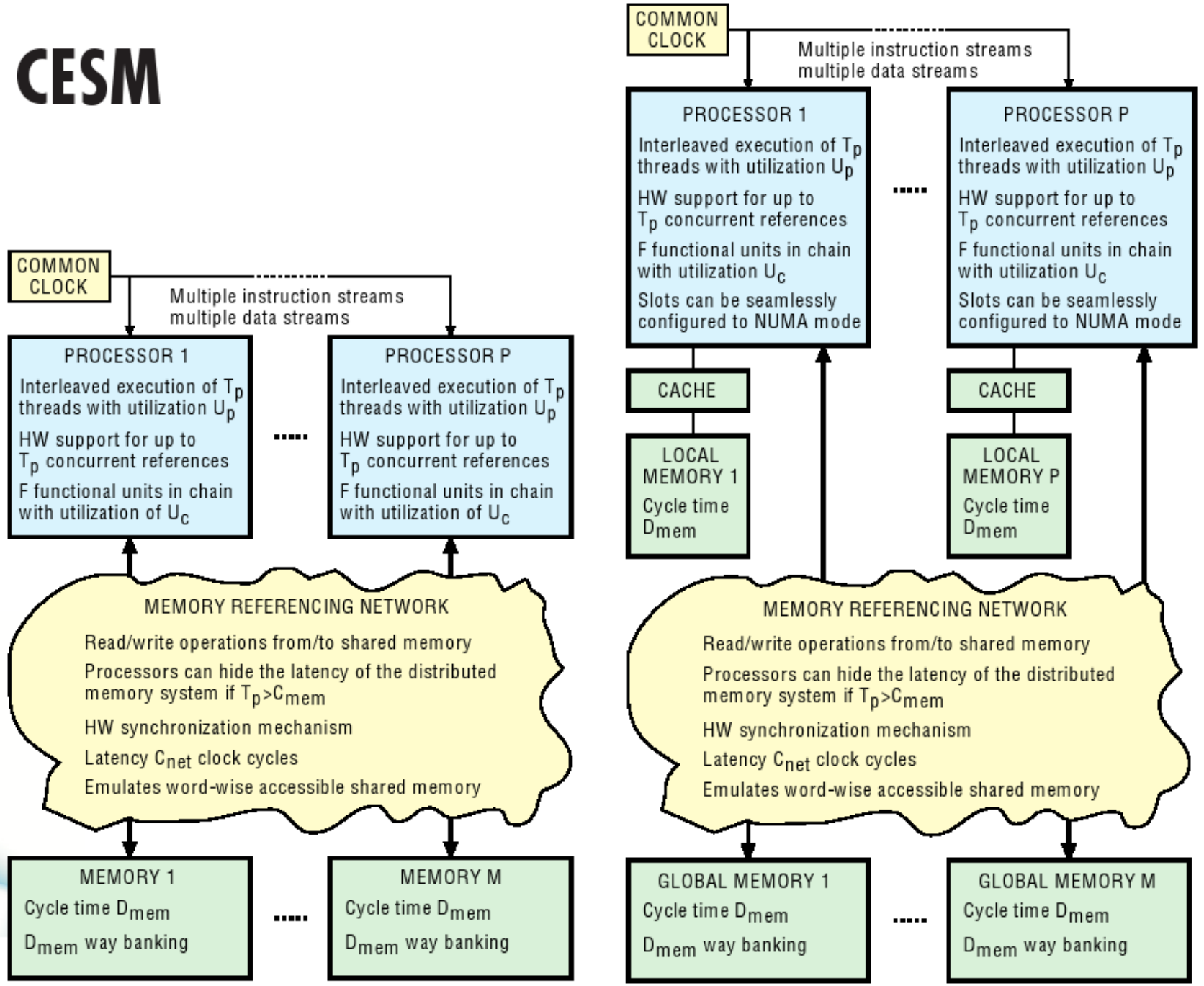
Relative throughput with respect to E4.

- E = Exclusive memory access
- C = Concurrent memory access
- C+ = Concurrent memory access + multioperations

