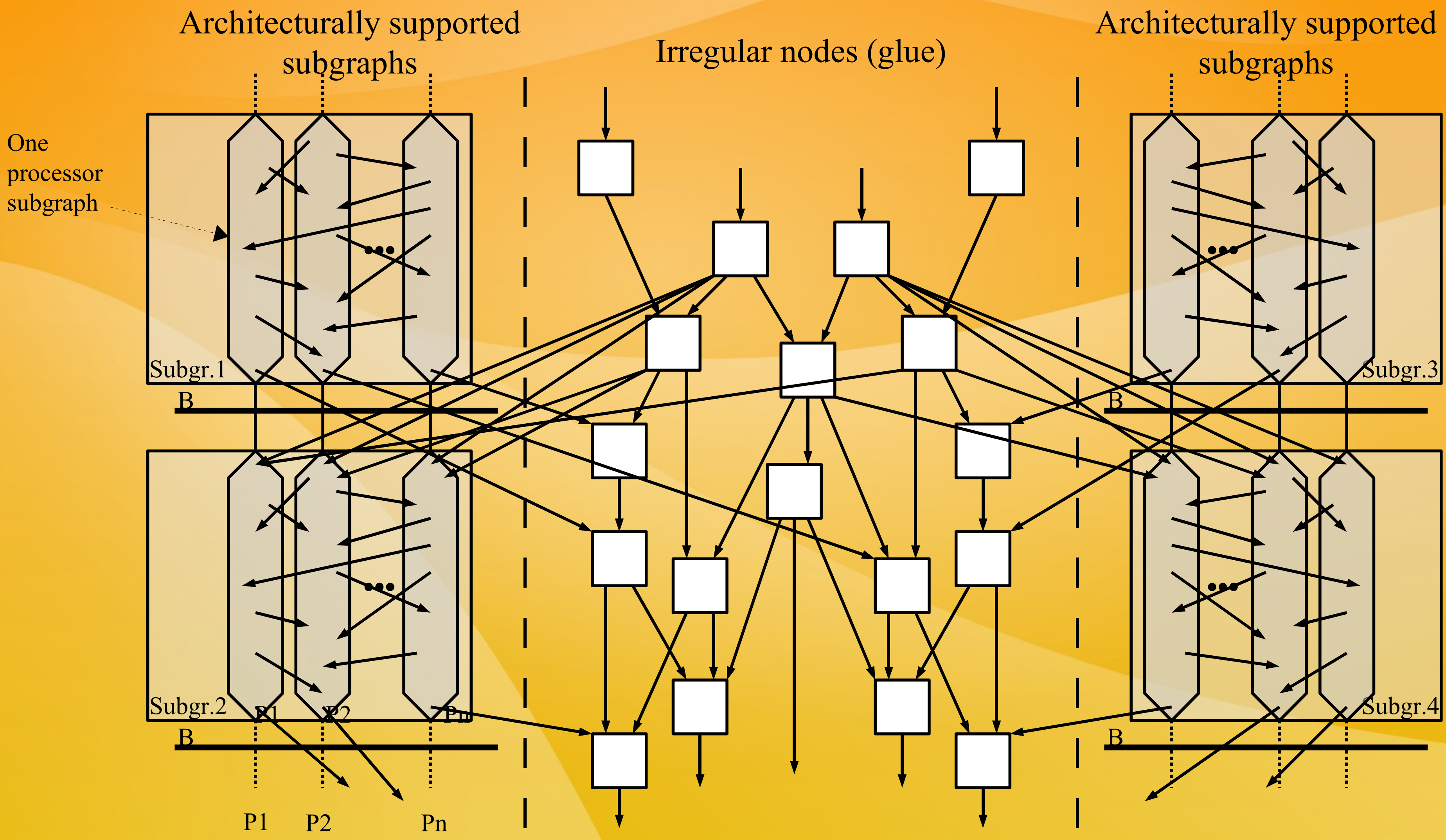
Inter-Processor Communication in SoC Systems for Intensive Fine-Grain Data Sharing

Marek Tudruj, Łukasz Maśko

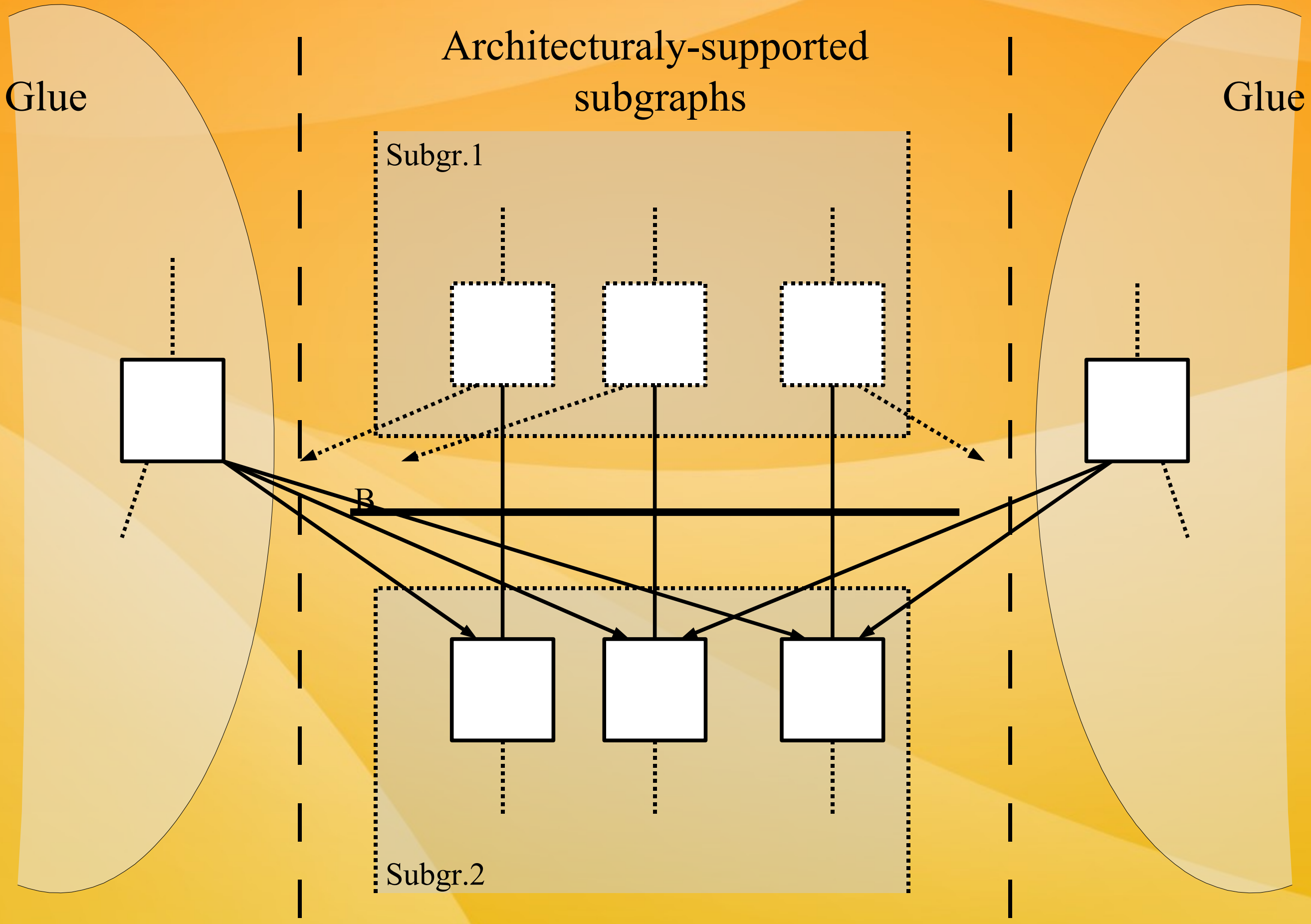
Institute of Computer Science Polish Academy of Sciences
Polish-Japanese Institute of Information Technology
Warsaw, Poland

Contents:

Program graph with architecturally-supported regions



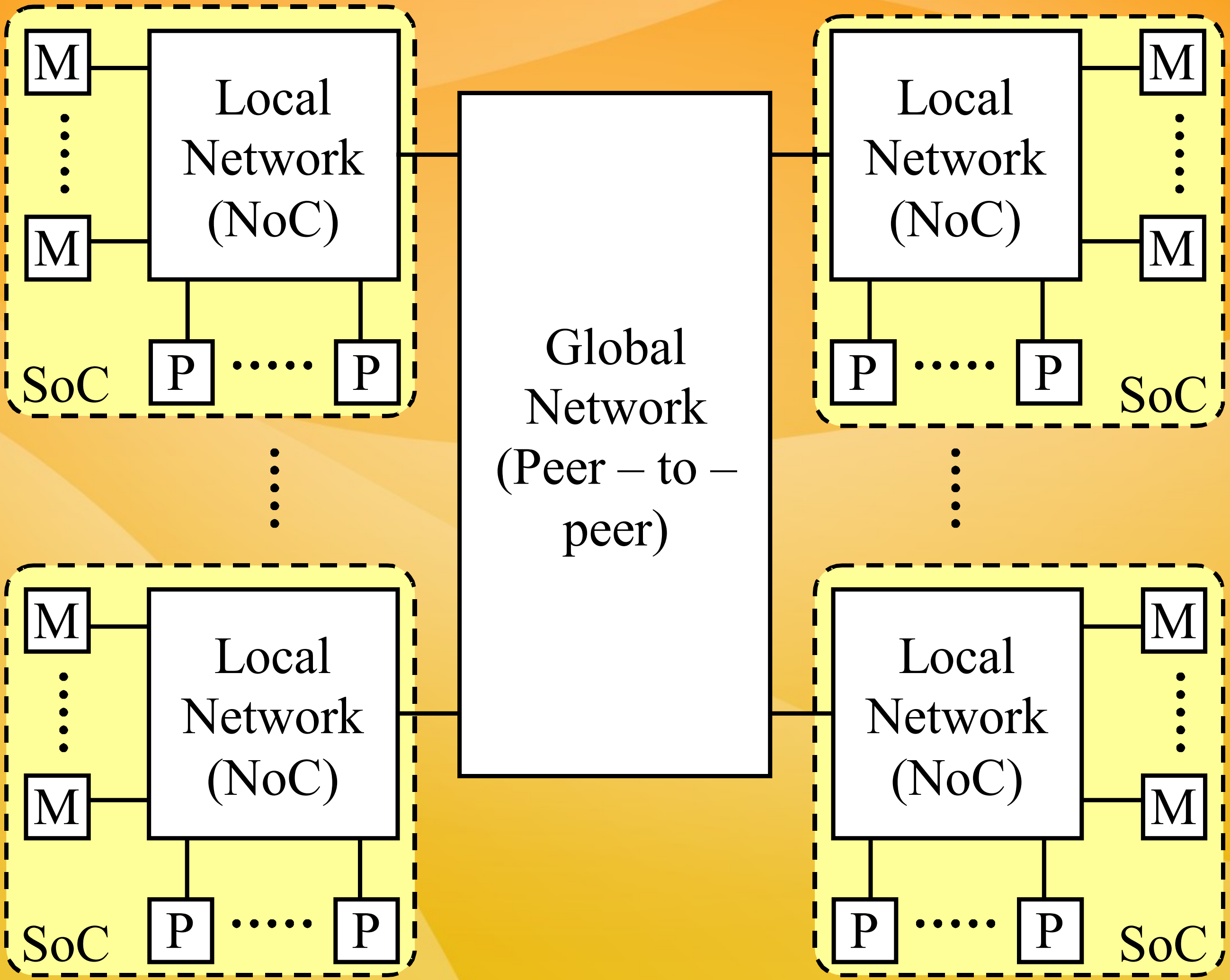
Initialization of an architecturaly-supported subgraph



