



The Plural Architecture

Shared Memory Many-core with Hardware Scheduling

Ran Ginosar
Technion, Israel

March 2015



Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Plural programming examples
- ManyFlow for the Plural architecture
- Scaling the Plural architecture
- Mathematical model of the Plural architecture



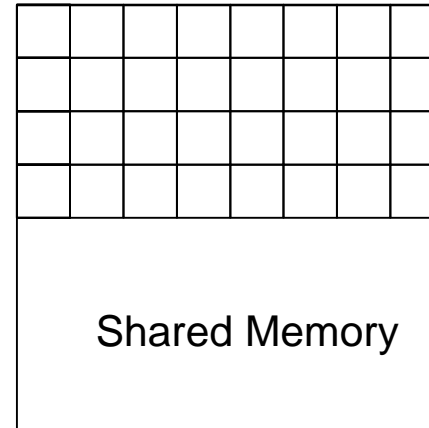
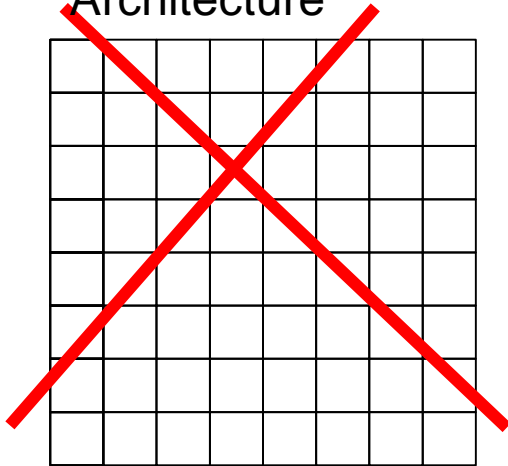
many-cores

- Many-core is:
 - a single chip
 - with many (how many?) cores and on-chip memory
 - running one (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

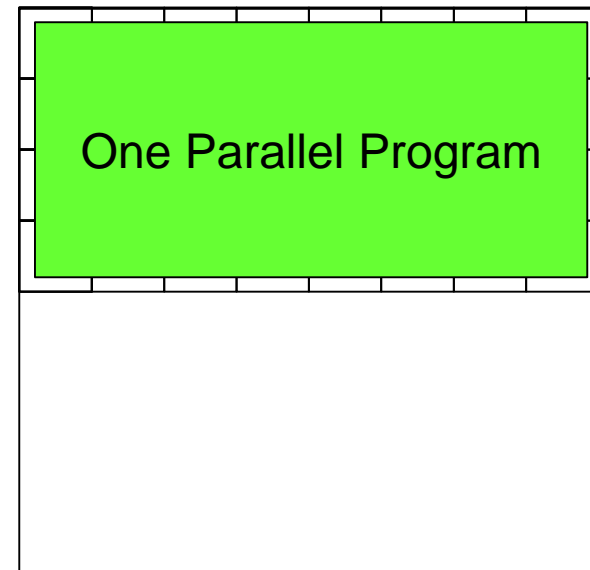
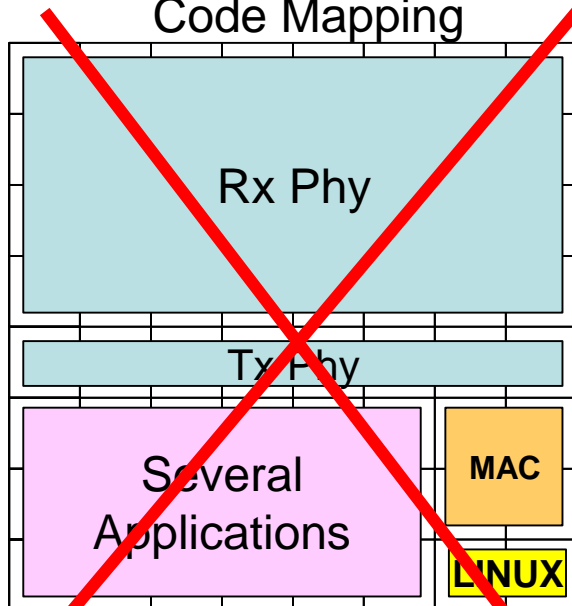