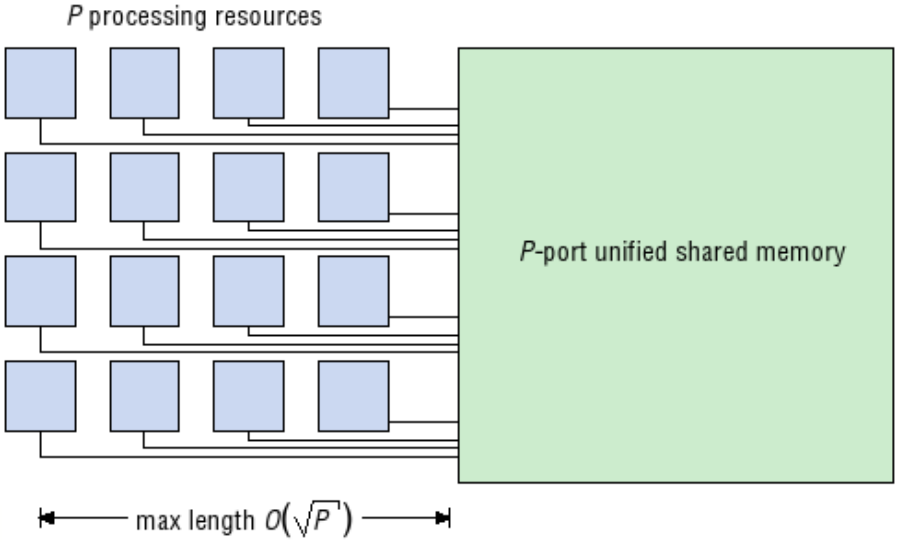
Abstraction becomes now

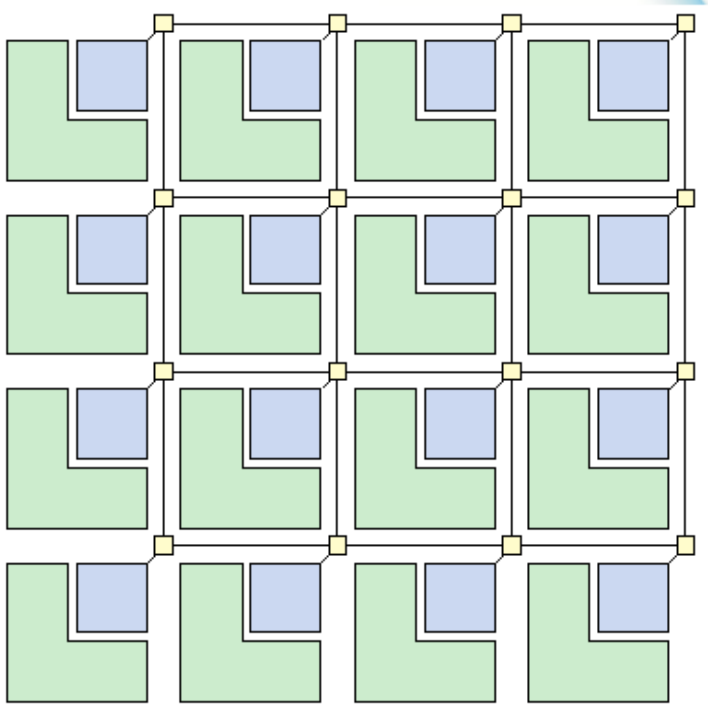


ESM, CESM


COMMON CLOCK
Multiple instruction streams multiple data streams
PROCESSOR 1
Interleaved execution of T_p threads with utilization U_p
HW support for up to T_p concurrent references
F functional units in chain with utilization of U_c
PROCESSOR P
Interleaved execution of T_p threads with utilization U_p
HW support for up to T_p concurrent references
F functional units in chain with utilization of U_c
MEMORY REFERENCING NETWORK
Read/write operations from/to shared memory
Processors can hide the latency of the distributed memory system if $T_p > C_{mem}$
HW synchronization mechanism
Latency C_{net} clock cycles
Emulates word-wise accessible shared memory
MEMORY 1
Cycle time D_{mem}
D_{mem} way banking
MEMORY M
Cycle time D_{mem}
D_{mem} way banking
COMMON CLOCK
Multiple instruction streams multiple data streams
PROCESSOR 1
Interleaved execution of T_p threads with utilization U_p
HW support for up to T_p concurrent references
F functional units in chain with utilization U_c
Slots can be seamlessly configured to NUMA mode
PROCESSOR P
Interleaved execution of T_p threads with utilization U_p
HW support for up to T_p concurrent references
F functional units in chain with utilization U_c
Slots can be seamlessly configured to NUMA mode
CACHE
CACHE
LOCAL MEMORY 1
Cycle time D_{mem}
LOCAL MEMORY P
Cycle time D_{mem}
MEMORY REFERENCING NETWORK
Read/write operations from/to shared memory
Processors can hide the latency of the distributed memory system if $T_p > C_{mem}$
HW synchronization mechanism
Latency C_{net} clock cycles
Emulates word-wise accessible shared memory
GLOBAL MEMORY 1
Cycle time D_{mem}
D_{mem} way banking
GLOBAL MEMORY M
Cycle time D_{mem}
D_{mem} way banking

Why distributed shared memory rather than unified? [Forsell94]





