The Plural Architecture

Shared Memory Many-core with Hardware Scheduling

Ran Ginosar

Technion, Israel

January 2012

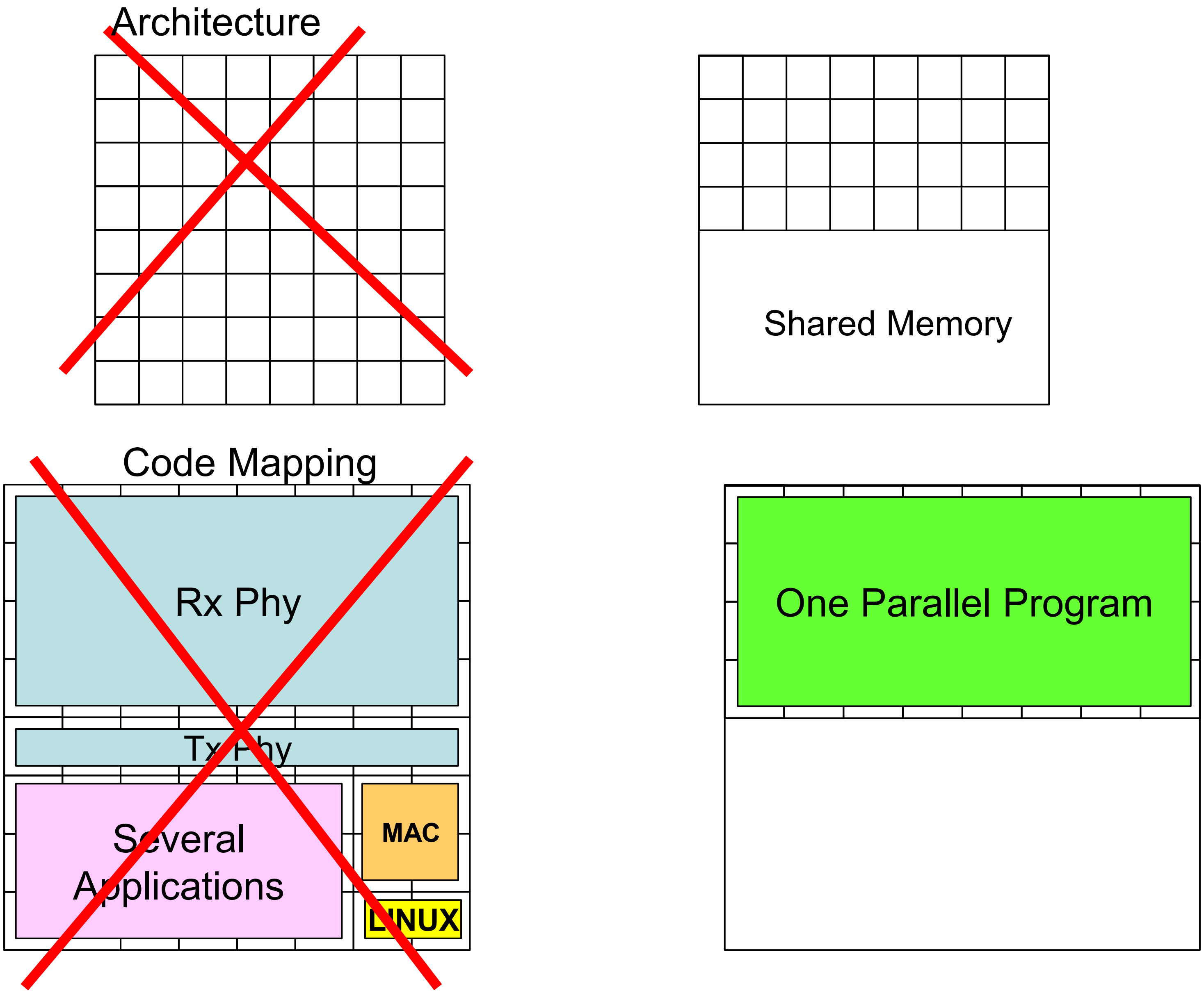
Outline

- Plural architecture and programming
- Mathematical model
- Examples

many-cores

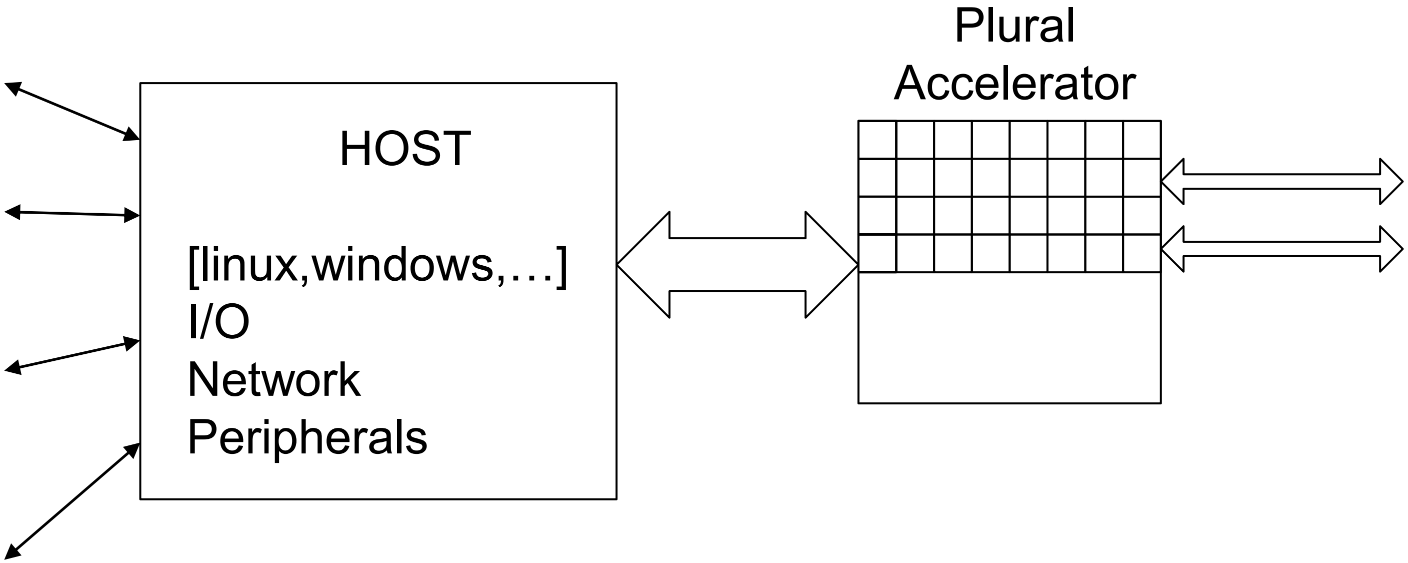
- Many-core is:
  - a single chip
  - with many (?) cores and on-chip memory
  - running a single (parallel) program at a time, solving one problem
  - an accelerator
- Many-core is NOT:
  - Not a “normal” multi-core
  - Not running an OS
- Contending many-core architectures
  - Shared memory (the Plural architecture, XMT)
  - Tiled (Tilera, Godson-T)
  - Clustered (Rigel)
  - GPU (Nvidia)
- Contending programming models

Plural shared memory architecture



Context

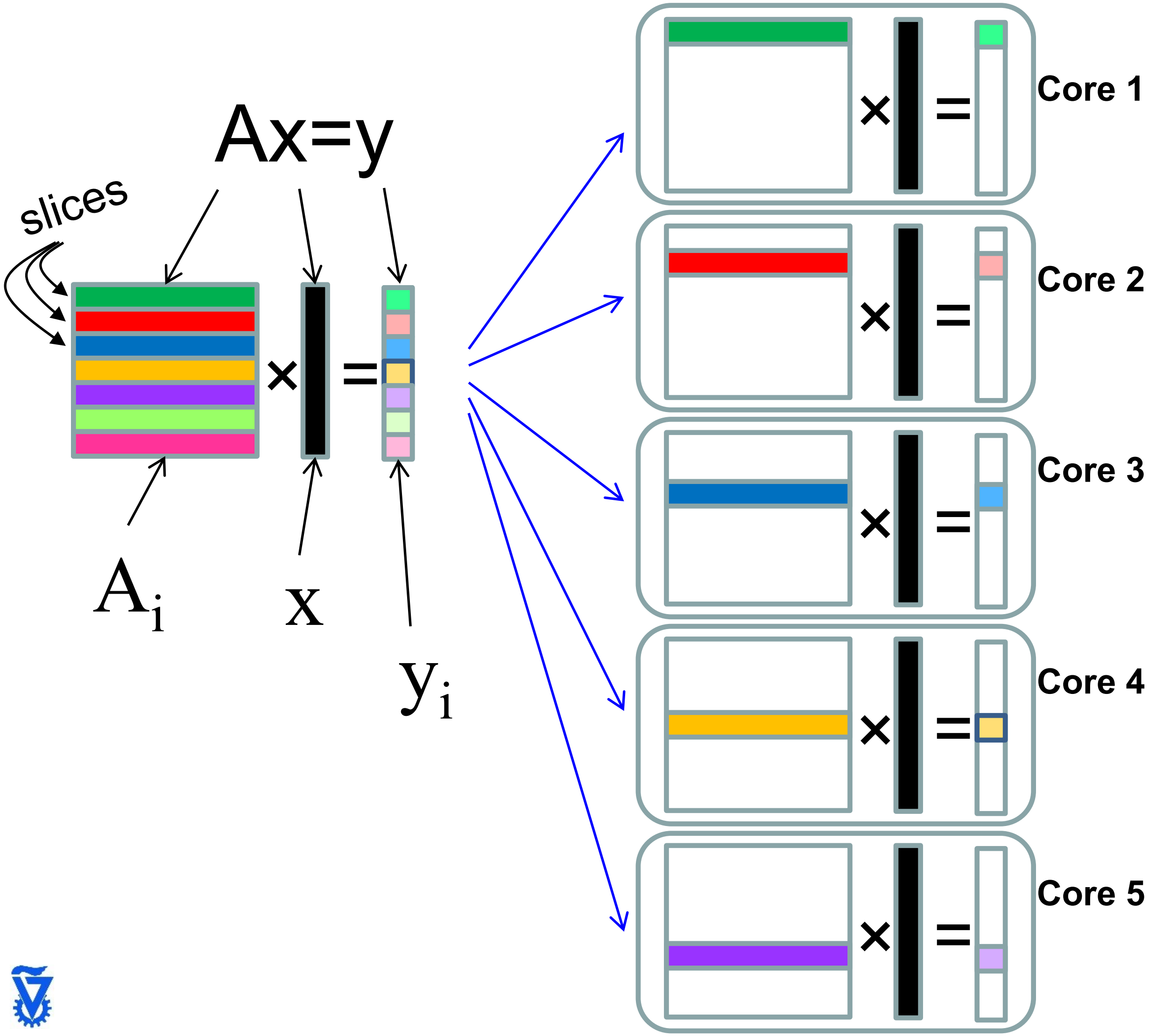
- Plural: homogeneous acceleration for heterogeneous systems



One (parallel) program ?

- Plural programming model: de-synchronized PRAM
  - Manages all cores as a single shared resource
  - Manages all memory as a single shared resource
- Plural programming model is *NOT* CSP
  - CSP [Hoare 1978], a.k.a. message-passing, means long processes
    - Plural PM is fine grain tasks
  - CSP blocks on communications and synchronization
    - Plural PM has no communications (only shared memory)
    - Plural PM kills when synchronization is needed

PRAM matrix-vector multiply



The PRAM algorithm

i is core index
AND slice index

Begin

$$y_i = A_i x$$

End

A,x,y in shared memory
(Concurrent Read of x)

Temp are in private memories (e.g. computing actual addresses given *i*)

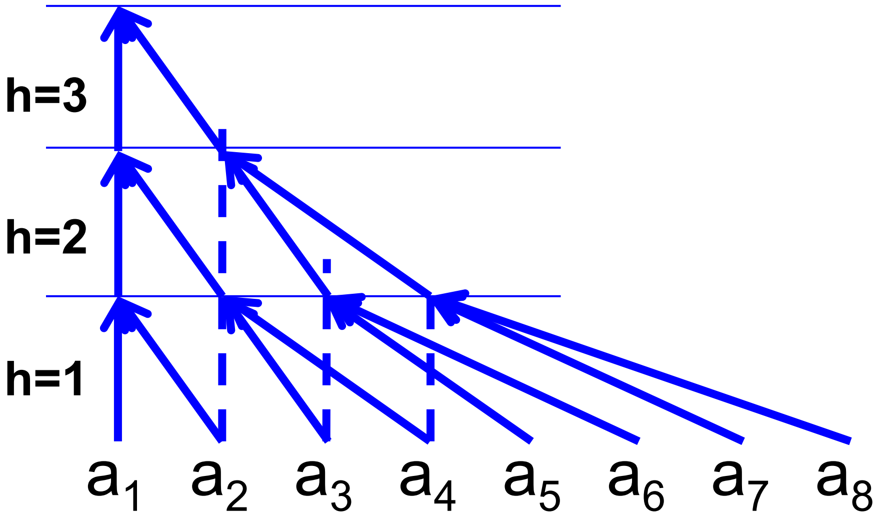
PRAM logarithmic sum

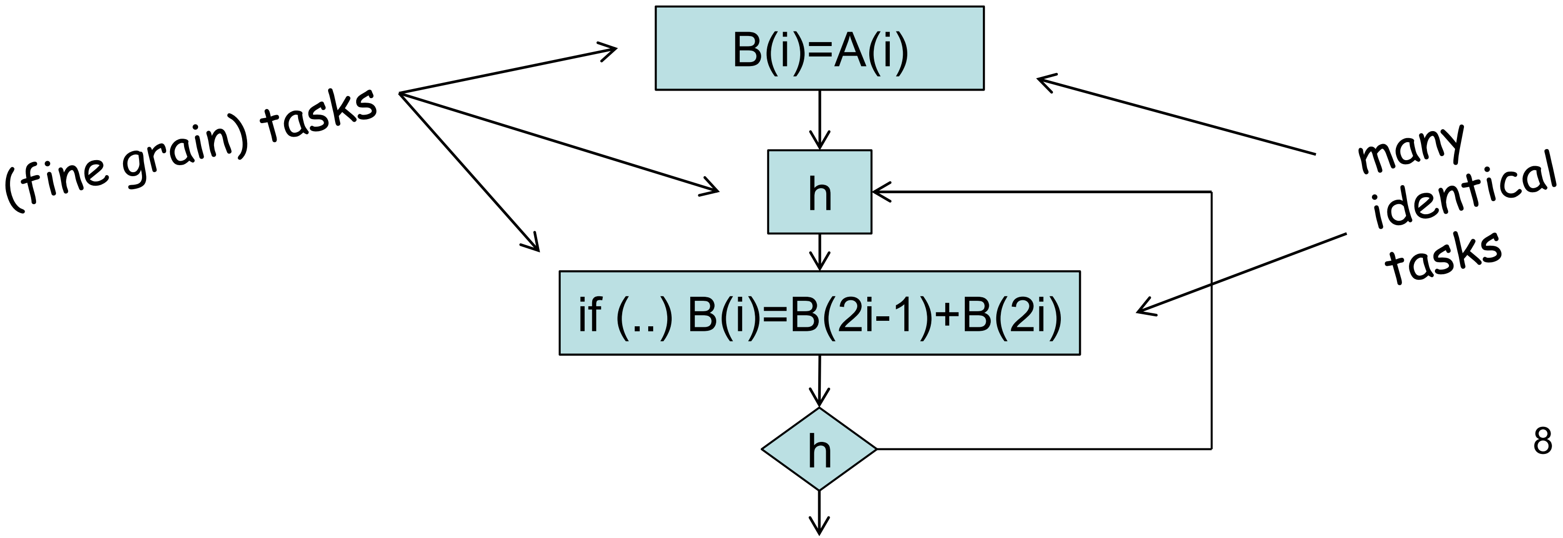
The PRAM algorithm

// Sum vector A(*)

```
Begin
    B(i) := A(i)
    For h=1:log(n)
        if $i \le n/2^h$ then
            B(i) = B(2i-1) + B(2i)
End
```

// B(1) holds the sum





B(i)=A(i)

h

if (..) B(i)=B(2i-1)+B(2i)

h

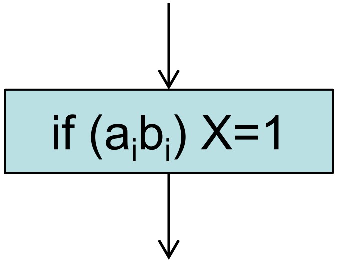
(fine grain) tasks

many identical tasks

PRAM SoP: Concurrent Write

- Boolean $X=a_1b_1+a_2b_2+\dots$
- The PRAM algorithm

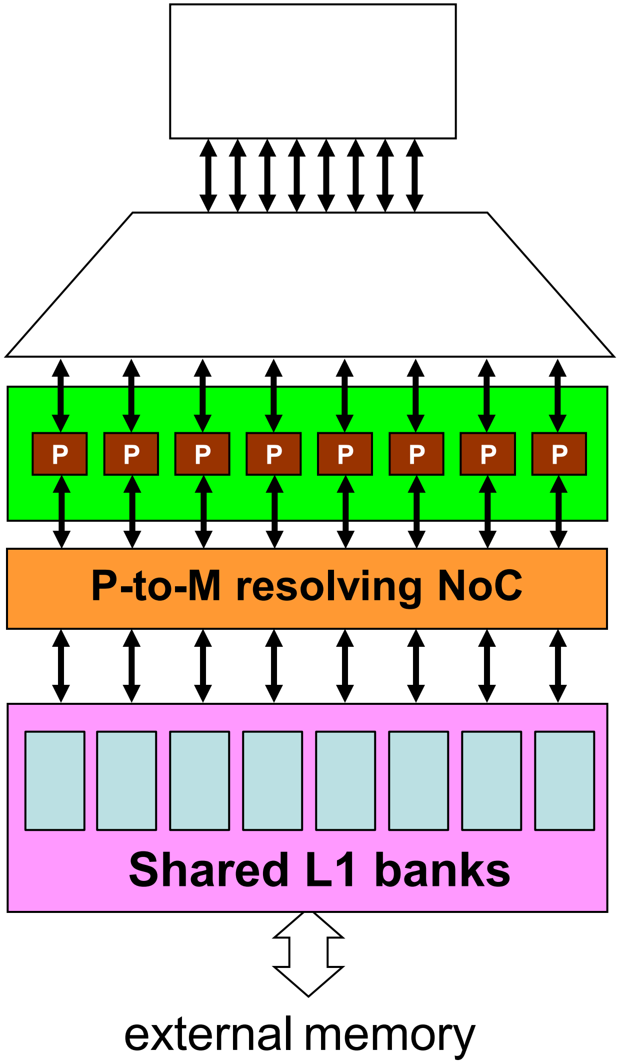
```
Begin
    if $(a_ib_i)$   $X=1$
End
```



if (a_ib_i) $X=1$

All cores which write into X, write the same value

The Plural Architecture: Part I



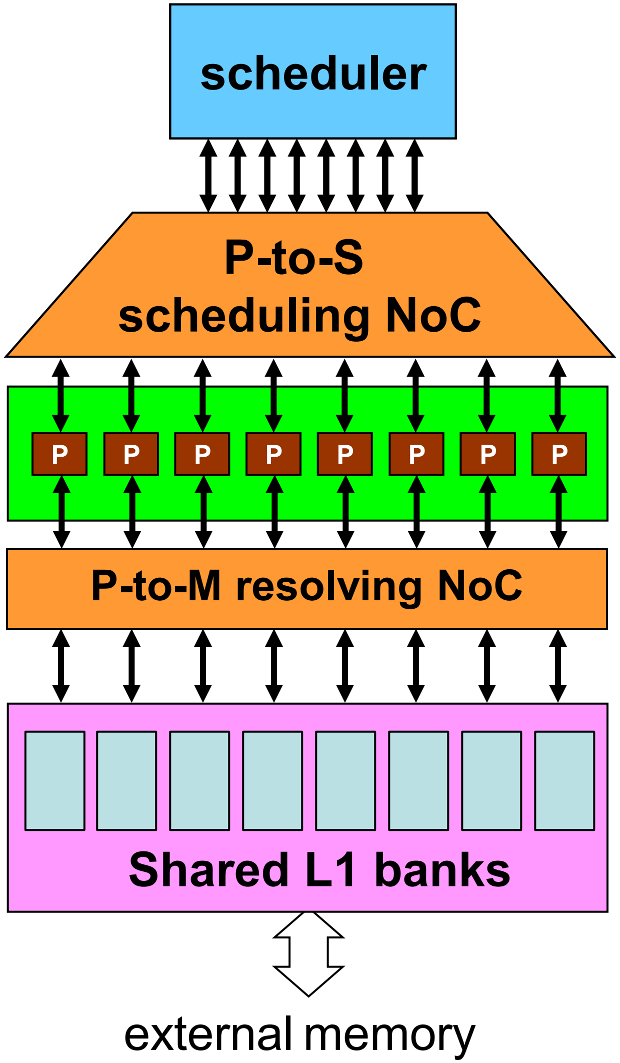
Many small processor cores
Small private memories

Fast NOC to memory
(Multistage Interconnection Network)
NOC resolves conflicts

SHARED memory (cache), many banks
~Equi-distant from cores (2-3 cycles)

“Anti-local” address interleaving
Negligible conflicts

The Plural Architecture: Part II



Hardware scheduler / dispatcher / synchronizer

Low (zero) latency parallel scheduling
enables fine granularity

Many small processor cores
Small private memories

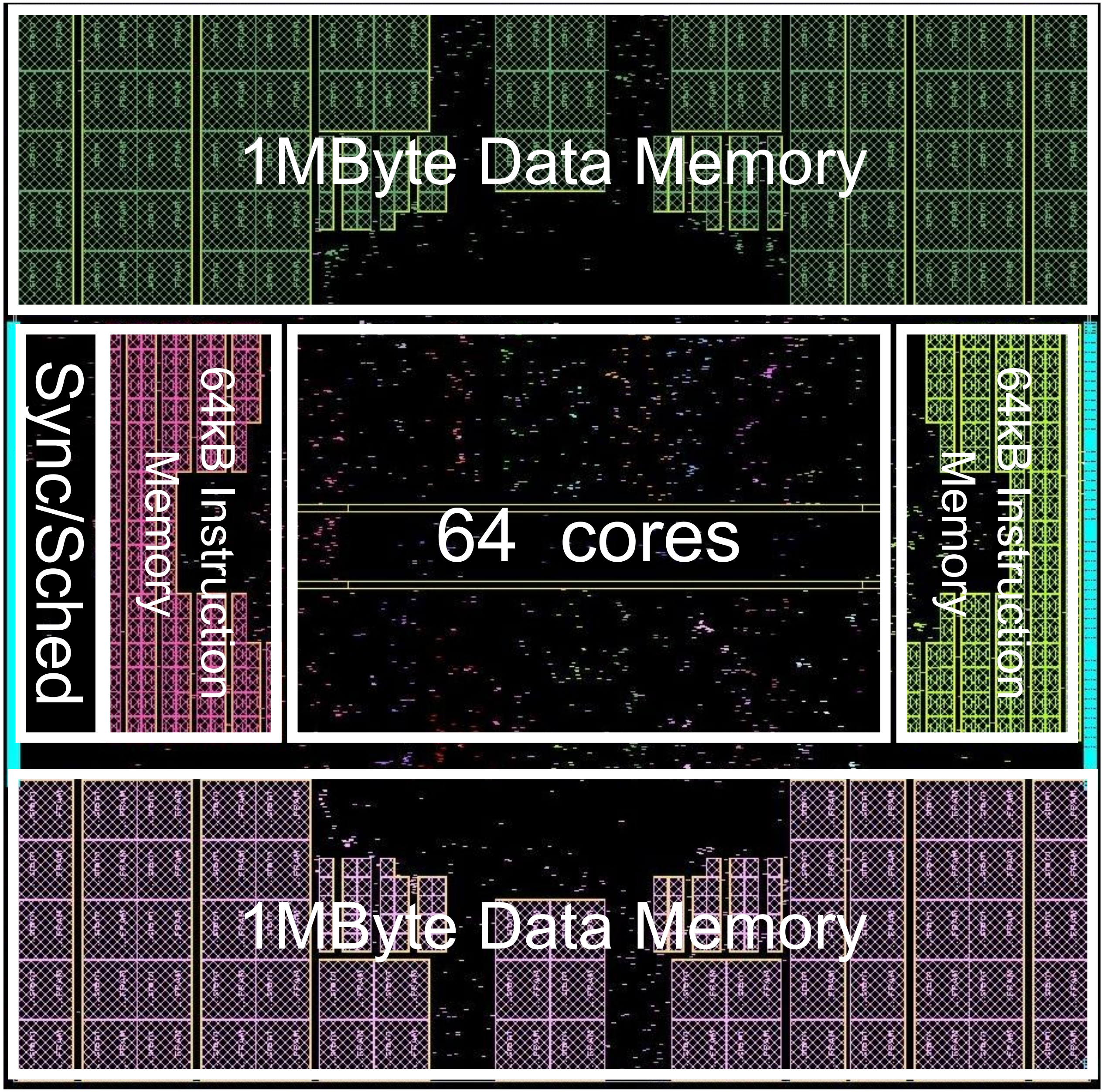
Fast NOC to memory
(Multistage Interconnection Network)
NOC resolves conflicts

SHARED memory (cache), many banks
~Equi-distant from cores (2-3 cycles)

“Anti-local” address interleaving
Negligible conflicts

PLURALITY

Example floorplan + layout



40nm GP

4×4mm

64 cores

16 FPU

2MB D$
in 128 banks

128kB I$

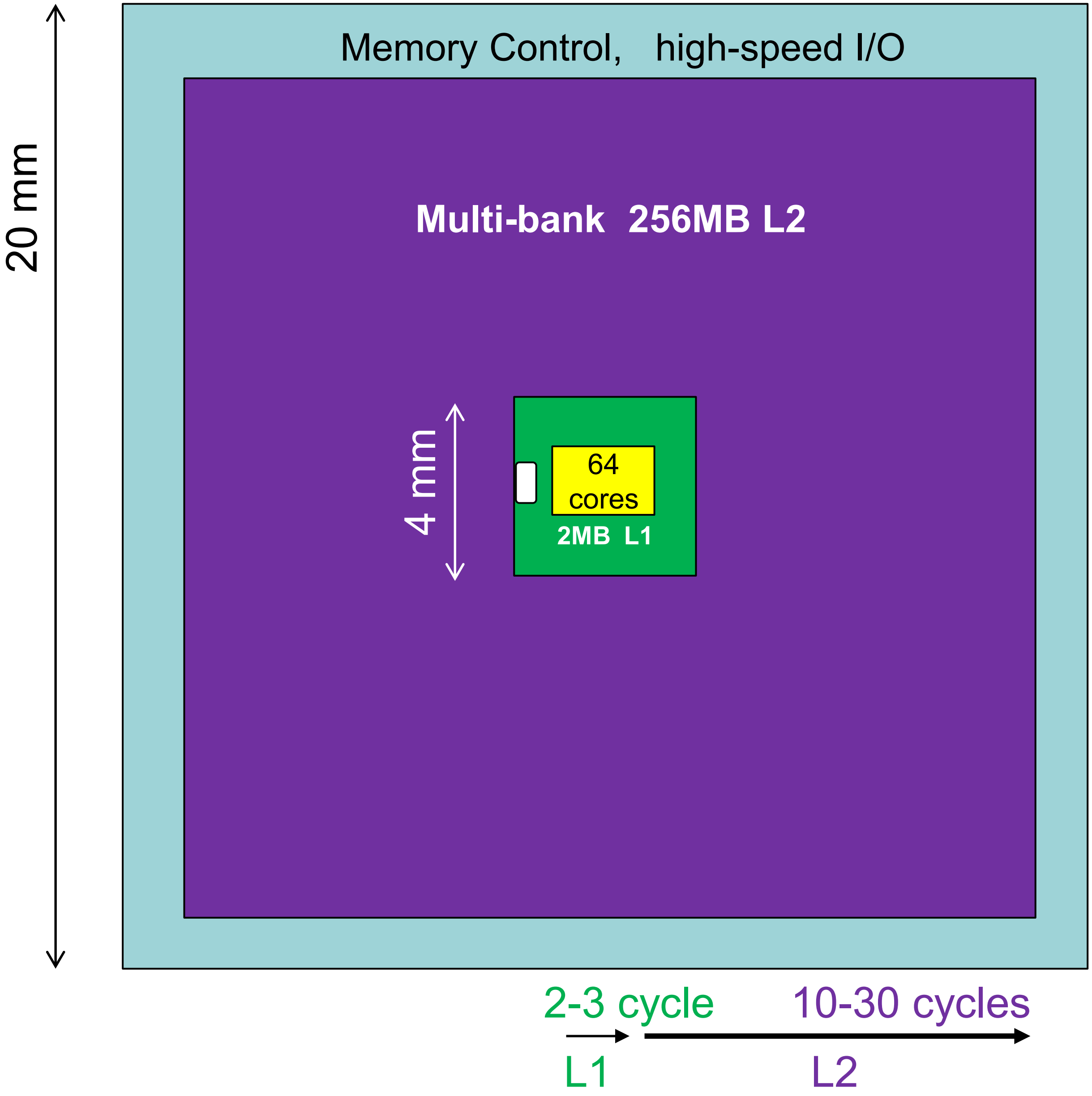
400 MHz

1 Watt

But does it scale?