Plural shared memory architecture

Architecture

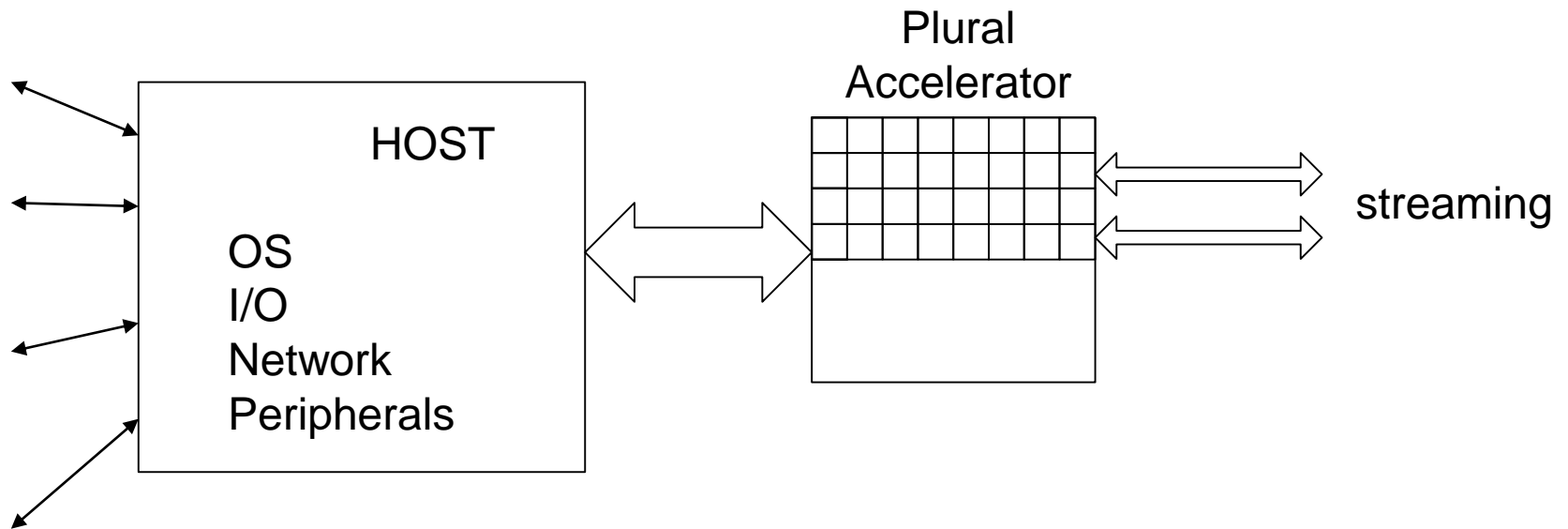


Code Mapping



Context

- Plural: homogeneous acceleration for heterogeneous systems

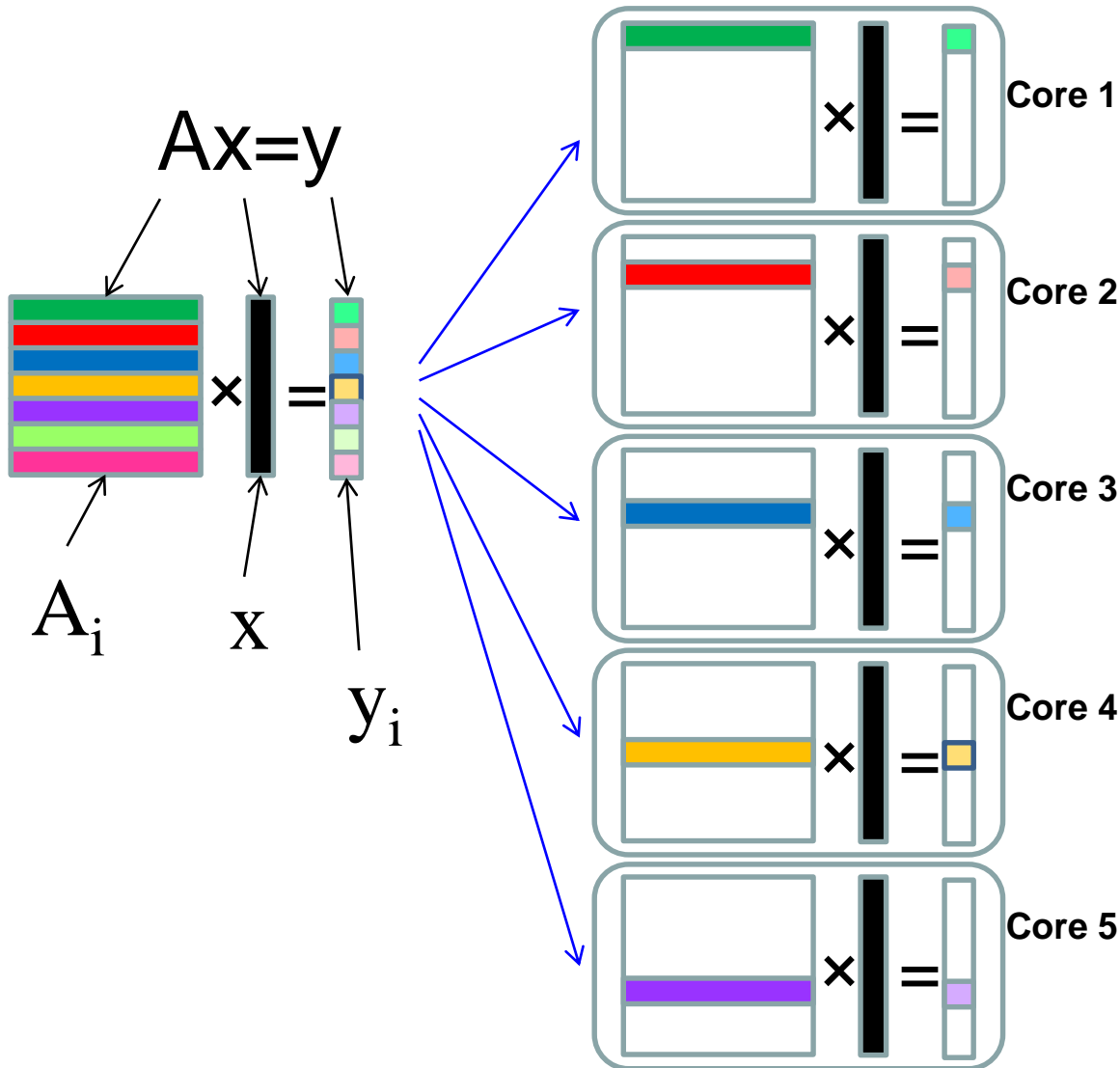


One (parallel) program ?

- Best formal approach to parallel programming is the PRAM model
- Manages
 - all cores as a single shared resource
 - all memory as a single shared resource
- and more...