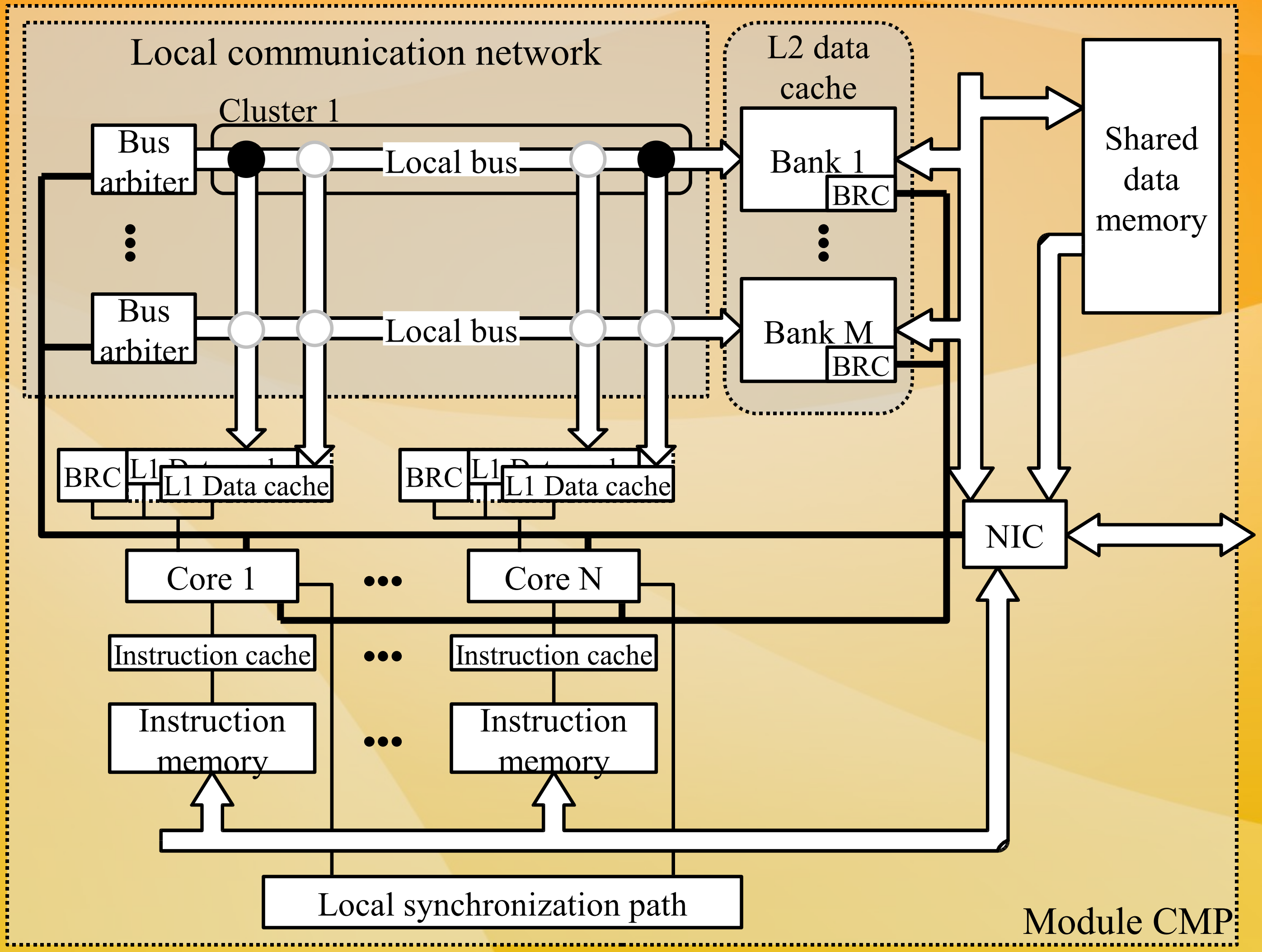
System on Chip technology

- Systems on Chip (SoC) technology offers new implementation perspectives for parallel processing.
- Processor centric SoC design has been replaced by interconnect-centric design.
- In the next several years the number of processors in SoC systems will increase up to hundreds and thousands.
- Reconsidering the problem of massively parallel systems has received a technical background.