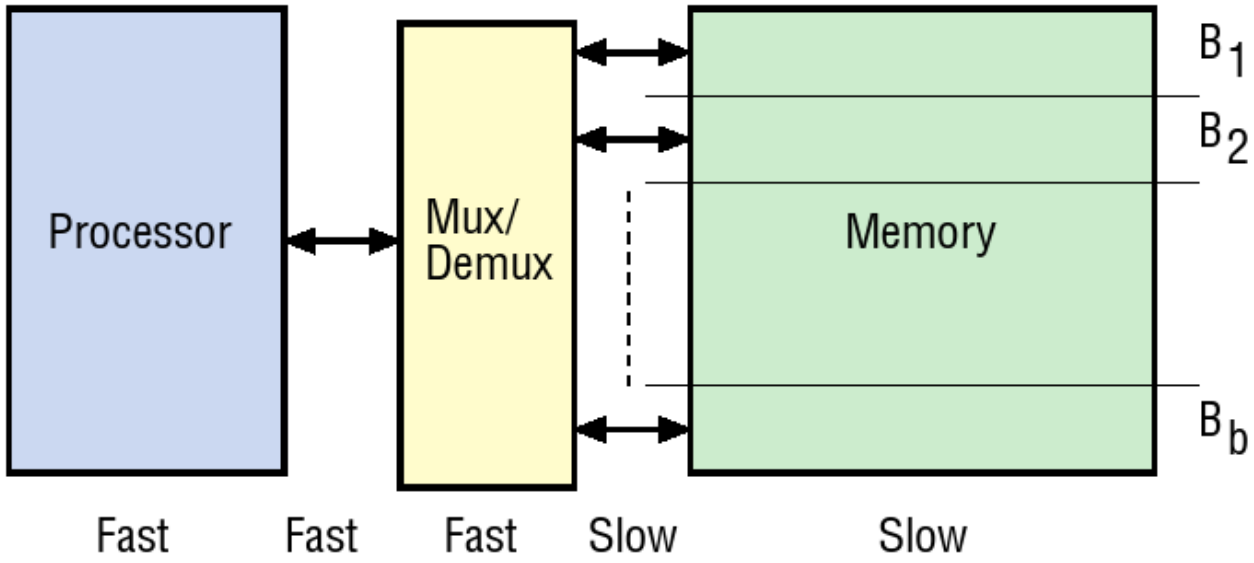
Unified shared memory (true multiport memory)

- Wiring takes P^2 more area than a single port memory of the same size [Forsell94]
- Max length of wire $O(P^{1/2})$

Distributed shared memory (multimesh)

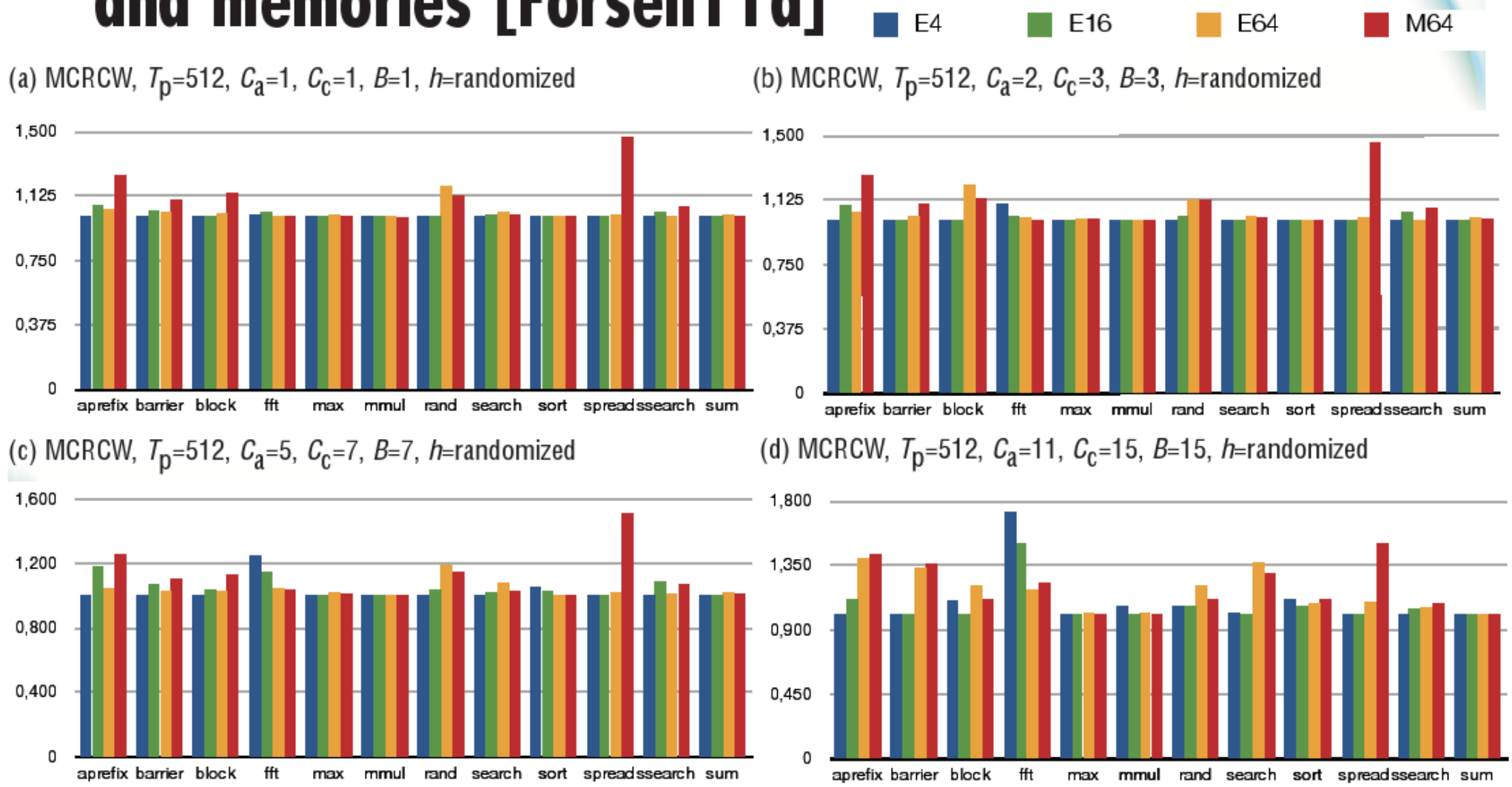
- Wring grows fast but e.g. a 64-core system requires just a dual mesh
- Max length of wire $O(1)$