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

$B(i) := A(i)$

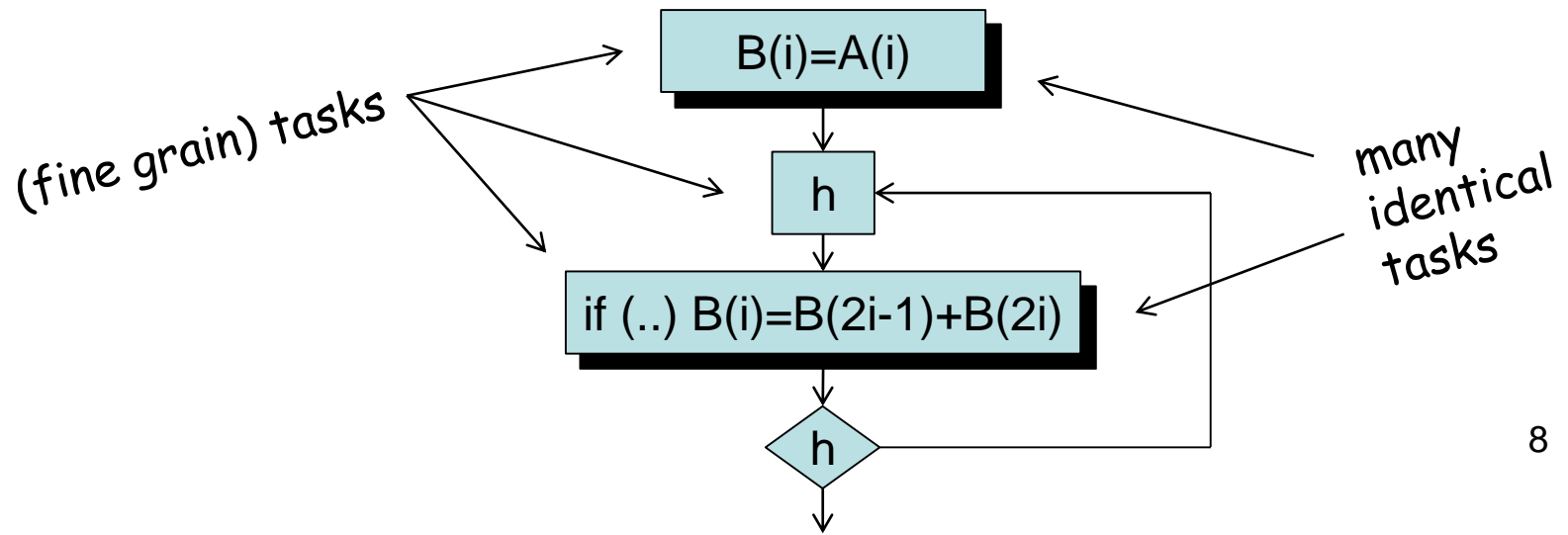
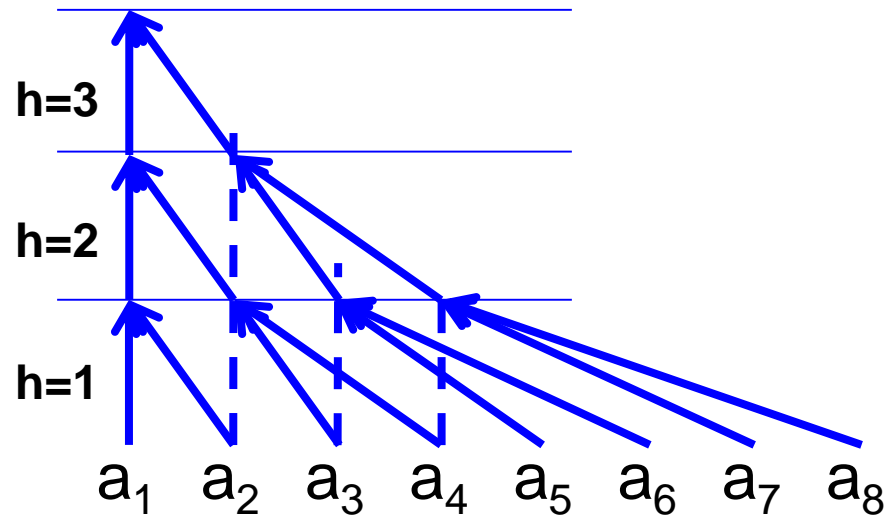
For $h=1:\log(n)$