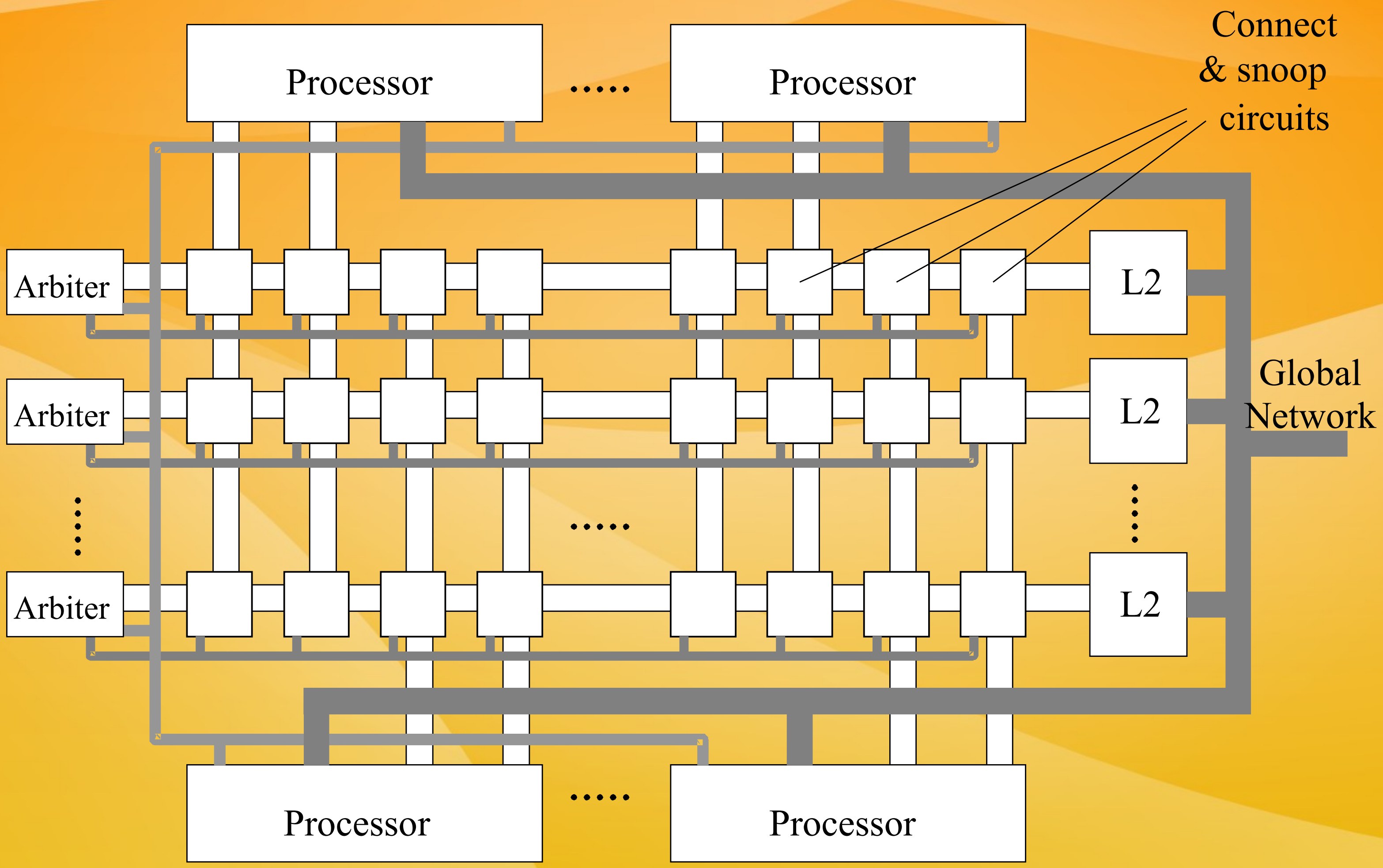
System architecture



System architecture



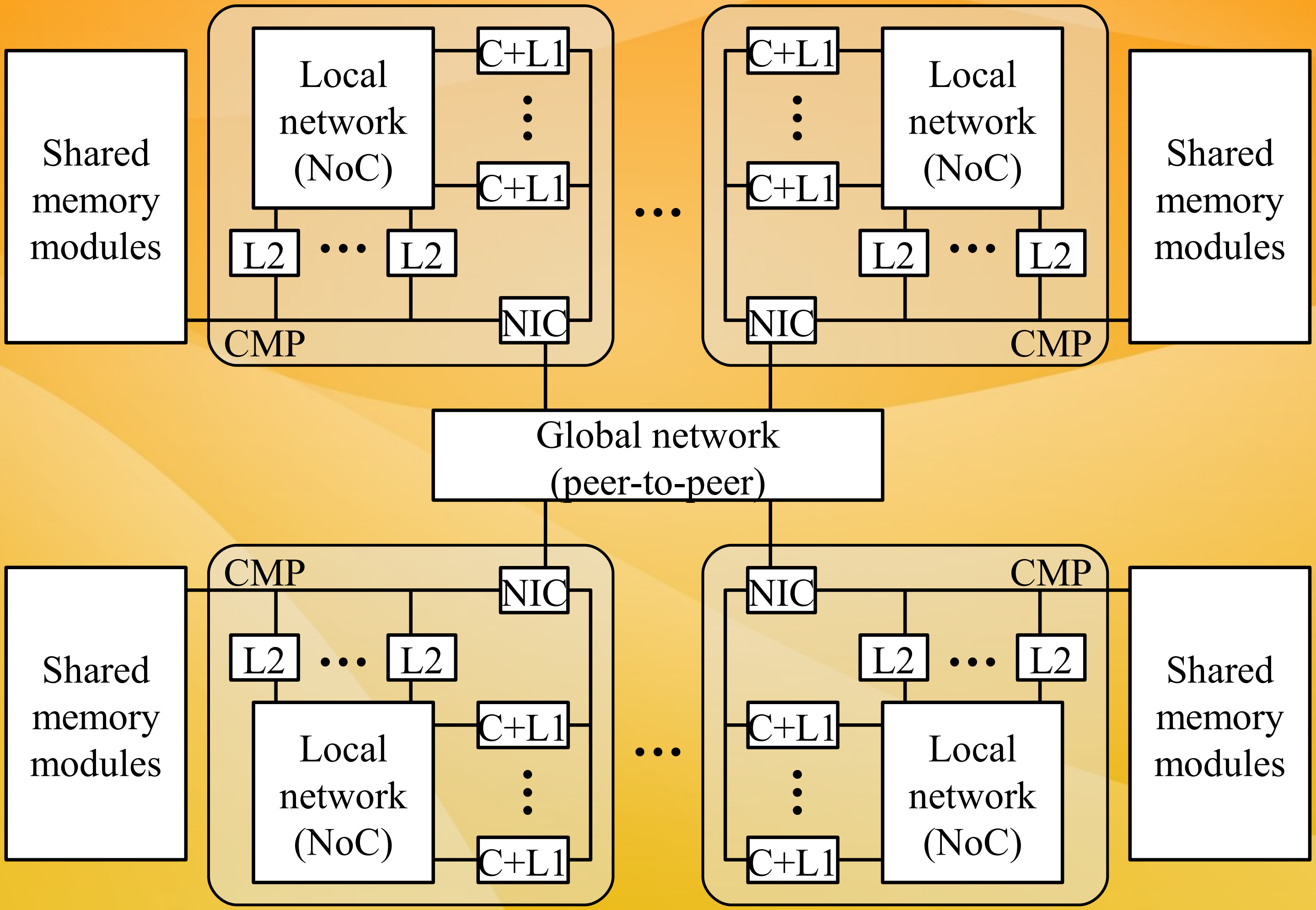
SoC implementation



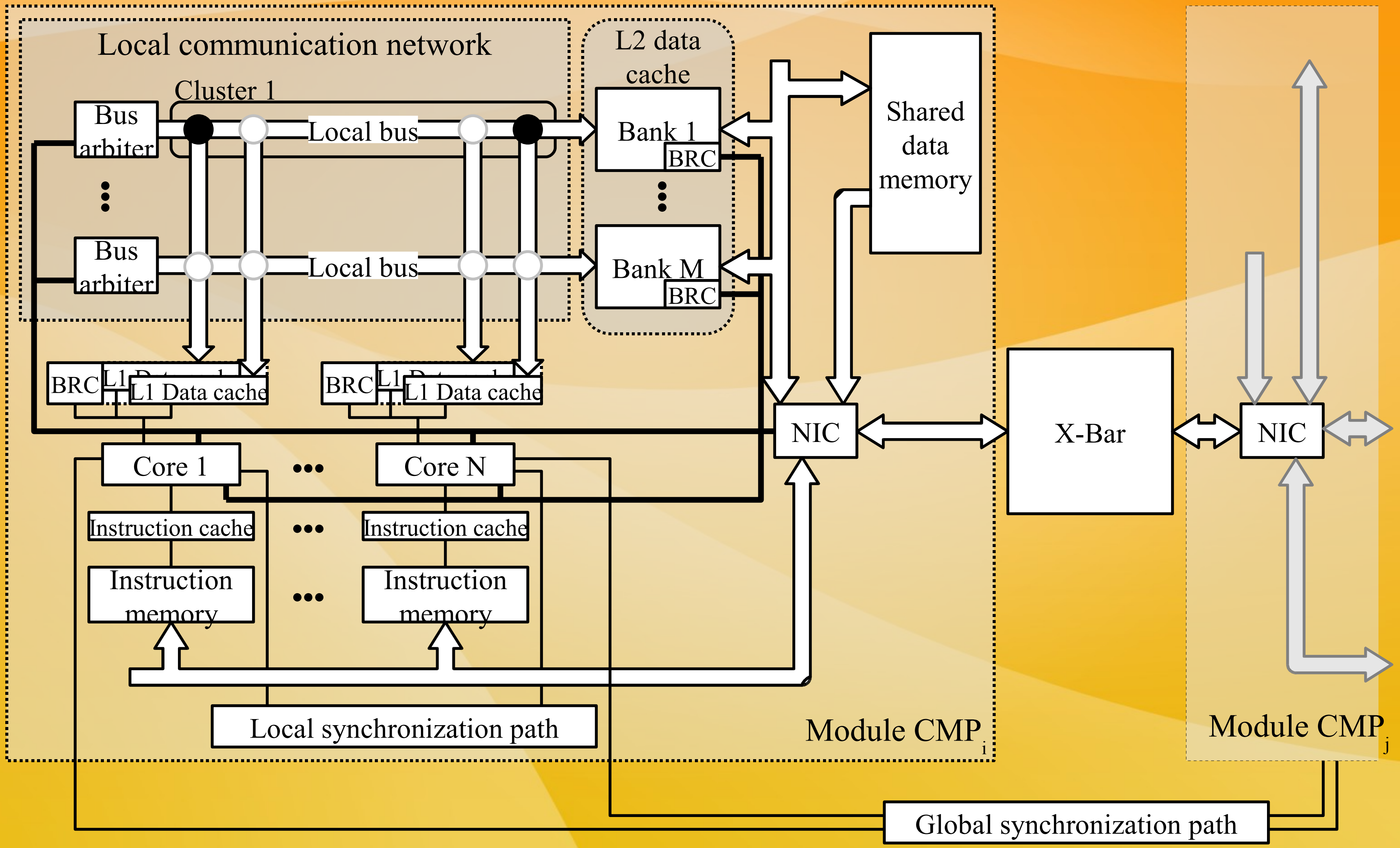
New architecture to reduce data communication time

- Introduction of shared L2 caches.
- Communication on the fly on memory to memory transfers.

General system structure with L1/L2 cache



CMP module internal structure



Reads on the fly

- The objective is to avoid multiple reads of the same data through a bus.
- Capturing data written by one processor on a memory bus by other processors (similar to cache injection).
- Synchronisation of the writing processor with reading processors is required.

Reads on the fly (cont.)

- Activities of reading processors:
  - deposing read requests to processor’s BRCs,
  - execution of a barrier,
  - reading data to the processor’s data cache during write operation on the memory bus.
- Activities of the writing processor:
  - barrier initialisation,
  - deposing the write request to the processor BRC.

We switch processors between clusters because:

- more processors can be needed to work in a target cluster on locally shared data,
- a processor can be supposed to carry data from one cluster to another for local use,
- a processor can be supposed to catch data in a cluster to be used in computation.

Processor switching is controlled by bus arbiters.

Communication on the fly

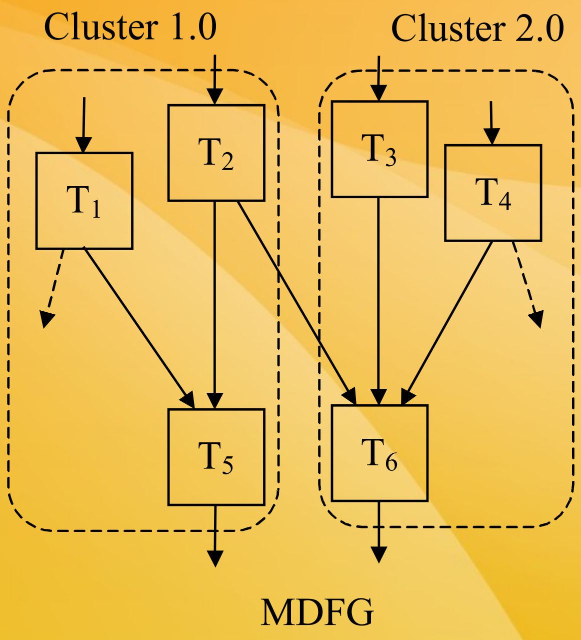
Communication on the fly is composed of :

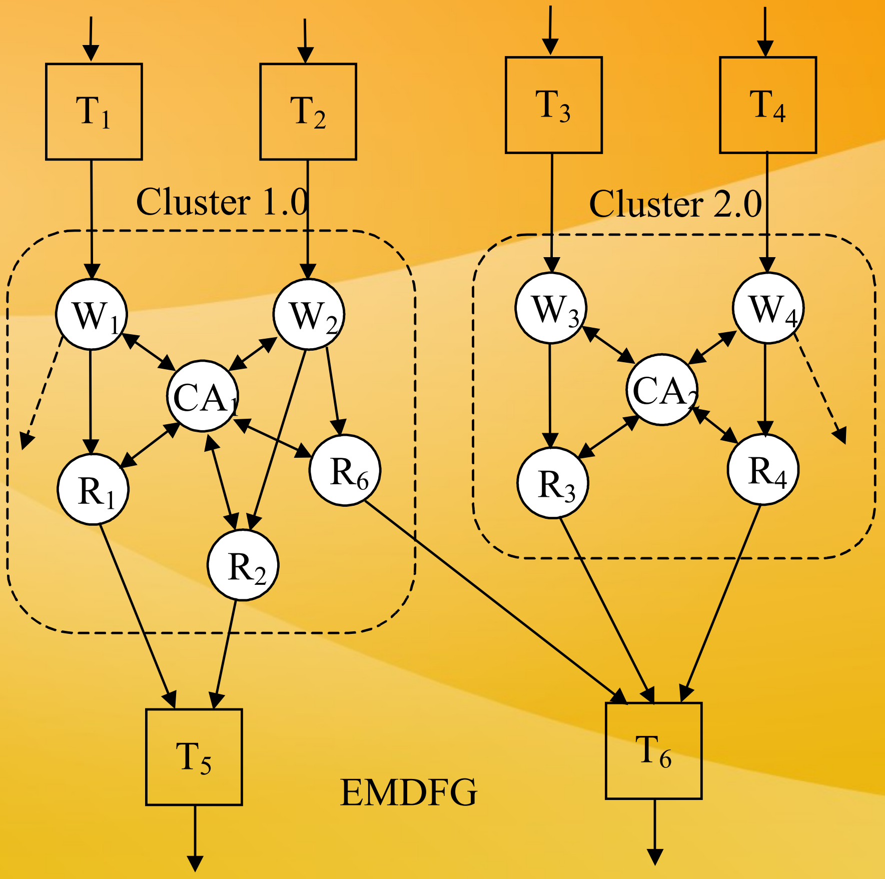
- processor switching into a cluster with its data cache contents,
- synchronisation of all reads with the write in the target cluster,
- data write and data reads on the fly in the cluster at the same time.

Program graph notation

- A program graph G=(V,E) consists of two sets of nodes: “standard” nodes (V_s) and “architectural” nodes (V_a), such that $V=V_s \cup V_a$, $V_s \cap V_a = \emptyset$.
- The edges correspond to data transmissions between the nodes from V_s and V_a or activation edges.
- The standard nodes are called program graph “glue nodes” since they usually provide computational interface between execution of subsequent architectural nodes of the graph.

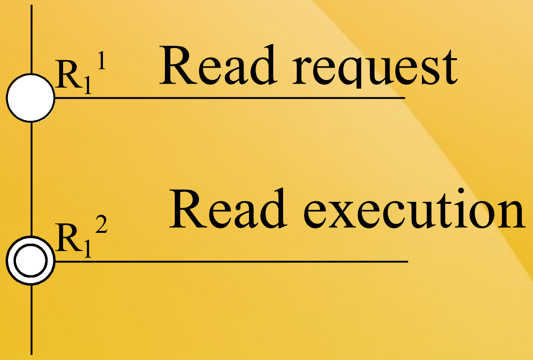
Graph representation of programs Extended Macro Data Flow Graph EMDFG