- Research question:
  - Access to distant memory is slow and energy-inefficient
  - What if we use the full chip
    - Instead of 4x4 mm?

Possible Full-Chip Plan



20 mm

Memory Control, high-speed I/O

Multi-bank 256MB L2

4 mm

64 cores

2MB L1

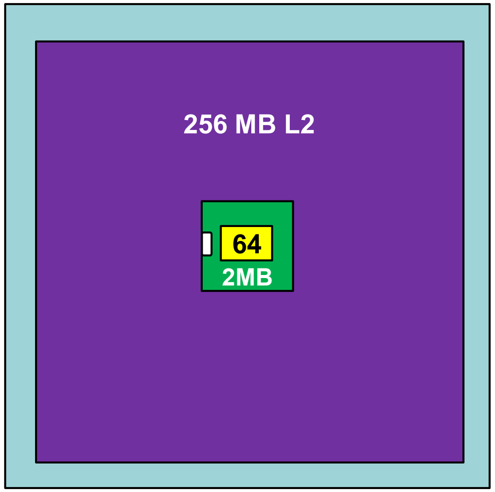
2-3 cycle

L1

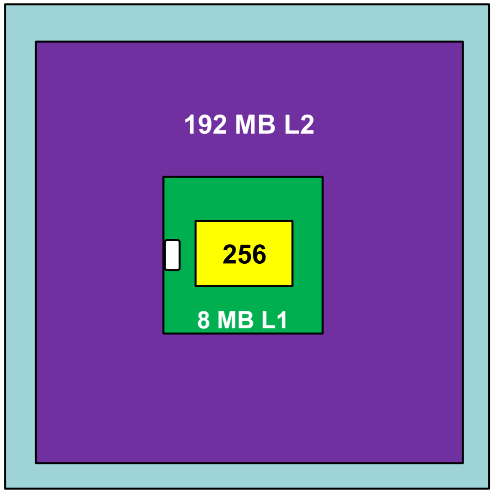
10-30 cycles

L2

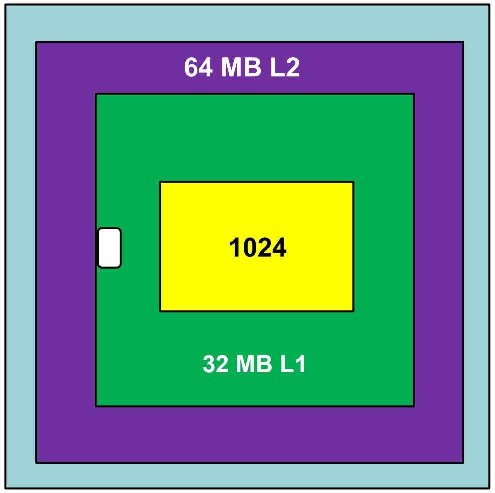
But does it scale (more processors)?



64 Cores
2 MB shared L1
256 MB shared L2

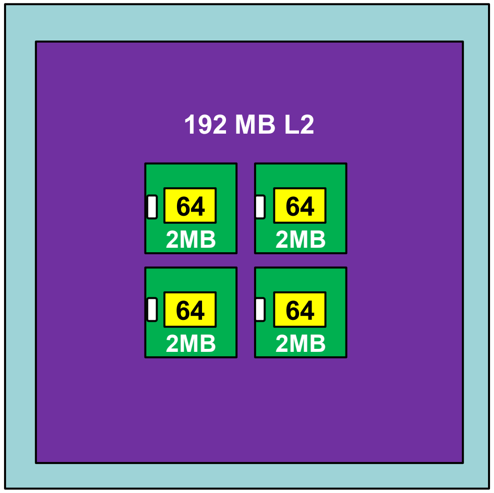


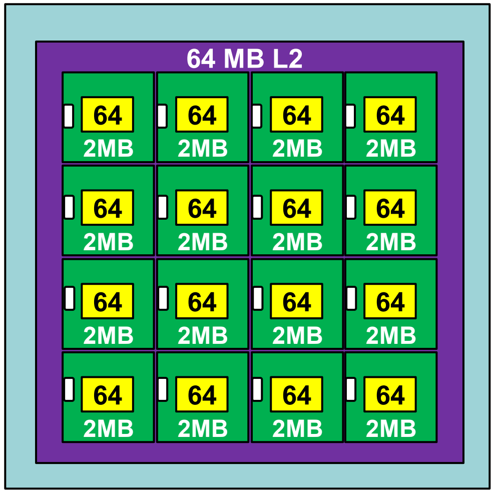
256 Cores
8 MB shared L1
192 MB shared L2



1024 Cores
32 MB shared L1
64 MB shared L2

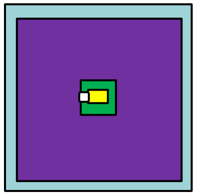
Long, high energy access to larger shared memory

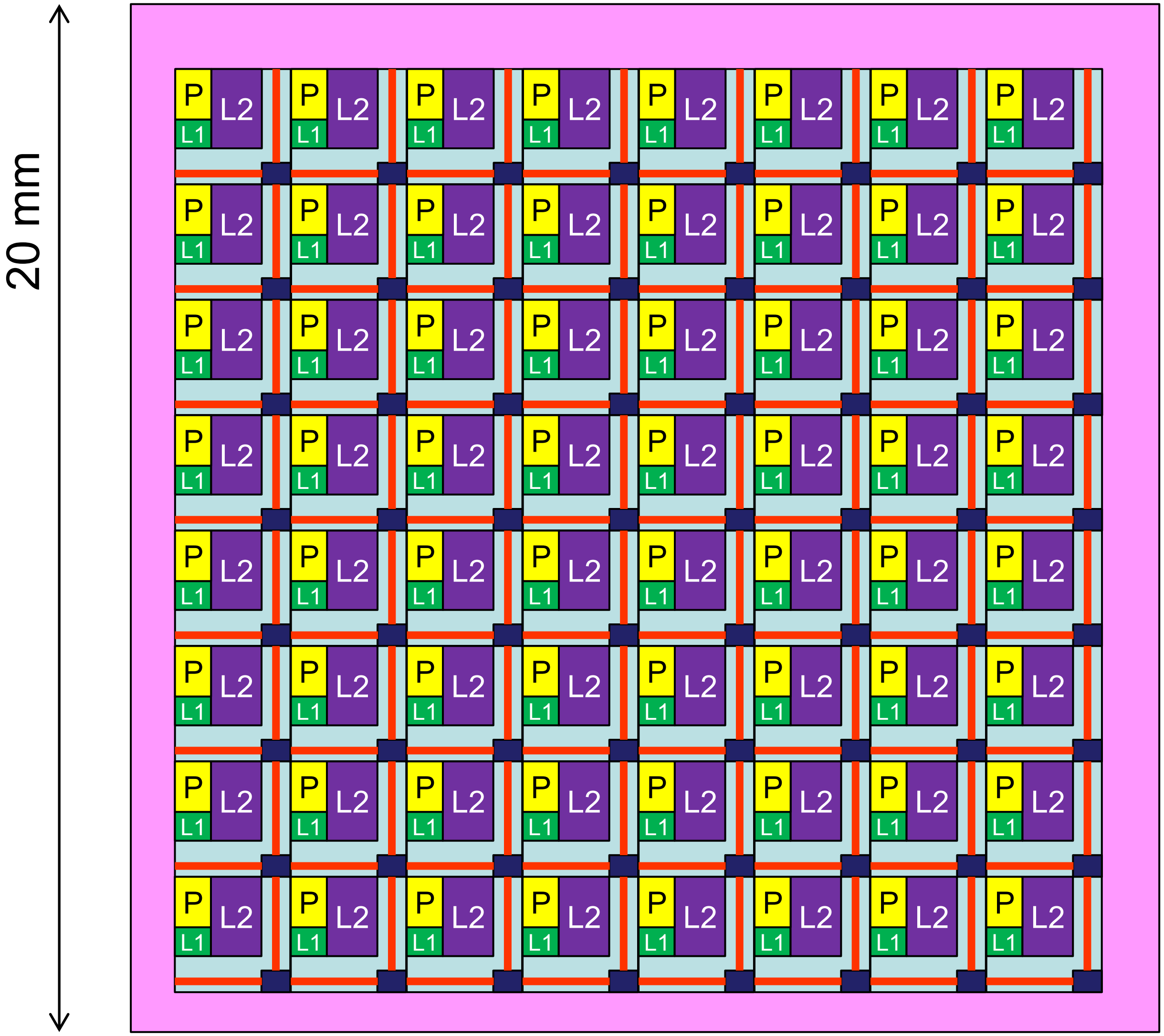




Cluster, Not a shared memory

Compare with “tiled” CMP using mesh NOC





20×20mm

64 tiles

32 kB L1 x64
= 2 MB

4 MB L2 x64
= 256 MB

Directory:
All L2’s = L3

1 cycle → L1

7cycles → L2

70cycles → L3

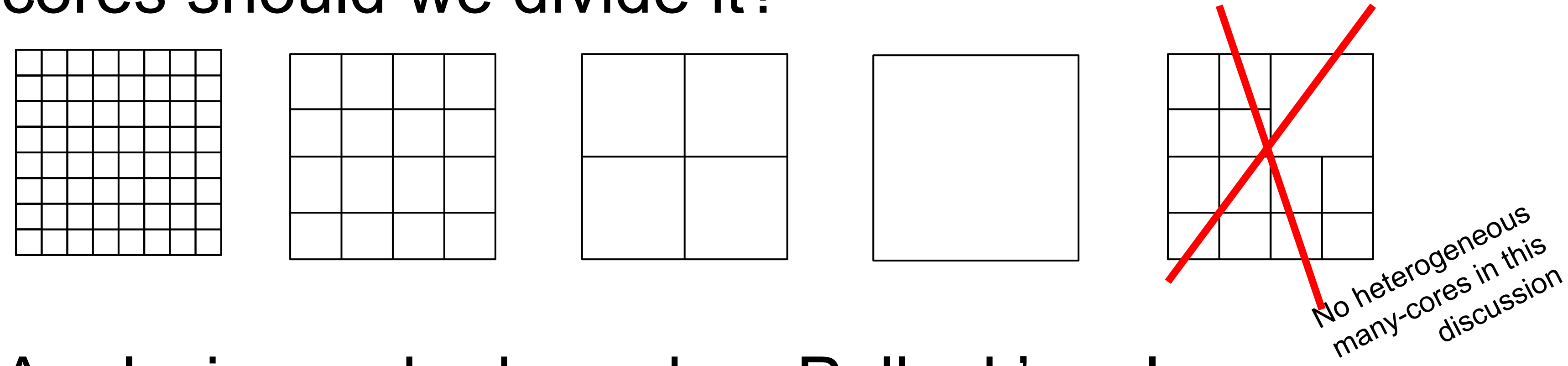
Modeling Many-core Architectures

Why model many-cores?

- Wish to fine-tune the Plural architecture
  - Cores:
    - How many? What type(s)?
    - What are performance, power, area?
      - Let's try Pollack's rule to inter-connect these parameters
  - Memories:
    - What architecture? How accessed?
    - What are performance, power, area?
- Wish to compare to other architectures
  - By performance, power, perf/power, area, …
    - And maybe ease of programming / parallelization

The many-core research question

- Given fixed ***area***, into how many processor cores should we divide it?



- Analysis can be based on Pollack's rule
- Other good questions (not dealt here):
  - Given fixed ***power***, how many cores? which cores?
  - Given fixed ***energy***, how many cores? which cores?
  - Given target performance, how many? Which?

Performance = Microarchitecture x Process

Process Technology

1.5µ 1.0µ 0.8µ 0.6µ 0.35µ 0.25µ 0.18µ 130nm 90nm 65nm 45nm 32nm

Microarchitecture

Processor

Intel Core ® i7

Pentium® Duo 2

Pentium M

Pentium® 4 Processor

Pentium® III Processor

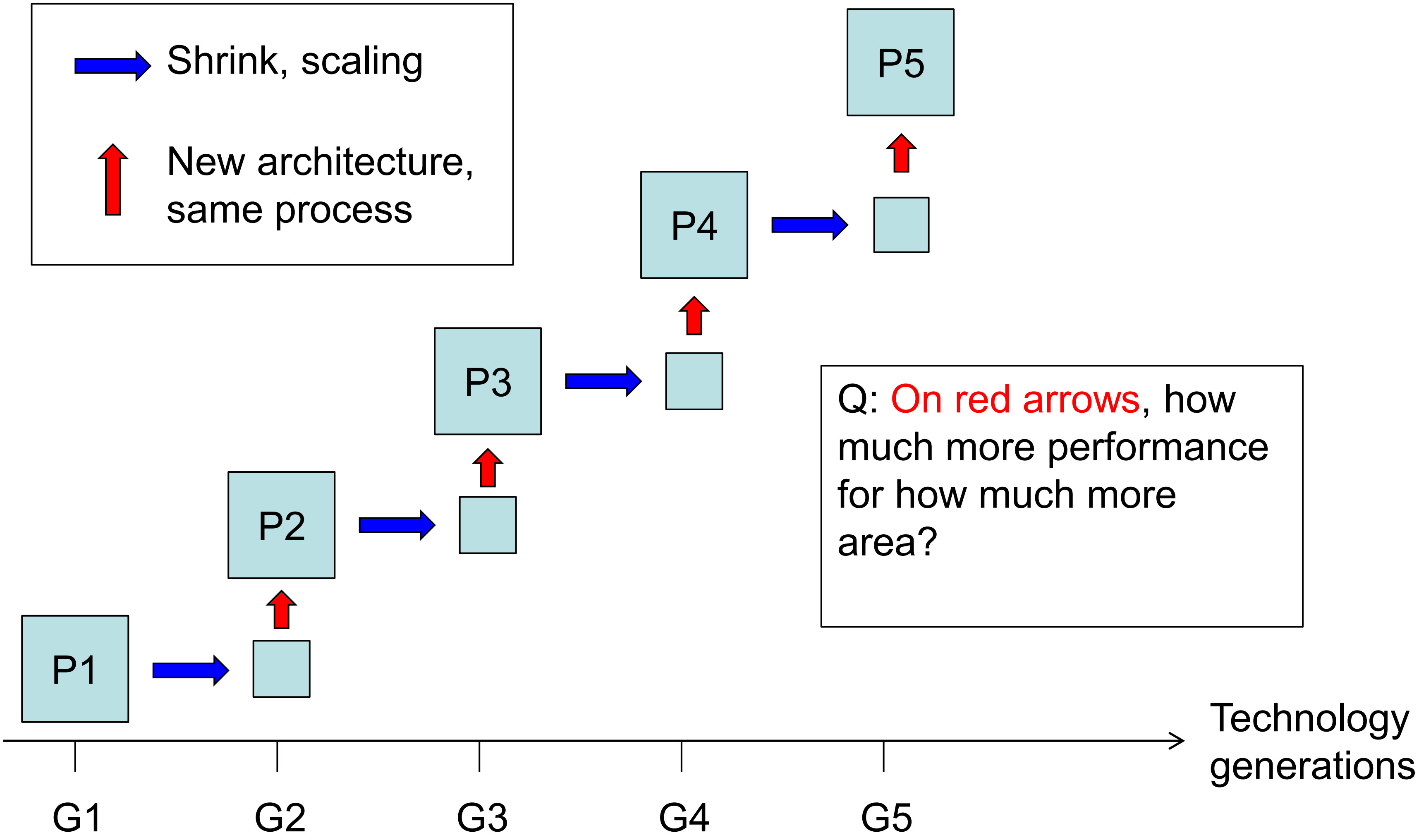
Pentium® II Processor

Pentium® Processor

Intel486™ DX Processor

Intel386™ DX Processor

The history at the basis of Pollack’s analysis



→ Shrink, scaling

↑ New architecture, same process

P1

P2

P3

P4

P5

G1

G2

G3

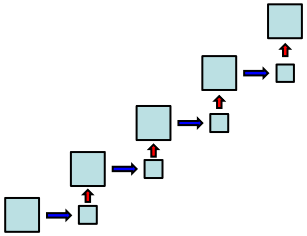
G4

G5

Technology generations

Q: On red arrows, how much more performance for how much more area?

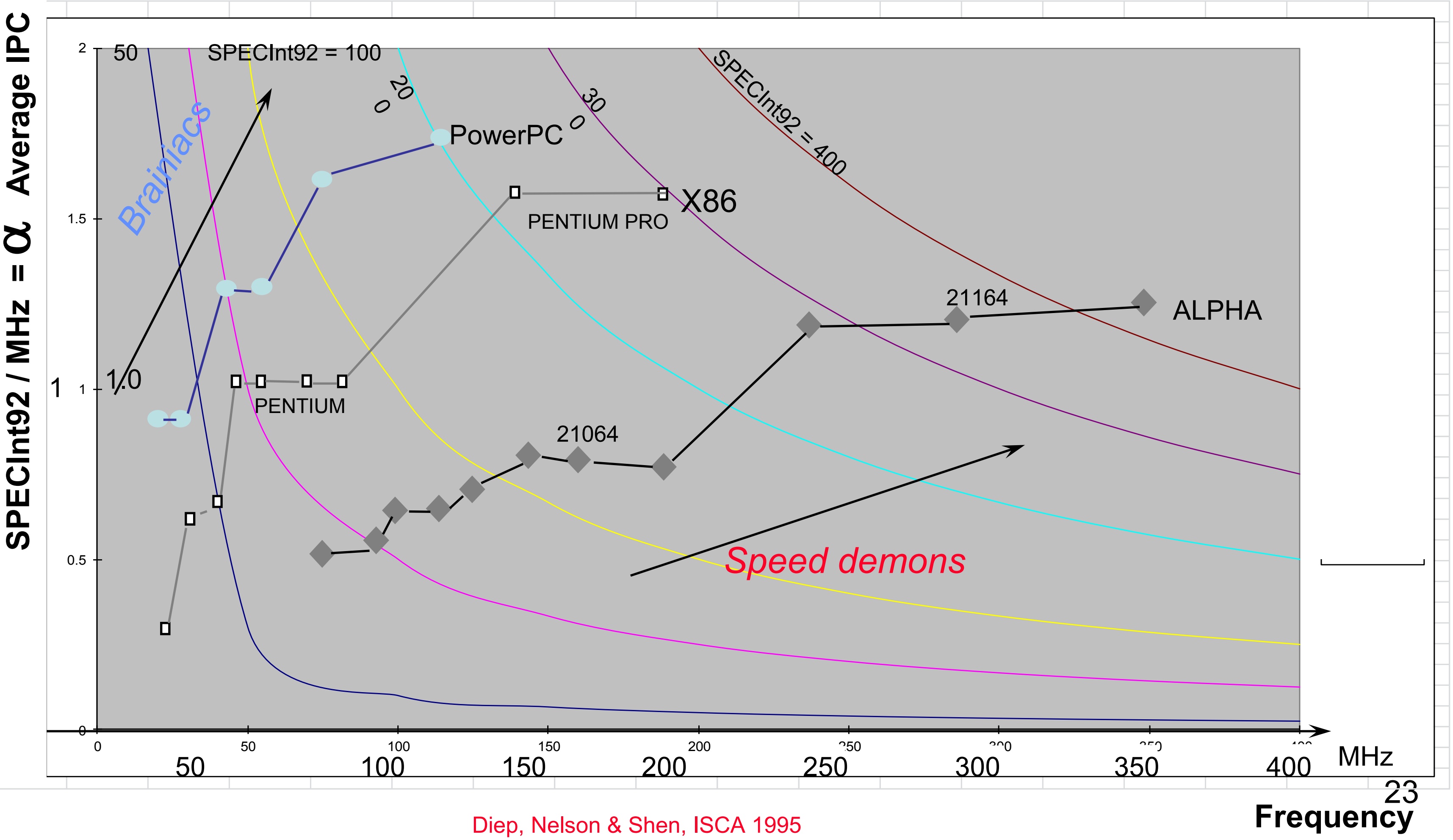
Pollack's rule for processors: Area or Power vs. Performance



- Pollack (& Borkar & Ronen, Micro 1999) observed many years of (intel) architecture
- In each Intel technology node, they compared:
  - Old uArch (shrink from previous node)
  - New uArch (faster clock and/or higher IPC)
- They noted:
  - New uArch used 2-3X larger area
  - New uArch achieved 1.5-1.7X higher performance
    - Resulting from both higher frequency and higher IPC
  - They did not consider power increase
    - Who thought about power in 1999?
- Observation: Performance ~ $\sqrt{area}$

Performance = IPC × Frequency

- Experience shows: for higher performance, both IPC and frequency must be increased