Balancing the speed difference between processors and memories [Forsell11a]

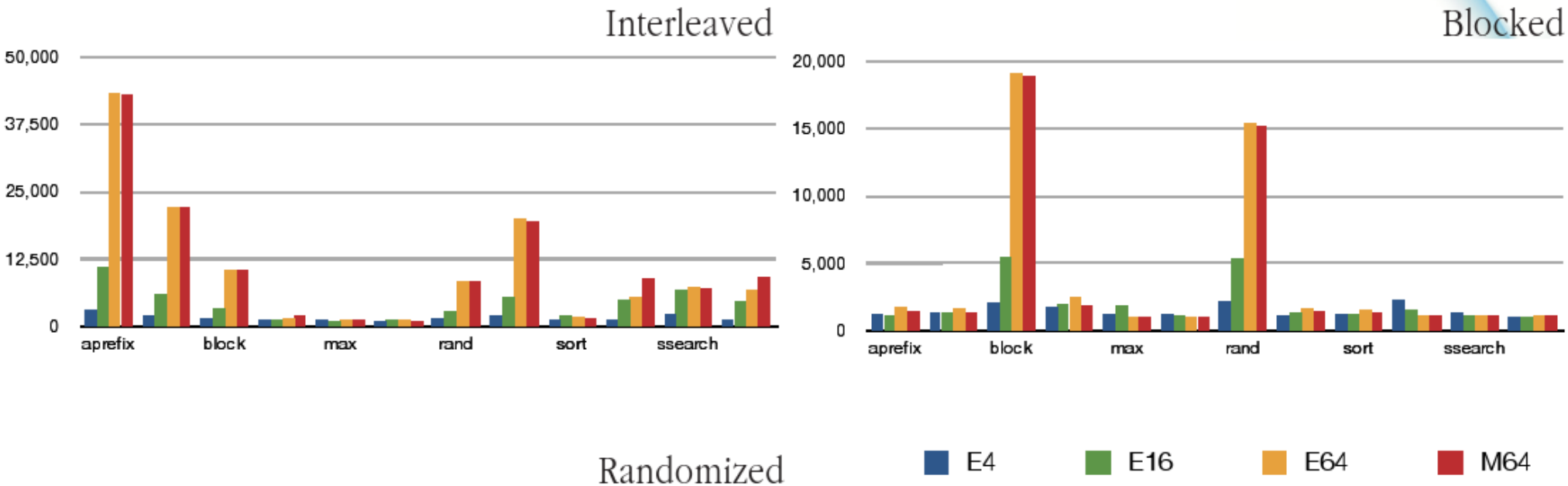


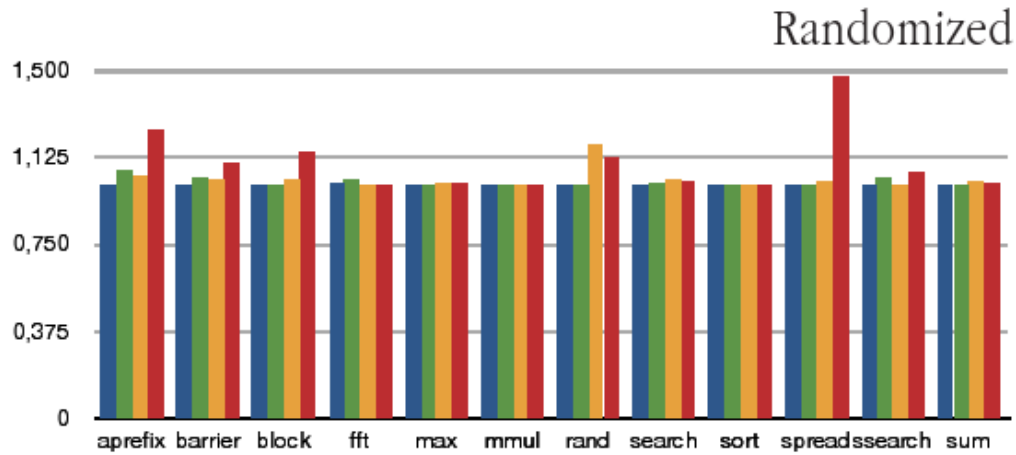
The speed difference between proessors and memory modules can be balanced by blocking (memory is divided into B banks)

Balancing the speed difference between processors and memories [Forsell11a]



Congestion avoidance with hashing [Forsell11a]





The cost of execution with respect to a similar machine with an ideal shared memory

- Randomized hashing can efficiently be used to avoid congestion in intercommunication

4. Summary

According to wide-spread consensus, current CMP/MP-SOC solutions making use of asynchronous shared memory and message passing are slow and hard to program.

- This is because of inefficient synchronization scheme, inapropriate abstraction, lack of bandwidth and support for parallel computing related techniques
- There exists a lot of parallelism in the applications and it is easily exploitable.
- Virtually all current solutions define **the same computability => the problem (with parallel computing) is mainly architectural**
- There exists alternatives that provide scalability and support for a good abstraction of parallelism.