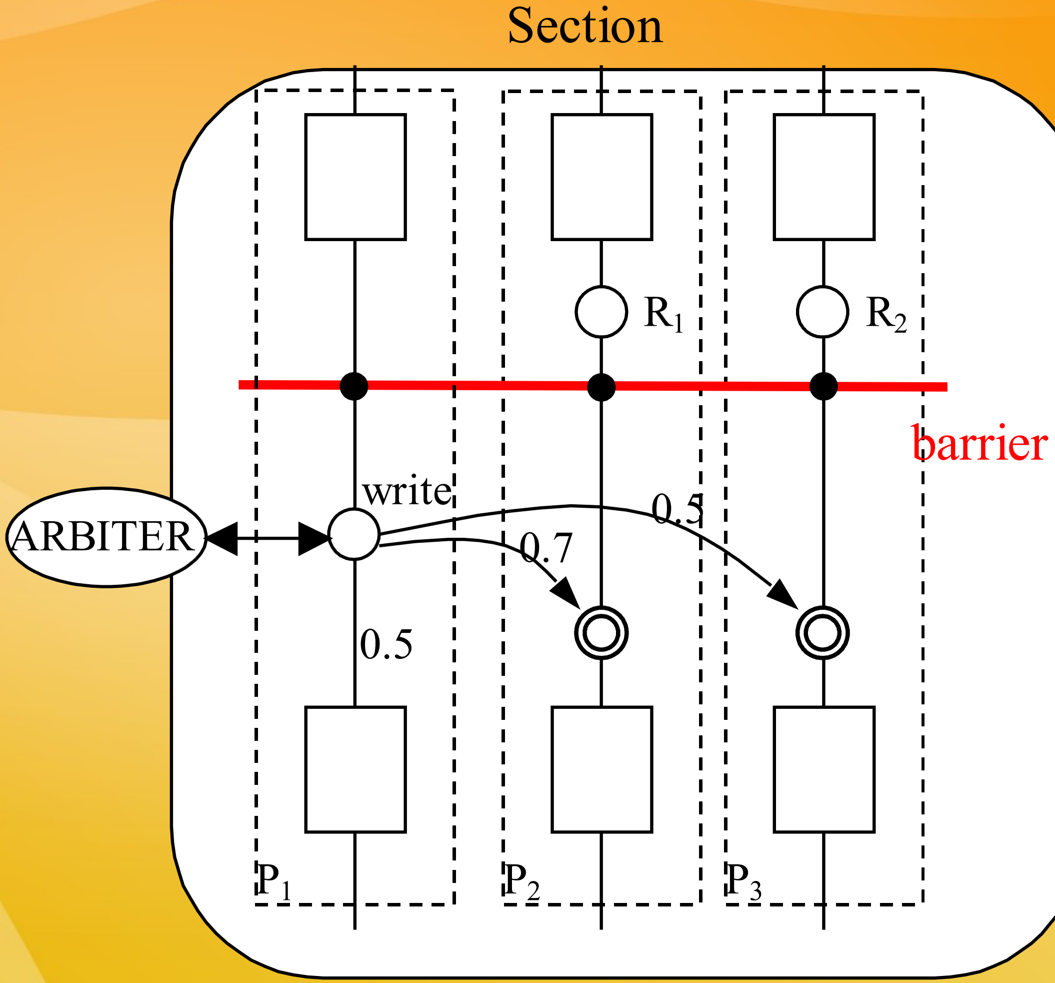




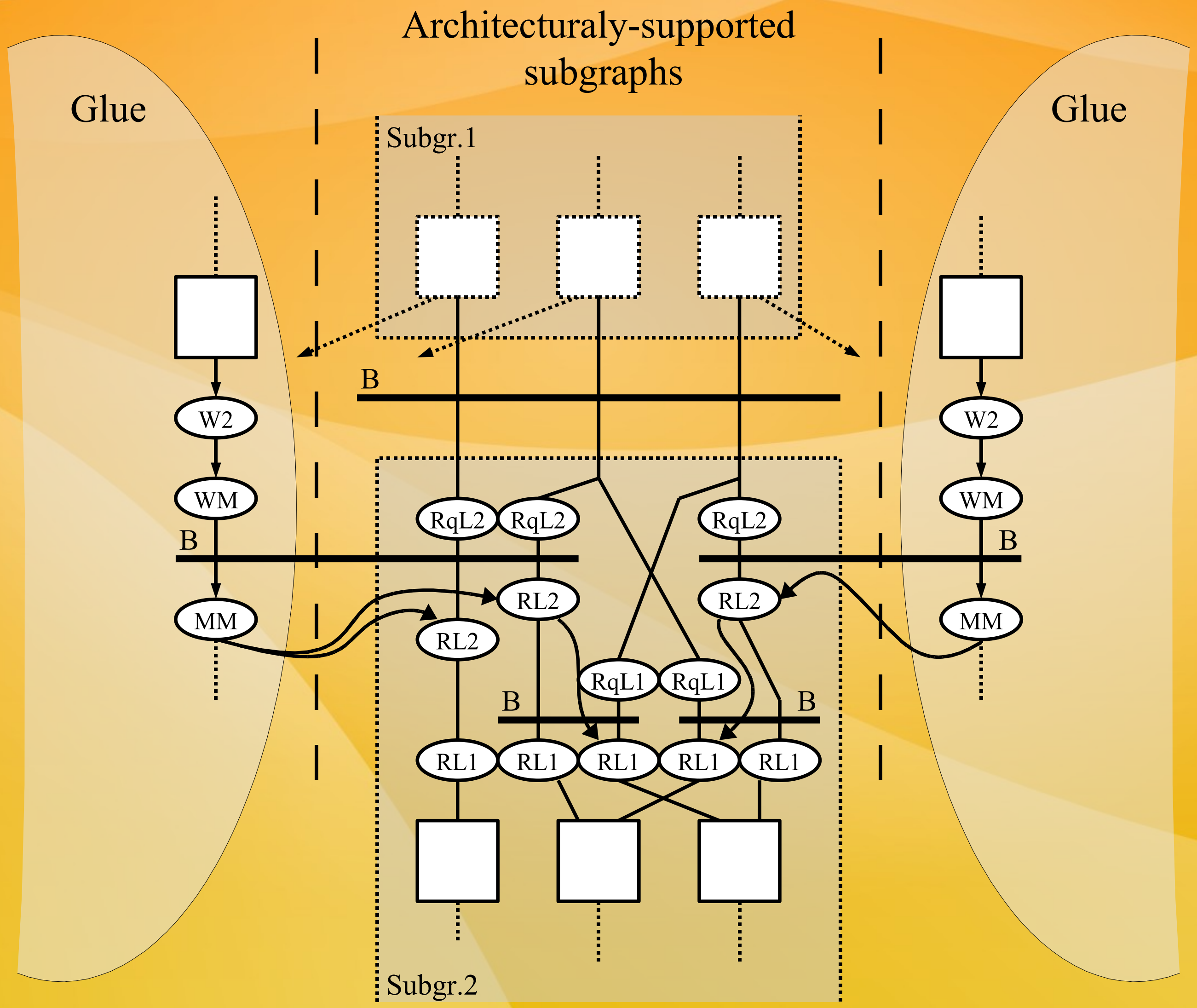
Graph representation of programs

Read on the fly





Initialization of an architecturaly-supported subgraph



The aim of the algorithm

- The algorithm schedules standard nodes to “general purpose” CMPs (GCMPs) and architectural nodes to architectural CMPs (ACMPs).
- It aims at minimal program execution time by equal loads of all available resources.
- A program graph ideal execution:
  - The program execution progress in standard and architectural CMP nodes is similar.
  - Both kinds of nodes are evenly distributed across a program graph.

Notation (cont.)

- A standard graph node corresponds to a classical, single-processor task in a macro dataflow program graph. It will be basically executed by one of processors in a GCMP module in a system. Such node can be also executed by a processor from an architectural CMP if necessary.
- Architectural graph nodes correspond to parallel subgraphs, for which a programmer denotes, that they are „regular” subgraphs and which should be executed using one of ACMPs in a system.

Outline of the algorithm

The algorithm is based on list scheduling with the ETF heuristics with additional priorities of graph nodes.

- The priorities decide on the order in which program graph nodes are taken into account by the scheduling algorithm.

Two types of priorities (1st and the 2nd level) are defined.

- The 1st level priorities are assigned to standard and architectural graph nodes, which are equally distant in paths from the beginning of the program graph.
- The 2nd level priorities are assigned to architectural nodes which are strongly bound by the activating glue nodes.

Priority notation

- The first level priority of a node v (denoted as $pr_1(v)$) is based on topological properties of the graph.
- The second level of priorities (denoted as $pr_2(v)$) is introduced to distinguish the nodes, which have the same 1st – level prioririty.
- For two nodes u and v (both must be either architectural or standard), $pr(u)<pr(v) \Leftrightarrow pr_1(u)<pr_1(v) \vee (pr_1(u)=pr_1(v) \wedge pr_2(u)<pr_2(v))$.

1st – level priority

The 1st – level priority aims at division of a set of architectural nodes into layers used to schedule the program nodes in the breadth-first-way.

- Each layer contains a subset of nodes, which are pairwise independent, i.e. there is no data dependency between any two of them.
- The scheduling algorithm tries to schedule nodes layer-by-layer.
- The layers are created using program graph paths analysis.

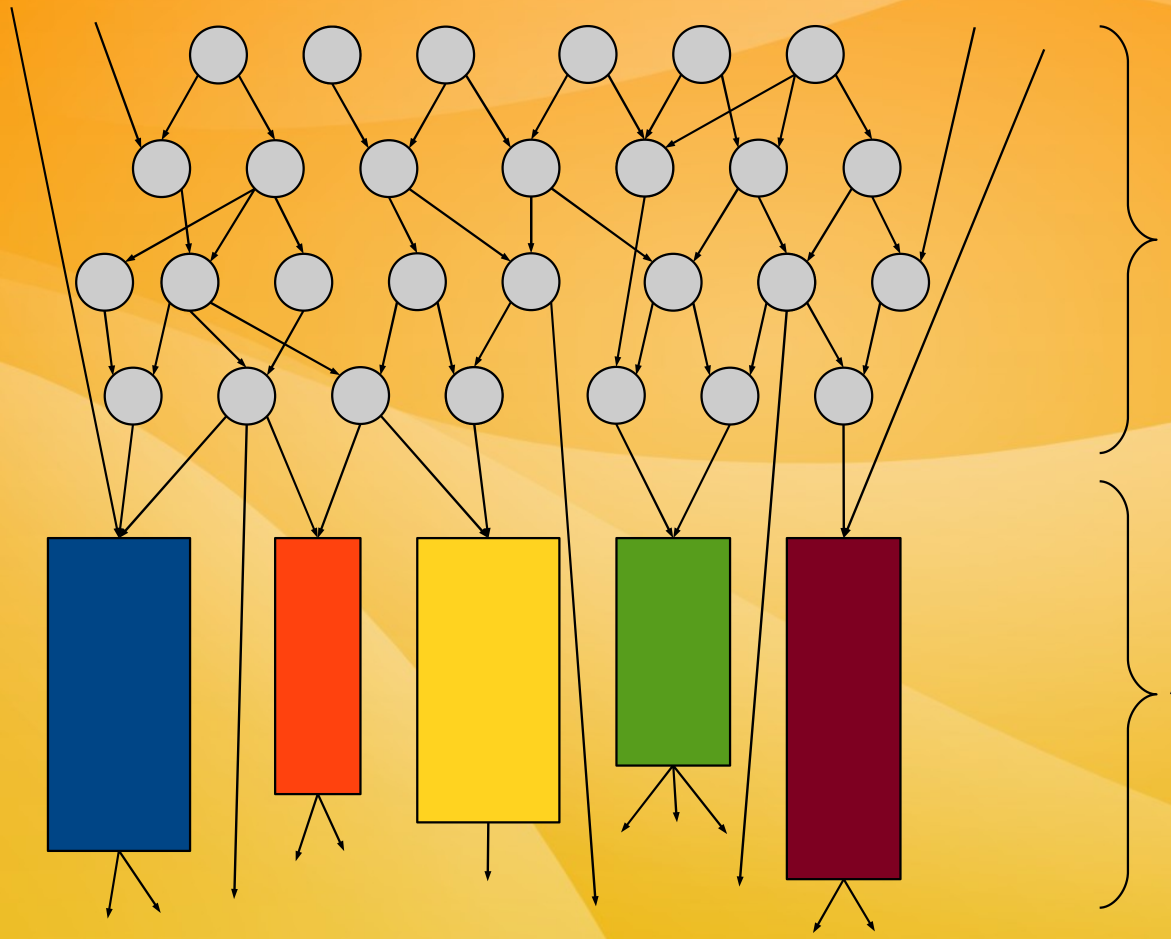
Assignment of the 1st – level priorities

- To compute 1st – level priorities, an "architectural task graph" $G_r=(V_r,E')$ is defined:
  - nodes correspond to regular tasks in graph G.
  - For two nodes $u,v \in V_r$, an edge $u \rightarrow v$ exists in G_r, if there is a directed path between these two nodes in original graph G such that this path contains only standard nodes.
- For each $u \in V_r$, priority of node u is equal to its depth in graph G_r (the number of nodes on the longest path leading to this node from one of the nodes which have no predecessors in G_r).