if $i \leq n/2^h$ then

$B(i) = B(2i-1) + B(2i)$

End

// $B(1)$ holds the sum



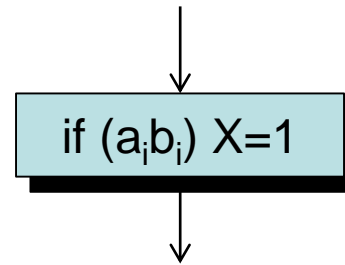
PRAM SoP: Concurrent Write

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

Begin

if ($a_i b_i$) $X = 1$

End



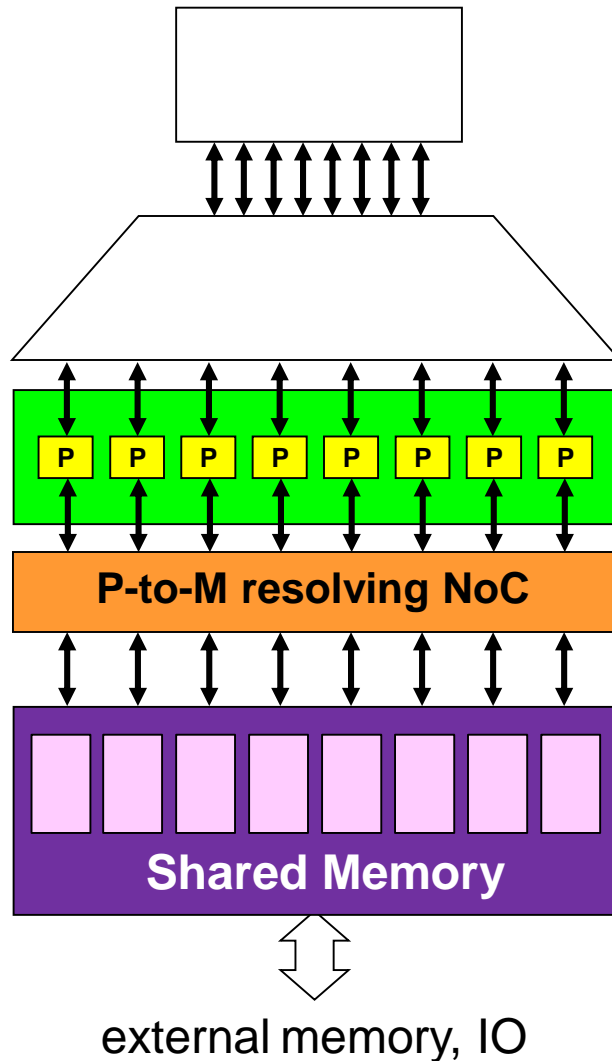
All cores which write into X , write the same value

Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Plural programming examples
- ManyFlow for the Plural architecture
- Scaling the Plural architecture
- Mathematical model of the Plural architecture

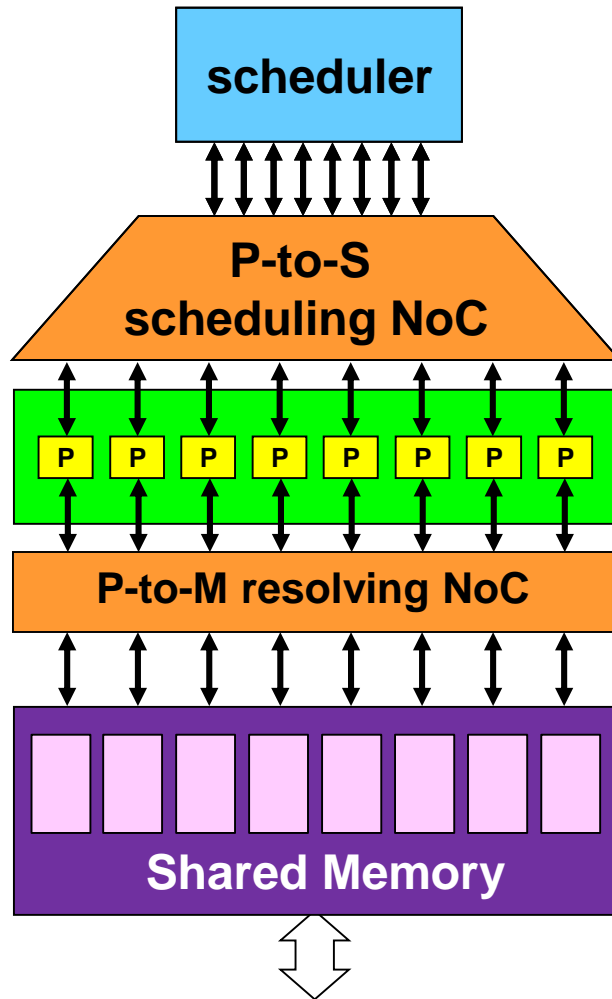


The Plural Architecture: Part I



- Many small processor cores
- Small private memories (stack, L1)
- Fast NOC to memory
(Multistage Interconnection Network)
- NOC resolves conflicts
- SHARED memory, many banks
~Equi-distant from cores (2-3 cycles)
- “Anti-local” address interleaving
- Negligible conflicts