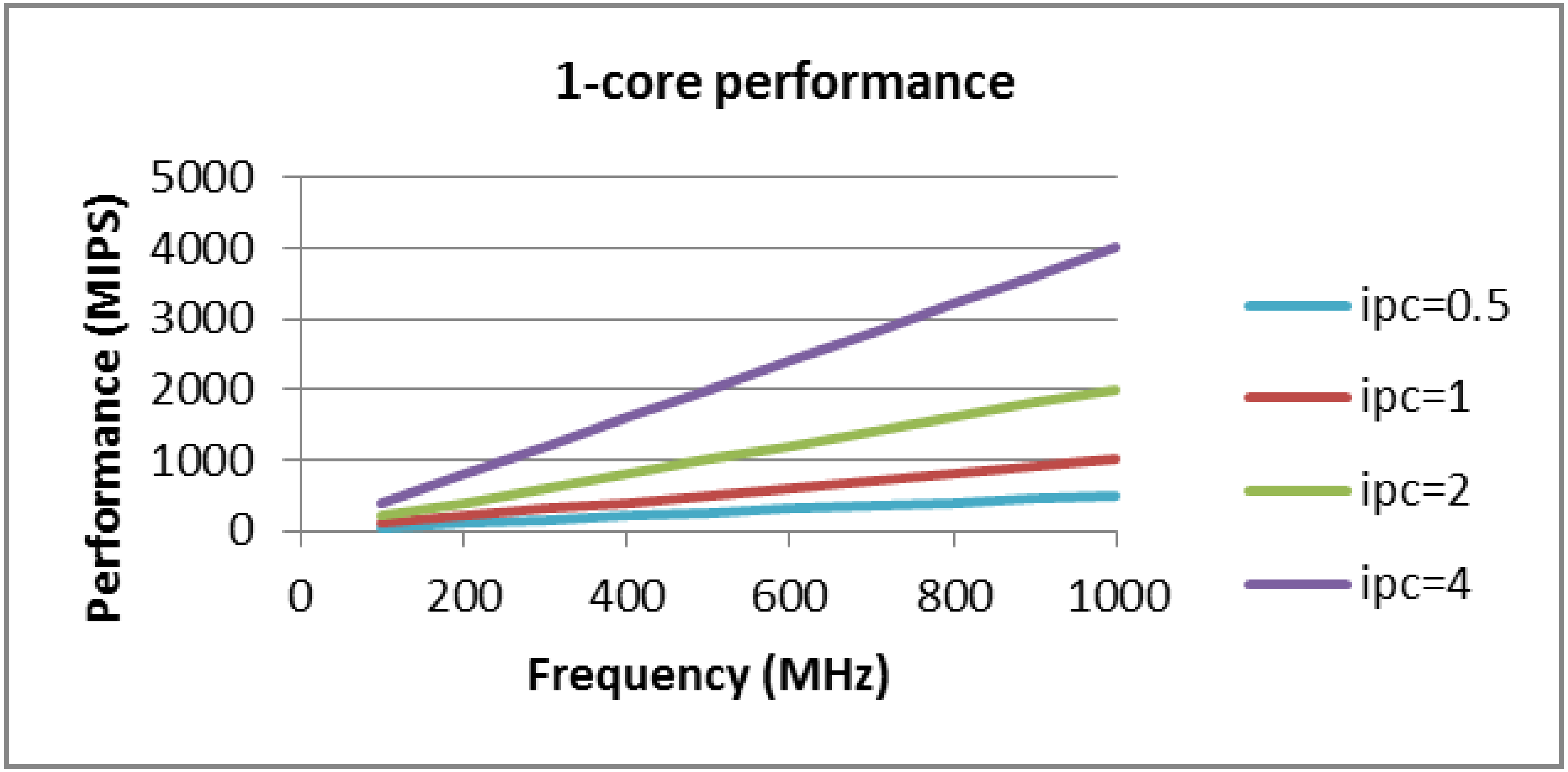
Diep, Nelson & Shen, ISCA 1995

The many-core *fixed-total-area* model

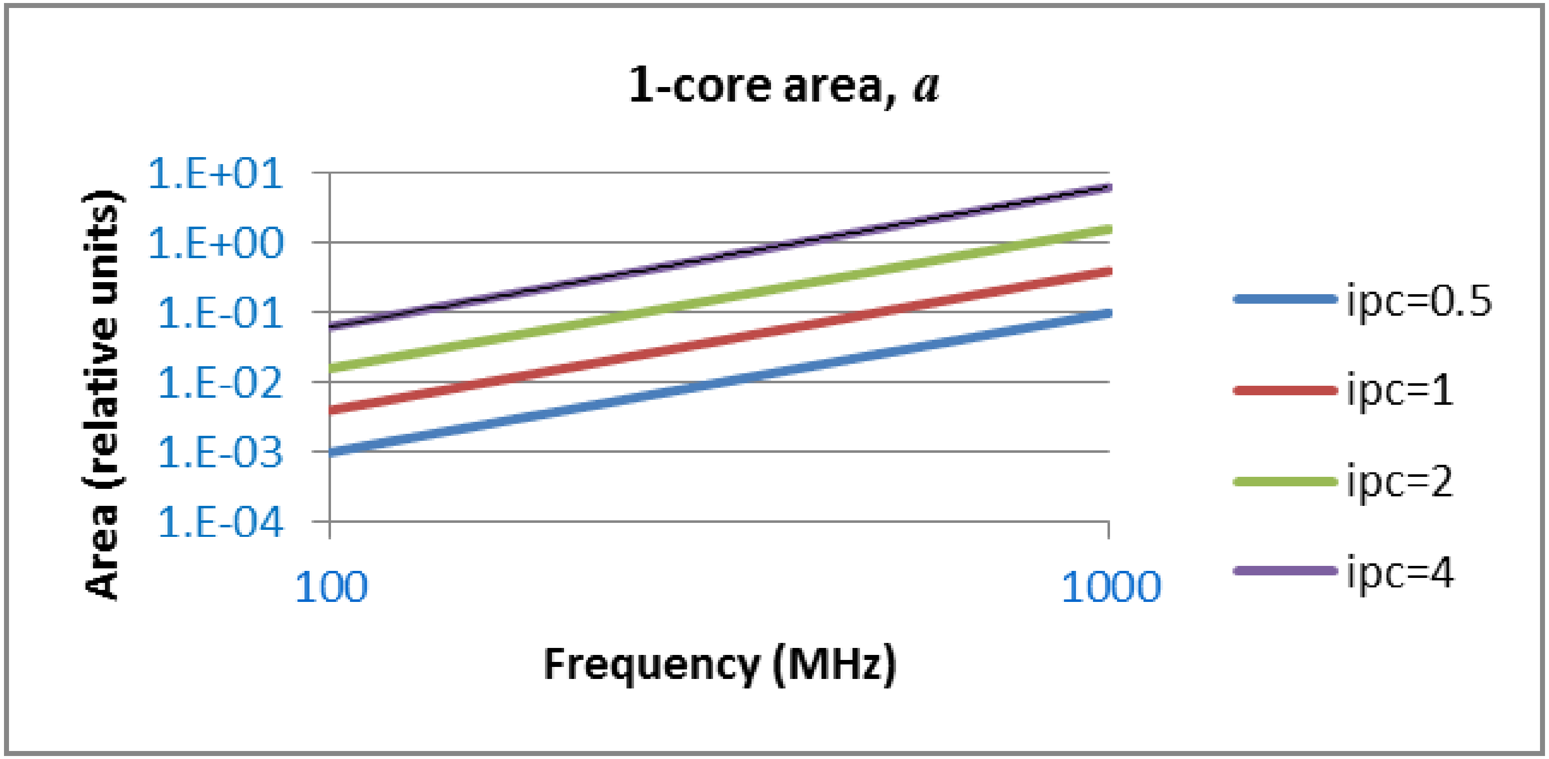
- Assume fixed chip area (typically 300-500 mm^2)
- Split chip area A = A$_{cores}$ + A$_{mem}$
  - Split (memory size) affects on-chip hit rate
  - A$_{mem}$ may be further split into A$_{L1}$+A$_{L2}$
- Divide A$_{cores}$ into m cores. How many ?
  - Area of each core: $a = \frac{A_{cores}}{m}$. $\quad Thus,\ \ m \sim {}^{1}/_{a}$
- [Pollack's]: core area determines core performance. Select *IPC* and frequency f so that:
  - $Performance\ (core) = IPC \times f \sim \sqrt{a}.\quad Thus,\ \ a \sim IPC^2 f^2\ ,\ m \sim {}^{1}/_{IPC^2 f^2}$
  - $Power\ (core) \sim a \times f \sim IPC^2 f^3$
- Assume perfect parallelism (at least as upper bound)
  - $Performance\ (m\ cores) = IPC \times f \times m \ \sim \frac{IPC \cdot f}{IPC^2 f^2} = \frac{1}{IPC \cdot f} \sim \frac{IPC \cdot m}{IPC\sqrt{m}} = \sqrt{m}$
  - $Power\ (m\ cores) = a \times f \times m \ \sim \frac{IPC^2 f^3}{IPC^2 f^2} = f \sim \ \frac{1}{IPC\sqrt{m}}$

Summary: Performance$\sim\frac{1}{f}\sim\sqrt{m},$ Power$\sim\frac{1}{\sqrt{m}}\sim f\,,$ $\quad m\sim\frac{1}{f^2}$

$$Performance\ (core) = IPC \times f$$

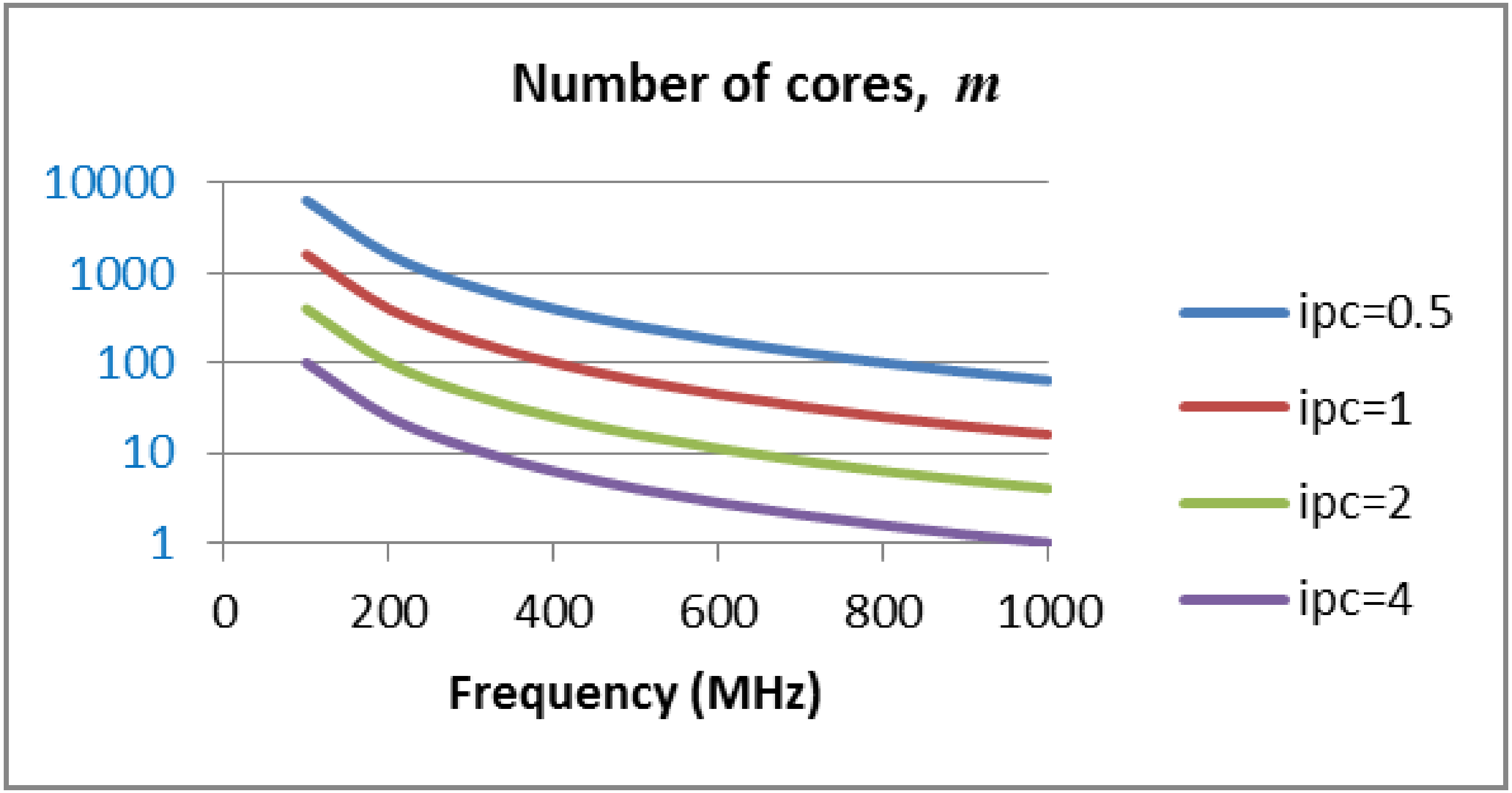


$$a \sim IPC^2 f^2$$



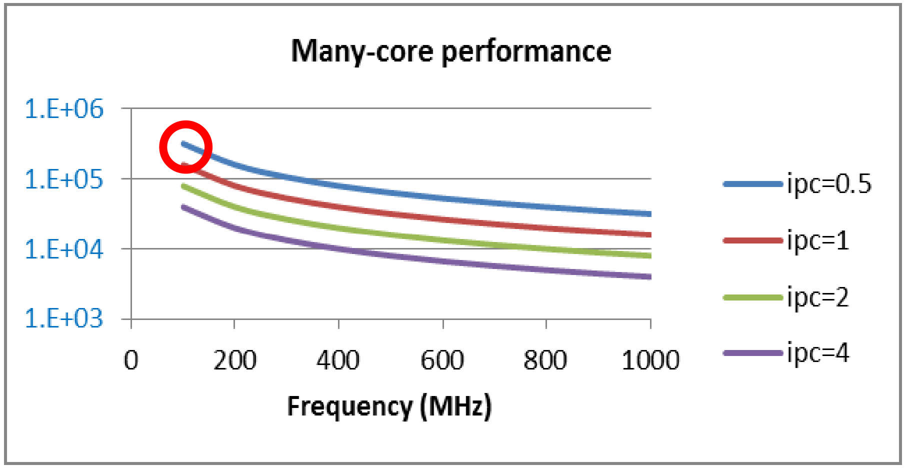
For each IPC curve, $a \sim f^2$

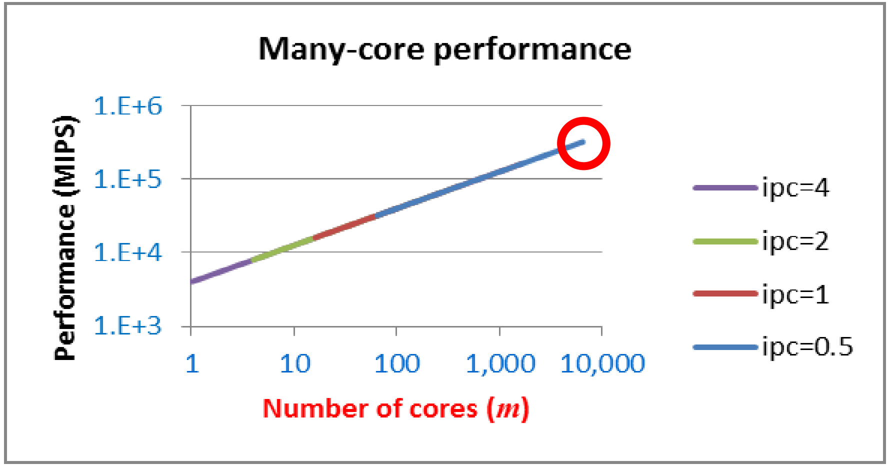
$$m \sim \frac{1}{IPC^2 f^2}$$



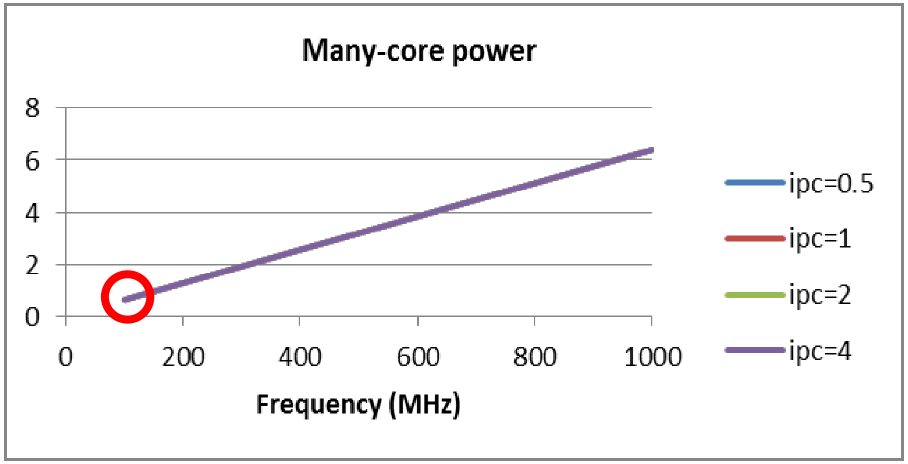
For each IPC curve, $m \sim \frac{1}{f^2}$

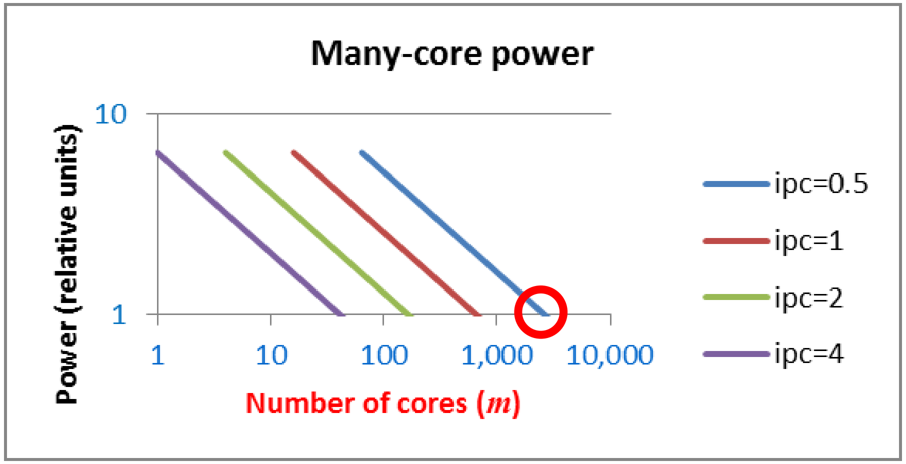
$$\text{Performance} \sim \frac{1}{f} \sim \sqrt{m}$$



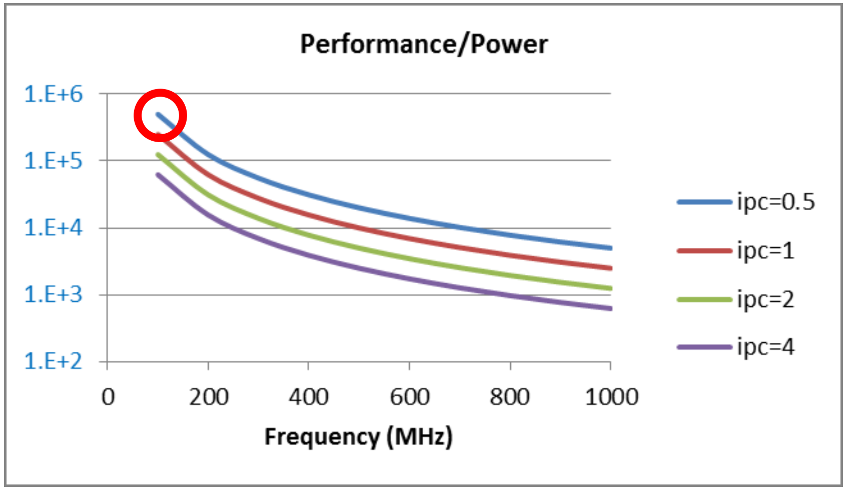


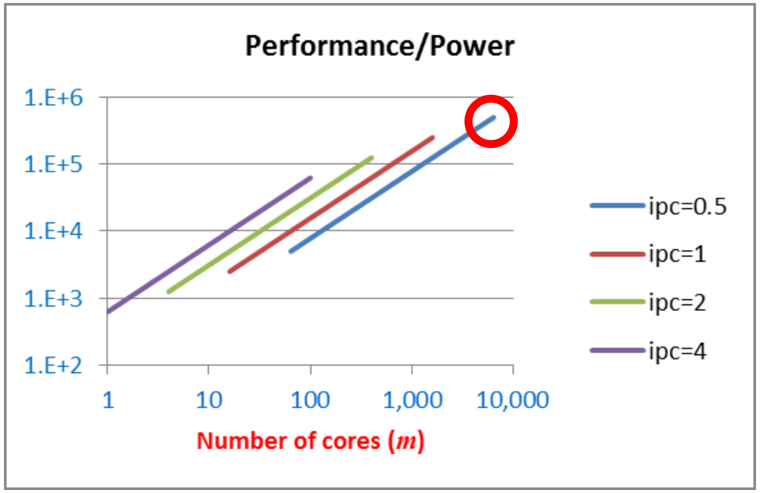
$$\text{Power} \sim f \sim \frac{1}{\sqrt{m}}$$





$$\frac{Performance}{Power} \sim \frac{1/f}{f} = \frac{1}{f^2} \sim \frac{\sqrt{m}}{1/\sqrt{m}} = m$$





Analysis of the results so far:

- Slower frequency and lower IPC → higher performance, lower power
- Thanks to Pollack’s square rule

But this changes when we also consider memory power…

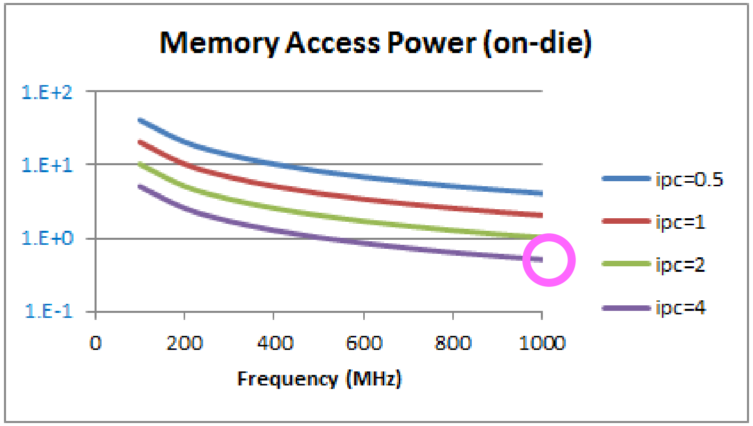
Now add memory

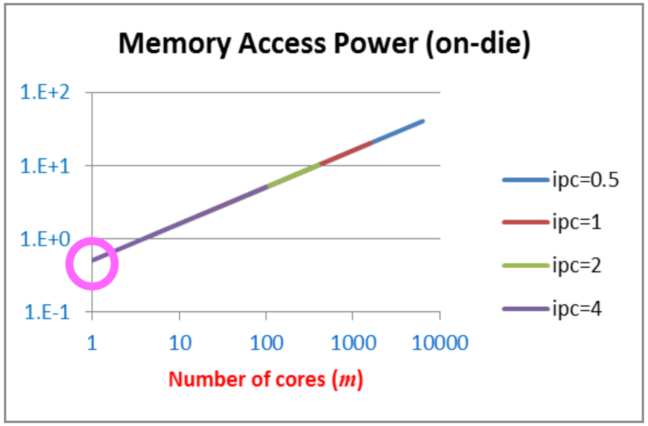
- So far, only computing power
    - Including power to access local cache/memory in each core
        - Only small private memory is local in the SM Plural architecture
- But we also need to access not-so-local shared memory
- Access rate to memory: once every r_m instructions
    - About every 20 instructions in the SM Plural architecture
    - Ignore cache misses, assume using only on-chip memory
- Need to add memory access power to the computing power
    - Relative energy: assume access is 10x higher than exec.

Comparing SM Plural to TILED architecture

- Local memory is:
  - SM: private memory
  - TILED: local L1 & L2 (L2 access less frequent but higher energy)
- Global memory is:
  - SM: Shared memory (L1), possibly LLC
    - Via fast cores-to-memories net
  - TILED: other cores' local L2 caches (L3)
    - Via complex NoC incl. directory access
- Access rate: global memory access every r_m instructions
  - SM: every 20 instructions to L1, every 5000 instructions to LLC
  - TILED: Assume every 500 instructions
- Energy to access global memory (higher than exec energy)
  - SM: 10x to L1, 100x to L2
  - TILED: 1x to L1, 5x to L2, 1000x to L3

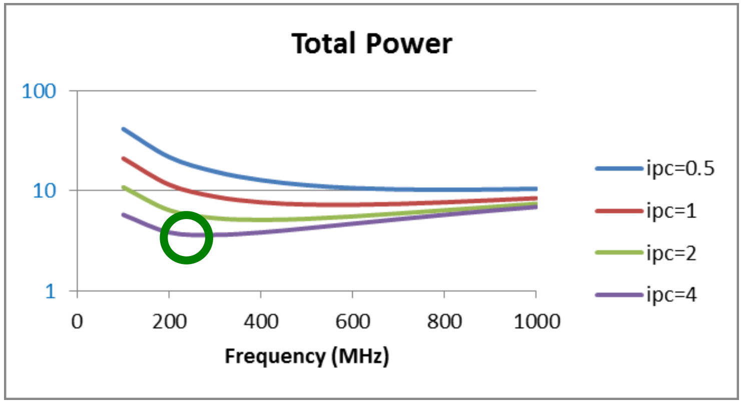
$$\frac{1}{f}$$

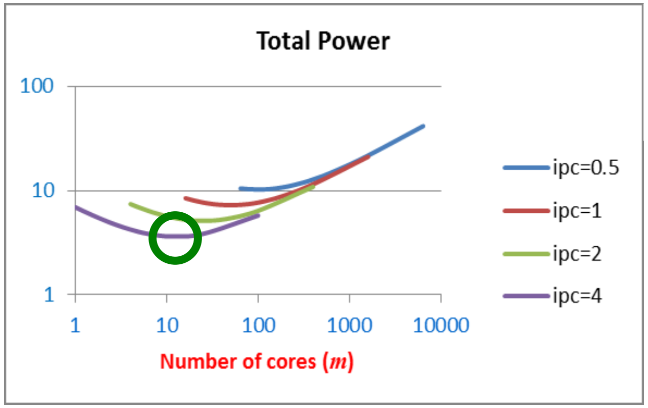




$$\sqrt{m}$$

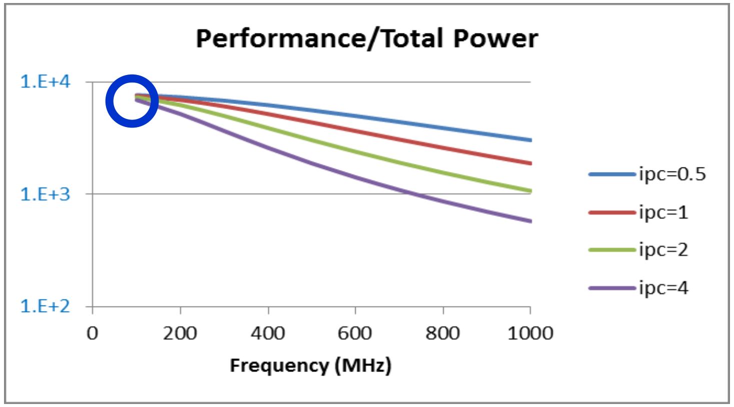
$$\frac{1}{f}+f$$

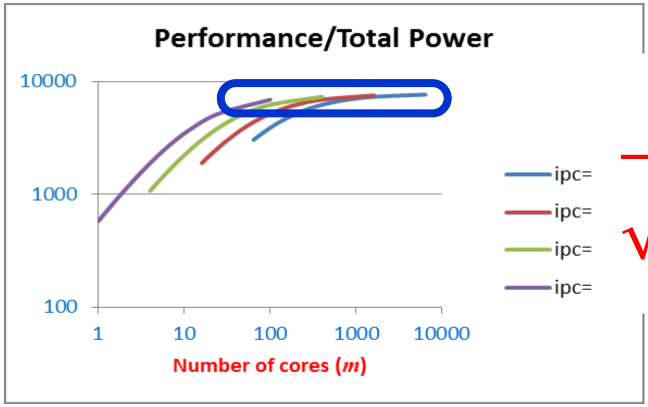




$$\sqrt{m}+\frac{1}{\sqrt{m}}$$

$$\frac{\frac{1}{f}}{\frac{1}{f}+f}$$





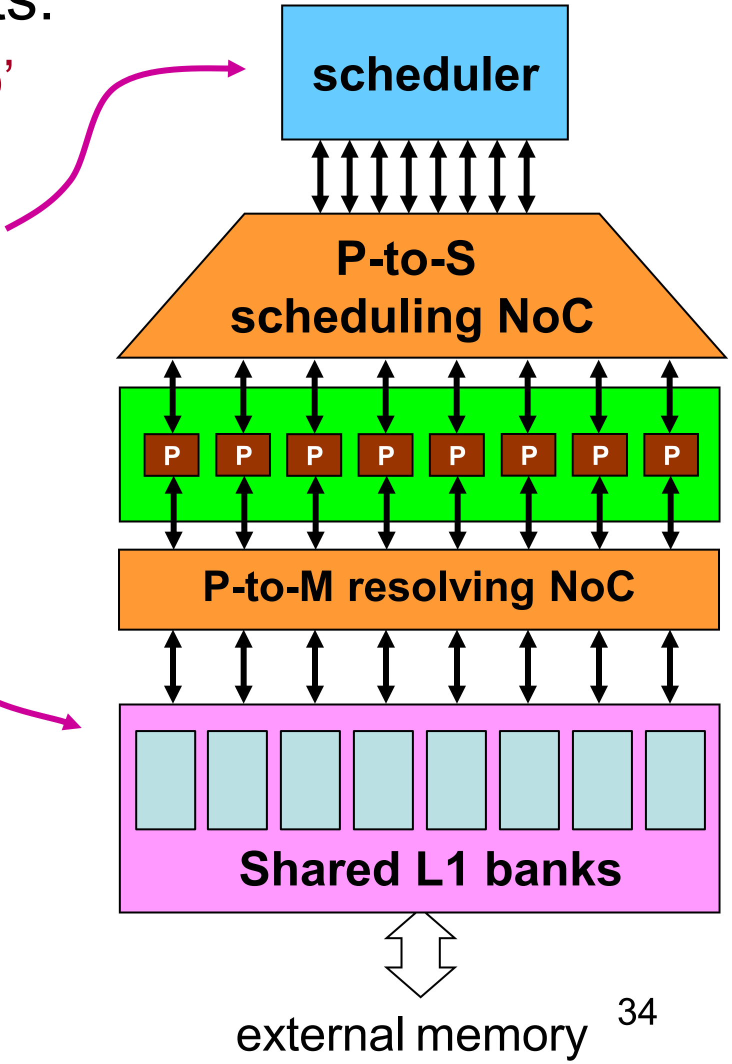
$$\frac{\sqrt{m}}{\sqrt{m}+\frac{1}{\sqrt{m}}}$$