Priorities for standard nodes

- Priorities for standard nodes depend on priorities of architectural nodes.
- For each $v \in V_s$, we determine a set $X \subseteq V_a$ of nodes such that there exists a path from node v to each of these nodes. If X is not empty, the priority of node v is equal to minimal priority over the nodes from X and is equal to $\max(pr_1(u \in V_r))+1$ otherwise.

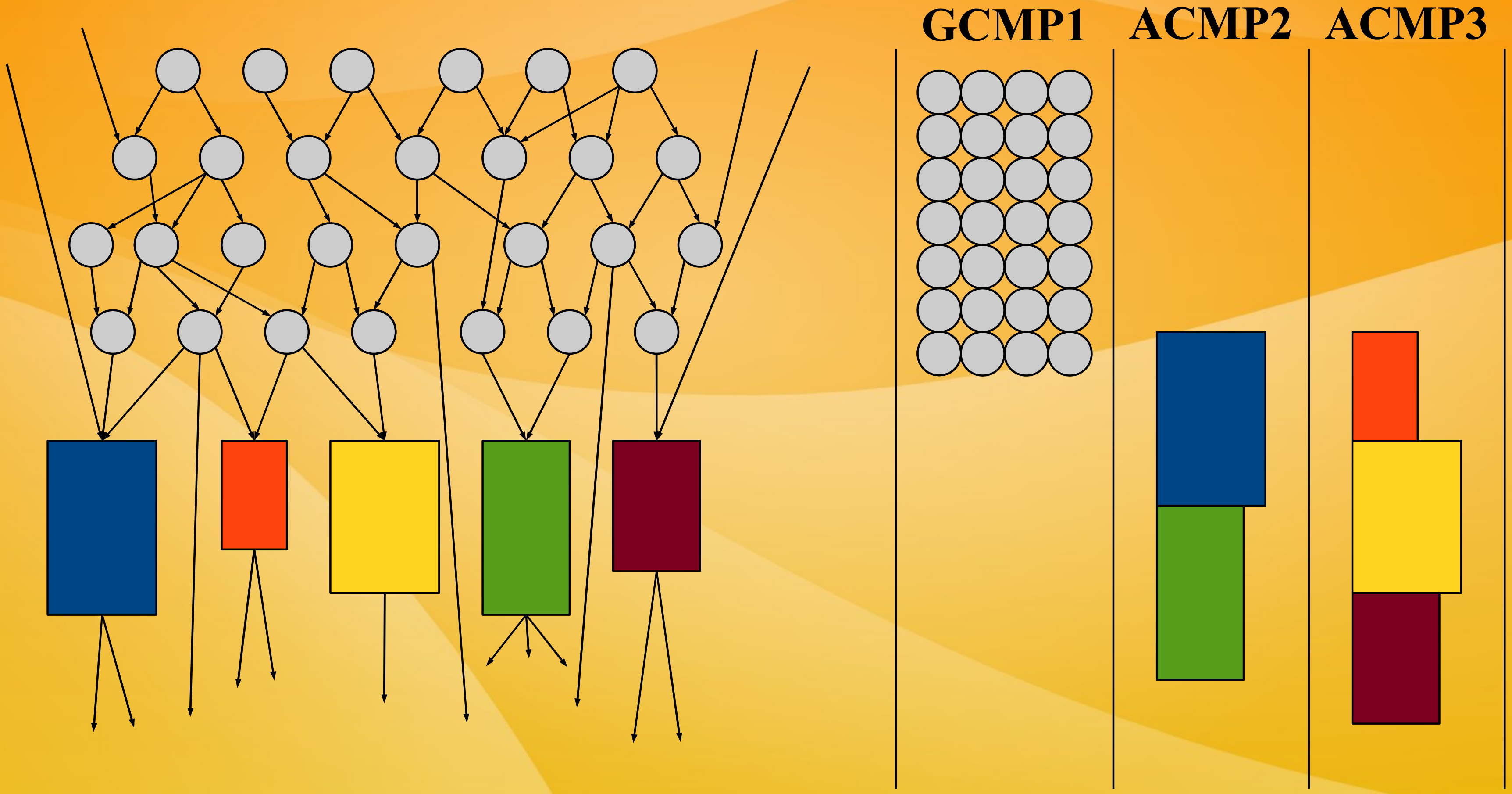
A layer of nodes



STANDARD TASKS („GLUE”)

ARCHITECTURAL TASKS

Standard ETF scheduling



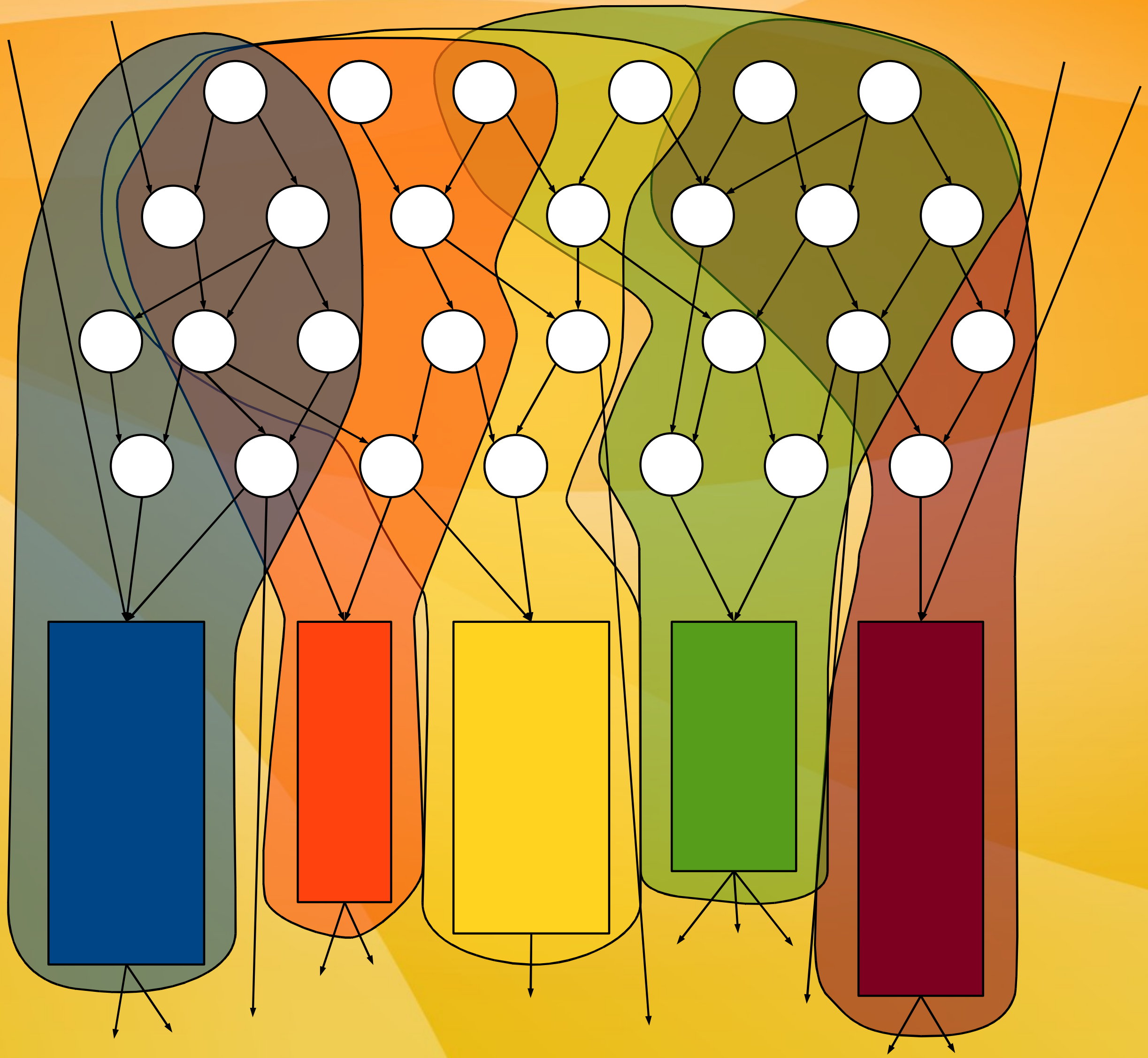
2nd – level priority asignment

- The 2nd – level priority aims at division of nodes in layers determined by the 1st – level priority, into subsets in such way, that we can obtain equal, high load of all computational resources in the system.
- Each node subset contains no more architectural nodes, than the number of ACMPs in the system dedicated for parallel execution of architectural nodes of the program graph.

Assignment of the 2nd – level priorities (notation)

- U be a set of architectural nodes, which have the same 1st-level priority. Let $p=0$ be the first value of 2nd priority for this set.
- For each u from U we define X_u as a set of standard nodes v with the same 1st-level priority as u, such, that there is a directed path from v to u, and which doesn't have a 2nd -level priority assigned yet.

How sets X_u are defined



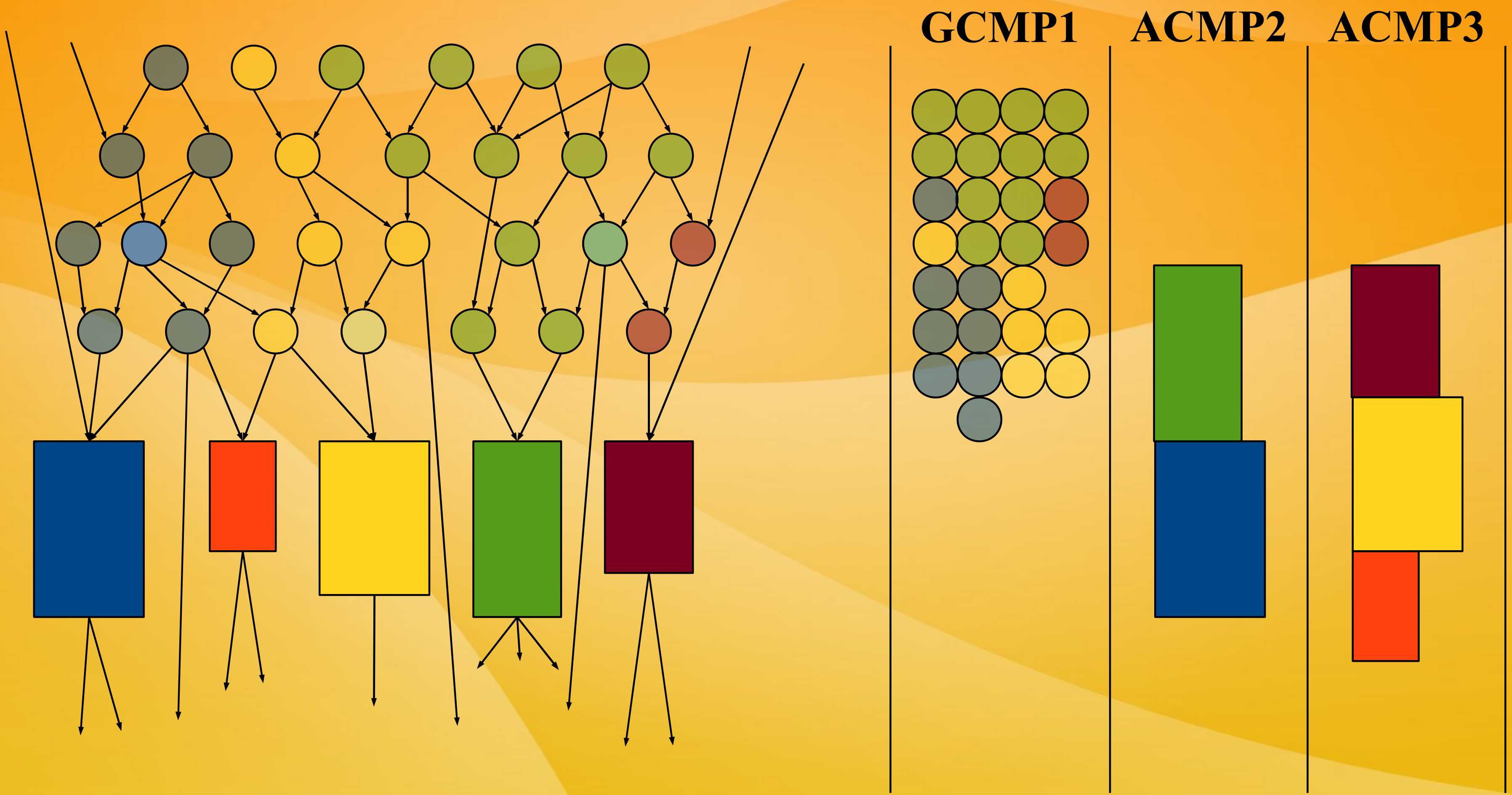
Assignment of the 2nd – level priorities

```
Let $p=0$ be the first value of priority for this set.
While U is not empty {
    Determine $X_u$ sets for all nodes from U.
    Determine a subset V of architectural nodes from U chosen to be assigned priority p and $X_V$ as a sum of sets $X_v$ for all $v \in V$.
    Assign $pr_2(u)=p$ for all nodes u from $V \cup X_V$.
    Remove architectural nodes from V from set U.
    Let $p=p+1$
}
```