The Plural Architecture: Part II



external memory, IO

Hardware scheduler / dispatcher / synchronizer

Low (zero) latency parallel scheduling enables fine granularity

Many small processor cores
Small private memories (stack, L1)

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

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

“Anti-local” address interleaving
Negligible conflicts

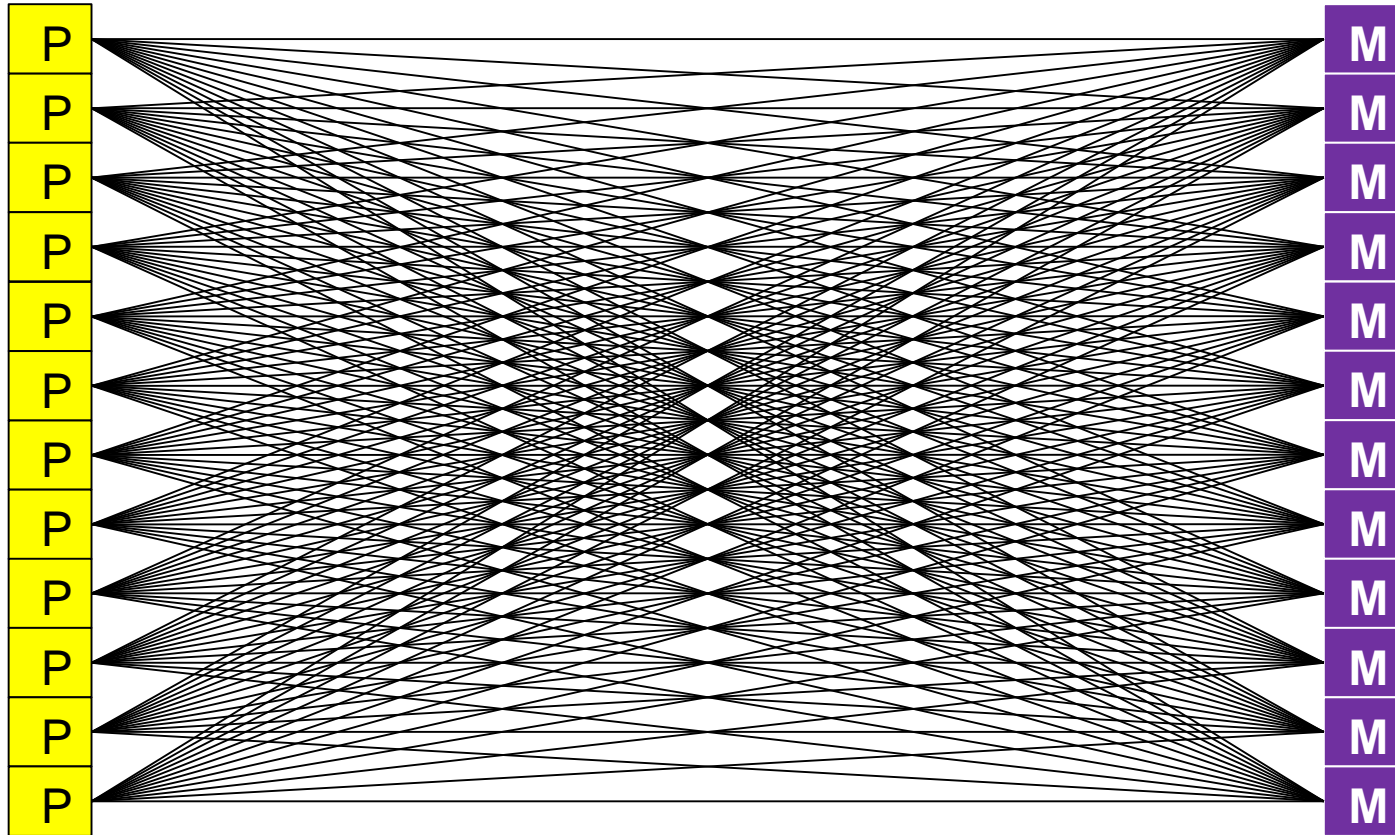


Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Plural programming examples
- ManyFlow for the Plural architecture
- Scaling the Plural architecture
- Mathematical model of the Plural architecture