Summary of the model

- Considering only cores, *fixed-total-area* model implies: for highest performance and lowest power, use
  - smallest / weakest cores (lowest IPC)
  - lowest frequency
- Adding on-chip access to memory leads to a different conclusion: for lowest power and highest performance/power ratio, use
  - Strongest cores (high IPC)
  - But stay with lowest frequency
    - Lower frequency → lower access rate to global memory

The Plural task-oriented programming model

- Programmer generates TWO parts:
  - Task-dependency-graph = ‘task map’
  - Sequential task codes
- Task maps loaded into scheduler
- Tasks loaded into memory



Task template:

```
singular
duplicable      task xxx( dependencies )
control
{
        … # ….  // # is instance number
        …..
}
```

Fine Grain Parallelization

Convert (independent) loop iterations

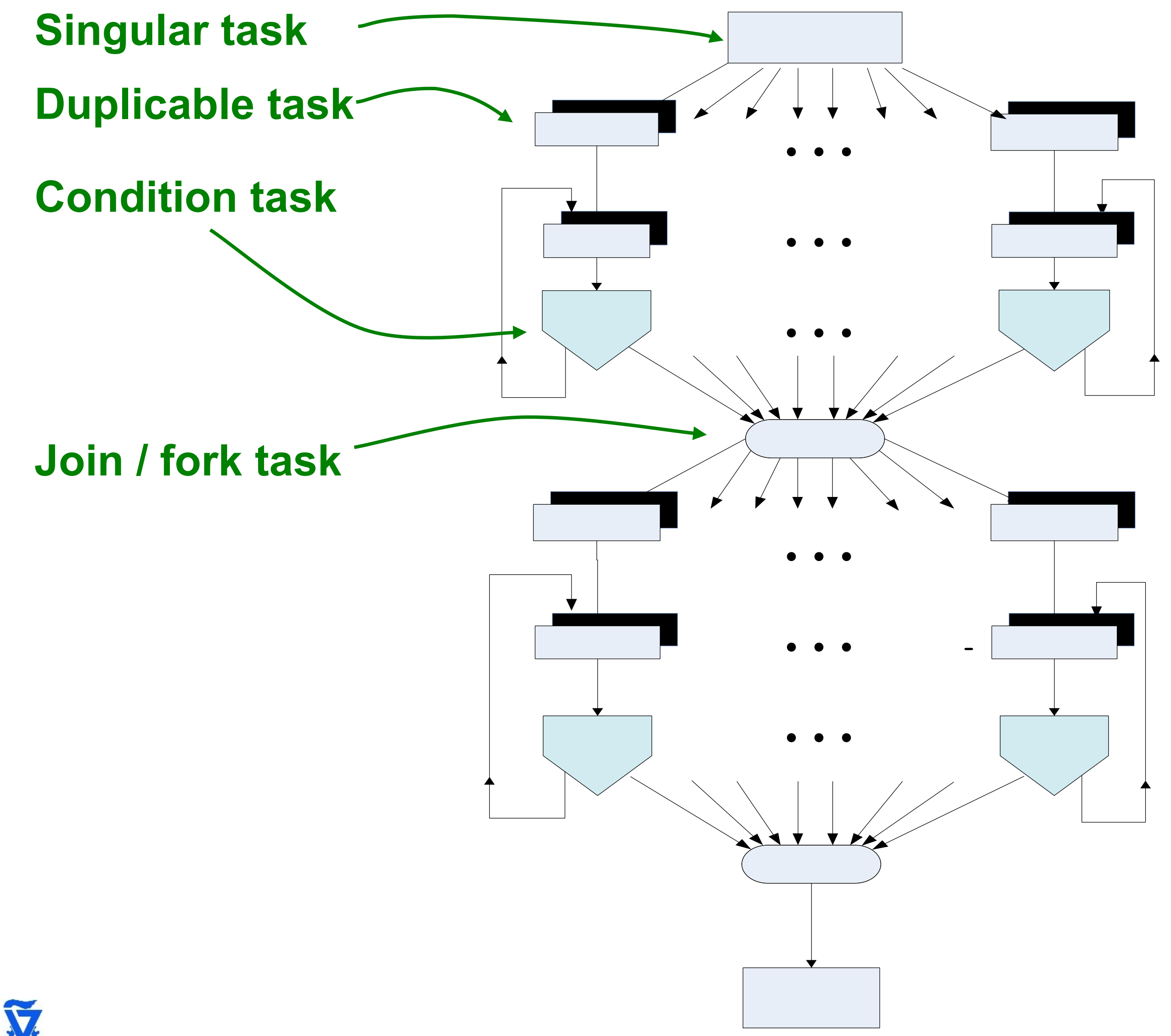
```
for ( i=0; i<10000; i++ ) { a[i] = b[i]*c[i]; }
```

into parallel tasks

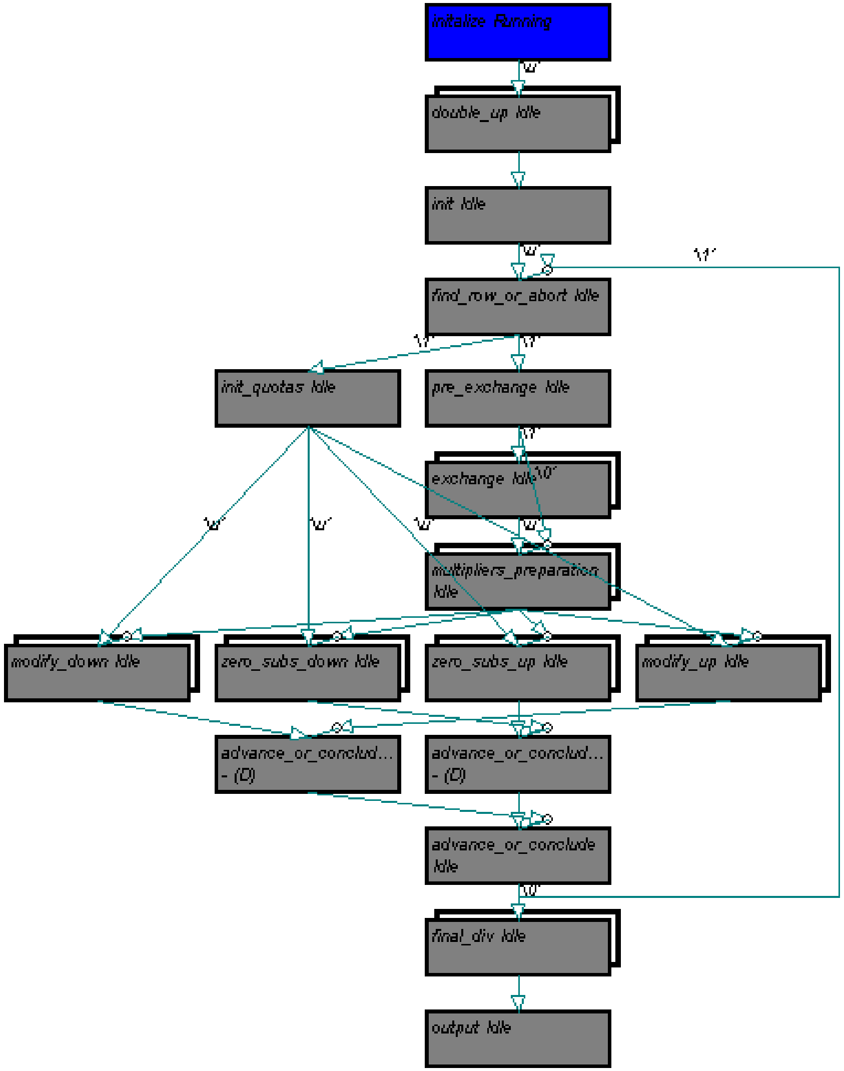
```
Singular task init { set_quota (XX,10000); }


duplicable task XX(init)
{ a[#] = b[#]*c[#]; }  // # is instance number
```

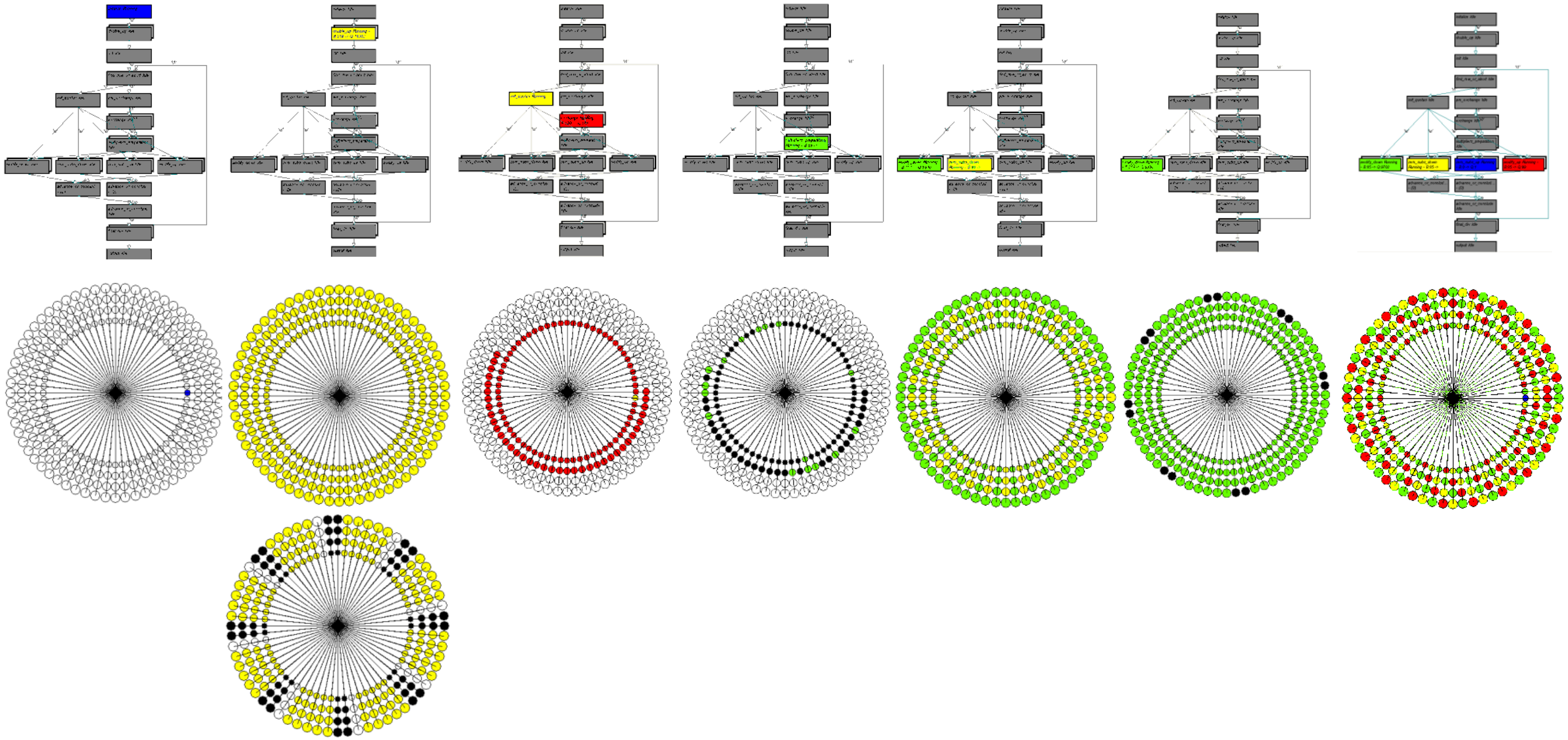
Task map example (2D FFT)



Another task map (linear solver)



Linear Solver: Simulation snap-shots



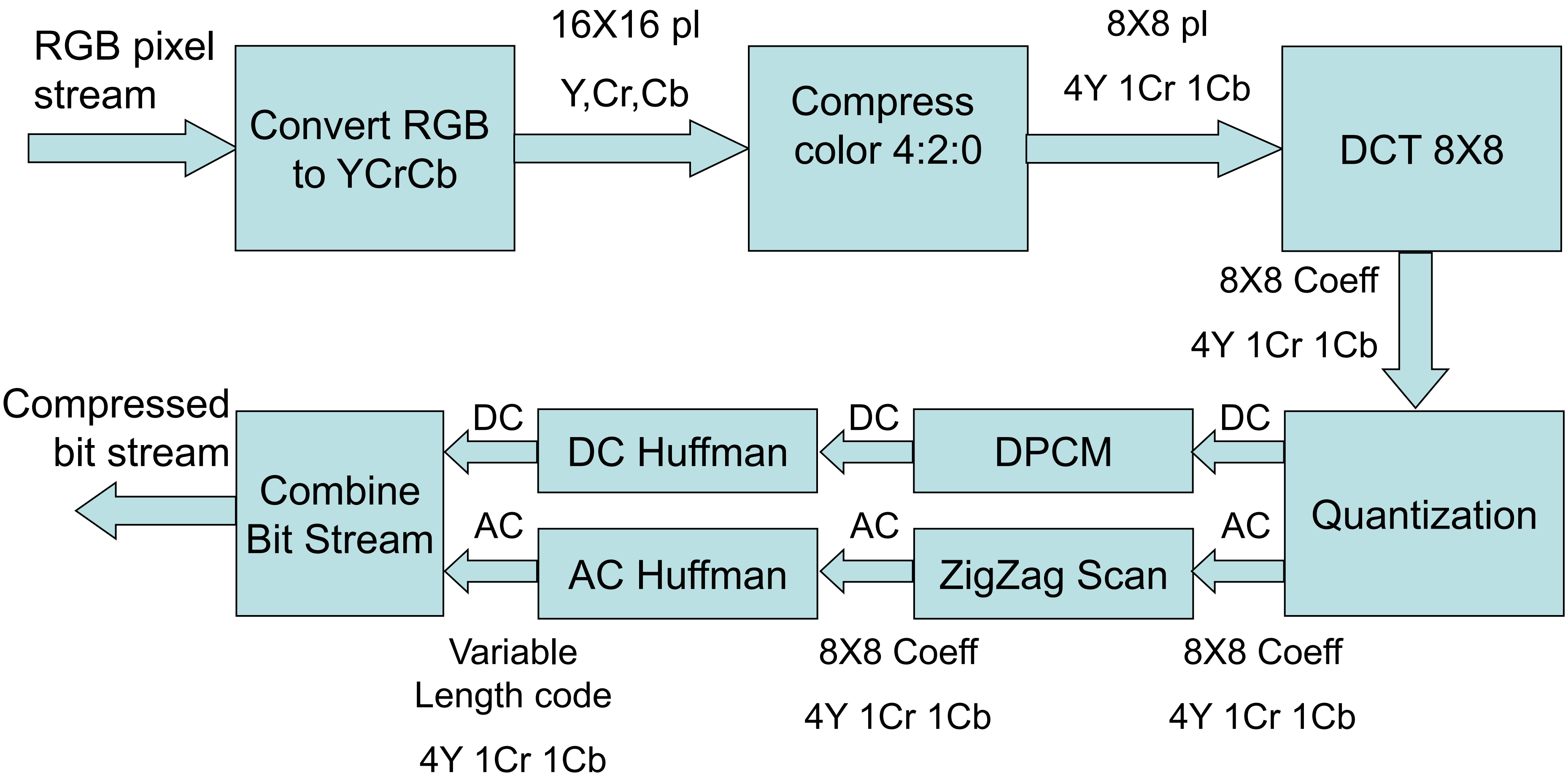
Task Rules 1

- Tasks are sequential
- All ready tasks, or any subset, can be executed in parallel on any number of cores
- All computing organized in tasks. All code lines belong to tasks
- Tasks use shared data in shared memory
  - May employ local private memory.
  - Its contents disappear once a task completes
- Precedence relations among tasks:
  - Described in task map
  - Managed by scheduler: receive task completion messages, schedule dependent tasks
- Nesting task spawning is easy and natural

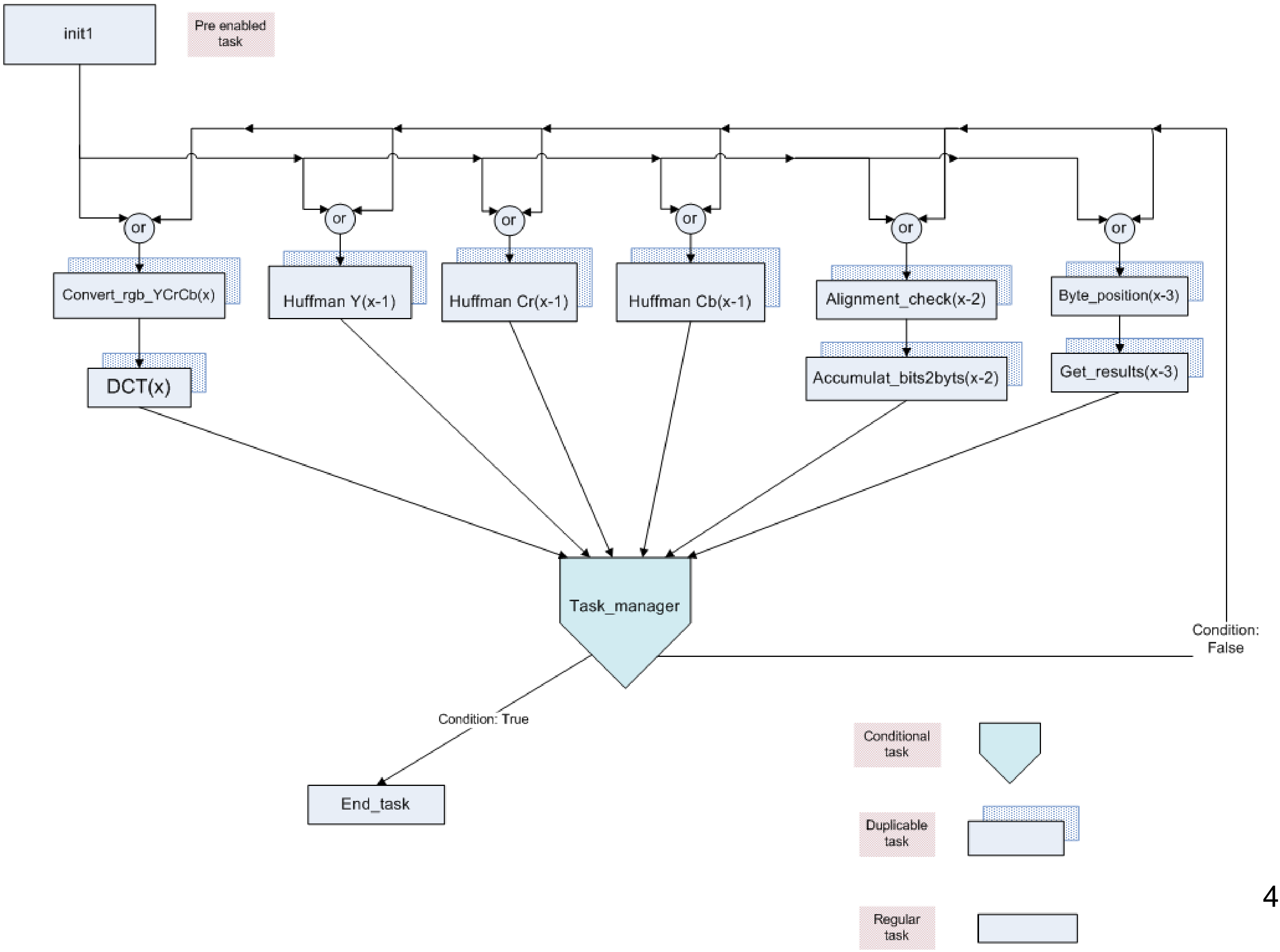
Task Rules 2

- 3 types of tasks:
  - Singular task (Executes once)
  - Duplicable task
    - Duplicated into quota=*d* independent concurrent instances
    - Identified by entry point (same for all *d* instances) and by unique instance number.
    - Task quota is actually a variable. The only reason for the synchronizer to access data memory
  - Control task
    - No executable code.
    - Controls branch, merge and conditional points in task map.
    - Executed by scheduler
- Tasks are not functions
  - No arguments, no inputs, no outputs
  - Share data only in shared memory
- No synchronization points other than task completion
  - No BSP, no barriers
- No locks, no access control in tasks
  - Conflicts are designed into the algorithm (they are no surprise)
  - Resolved only by NoC

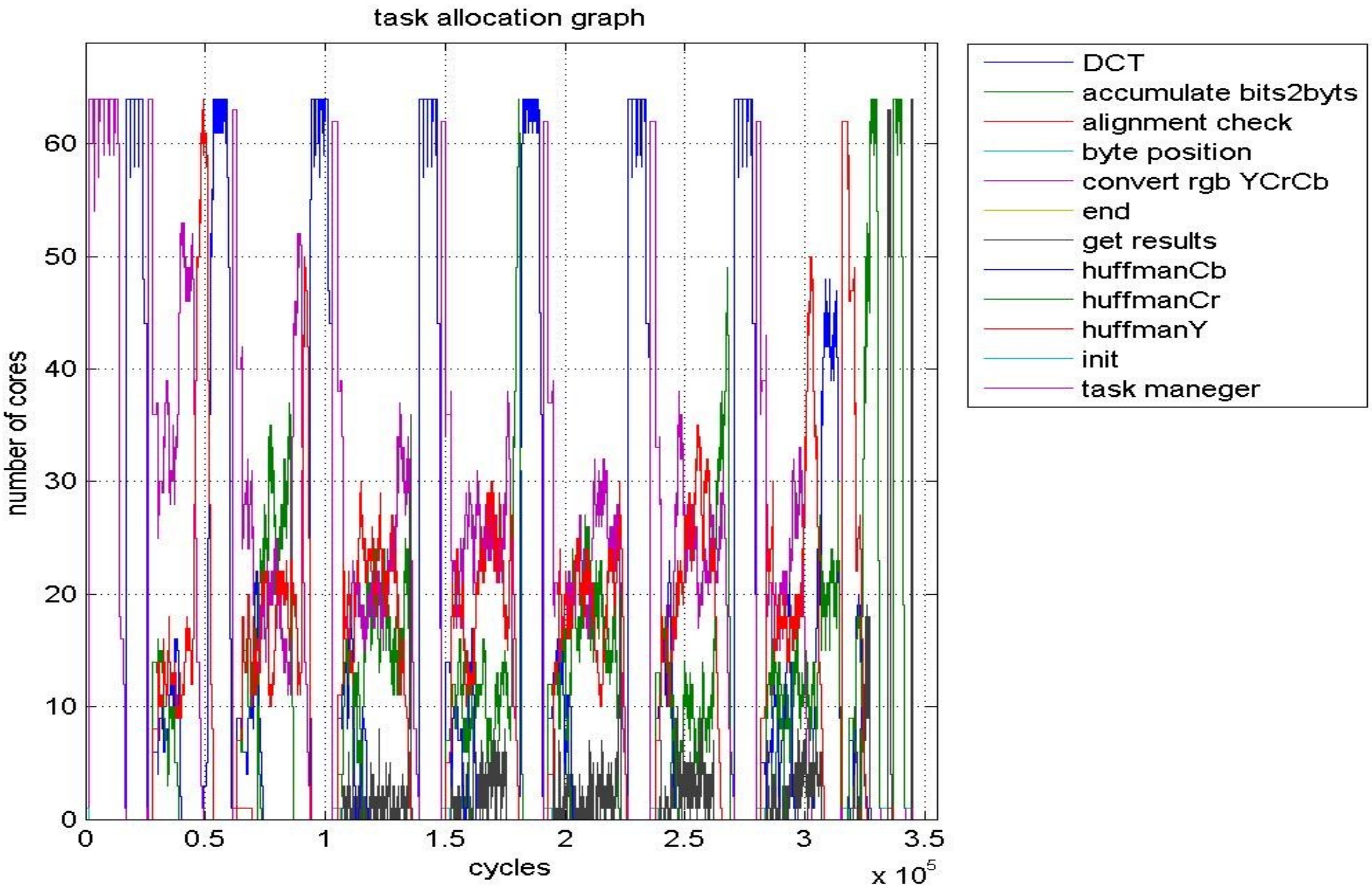
JPEG compression algorithm: Pipelined, limited parallelism per block