Selection of nodes to the V subset

```
Let $V=\emptyset$ and $X_V=\emptyset$
while (|V| is smaller then the number of resources for architectural nodes) {
    if V is empty {
        for all tasks u from U
            { Schedule a subgraph $X_u$ on available resorces dedicated for execution of standard nodes, using ETF-based list scheduling. }
        Select such node u from U, for which its $X_u$ set gives the shortest schedule in the previous step.
    } else
        { Select node u from U such, that $X_u \cap X_V$ is the biggest. }
     Let $V=V\cup\{u\}$ and $X_V=X_V\cup X_u$
}
```

Result of scheduling with 2nd – level priorities assigned



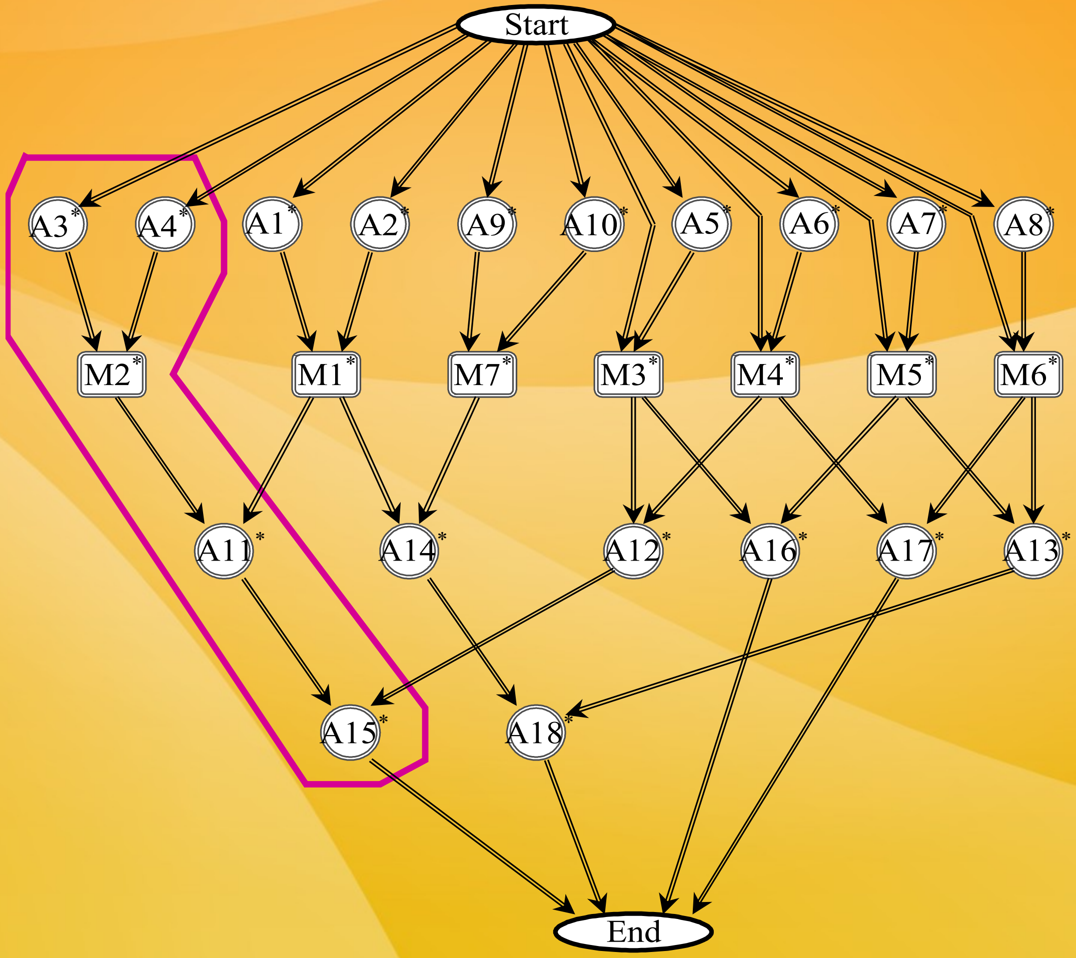
Strassen’s matrix multiplication

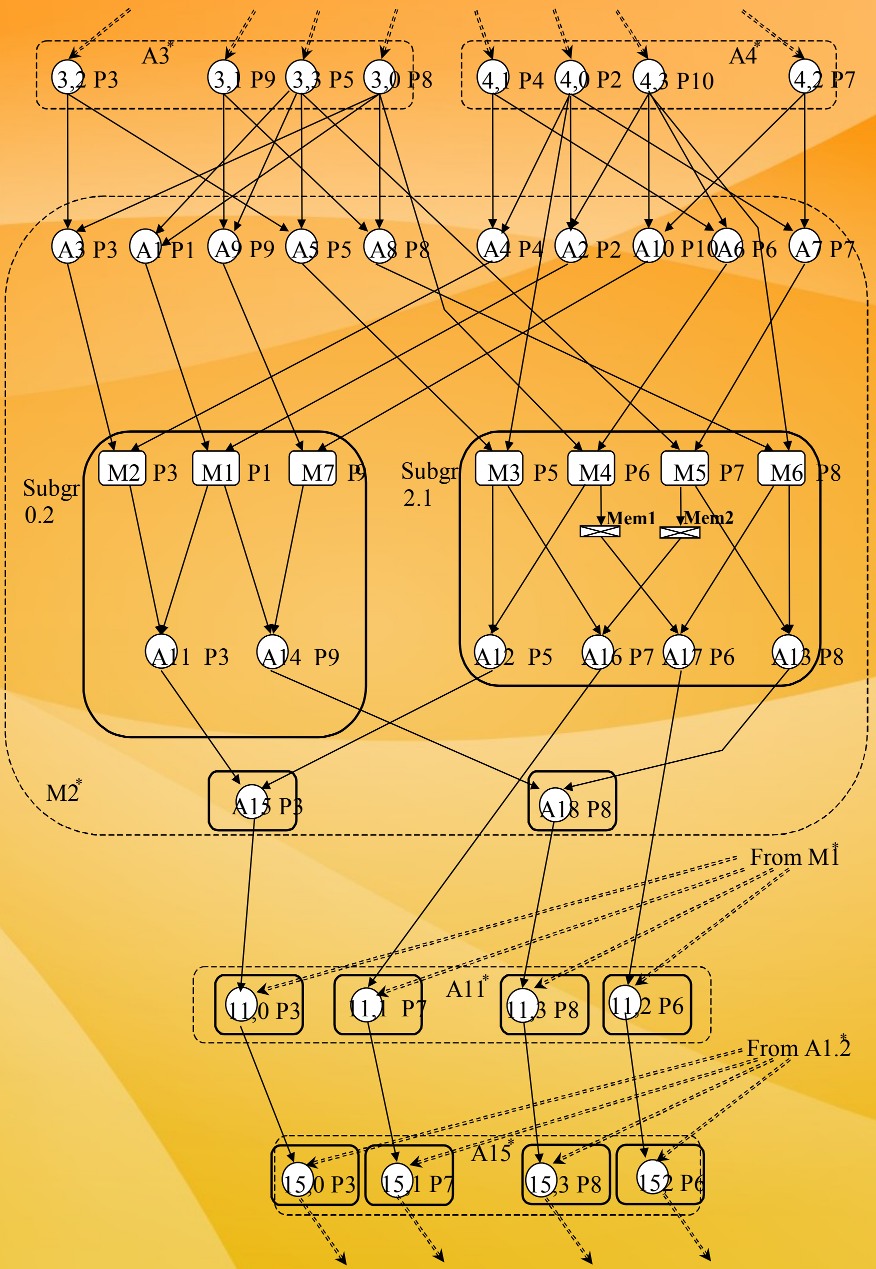
$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \bullet \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