REPLICA

100 Billion Dollar Question

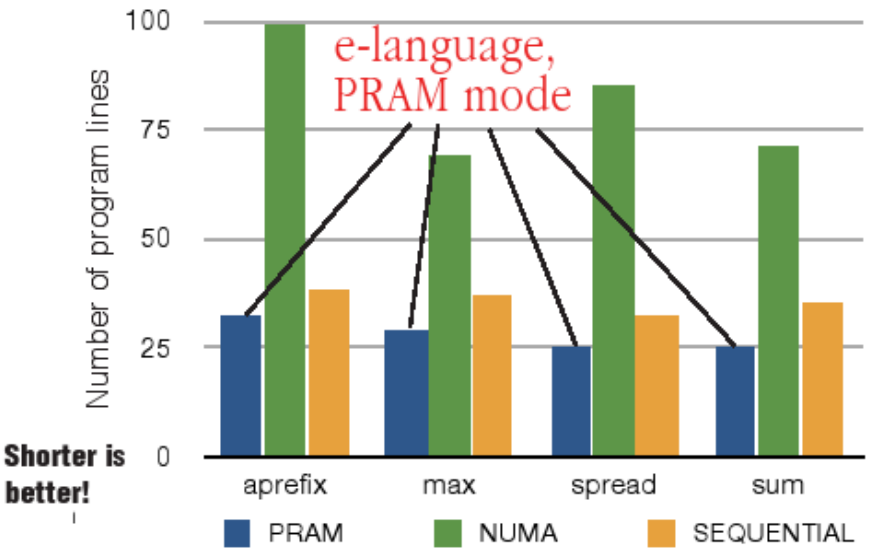
Current CMPs architectures (SMP, NUMA, CC-NUMA, MP, VC) are **tedious to program** and often provide **poor speedup** compared to conventional sequential (single core) processors.

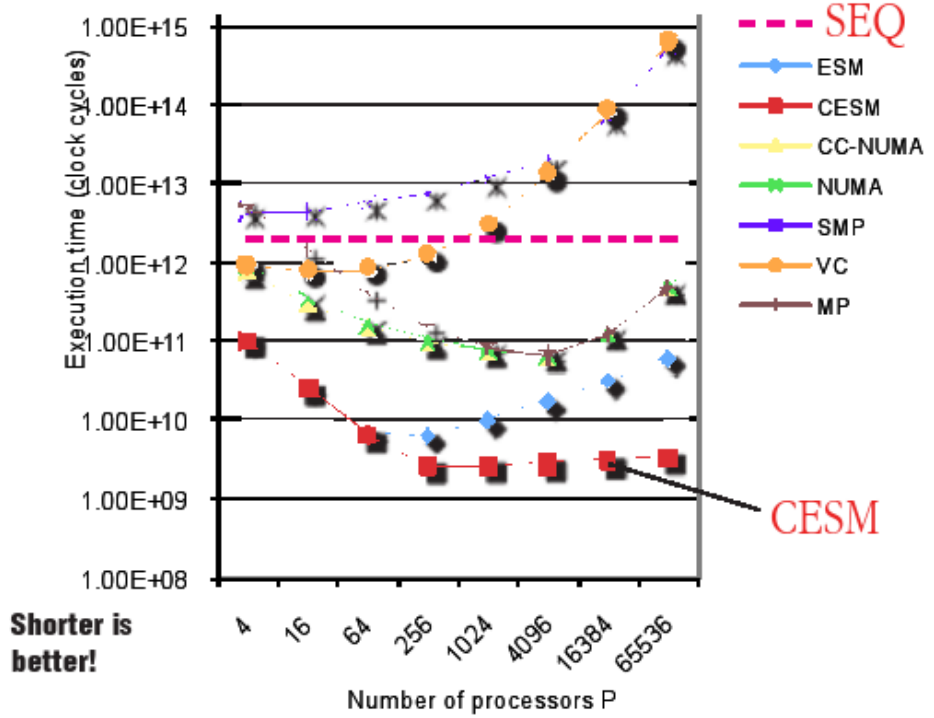
This is because of **lack of fast synchronization** and **latency hiding** mechanism, i.e. weak models of computation.

The **REPLICA** project aims developing an architecture (CESM) and methodology that would enable radically **easier programming** and **higher performance** with a help of the PRAM model of computing.

We can e.g. **drop the cost** of synchronization **from 100 to 1/100** + provide additional performance & programmability gains, but a **convincing proof of concept** is **needed**!

A proof of concept prototype will be built!