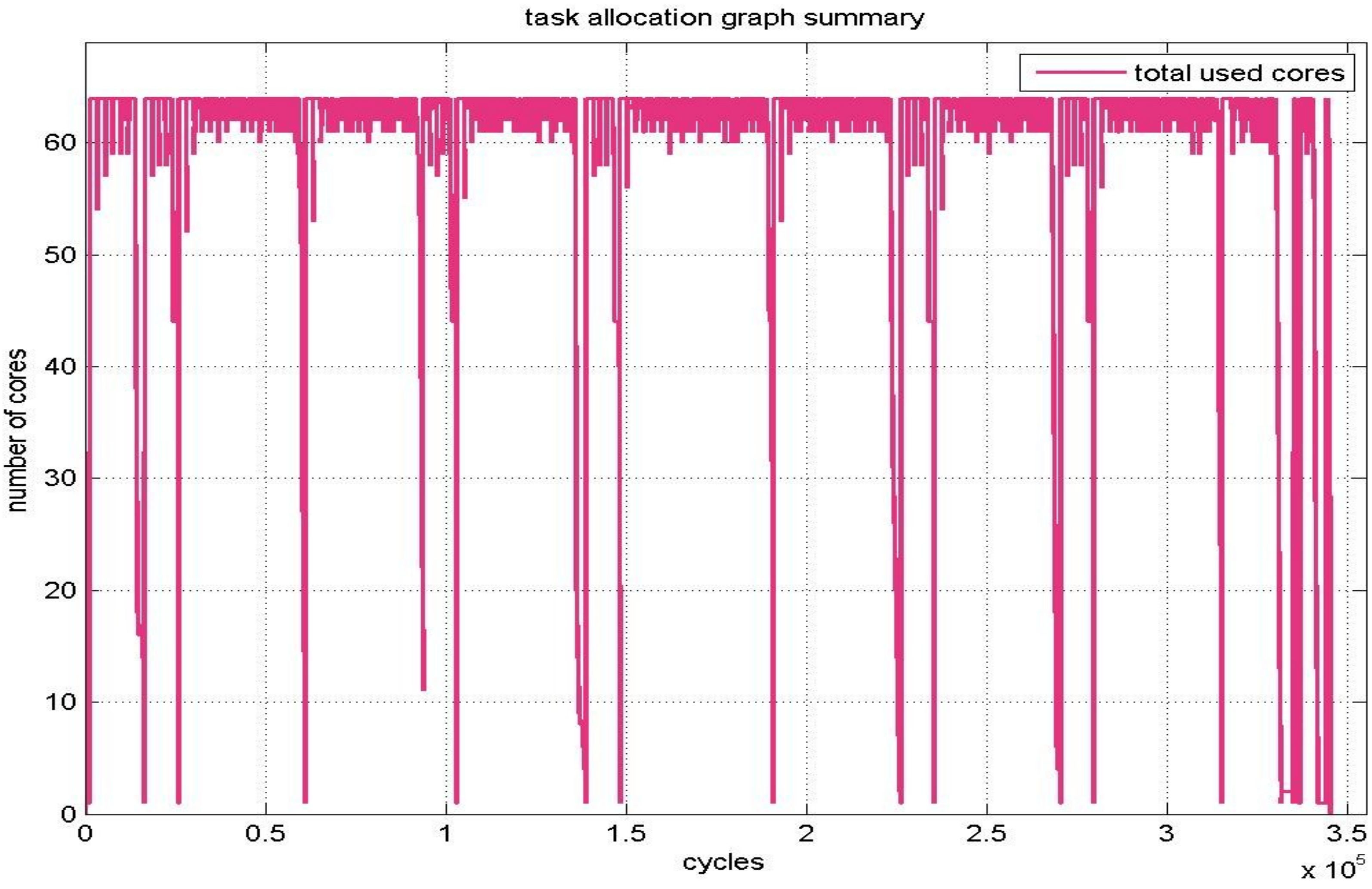
JPEG compression: pipelined / parallel



JPEG compression: Task Allocation

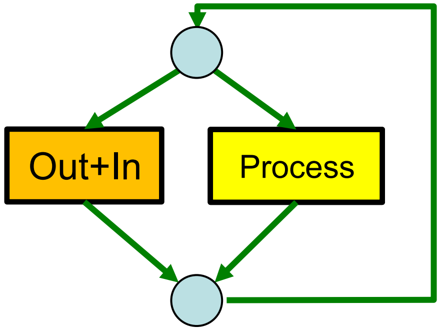


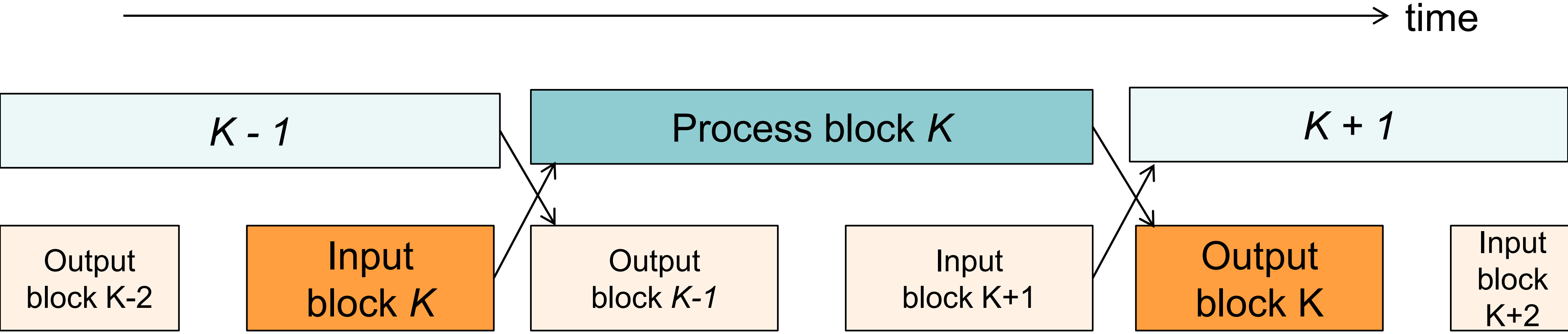
JPEG compression: Most cores active



Memory pattern

- For stream processing (e.g. JPEG), cache management is inefficient
  - Even with pre-fetching
- Instead, manage memory as local store
  - Planned data I/O in parallel with processing
  - On DMA unit or by one core
- But stream I/O is not a must
  - Any random access to data is allowed





Example: JPEG2000 Encoder

Image: $1K \times 1K$ 8b pixels

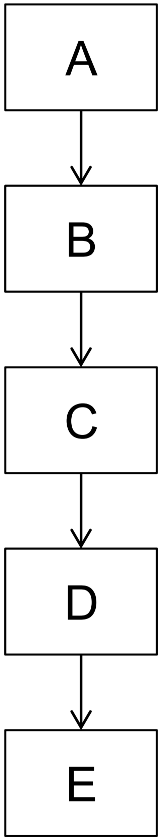
Core frequency $F_1 = 250$ MHz

Serial time $T_1 = 3.55$ sec

Parallel time $T_{64} = 400$ msec

Speed-up: $SU(64) = T_1/T_{64} \approx 9$

Efficiency: $E(64) = \frac{SU(64)}{64} = 0.14$



Serial: 220 msec

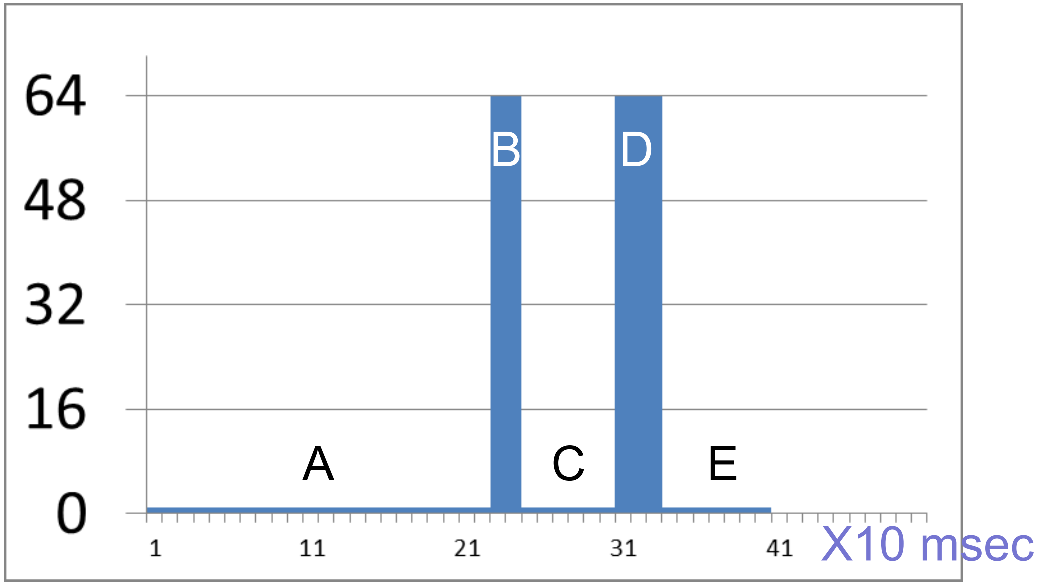
Parallel:
1280/64=20 msec

Serial: 60 msec

Parallel:
1920/64 = 30 msec

Serial: 70 msec

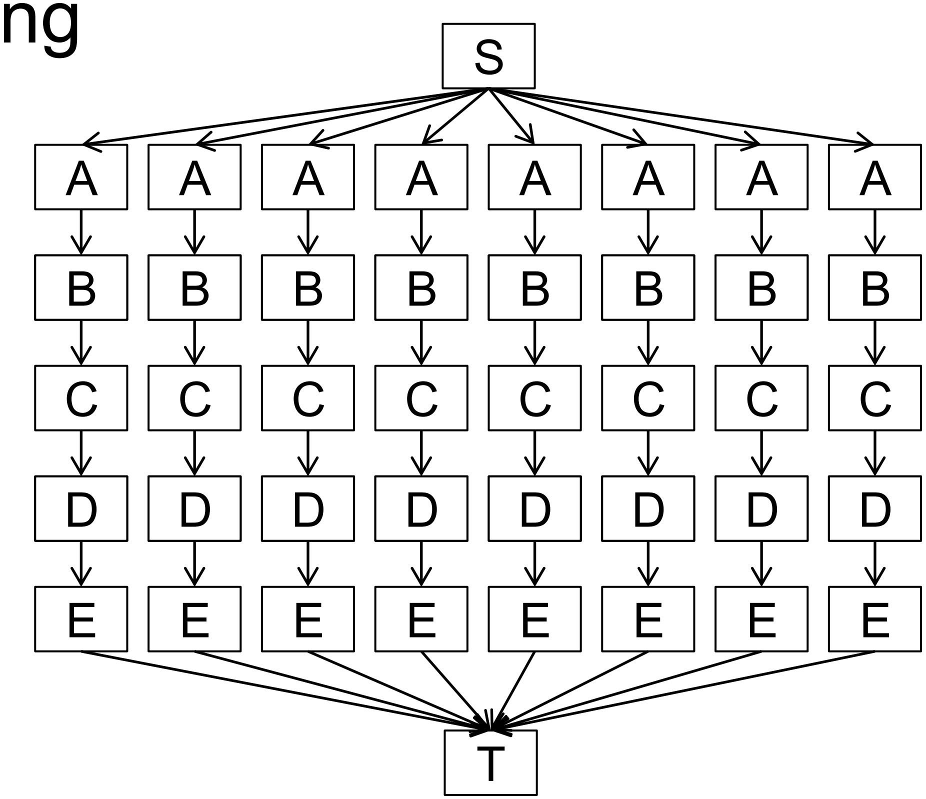
Number of busy cores



Parallel fraction f=95%

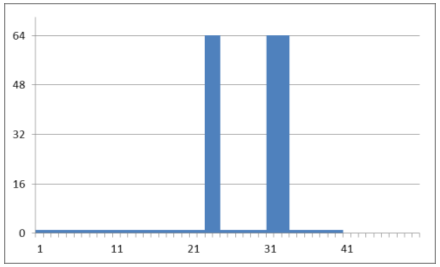
Multi-Job Scheduling

- Let’s fix the low speed-up and low efficiency
- Run multiple serial sections in parallel
- Achieved automatically with this task map and Plural scheduling



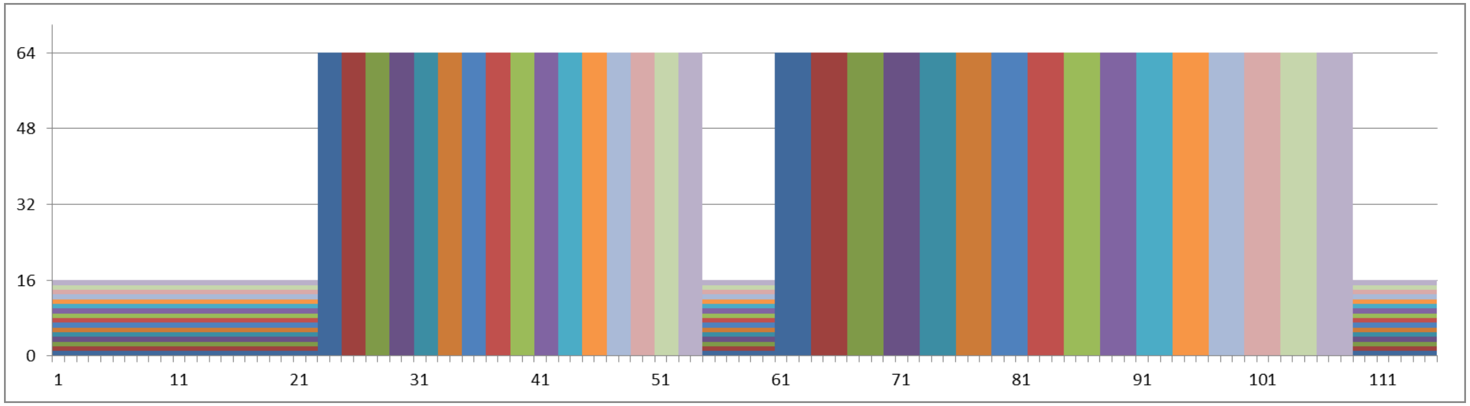
Multi-Job Scheduling

- Fixed number of cores p=64
- Job with fraction f parallel, $(1-f)$ serial
  - Time of parallel section fT_1/p
- Variable number of Jobs J=1,2,…
- Schedule:
  - J serial sections in parallel, time $T_{PS} = (1-f)T_1$
  - J parallel sections in series, time $T_{PP} = J \times fT_1/p$
- Serial time $T_S(J) = J \times T_1$
- Parallel time $T_P(J) = T_{PS} + T_{PP}$

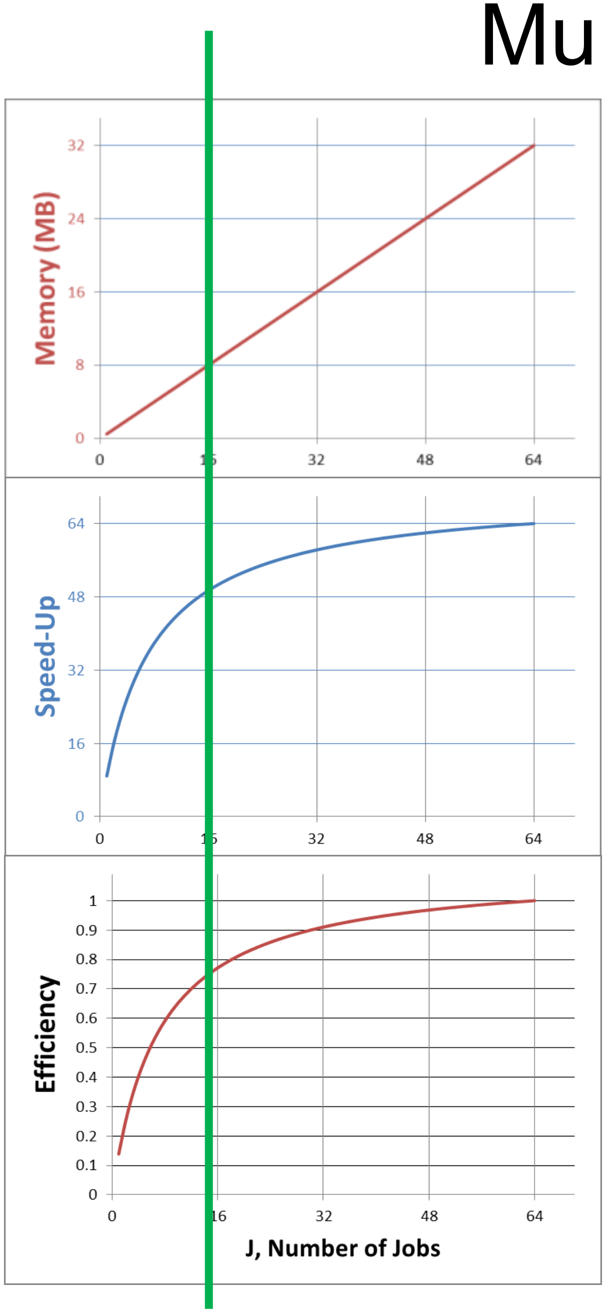


JPEG2000, J=1, f=95%

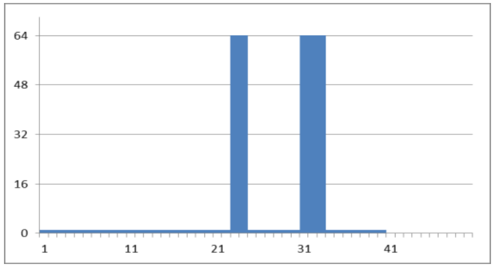
J=16



Multi-Job Scheduling

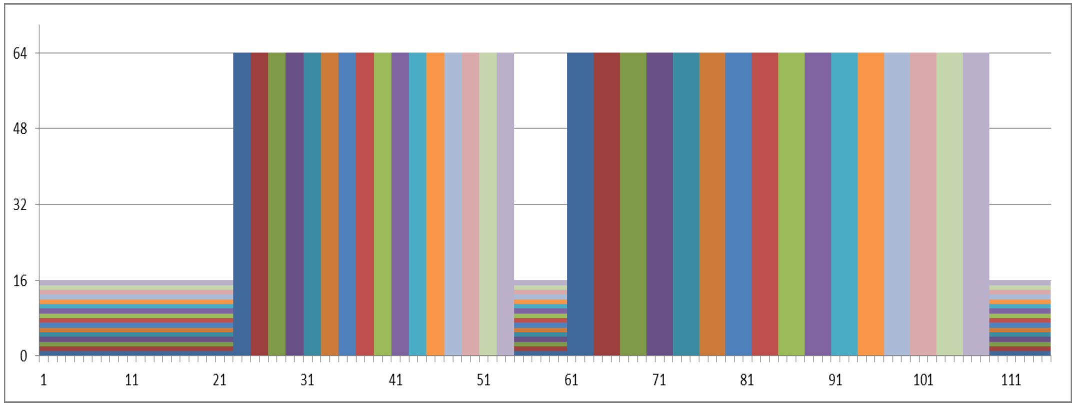


- Memory-limited
- 8MB (¼ max memory) enables:
  - J=16 jobs
  - Speed-up 50 (cf. 9)
  - 0.8 efficiency (cf. 0.14)



JPEG2000, J=1

J=16



The Plural Architecture: Some benefits

- Shared, uniform (~equi-distant) memory
  - no worry which core does what
  - no advantage to any core because it already holds the data
- Many-bank memory + fast P-to-M NoC
  - low latency
  - no bottleneck accessing shared memory
- Fast scheduling of tasks to free cores (many at once)
  - enables fine grain data parallelism
  - harder in other architectures due to:
    - task scheduling overhead
    - data locality
- Any core can do any task equally well on short notice
  - scales well
- Programming model:
  - intuitive to programmers
  - “easy” for automatic parallelizing compiler (?)

Patterns

http://parlab.eecs.berkeley.edu/wiki/_media/patterns/20090923_patterns_workshop_architecture.pptx

Productivity Layer

Applications

Structural Patterns

Choose your high level structure

Agent and repository	Layered systems
Arbitrary static task graph	Map reduce
Iterative refinement	Model view controller
Process control	Pipe-and-filter
Event based, implicit invocation	Puppeteer

Computational Patterns

Identify the key computations

Dense linear algebra	Backtrack branch and bound	Monte carlo methods
Sparse linear algebra	Finite state machine	Dynamic programming
Unstructured grids	Graphical models	Graph algorithms
Structured grids	N-body methods	Circuits
		Spectral methods

Parallel Algorithm Strategy Patterns

Refine the structure - what concurrent approach do I use? Guided re-organization

Task Parallelism	Geometric Decomposition	Data Parallelism	Pipeline	Discrete Event	Recursive Splitting

Implementation Strategy Patterns

Utilize Supporting Structures – how do I implement my concurrency? Guided mapping

Program Structure				Data Structure	
Actors	SPMD	Master/Worker		Shared queue	Distributed array
Task queue	Strict data parallel	Loop parallelism		Shared data	Graph partitioning
Fork/Join	BSP			Shared hash table	Memory parallelism

Efficiency Layer

Concurrent Execution Patterns

Implementation methods – what are the building blocks of parallel programming? Guided implementation

Advancing Program Counters		Coordination		
MIMD	Thread pool	Message passing	Mutual exclusion	Digital circuits
Task graph	Speculation	Collective communication	Transaction al memory	
SIMD	Data flow	Collective synchronization	P2P synchronization	

On-going Research

- Mathematical model incl. memories
- Scaling: full chip, multiple chips
- Accelerator for super-computing
- Plural algorithms and Plural programming
- FPGA versions
- Better NoC to shared memory
- Better scheduler and NoC to scheduler
- Near/sub-threshold for extremely low energy/power
  - Using asynchronous logic design
- 3D for larger ‘on-chip’ memory
- High-reliability version (rad-hard)
- Converting large message-passing programs to shared-memory plus message passing codes

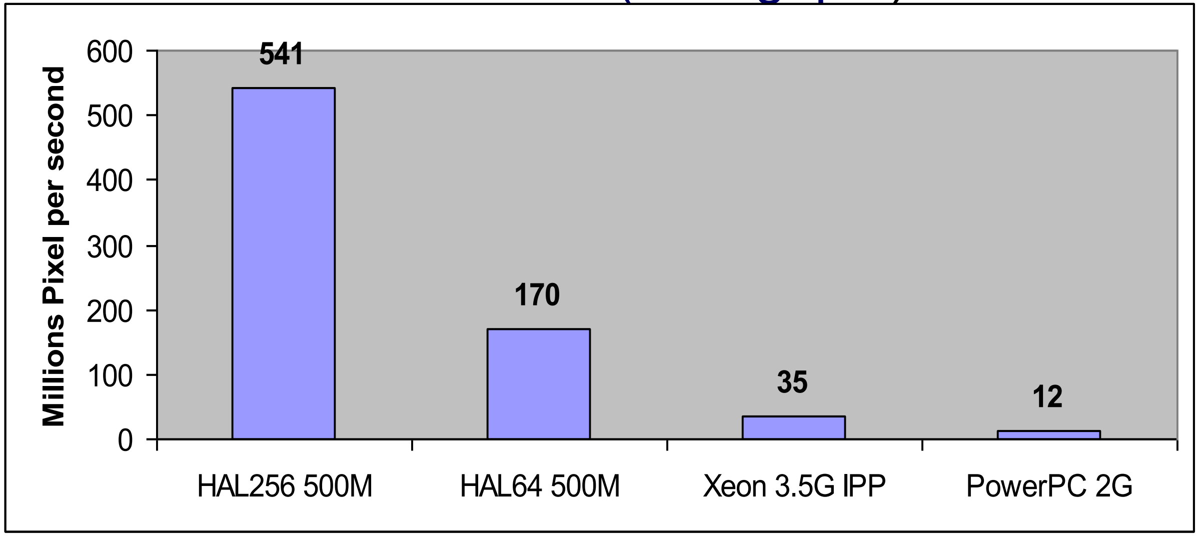
Summary

- Simple many-core architecture
  - Inspired by PRAM
- Hardware scheduling
- Task-based programming model
- Designed to achieve the goal of 'more cores, less power'
- Developing model to illuminate / investigate

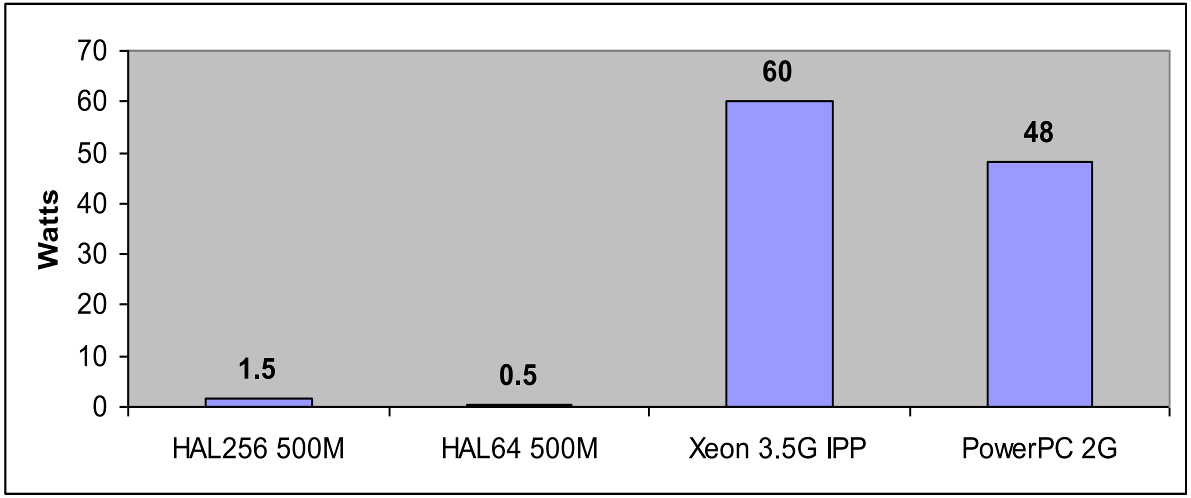
Backup

JPEG Benchmark
Plural vs. Xeon 3.5G & PowerPC 2G

Performance (throughput)



Power Consumption



EEMBC Imaging/Graphics Benchmark

Iterations/sec

	Frequency	Power	High-Pass grayscale filter	RGB to CMYK Conversion	RGB to YIQ Conversion
Plural 64	500 MHz	1 W	5,120	8,752	5,600
Plural 256	500 MHz	5 W	18,897	30,270	23,012
IBM 970FX (PowerPC)	2 GHz	60 W	1,609	1,404	1,131
AMD Geode LX800	500 MHz	1.8 W	104	205	82
ST231	300 MHz		313	237	207

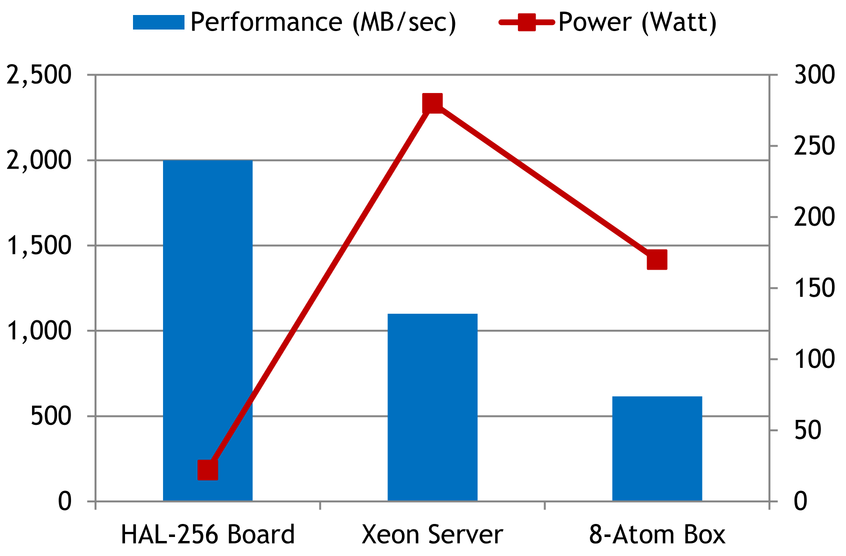
Benchmark Example: Data Compression for Storage Applications

Processor Comparison

Processor	#Cores	Frequency	Performance (MB/sec)	Power (Watt)	Performance /Watt
Plural-128 (FPGA)	128	200 MHz	1,000	9	111
Xeon	4	3.3 GHz	1,100	130	8
Atom	1	1.6 GHz	77	3	31

System Comparison

System Type	Performance (MB/sec)	Power (Watt)	Performance /Watt
HAL-256 Board	2,000	22	91
Xeon Server	1,100	280	4
8-Atom Box	616	170	4