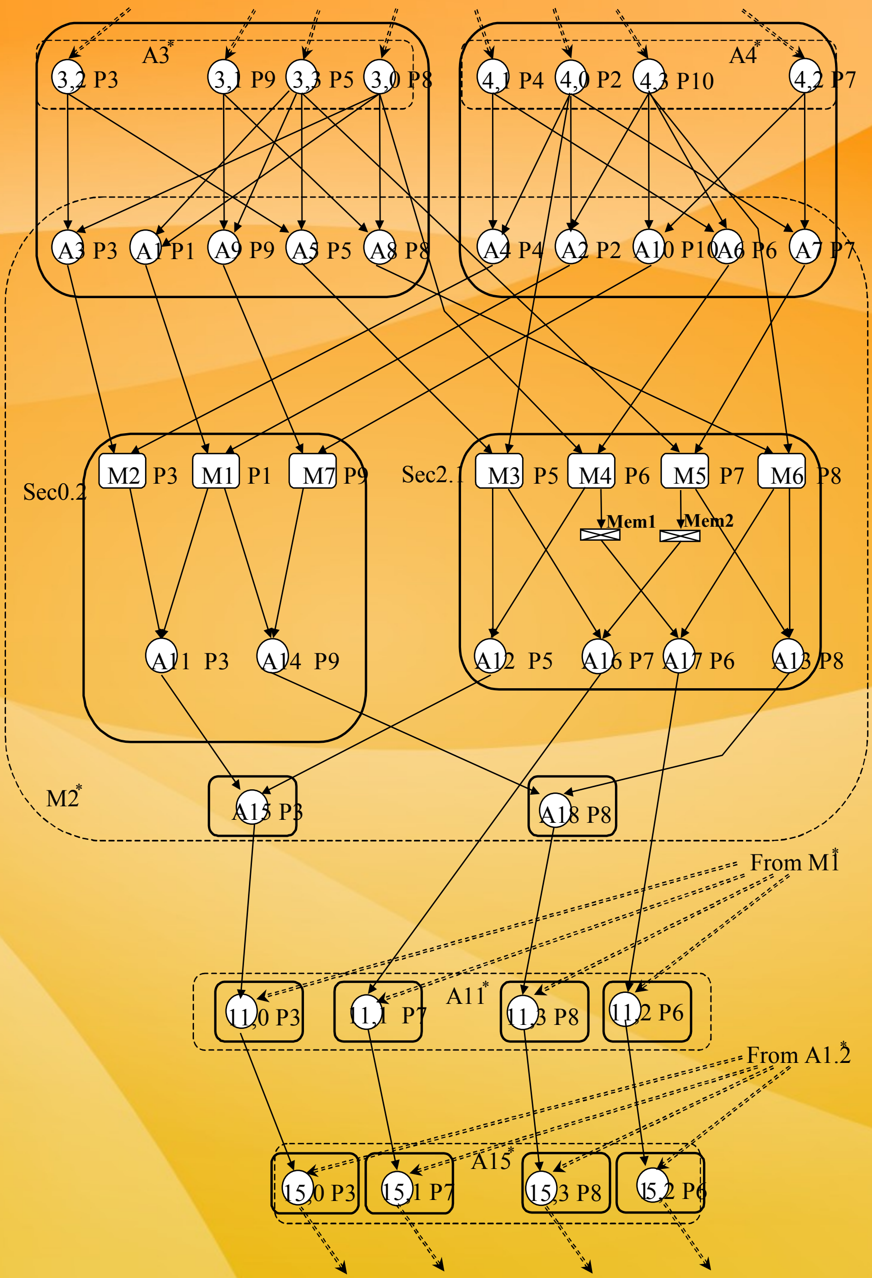
$$M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22}) \cdot B_{11}$$
$$M_3 = A_{11} \cdot (B_{12} - B_{22})$$
$$M_4 = A_{22} \cdot (B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12}) \cdot B_{22}$$
$$M_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{21} = M_2 + M_4$$
$$C_{12} = M_3 + M_5$$
$$C_{22} = M_1 + M_3 - M_2 + M_6$$

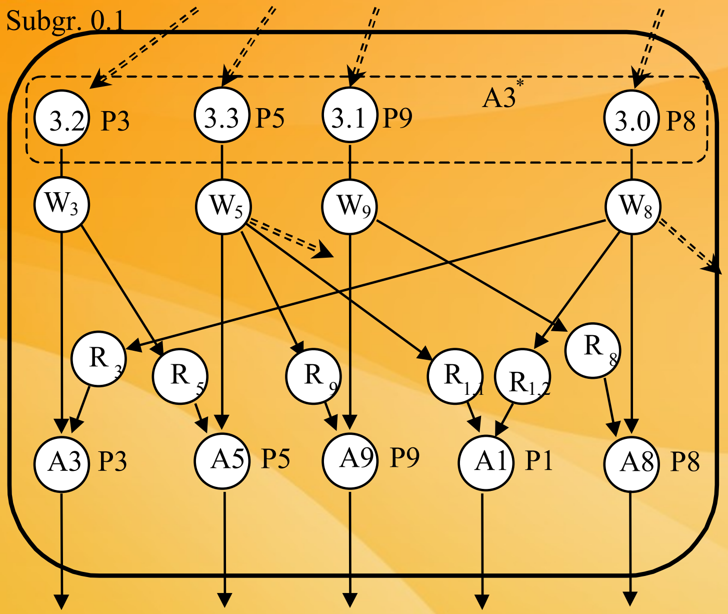
Numerical examples: Strassen's matrix multiplication

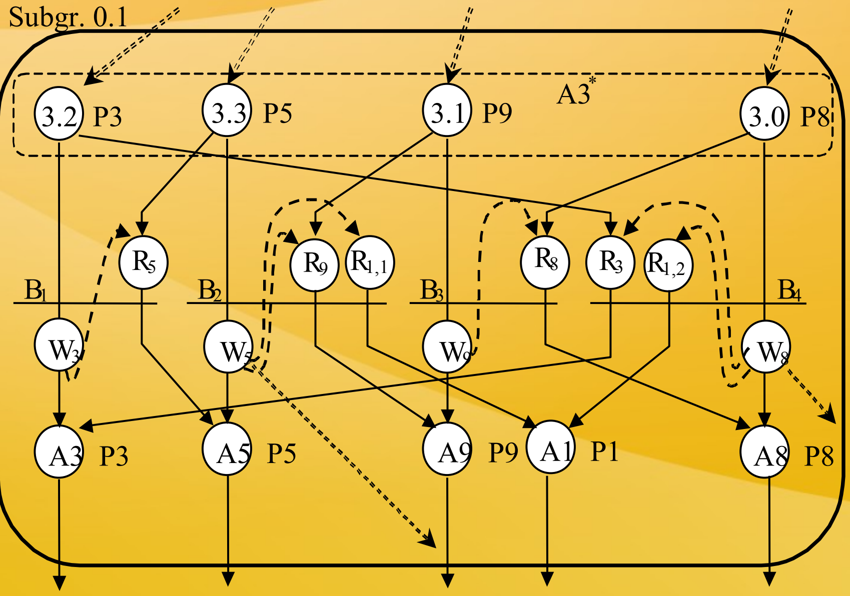



$A3^*$
3,2 P3
3,1 P9
3,3 P5
3,0 P8
4,1 P4
4,0 P2
4,3 P10
$A4^*$
4,2 P7
A3 P3
A1 P1
A9 P9
A5 P5
A8 P8
A4 P4
A2 P2
A10 P10
A6 P6
A7 P7
Subgr 0.2
M2 P3
M1 P1
M7 P9
Subgr 2.1
M3 P5
M4 P6
M5 P7
M6 P8
Mem1
Mem2
A11 P3
A14 P9
A12 P5
A16 P7
A17 P6
A13 P8
$M2^*$
A15 P3
A18 P8
From $M1^*$
11,0 P3
11,1 P7
$A11^*$
11,3 P8
11,2 P6
From $A1.2^*$
$A15^*$
15,0 P3
15,1 P7
15,3 P8
152 P6

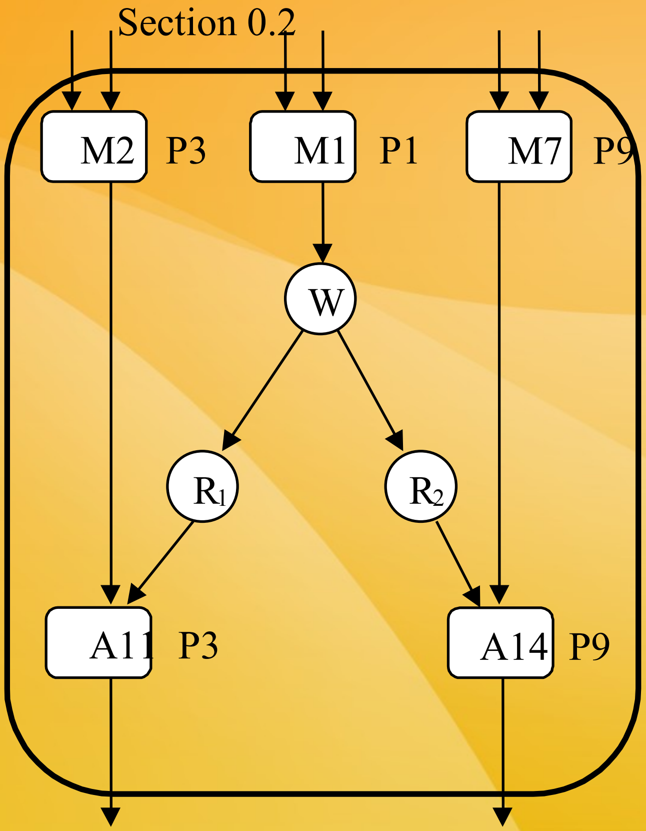

A3*
3,2 P3
3,1 P9
3,3 P5
3,0 P8
A4*
4,1 P4
4,0 P2
4,3 P10
4,2 P7
A3 P3
A1 P1
A9 P9
A5 P5
A8 P8
A4 P4
A2 P2
A10 P10
A6 P6
A7 P7
Sec0.2
M2 P3
M1 P1
M7 P9
Sec2.1
M3 P5
M4 P6
M5 P7
M6 P8
Mem1
Mem2
A11 P3
A14 P9
A12 P5
A16 P7
A17 P6
A13 P8
M2*
A15 P3
A18 P8
From M1*
11,0 P3
11,1 P7
A11*
11,3 P8
11,2 P6
From A1.2*
A15*
15,0 P3
15,1 P7
15,3 P8
15,2 P6

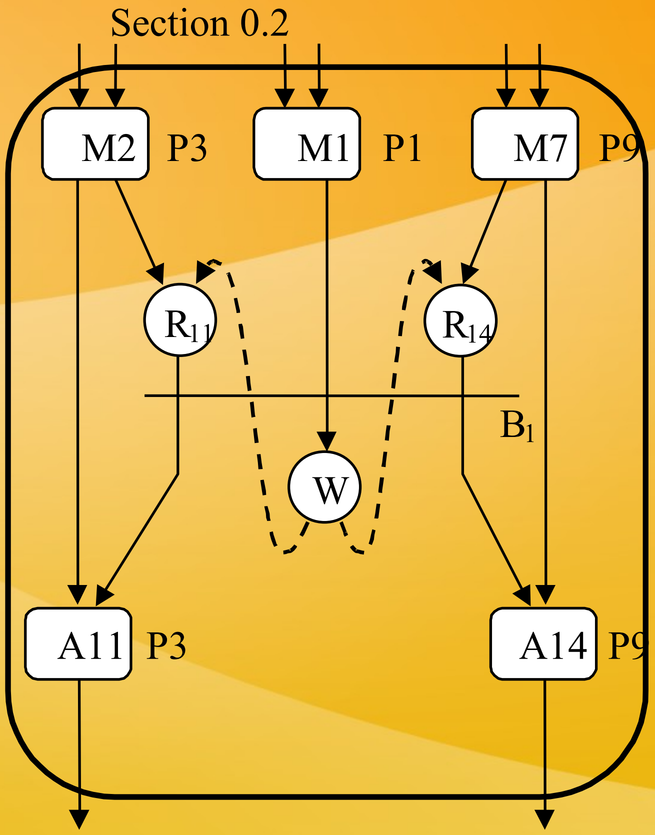
Numerical examples: Strassen's matrix multiplication – graph transformations