VTT

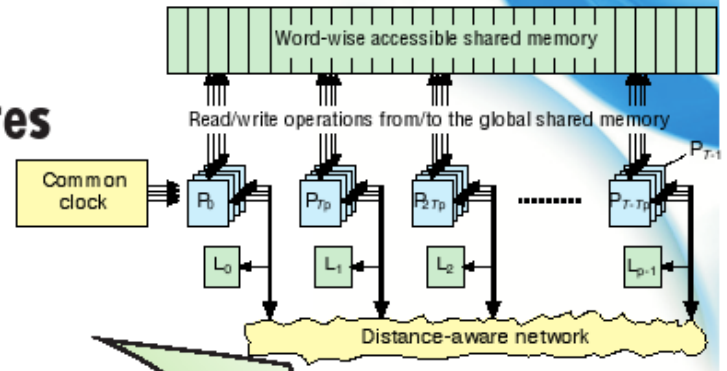
REPLICA = Removing Performance and Programmability Limitations of Chip Multiprocessor Architectures

A 3-year Frontier research project funded entirely by VTT

Funding: 500 000 €/year, in total 1 500 000 €

Amount of work: 129 pm, duration 3 year

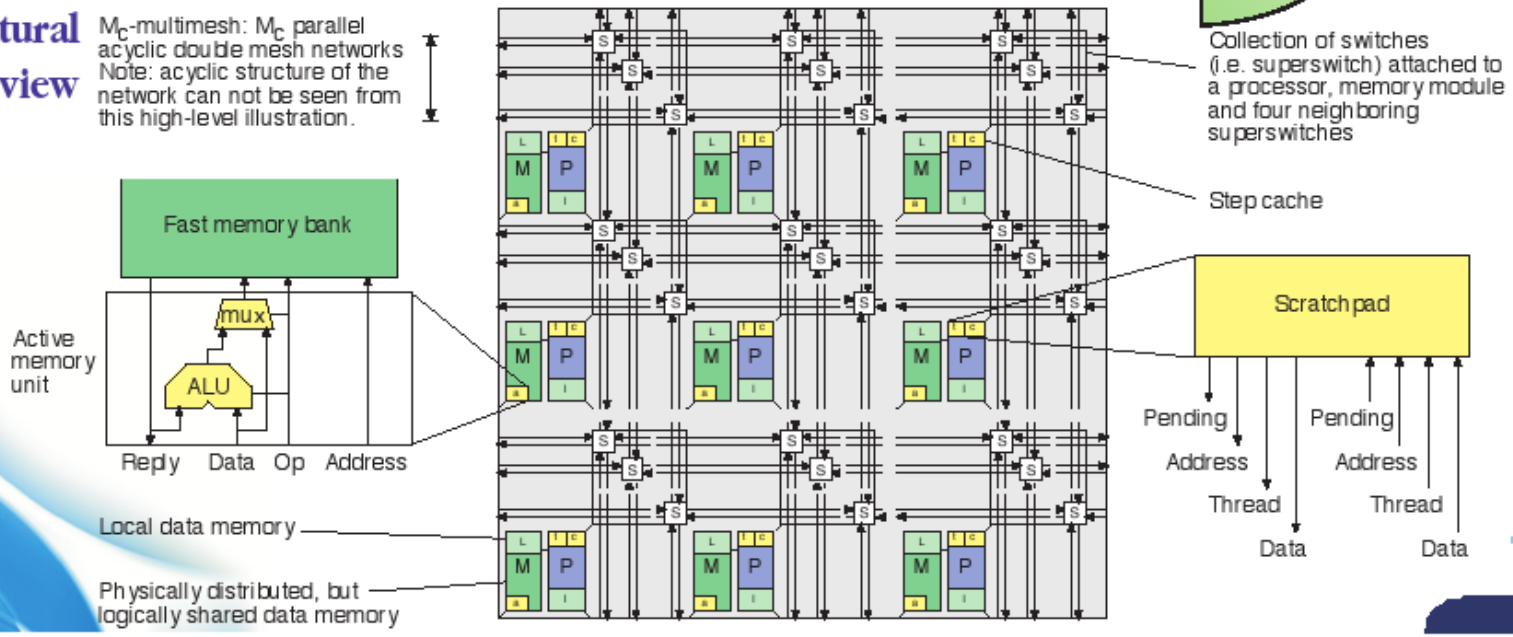
Target business: **Companies that design or manufacture general purpose and application-specific CMPs or develop software/functionality for them**



Programmers view

Architectural view

M_C-multimesh: M_C parallel acyclic double mesh networks
Note: acyclic structure of the network can not be seen from this high-level illustration.



Novel techniques:

- Latency hiding
- Efficient wave synchronization
- Concurrent memory access
- Multioperations
- Virtual ILP exploitation
- Pipeline hazard elimination
- Memory hashing

Home work (voluntary)

1. **Outline/write a program that computes a prefix sum of a vector of N integers making use of the work-optimal blocking algorithm (page 14) with the e-language (or similar algorithm language)**

   - How many synchronizations are needed (in the code) if it will be executed on an asynchronous machine and how many on a synchronous machine?
   - Why it might be efficient for current architectures?