- Benchmark based on Calgary Corpus http://corpus.canterbury.ac.nz/index.html
- Algorithm used: QuickLZ level 1

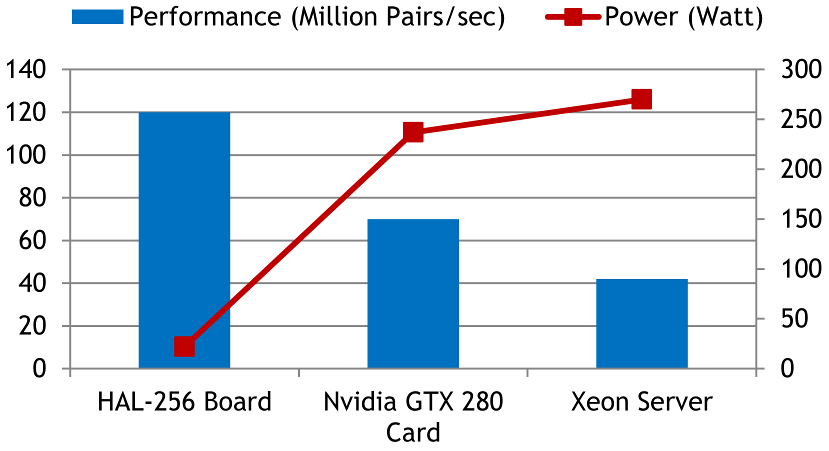
Benchmark Example:
Radix Sort for Database Applications

Processor Comparison

Processor	#Cores	Frequency	Performance (Million Pairs/sec)	Power (Watt)	Performance / Watt
Plural (FPGA)	128	200 MHz	60	9	6.7
Nvidia GTX 280	30 SM	1.3 GHz	70	237	0.3
Intel Core2 Xeon E5345 ("Clovertown")	4	2.16 GHz	42	120	0.4

System Comparison

Server Type	Performance (Million Pairs/sec)	Power (Watt)	Performance/ Watt
Plural-256 FPGA Boar	120	12	10.3
Nvidia Graphix Card	70	237	0.3
Xeon Server	42	270	0.2



Benchmark based on Satish,Harris & Garland ,"*Designing Efficient Sorting Algorithms for Manycore GPUs*", 23rd IEEE Int. Parallel & Distributed Processing Symposium, May 2009

MVM Covariance

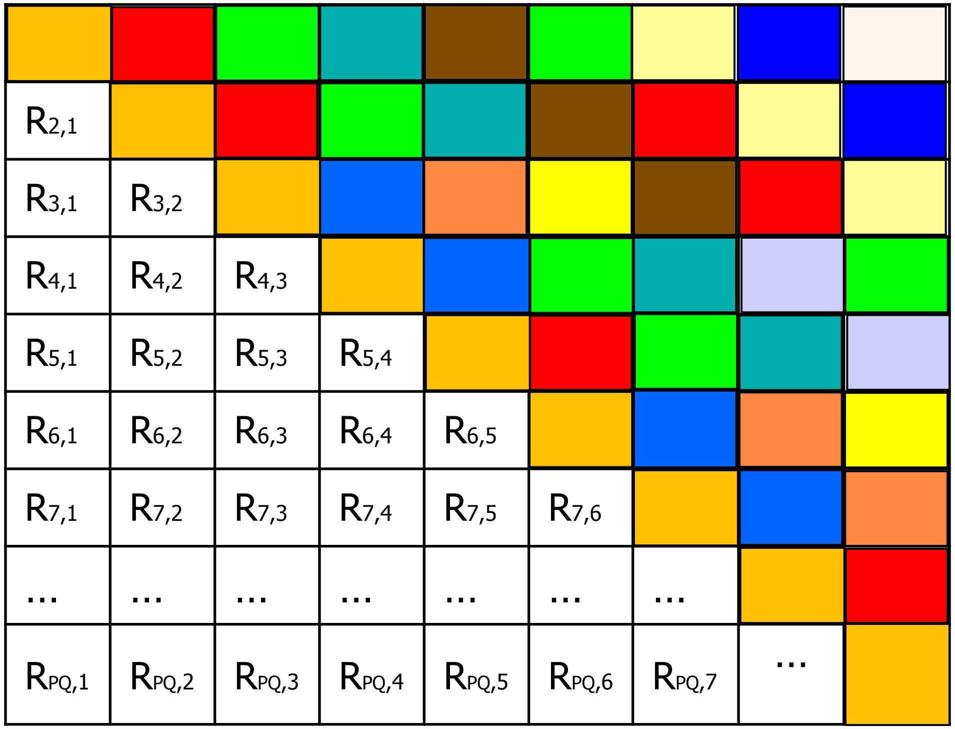
- Compute covariance matrix for SAR imaging
- Each product repeated many times
  - Naïve 32x32 block: 11.4M products
  - Ideally, only need 208K products (x28)
- Basic algorithm is difficult to parallelize
- Main difficulties:
  - Simultaneous writes into same cell require locks
  - Large memory needed to hold Cov matrix
    - Each matrix 125KB

New algorithm key features

- Optimal number of products (each product executed only once)
- Memory efficient—data reuse
- Highly parallel

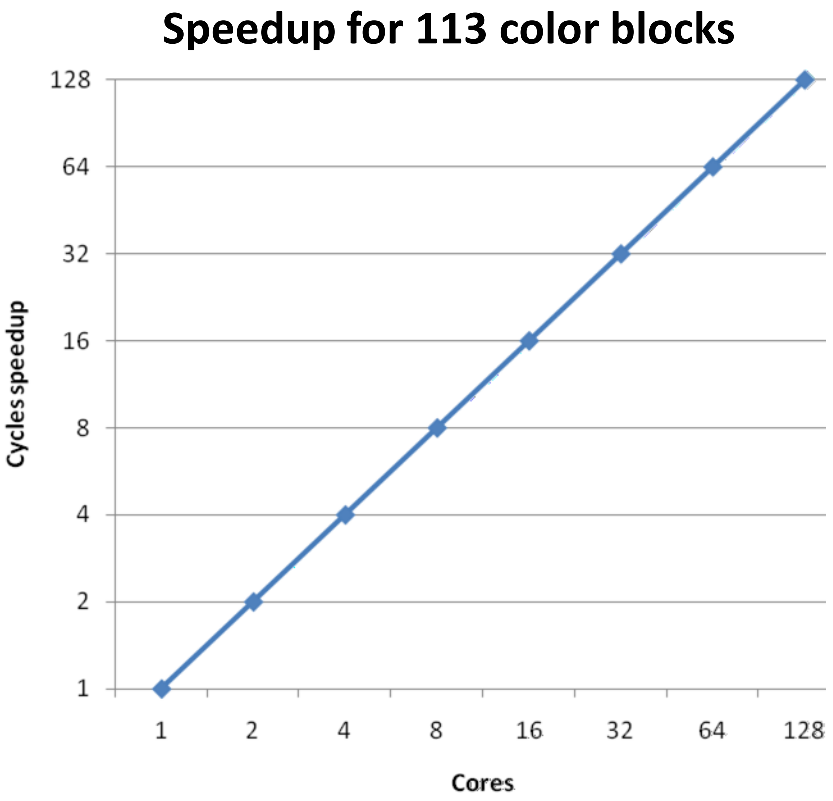
New MVM algorithm

- A product is computed once, and added to all cells that need it
- Always on same diagonal
- Only to same-color cells
- No overlap between cells of different colors
- Each color is one task



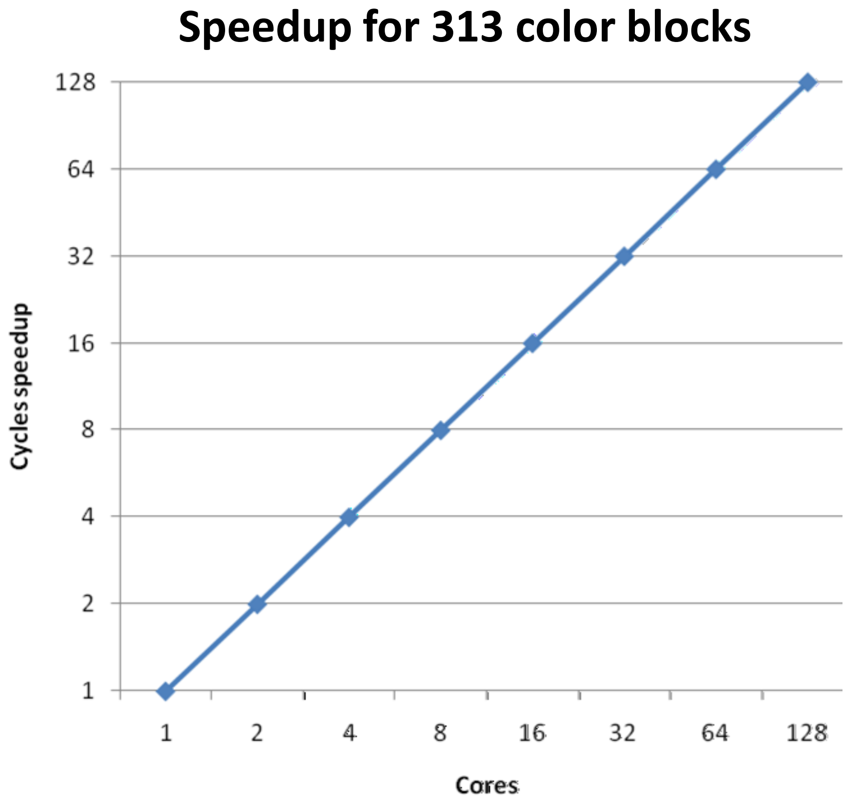
$R_{2,1}$								
$R_{3,1}$	$R_{3,2}$							
$R_{4,1}$	$R_{4,2}$	$R_{4,3}$						
$R_{5,1}$	$R_{5,2}$	$R_{5,3}$	$R_{5,4}$					
$R_{6,1}$	$R_{6,2}$	$R_{6,3}$	$R_{6,4}$	$R_{6,5}$				
$R_{7,1}$	$R_{7,2}$	$R_{7,3}$	$R_{7,4}$	$R_{7,5}$	$R_{7,6}$			
…	…	…	…	…	…	…		
$R_{PQ,1}$	$R_{PQ,2}$	$R_{PQ,3}$	$R_{PQ,4}$	$R_{PQ,5}$	$R_{PQ,6}$	$R_{PQ,7}$	…	

Speedup is linear



Speedup for 113 color blocks

Chip size: 20x20
Sub-Aparture size: 8x8



Speedup for 313 color blocks

Chip size: 32x32
Sub-Aparture size: 13x13

Compare RC64 to Intel dual-core

- RC64: 3 watt, 200MHz, IPC=1x64
- Intel Core 2 Duo: 65 Watt, 2.4 GHz, IPC=1.3

$$\frac{Perf}{Power} ratio \frac{RC64}{Intel\ Core2duo} = \frac{\frac{freq \cdot IPC}{power}}{\frac{freq \cdot IPC}{power}} = \frac{0.2 \cdot 64}{2.5 \cdot 1.3} \times \frac{65}{3} = 85$$

Performance ratio: RC64 4× faster

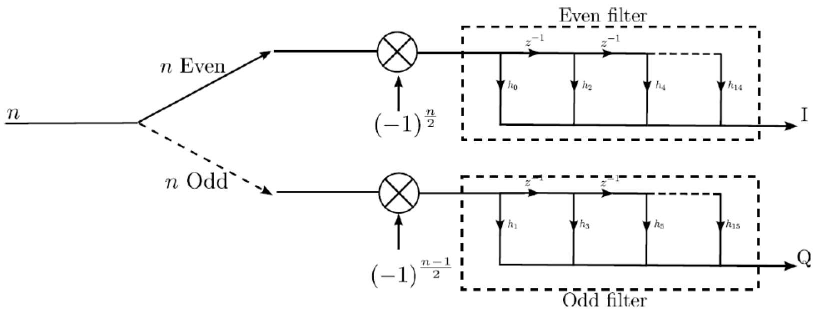
Power ratio: RC64 22× lower power

ESA NGDSP Benchmark B5

- Benchmark #5 on ESA/ESTEC *Next generation space digital signal processor software benchmark* 2008
- Demodulator followed by LPF-based 4/5 decimator

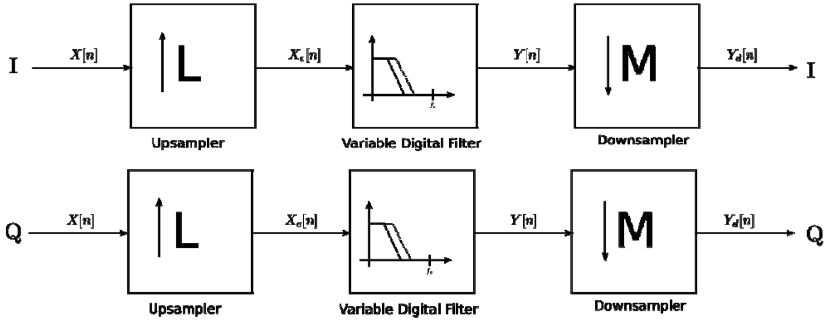
FIR-I

FIR-Q

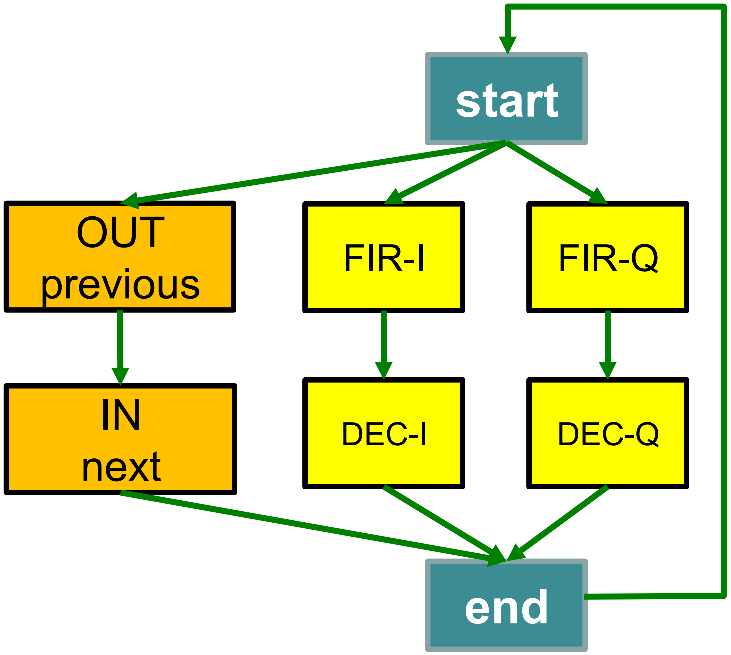


DEC-I

DEC-Q



B5 task map



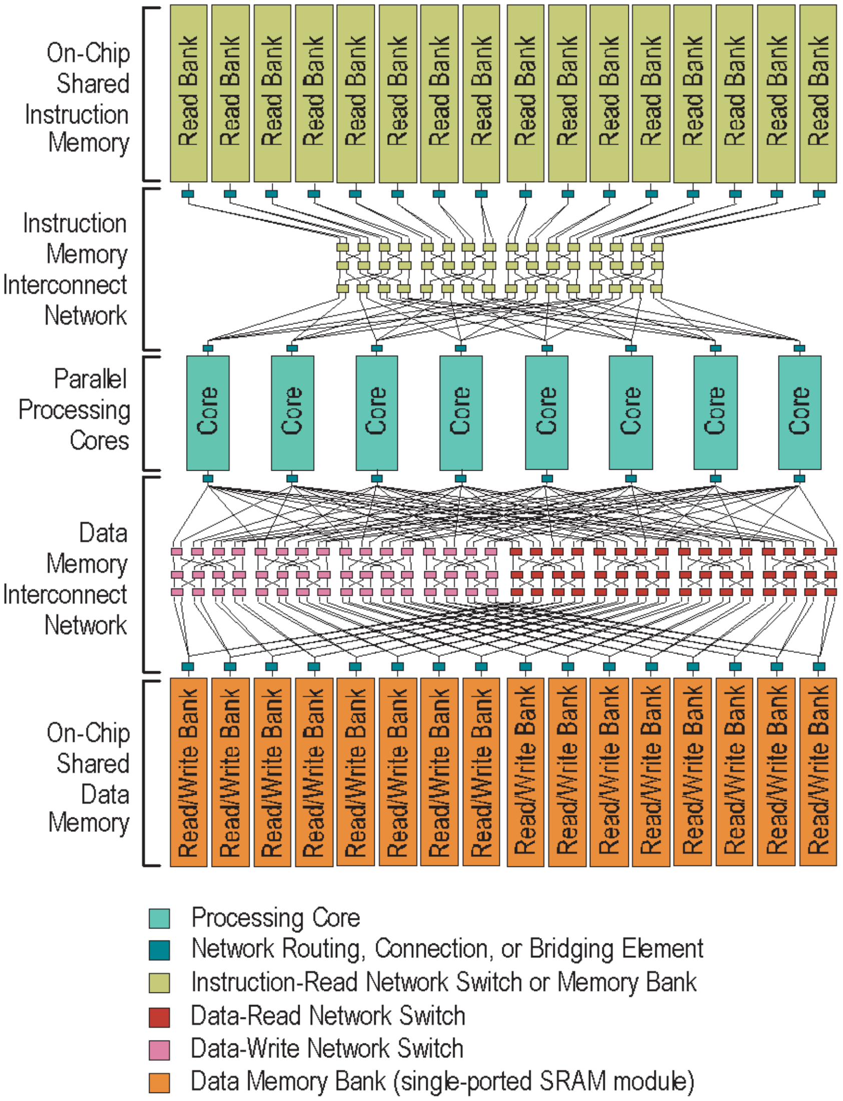
B5 results

Clock frequency	200MHz
Cores	64
FPU	N/A
Multipliers	16
GIPS	10 GIPS
GFLOPS	2.5 GFLOPS
Memory on-chip	512 KByte
I/O data rate	1.6 GByte/sec

I/O data rate required for B5: 331 Mbytes/sec

Benchmark 5: Radar Remote Sensing data processing	Maximum data throughput [samples/sec]	Processing latency [clock cycles / ns]	Consumed processing power [%]
I/Q demodulation only	124M sps	10M samples: 16.1Mcycles, 80ms	67%
Decimation filter only (I and Q in parallel using synthetic input)	249M sps	10M samples: 8Mcycles, 40ms	33%
Overall (I/Q demodulation and decimation of I and Q)	82.8M sps	Per one sample: 2.4 cycles, 12.1 ns For 10M samples: 24.1Mcycles, 121 msec	100%

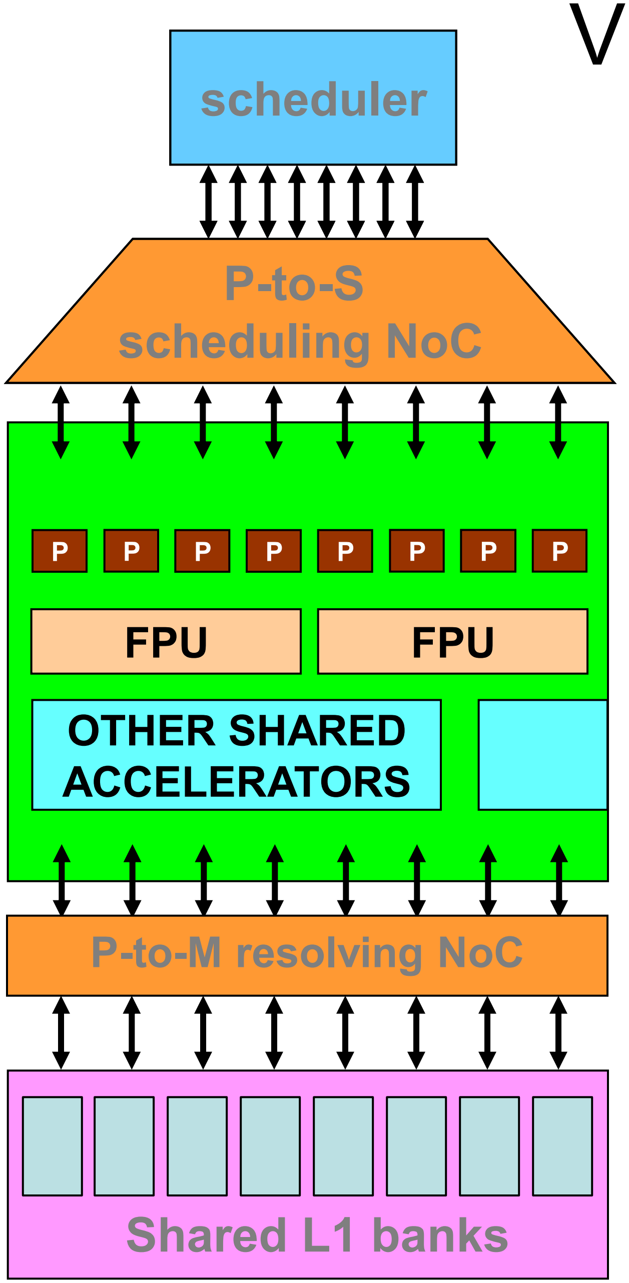
Shared Memory System


On-Chip Shared Instruction Memory
Read Bank
Instruction Memory Interconnect Network
Parallel Processing Cores
Core
Data Memory Interconnect Network
On-Chip Shared Data Memory
Read/Write Bank
🟩 Processing Core
🟦 Network Routing, Connection, or Bridging Element
🟨 Instruction-Read Network Switch or Memory Bank
🟥 Data-Read Network Switch
🟪 Data-Write Network Switch
🟧 Data Memory Bank (single-ported SRAM module)

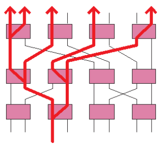
Shared memory system

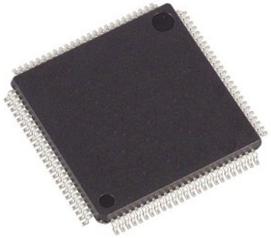
- Many SRAM blocks
- Per cycle arbitration
- 2-3 cycle latency
- Cores retry if fail
- Concurrent Read

Variations



- Shared accelerators, e.g. FPU, mult/div, collectives
- Instructions for specific applications, e.g. encrypt, compress, search, vectors
- Ext. memory interfaces for higher BW
  - Prefetch, DMA
- Separate data / instructions memories
  - Multicast instruction fetch
  - Duplicate instruction caches
  - Separate data and instruction access NoCs
- Separate read/write NoCs
  - Read conflict resolution: multicast
  - Write conflict resolution:
    - Serialize, or
    - combine (fetch-&-op, prefix-sum)
    - Support CRCW PRAM
      - Common / priority / arbitrary
- Modified scheduling / dispatch / synchronization
  - Pre-dispatch and queue tasks to busy cores





Many-cores: Supercomputer-on-chip
How many? And how?

(how not to?)

Ran Ginosar

Technion

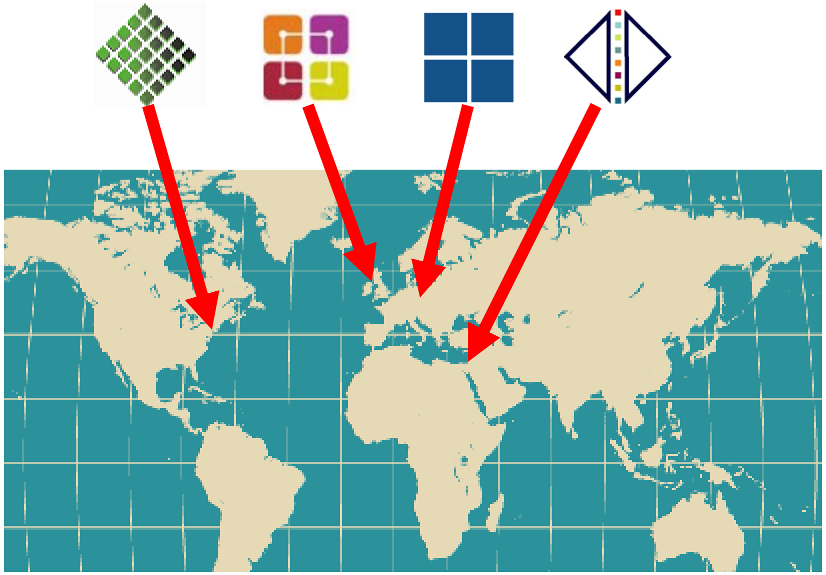
Mar 2010

Many-cores

- CMP / Multi-core is “more of the same”
  - Several high-end complex powerful processors
  - Each processor manages itself
  - Each processor can execute the OS
  - Good for many unrelated tasks (e.g. Windows)
  - Reasonable on 2–8 processors, then it breaks
- Many-cores
  - 100 – 1,000 – 10,000
  - Useful for heavy compute-bound tasks
  - So far (50 years) many disasters
    - But there is light at the end of the tunnel ☺

Agenda

- Review 4 cases
- Analyze
- How *NOT* to make a many-core



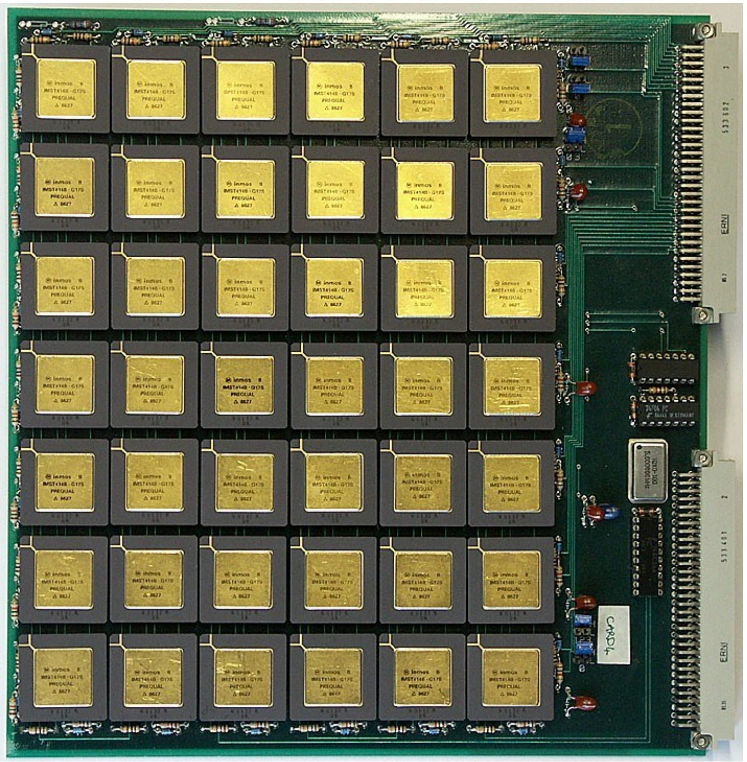
Many many-core contenders

- Ambric
- Aspex Semiconductor
- ATI GPGPU
- BrightScale
- ClearSpeed Technologies
- Coherent Logix, Inc.
- CPU Technology, Inc.
- Element CXI
- Elixent/Panasonic
- IBM Cell
- IMEC
- Intel Larrabee
- Intellasys
- IP Flex
- MathStar
- Motorola Labs
- NEC
- Nvidia GPGPU
- PACT XPP
- Picochip
- Plurality
- Rapport Inc.
- Recore
- Silicon Hive
- Stream Processors Inc.
- Tabula
- Tilera