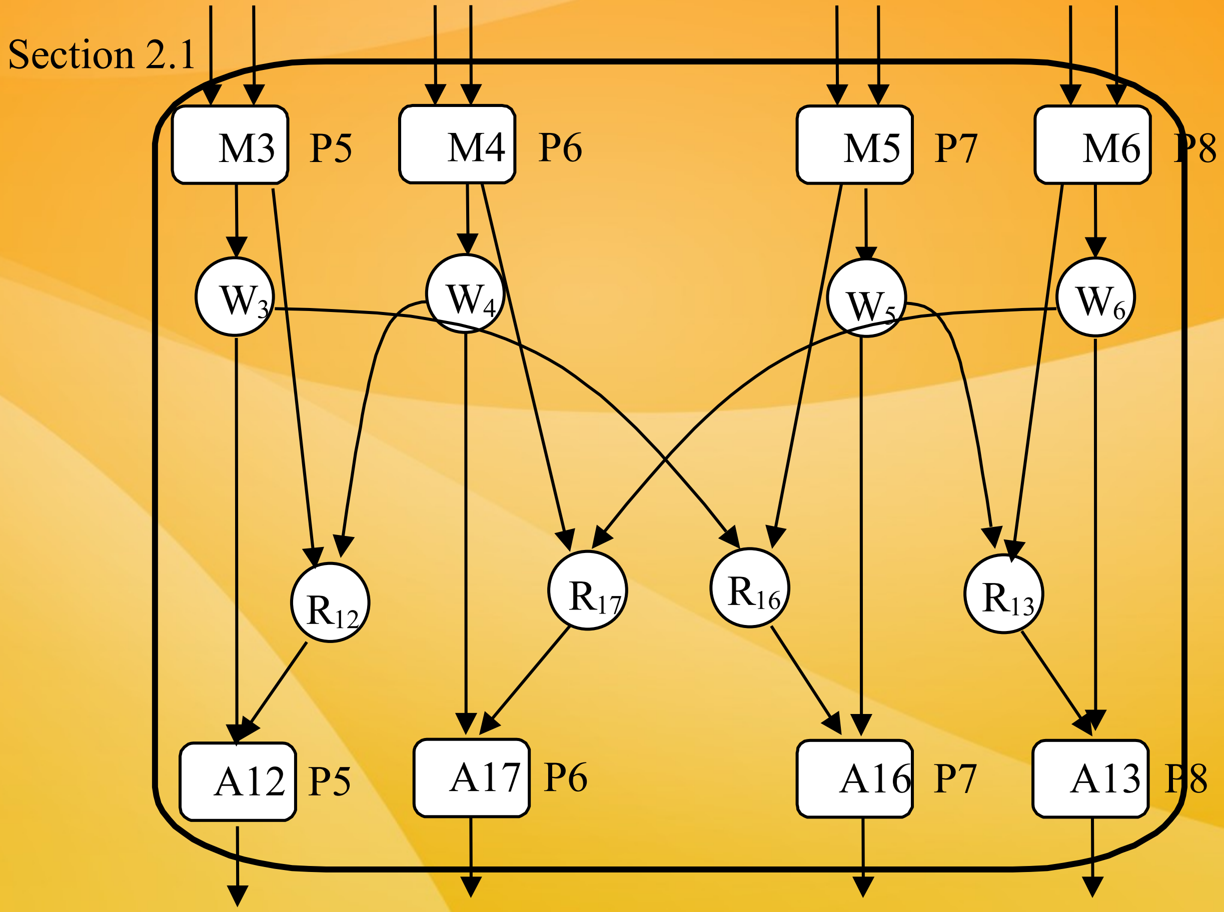


Numerical examples: Strassen’s matrix multiplication – graph transformations

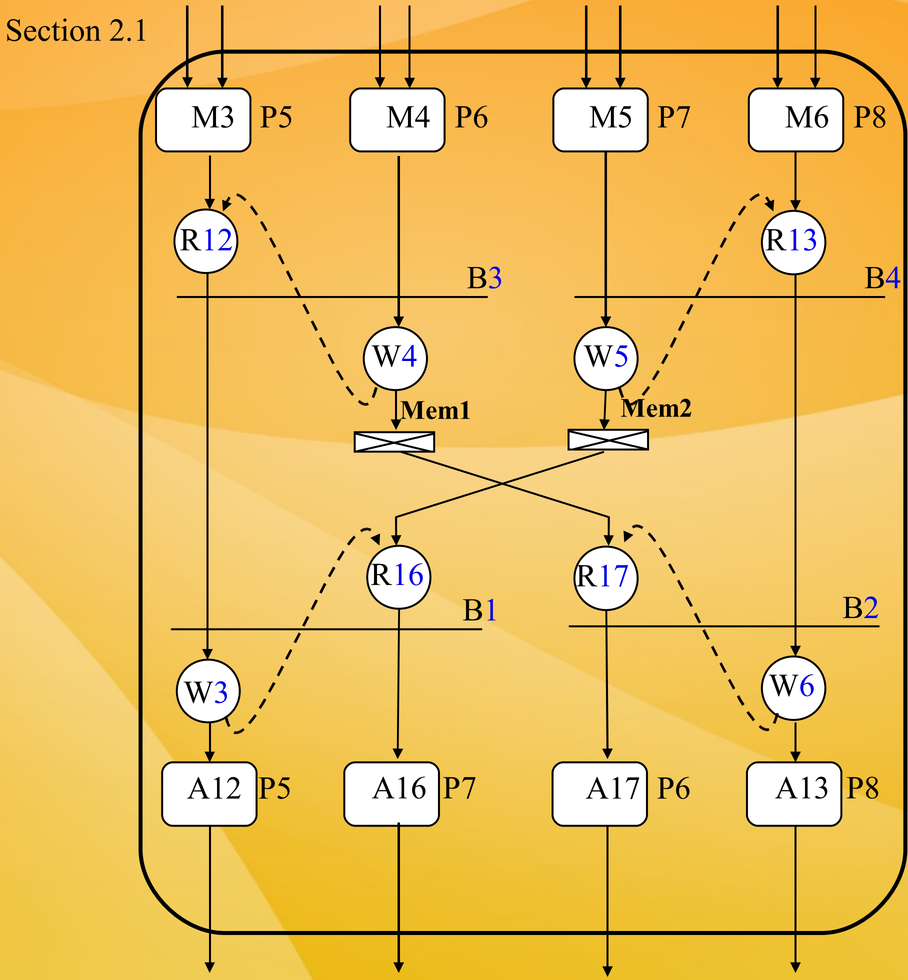




Numerical examples: Strassen's matrix multiplication – graph transformations



Numerical examples: Strassen's matrix multiplication – graph transformations



Speedup and efficiency for parallel Strassen's matrix multiplication of the size 256 in multiple CMPs

recursion level	processors used	matrix size per processor	architecture variant	communication to computation ratio 1 to 2	1 to 4	1 to 8
				Speedup		
1	10	128	A	6,59	6,25	5,70
			B	6,52	6,14	5,52
			C	4,89	3,77	2,62
2	70	64	A	30,19	22,12	14,55
			B	28,91	20,79	13,43
			C	18,72	13,56	8,21
				Efficiency		
1	10	128	A	65,9%	62,5%	57,0%
			B	65,2%	61,4%	55,2%
			C	48,9%	37,7%	26,2%
2	70	64	A	43,1%	31,6%	20,8%
			B	41,3%	29,7%	19,2%
			C	26,7%	19,4%	11,7%

A: 10 busses per CMP, with processor switching and data transfers on the fly.

B: 10 buses per CMP, with processor switching, without transfers on the fly.

C: 1 bus perCMP, no processor switching, no data transfers on the fly.

Speedup and efficiency for parallel Strassen’s matrix multiplication of the size 2048 in multiple CMPs

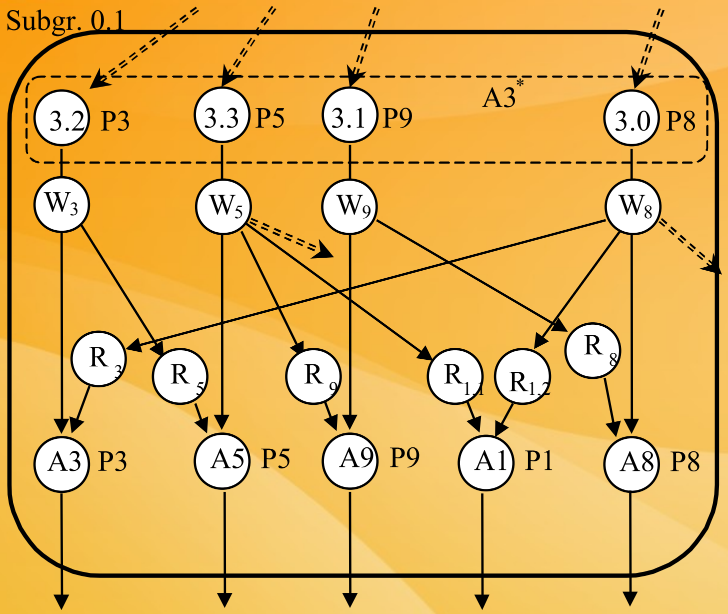
recursion level	processors used	matrix size per processor	architecture variant	communication to computation ratio 1 to 2	1 to 4	1 to 8
				Speedup		
1	10	1024	A	6,92	6,85	6,71
			B	6,91	6,82	6,66
			C	6,48	6,03	5,30
2	70	512	A	43,94	39,96	33,86
			B	43,42	39,12	32,66
			C	38,74	23,82	21,22
				Efficiency		
1	10	1024	A	69,2%	68,5%	67,1%
			B	69,1%	68,2%	66,6%
			C	64,8%	60,3%	53,0%
2	70	512	A	62,8%	57,1%	48,4%
			B	62,0%	55,9%	46,7%
			C	55,3%	34,0%	30,3%

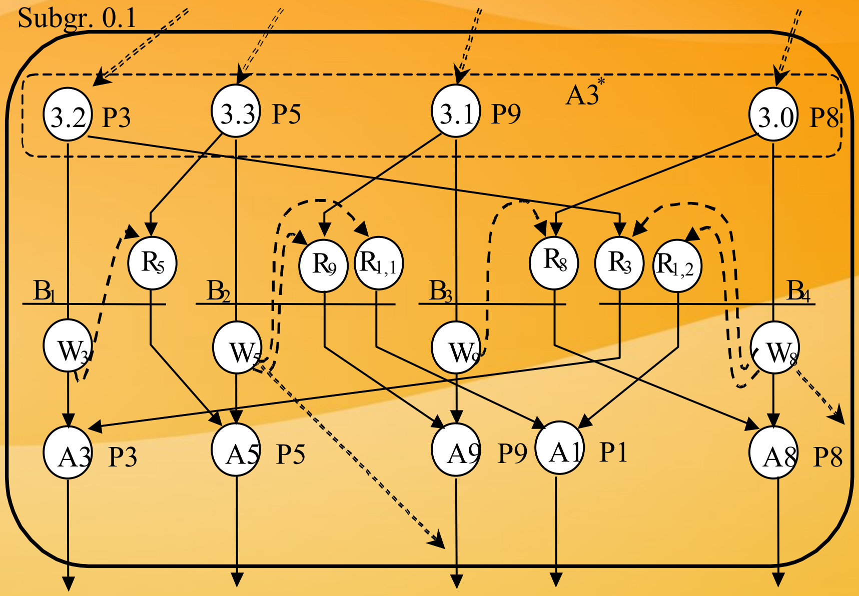
A: 10 busses per CMP, with processor switching and data transfers on the fly.

B: 10 buses per CMP, with processor switching, without transfers on the fly.

C: 1 bus perCMP, no processor switching, no data transfers on the fly.

Numerical examples: Strassen’s matrix multiplication – graph transformations





- Execution with architecture-supported CMP modules is 5-6 times faster, depending on the speed of the local communication.

Conclusions

- The proposed architecture with dynamic shared memory clusters, communication on the fly and dual-ported data caches is efficient for communication in fine grain and coarse grain parallel computations.
- Communication on the fly can eliminate many transactions on intra-cluster and inter-cluster data transmission networks.
- In the matrix multiplication example, all standard communication could be replaced by communication and reads on the fly, also communication through the global network could be eliminated .

Conclusions (cont.)

- Parallelization efficiency from 0.88 to 0.68 was observed for execution of small size matrix 64x64 multiplication with fine parallel grain determined by 32x32 to 8x8 serial multiplications.
- Parallelization efficiency from 0.88 to 0.59 was observed for execution of larger size matrix 2048x2048 multiplication with parallel grain size from 1024x1024 to 128x128.
- That confirms high potential of the proposed solutions for designing parallel systems for numerical computations of synchronous character.
- The proposed architecture is convergent with current technology trends.