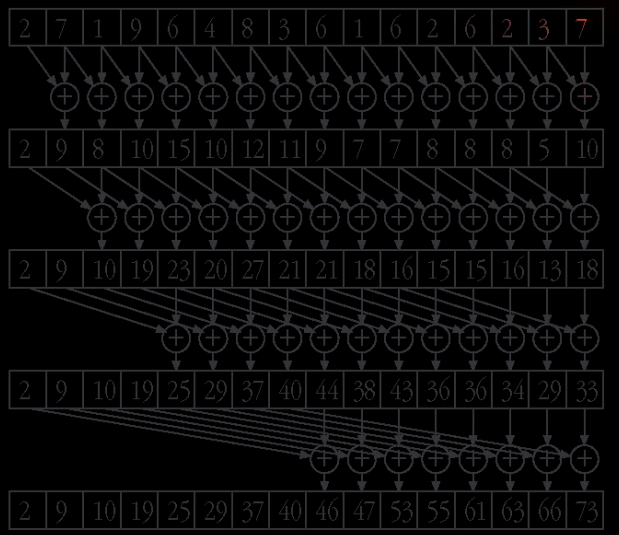
2. **Outline/write a program that sorts a vector of N integers in constant time with the e-language (or similar algorithm language)**

   - How many processors is needed?
   - Why the program might not practical for long vectors?

Hint: Make use of synchronous execution and constant time multiprefix operations.

The solutions will be available at my www page: http://www.ee.oulu.fi/~mforsell/work.html

E-language [Forsell04a]



The e-language is our experimental **TLP programming language**:

- For optimality reasons the **number of threads** is fixed **by** an **underlying machine** rather than by a construct boundaries
- The syntax is an **extension** of the syntax of familiar **c**
- Supports **parallely recursive** and **synchronous MIMD** programming under the computing model.
- **Shared and private variables** as globals and locals
- Multiple **simultaneous** execution **groups of threads**
- Groups can be **split into subgroups** hierarchically
- Supports both **synchronous** and **asynchronous** programming **style**
- Thread-level parallelism can be expressed with **thread identifiers**

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

Processing data in parallel, retaining synchronicity through uneven paths

With e, it very simple to process data by multiple threads in parallel

```
int table_ [ 255 ];                    // Declare shared variable
table_ [ _thread_id ] = f(thread_id);  // Assign f(element) to each element in parallel
```

Divide current group of thread into two subgroups and retain synchronicity over their sub-tasks of different duration

```
_if_else_ ( _thread_id < _number_of_threads >> 1 ,
      task for subgroup 1 taking  80 steps,
      task for subgroup 2 taking 57 steps
);
```

Splitting a parallel machine to submachines

Consider executing three applications in **parallel** with a *T*-threaded machine:

A	sequential application	1 thread
B	parallel application	*T*/3-1 threads
C	parallel application	2*T*/3 threads

```
_if_else_ ( _thread_id < _number_of_threads / 3 ,
      _if_else ( _thread_id = 0 ,
            A ,
            B ); ,
      C );
```

- These applications can be executed **independently or** they can **communicate with each other** via shared variables.
- Each application A, B, and C sees a **virtual machine** featuring 1, *T*/3-1, and 2*T*/3 threads without performance loss.

Handling non-idealities

If there are **more data elements** than threads

```
for_ ( i=_thread_id , i < number_of_elements , i += _number_of_threads ,
        process elements
);                                  // loop, i is thread private variable
```

If there are **less data elements** than threads

```
if_else_ ( _thread_id < number_of_elements ,
        process elements ,                                          // then part
        do something else with the rest of the threads or just wait // else part
);
```

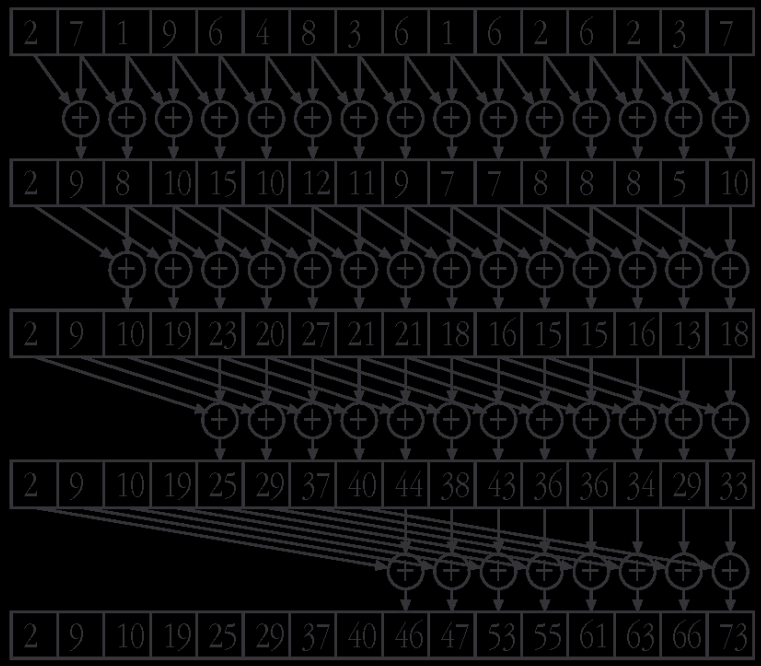
```
#define      size 2047          // SEQUENTIAL C-LANGUAGE VERSION
int source[size];
int main()
{
    int i;
    for ( i=0 ; i <size ; i+=1 )  // Initialize the array iteratively O(N) steps
        source[i] = i;
    for ( i=1 ; i<size ; i+=1 )   // Compute iteratively O(N) steps
        source[i] += source[i-1];
}
```

```
#include "e.h"                    // PARALLEL E-LANGUAGE VERSION
#define      size 2047
int source_[size];
int main()
{
    int i;
    source_[_thread_id] = _thread_id;     // Initialize in parallel, time O(1)
    for_ ( i=1 , i<size , i<<=1 ,         //  Compute in parallel, O(log N)
        if (_thread_id-i>=0) source_[_thread_id] += source_[_thread_id-i]; );
}
```

All threads execute in parallel unless thread-specific control requires threads to split groups

Example: Parallel prefix sum

- Sounds as strictly sequential problem but can be solved in parallel easily & efficiently
- Does not contain much locality!