(many are dead / dying / will die / should die)

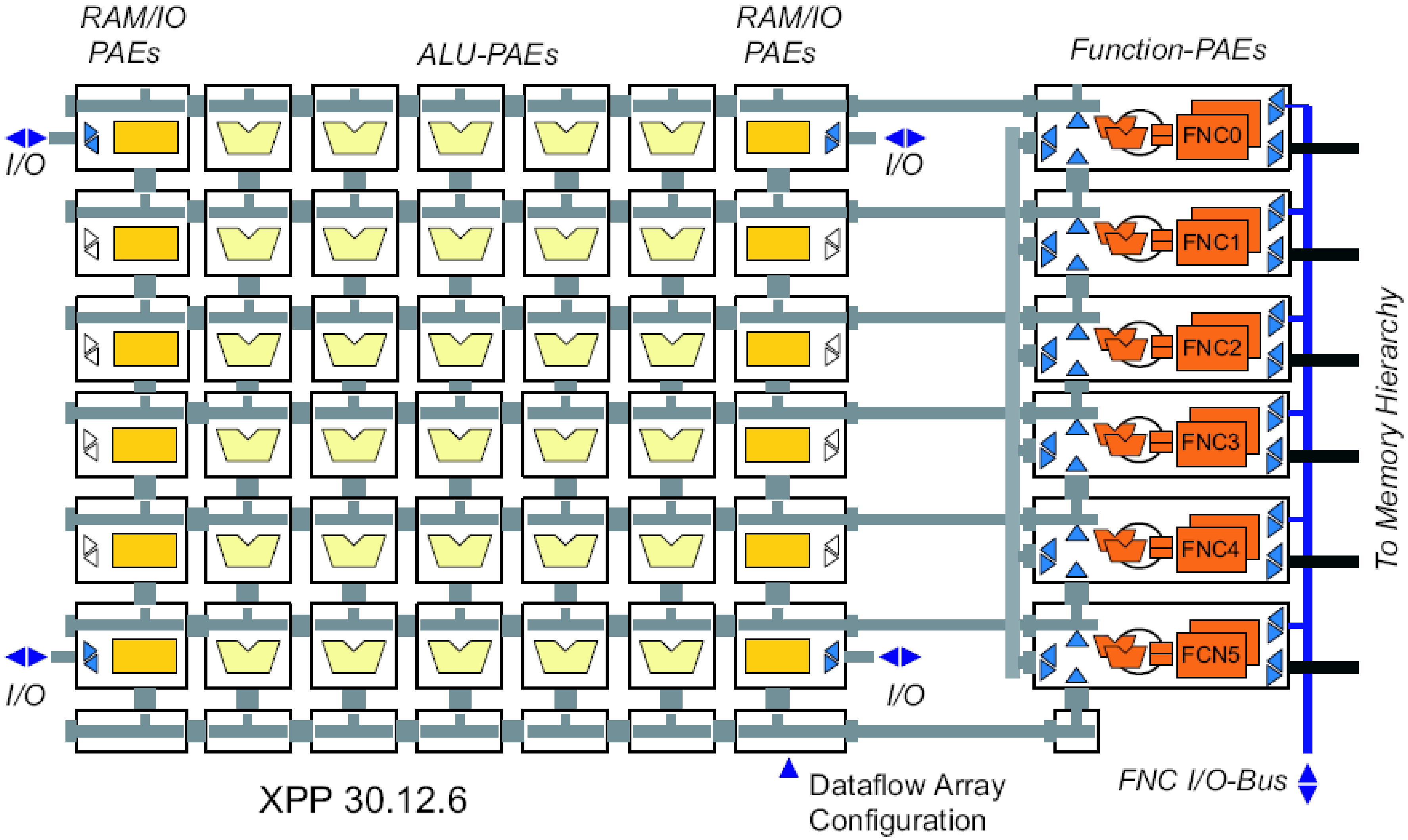
PACT XPP

- German company, since 1999
  - Martin Vorbach,
    an ex-user of Transputers

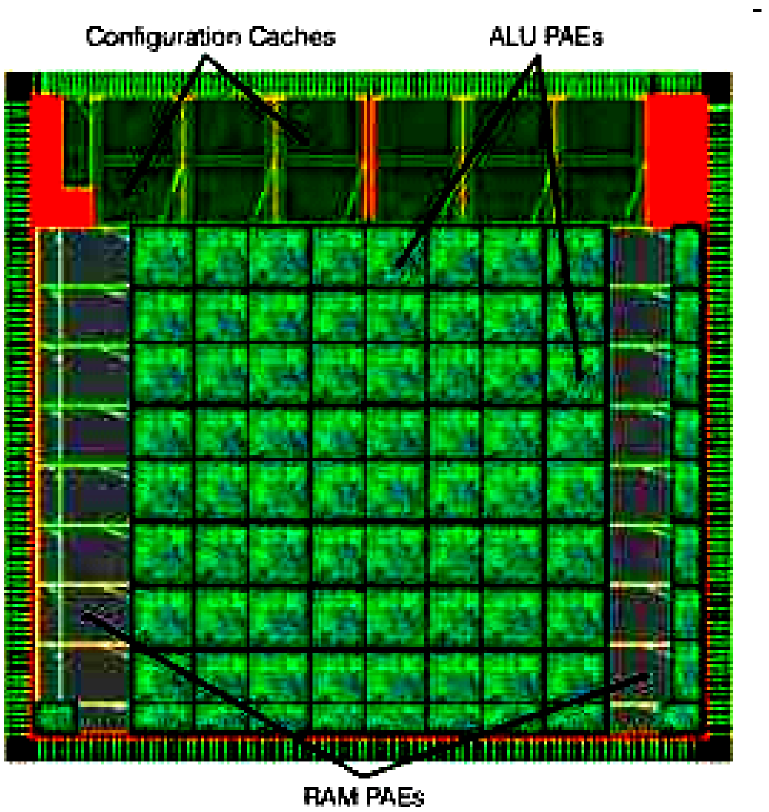


42x Transputers mesh 1980s

PACT XPP (96 elements)

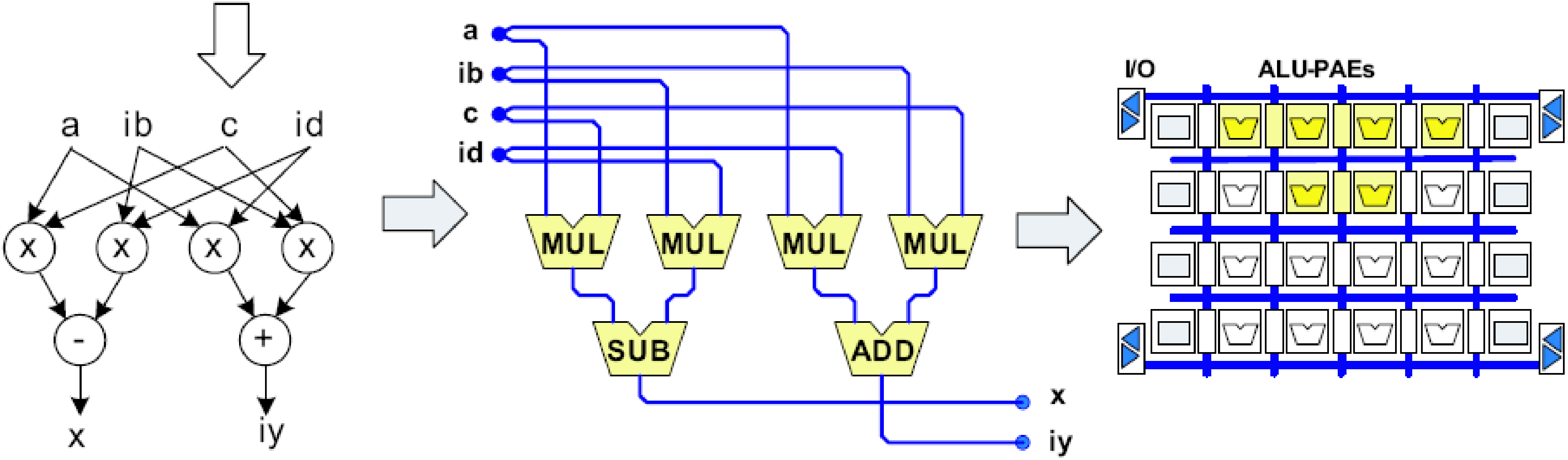


PACT XPP die photo

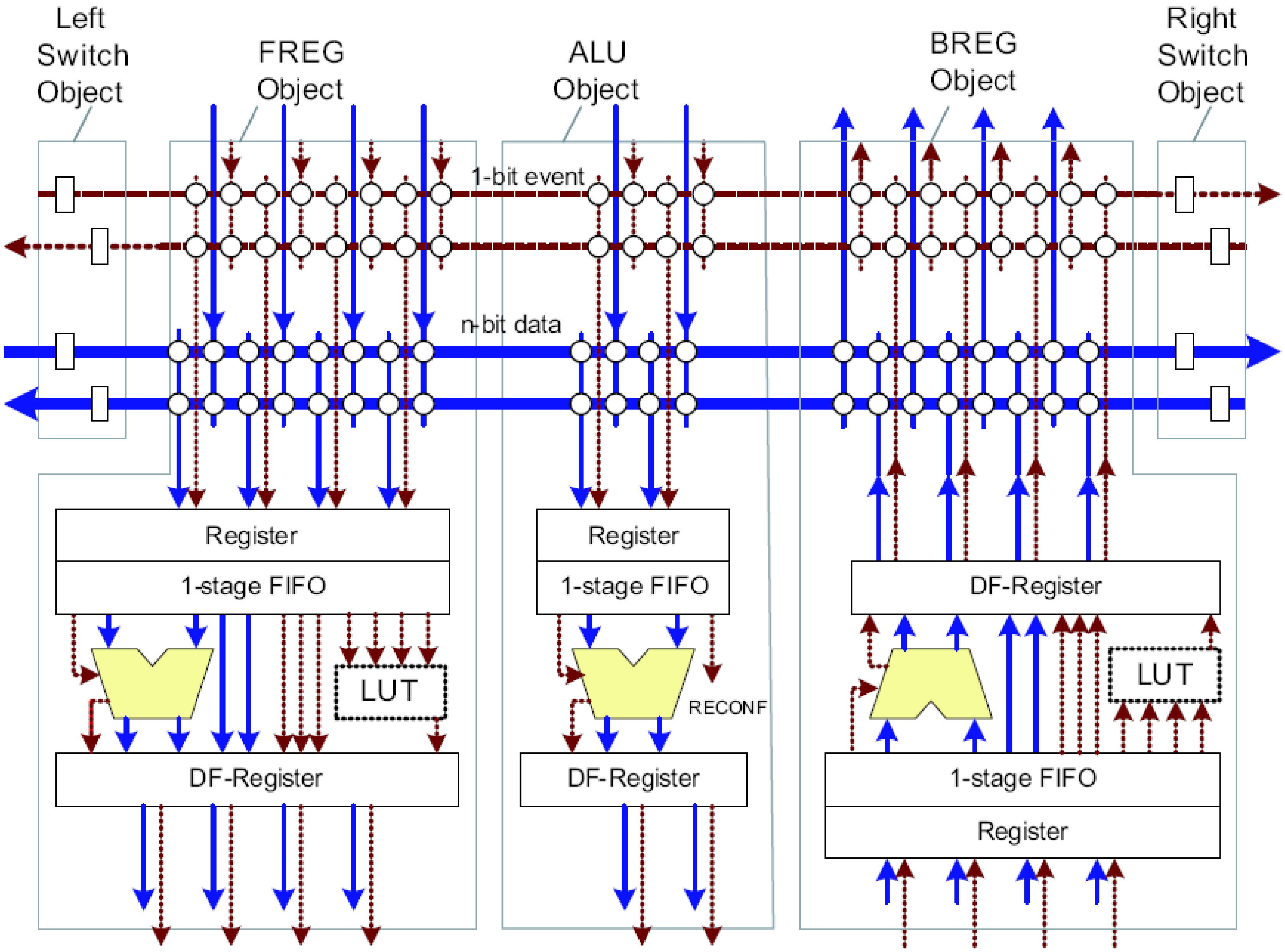


PACT: Static mapping, circuit-switch reconfigured NoC

$$
\begin{aligned}
x + iy &= (a+ib) * (c+id) \\
&= (ac - bd) + i\,(ad + bc)
\end{aligned}
$$

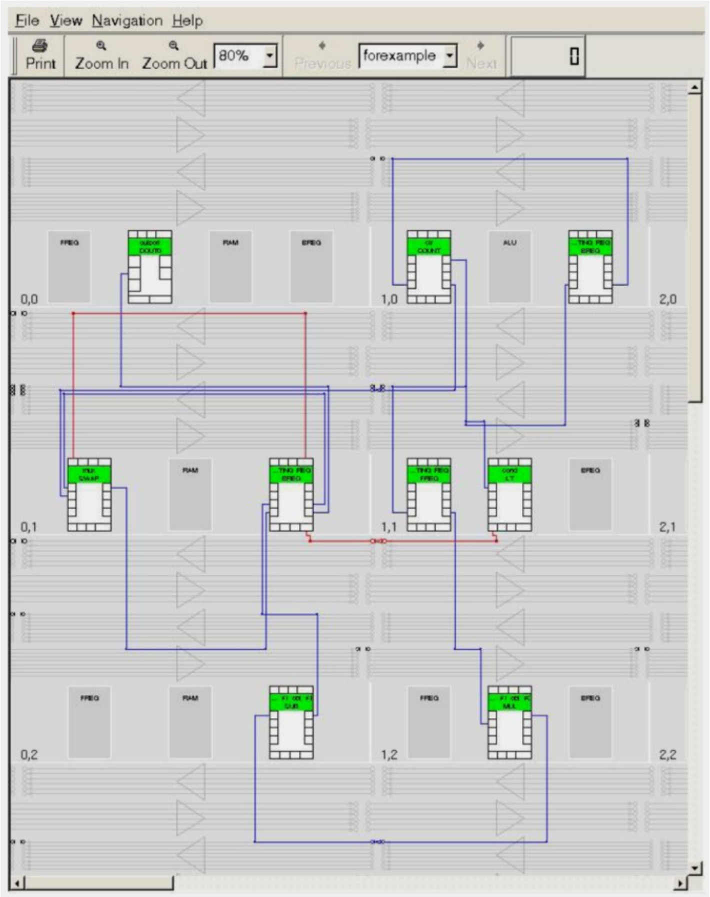


PACT ALU-PAE


Left Switch Object
FREG Object
ALU Object
BREG Object
Right Switch Object
1-bit event
n-bit data
Register
1-stage FIFO
LUT
DF-Register
Register
1-stage FIFO
RECONF
DF-Register
DF-Register
LUT
1-stage FIFO
Register

PACT

- Static task mapping ☹
  - And a debug tool for that

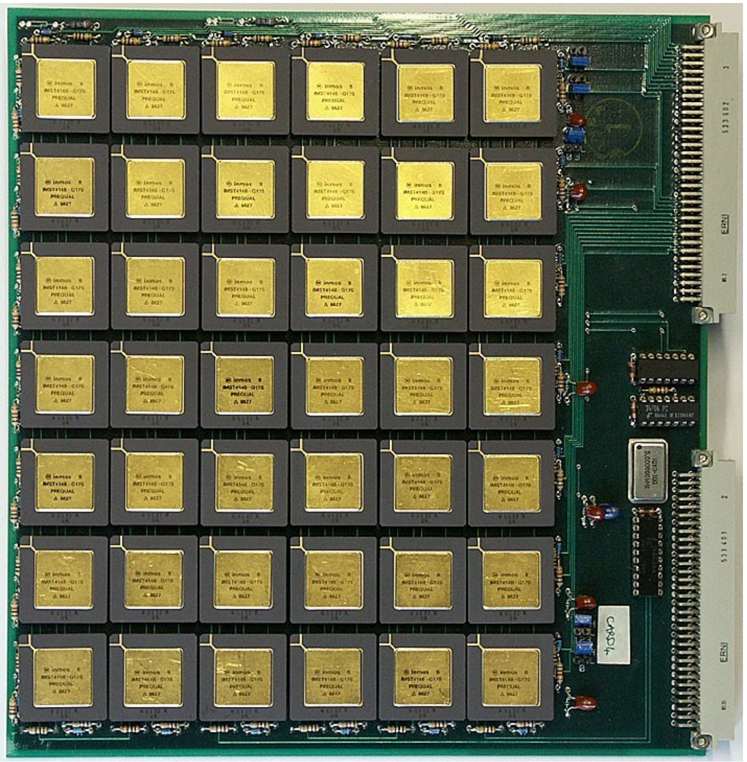


PACT analysis

- Fine granularity computing ☺
- Heterogeneous processors ☹
- Static mapping
  - → complex programming ☹
- Circuit-switched NoC → static reconfigurations
  - → complex programming ☹
- Limited parallelism
- Doesn’t scale easily

picoChip

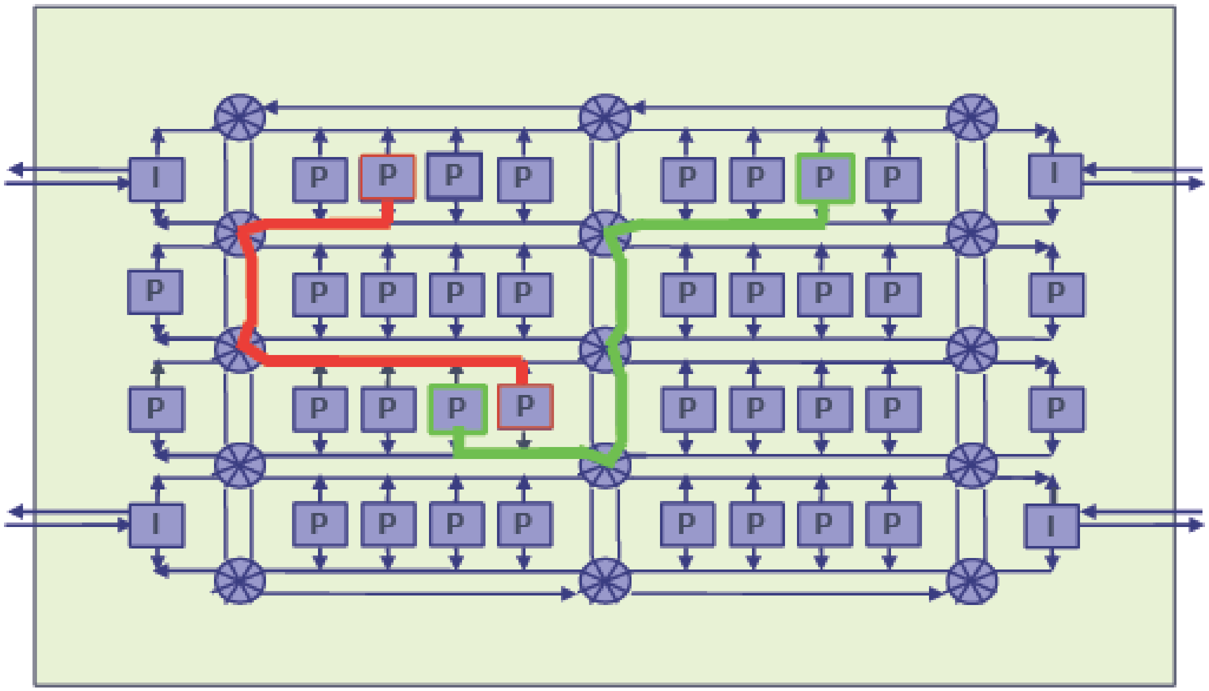
- UK company
- Inspired by Transputers (1980s), David May



42x Transputers mesh 1980s

picoChip

The picoArray concept : Architecture overview



322x
16-bit LIW RISC

P Processor

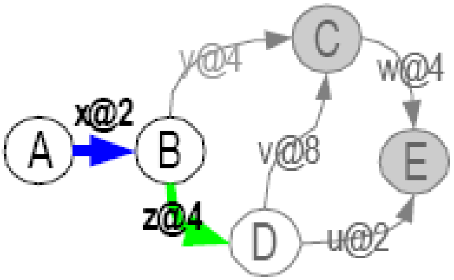
I Inter-picoArray Interface or Asynchronous Data Interface