References

[Forsell94] M. Forsell, Are Multiport Memories Physically Feasible?, Computer Architecture News 22, 4 (September 1994), 47-54.

[Forsell02a] M. Forsell, Architectural differences of efficient sequential and parallel computers, Journal of Systems Architecture 47, 13 (July 2002), 1017-1041.

[Forsell02b] M. Forsell, A Scalable High-Performance Computing Solution for Network on Chips, IEEE Micro 22, 5 (September-October 2002), 46-55.

[Forsell03] M. Forsell, Using Parallel Slackness for Extracting ILP from Sequential Threads, In the Proceedings of the SSGRR-2003s, July 28 - August 3, 2003, L'Aquila, Italy.

[Forsell04a] M. Forsell, E—A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs, WSEAS Transactions on Computers 3, 3 (July 2004), 807-812.

[Forsell04b] M.Forsell, Compiling Thread-Level Parallel Programs with a C-Compiler, In the Proceedings of the IV Jornadas sobre Programacion y Lenguajes (PROLE'04), November 11-12, 2004, Malaga, Spain, 215-226.

[Forsell06] M. Forsell, Realizing Multioperations for Step Cached MP-SOCs, In the Proceedings of the International Symposium on System-on-Chip 2006 (SOC'06), November 14-16, 2006, Tampere, Finland, 77-82.

[Forsell07] M. Forsell, A Cost-Efficient Algorithm for Arbitrary CRCW PRAM Simulation, In the Proceedings of the 2007 ECTI International Conference (ECTI-CON 2007), May 9-12, 2007, Chiang Rai, Thailand, 1190-1193.

References

[Forsell08] M. Forsell and J. Roivainen, Performance, Area and Power Trade-Offs in Mesh-Based Emulated Shared Memory CMP Architectures, In the Proceedings of the 2008 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'08), July 14-17, 2008, Las Vegas, USA, 471-477.

[Forsell09a] M. Forsell, Configurable Emulated Shared Memory Architecture for general purpose MP-SOCs and NOC regions, In the Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip, May 10-13, 2009, San Diego, USA, 163-172.

[Forsell09b] M. Forsell, On the performance and cost of some PRAM models on CMP hardware, to appear in International Journal of Foundations of Computer Science, 2009.

[Forsell10a] M. Forsell, A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads, In the Proceedings of the 12th Workshop on Advances in Parallel and Distributed Computational Models (in conjunction with the 24th IEEE International Parallel and Distributed Processing Symposium, IPDPS'10), April 19, 2010, Atlanta, USA, 1-8.

[Forsell10b] M. Forsell, TOTAL ECLIPSE—An Efficient Architectural Realization of the Parallel Random Access Machine, In Parallel and Distributed Computing Edited by Alberto Ros, IN-TECH, Vienna, 2010, 39-64. (ISBN 978-953-307-057-5)

[Forsell10c] M. Forsell, On the performance and cost of some PRAM models on CMP hardware, International Journal of Foundations of Computer Science 21, 3 (2010), 387-404.

[Forsell10f] M. Forsell, P. Hofstee, A. Jerraya, C. Jesshope, U. Vishkin and J. Träff, HPPC 2009 Panel: Are Many-Core Computer Vendors on Track?, Lecture Notes in Computer Science 6043, (2010), 9-15.

[Forsell11a] M. Forsell, Performance comparison of some shared memory organizations for 2D mesh-like NOCs, Microprocessors and Microsystems 35, 2 (March 2011), 274-284.

[Forsell11b] M. Forsell, A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads, International Journal of Networking and Computing 1, 1 (January 2011), 21-35.

[Forsell11d] M. Forsell and J. Roivainen, Supporting Ordered Multiprefix Operations in Emulated Shared Memory CMPs, In the Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11), July 18-21, 2010, Las Vegas, USA, 506-512.

[Fortune78] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, Proceedings of 10th ACM STOC, Association for Computing Machinery, New York, 1978, 114-118.

[Jaja92] J. Jaja, Introduction to Parallel Algorithms, Addison-Wesley, Reading, 1992.

[Keller01] J. Keller, C. Keßler, and J. Träff, Practical PRAM Programming, Wiley, New York, 2001.

[Leppänen96] V. Leppänen, Studies on the realization of PRAM, Dissertation 3, Turku Centre for Computer Science, University of Turku, Turku, 1996.

[Ranade91] A. Ranade, How to Emulate Shared Memory, Journal of Computer and System Sciences 42, (1991), 307-326.

[Vihkin08] U. Vishkin, G. Caragea, Aand B. Lee, Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform, In Handbook of Parallel Computing—Models, Algorithms and Applications (editors S. Rajasekaran and J. Reif), Chapman & Hall/CRC, Boca Raton, 2008, 5-1—5-60.