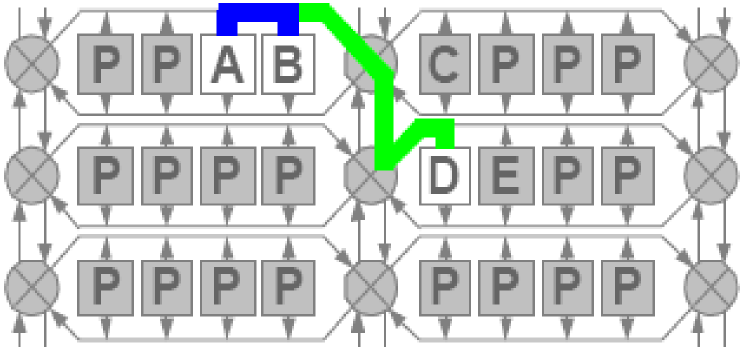
✳ Switch Matrix

🟥🟩 Example signal flows

picoChip

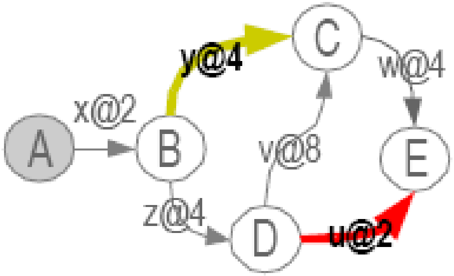
The picoArray concept : picoBus

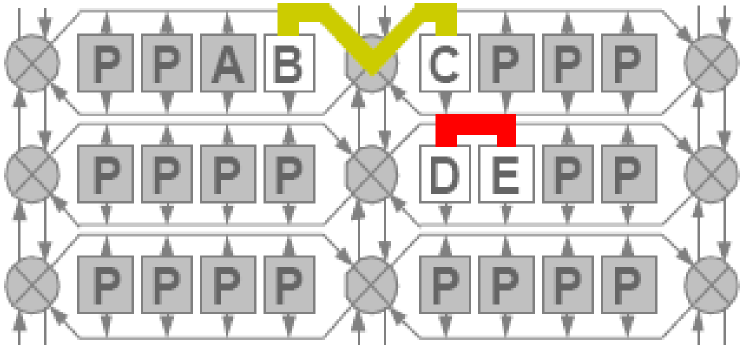




	0	1	2	3	4	5	6	7
U		🟥		🟥		🟥		🟥
V								▨
W			🟪				🟪	
X	🟦		🟦		🟦		🟦	
Y		🟨				🟨		
Z	🟩				🟩			

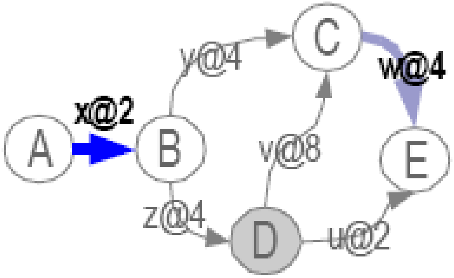
picoChip

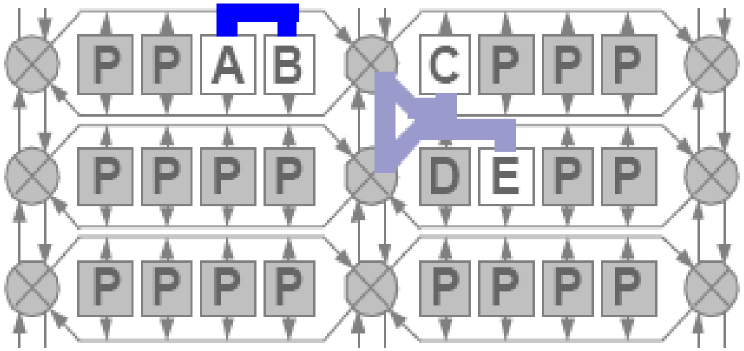




	0	1	2	3	4	5	6	7
U		🟥		🟥		🟥		🟥
V								▨
W			🟪				🟪	
X	🟦		🟦		🟦		🟦	
Y		🟨				🟨		
Z	🟩				🟩			

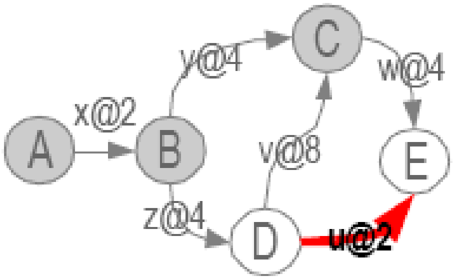
picoChip

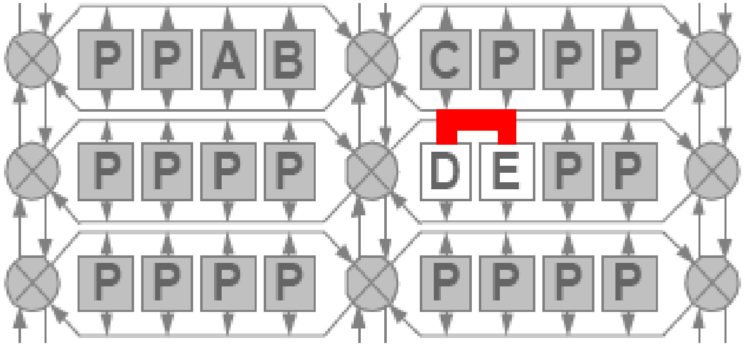




	0	1	2	3	4	5	6	7
U		🟥		🟥		🟥		🟥
V								🟦
W			🟪				🟪	
X	🟦		🟦		🟦		🟦	
Y		🟨				🟨		
Z	🟩				🟩			

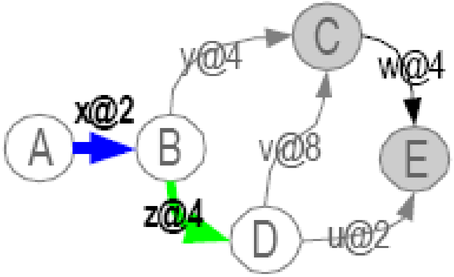
picoChip

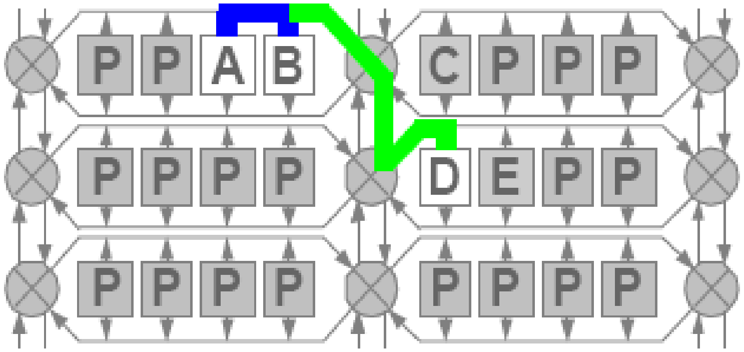




	0	1	2	3	4	5	6	7
U		🟥		🟥		🟥		🟥
V								🟦
W			🟪				🟪	
X	🟦		🟦		🟦		🟦	
Y		🟨				🟨		
Z	🟩				🟩			

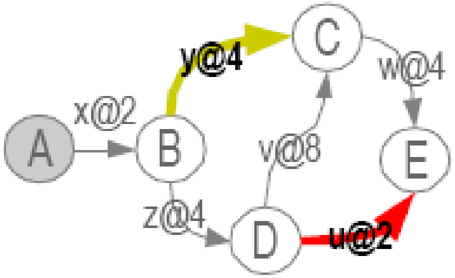
picoChip

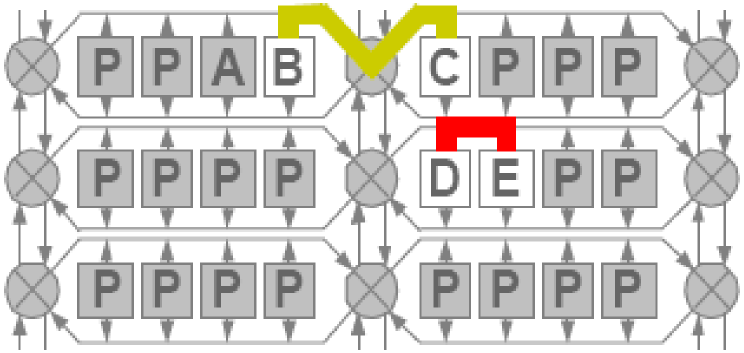




	0	1	2	3	4	5	6	7
U		🟥		🟥		🟥		🟥
V								▨
W			🟪				🟪	
X	🟦		🟦		🟦		🟦	
Y		🟨				🟨		
Z	🟩				🟩			

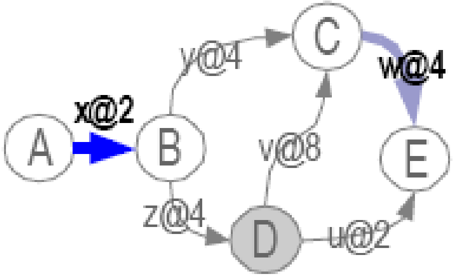
picoChip

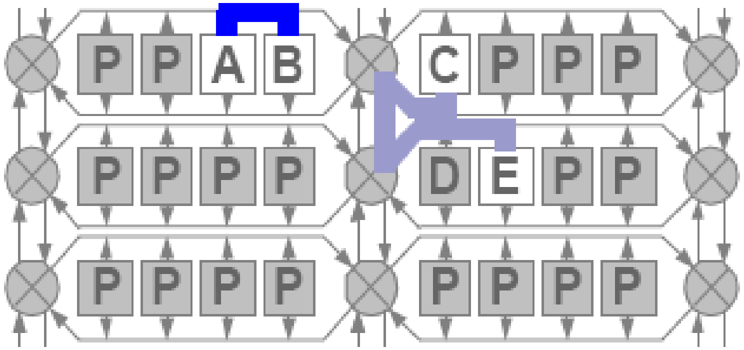




	0	1	2	3	4	5	6	7
U		🟥		🟥		🟥		🟥
V								■
W			🟪				🟪	
X	🟦		🟦		🟦		🟦	
Y		🟨				🟨		
Z	🟩				🟩			

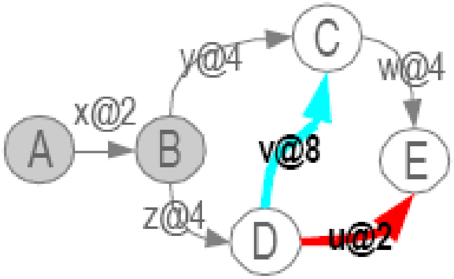
picoChip

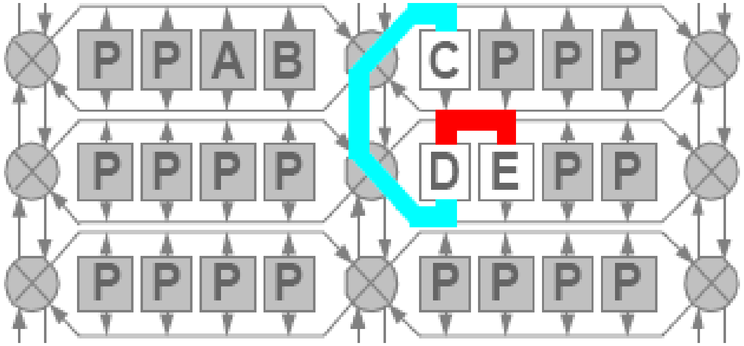




	0	1	2	3	4	5	6	7
U		🟥		🟥		🟥		🟥
V								■
W			🟪				🟪	
X	🟦		🟦		🟦		🟦	
Y		🟨				🟨		
Z	🟩				🟩			

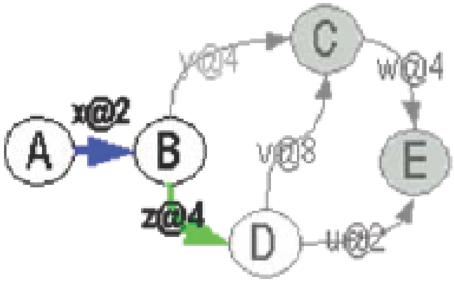
picoChip

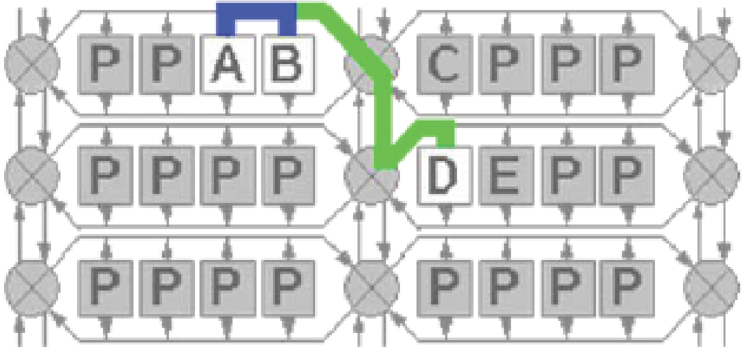


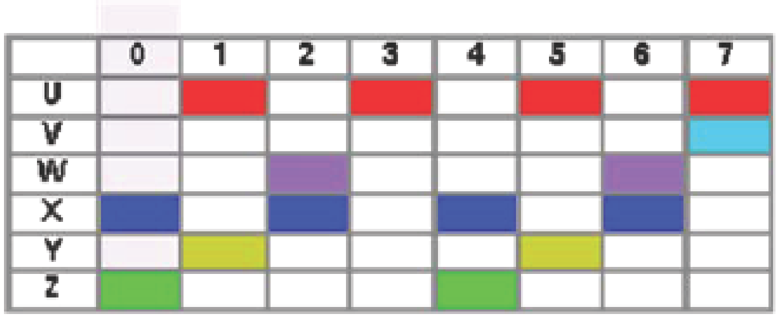


	0	1	2	3	4	5	6	7
U		🟥		🟥		🟥		🟥
V								🟦
W			🟪				🟪	▨
X	🟦		🟦		🟦		🟦	▨
Y		🟨				🟨		▨
Z	🟩				🟩			▨

picoChip





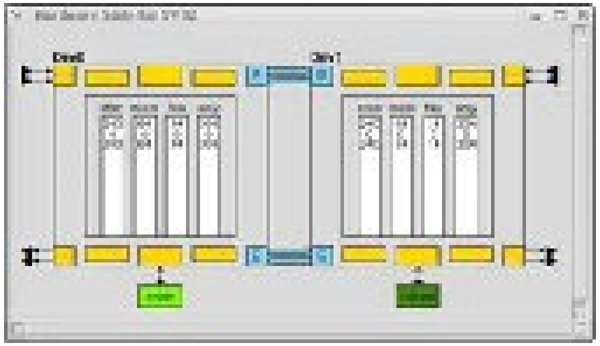


	0	1	2	3	4	5	6	7
U		🟥		🟥		🟥		🟥
V								■
W			🟪				🟪	
X	🟦		🟦		🟦		🟦	
Y		🟨				🟨		
Z	🟩				🟩			

picoChip: Static Task Mapping ☹

Compile

⇩



⇩

picoChip analysis

- MIMD, fine granularity, homogeneous cores ☺
- Static mapping

  → complex programming ☹
- Circuit-switched NoC → static reconfigurations

  → complex programming ☹
- Doesn’t scale easily
  - Can we create / debug / understand static mapping on 10K?

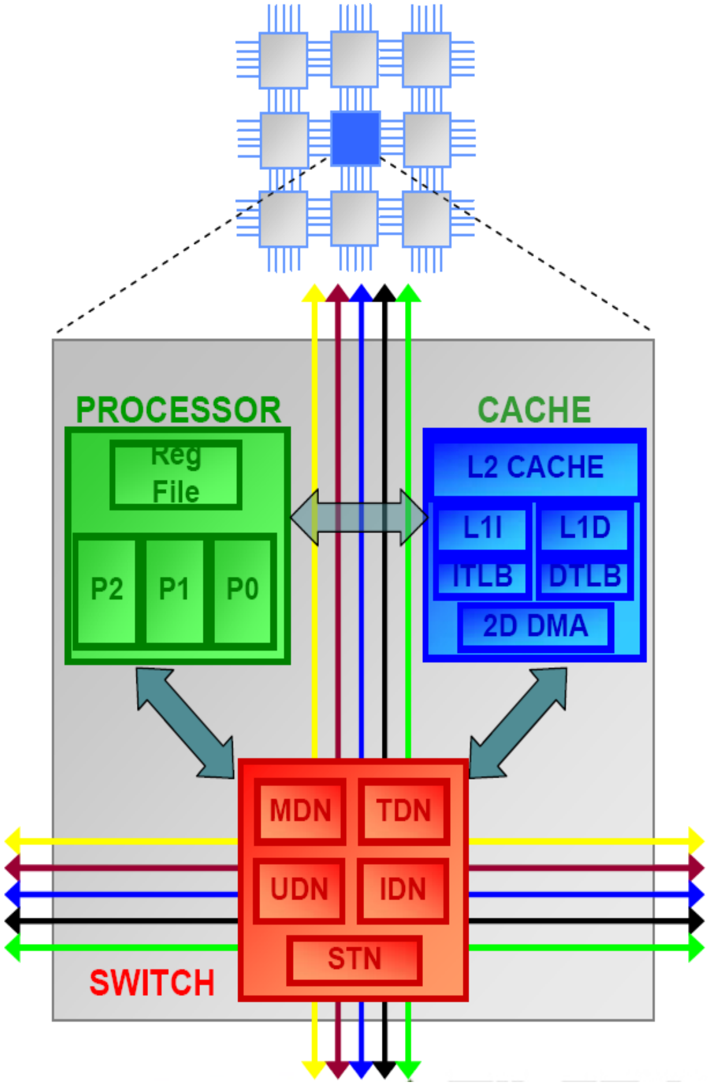
TILERA®

- USA company
- Based on RAW research @ MIT (A. Agarwal)

  - Heavy DARPA funding, university IP
- Classic homogeneous MIMD on mesh NoC
  - “Upgraded” Transputers with “powerful” uniprocessor features
    - Caches ☹
    - Complex communications ☹
- “tiles era”

TILERA Tiles

- Powerful processor
- High freq: ~1 GHz
  - High power (0.5W) ☹
- 5-mesh NoC
  - P-M / P-P / P-IO
- 2.5 levels cache ☹☹
  - L1+ L2
  - Can fetch from L2 of others
- Variable access time
  - 1 – 7 – 70 cycles

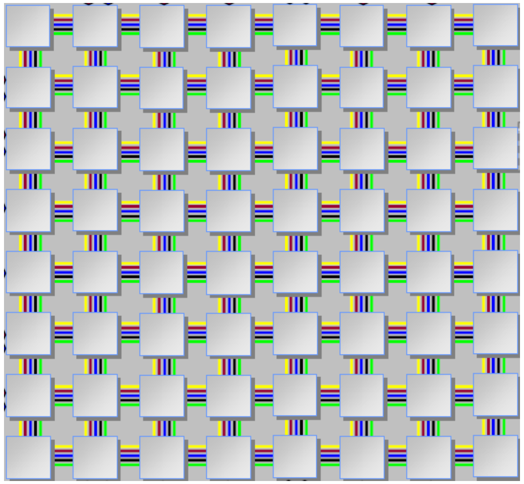


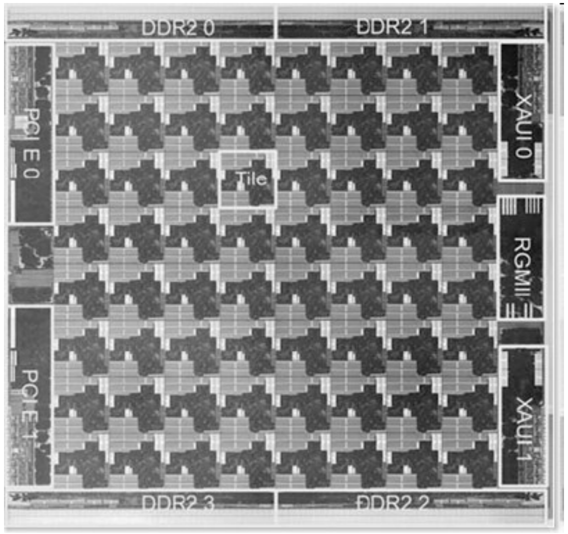
Caches Kill Performance

- Cache is great for a single processor
  - Exploits locality (in time and space)
- Locality only happens locally on many-cores
  - Other (shared) data are buried elsewhere
- Caches help speed up parallel (local) phases
  - Amdahl [1967]: the challenge is NOT the parallel phases
- Need to program many looong tasks on same data
  - Hard ! That’s the software gap in parallel programming

TILERA Array

- 36-64 processors
  - MIMD / SIMD ☹
- Total 5+ MB memory
  - In distributed caches
- High power
  - ~27W ☹☹

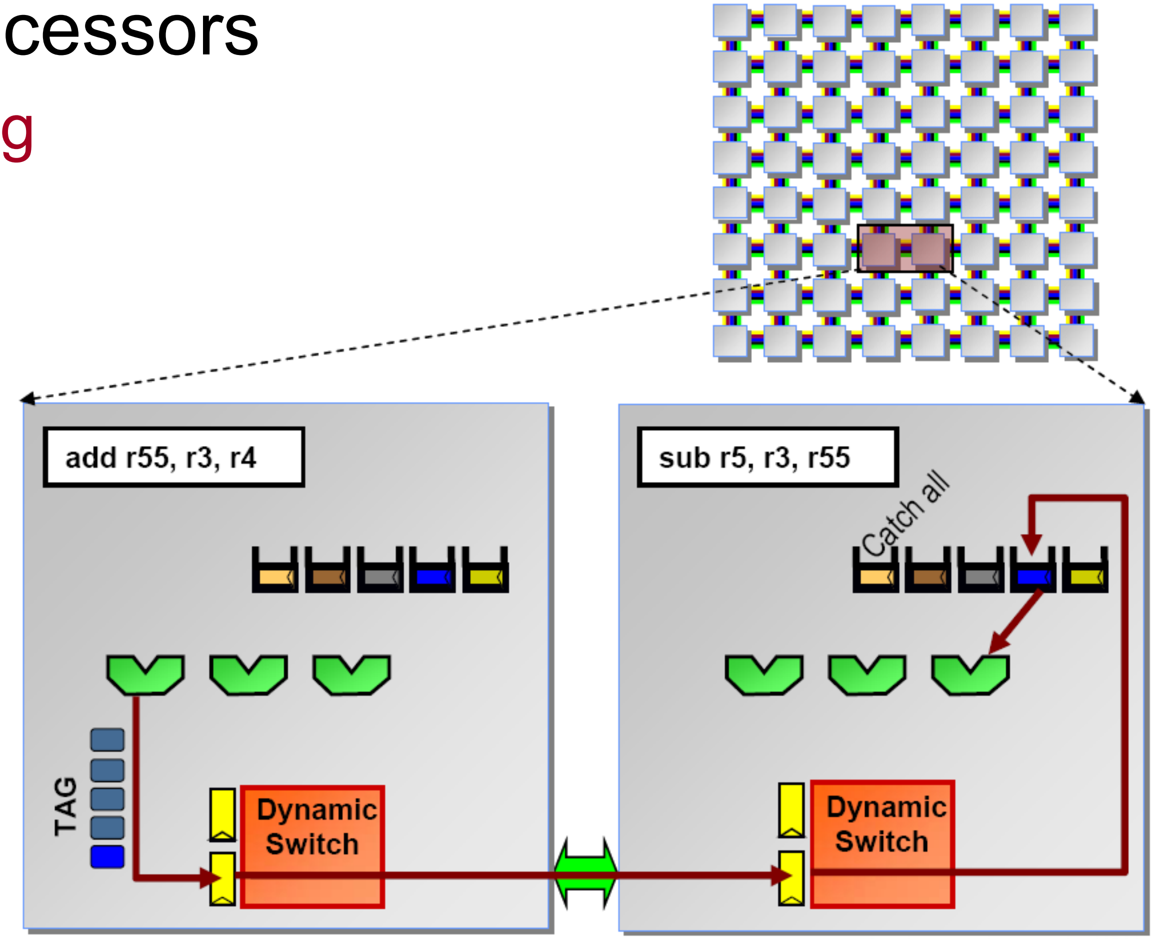




Die photo

TILERA allows statics

- Pre-programmed streams span multi-processors
  - Static mapping

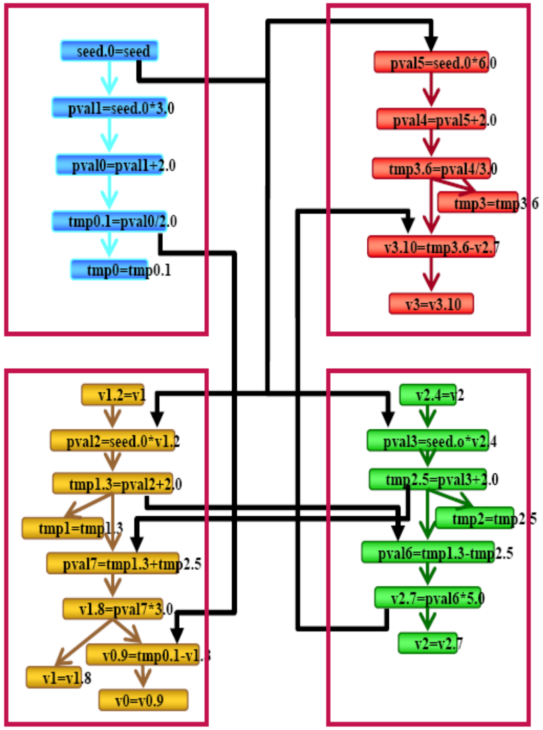


TILERA co-mapping: code, memory, routing ☹

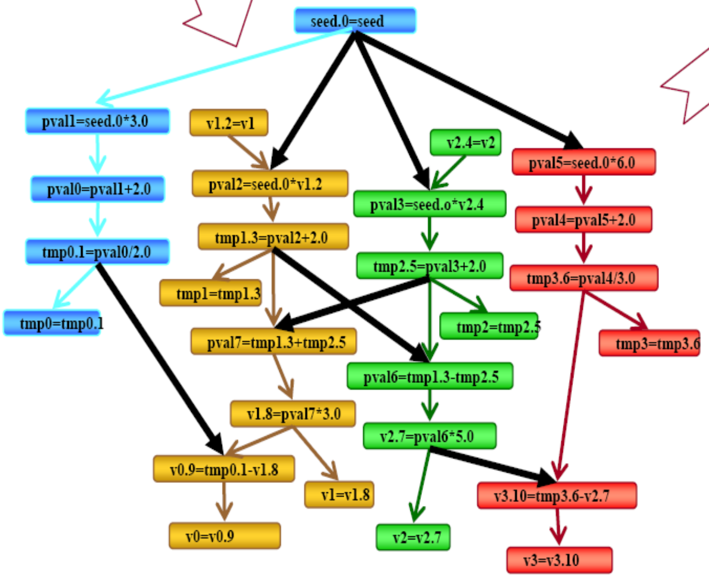
C

```
tmp0 = (seed*3+2)/2
tmp1 = seed*v1+2
tmp2 = seed*v2 + 2
tmp3 = (seed*6+2)/3
v2 = (tmp1 - tmp3)*5
v1 = (tmp1 + tmp2)*3
v0 = tmp0 - v1
v3 = tmp3 - v2
```

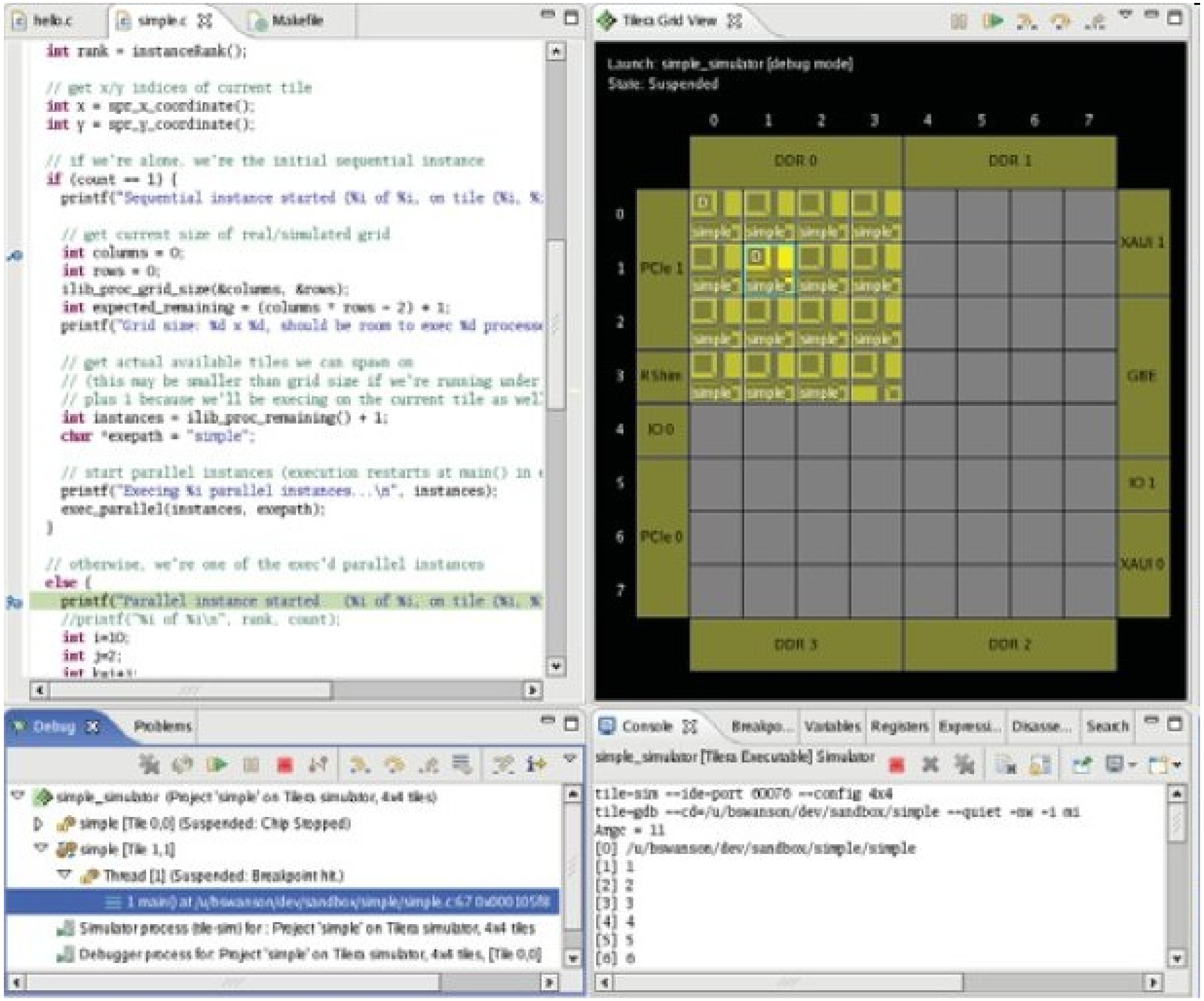
Place, Route, Schedule



Partitioning



TILERA® static mapping debugger ☹



```
int rank = instanceRank();

// get x/y indices of current tile
int x = spr_x_coordinate();
int y = spr_y_coordinate();

// if we're alone, we're the initial sequential instance
if (count == 1) {
  printf("Sequential instance started (%i of %i, on tile (%i, %

  // get current size of real/simulated grid
  int columns = 0;
  int rows = 0;
  ilib_proc_grid_size(&columns, &rows);
  int expected_remaining = (columns * rows - 2) + 1;
  printf("Grid size: %d x %d, should be room to exec %d processe

  // get actual available tiles we can spawn on
  // (this may be smaller than grid size if we're running under
  // plus 1 because we'll be execing on the current tile as wel
  int instances = ilib_proc_remaining() + 1;
  char *exepath = "simple";

  // start parallel instances (execution restarts at main() in
  printf("Execing %i parallel instances...\n", instances);
  exec_parallel(instances, exepath);
}

// otherwise, we're one of the exec'd parallel instances
else {
  printf("Parallel instance started   (%i of %i, on tile (%i, %
  //printf("%i of %i\n", rank, count);
  int i=10;
  int j=2;
  int k=i+j;
```

```
simple_simulator (Project 'simple' on Tilera simulator, 4x4 tiles)
  simple [Tile 0,0] (Suspended: Chip Stopped)
  simple [Tile 1,1]
    Thread [1] (Suspended: Breakpoint hit.)
      1 main() at /u/bswanson/dev/sandbox/simple/simple.c:67 0x00010578
  Simulator process (tile-sim) for : Project 'simple' on Tilera simulator, 4x4 tiles
  Debugger process for: Project 'simple' on Tilera simulator, 4x4 tiles, [Tile 0,0]
```

simple_simulator [Tilera Executable] Simulator

```
tile-sim --ide-port 60076 --config 4x4
tile-gdb --cd=/u/bswanson/dev/sandbox/simple --quiet -nw -i mi
Argc = 11
[0] /u/bswanson/dev/sandbox/simple/simple
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
```

TILERA® analysis

- Achieves good performance
- Bad on power
- Hard to scale
- Hard to program