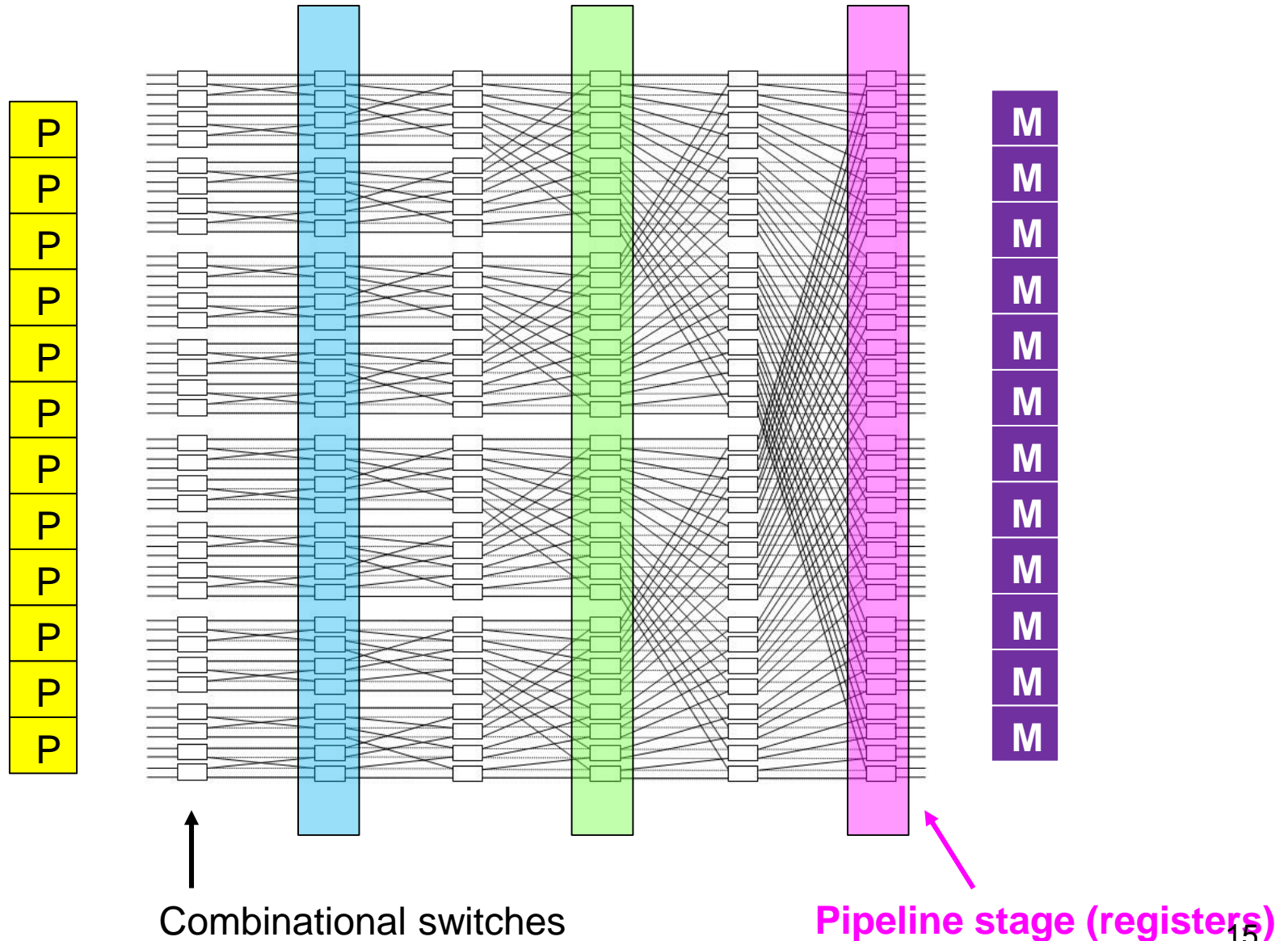


How does the P-to-M NOC look like?

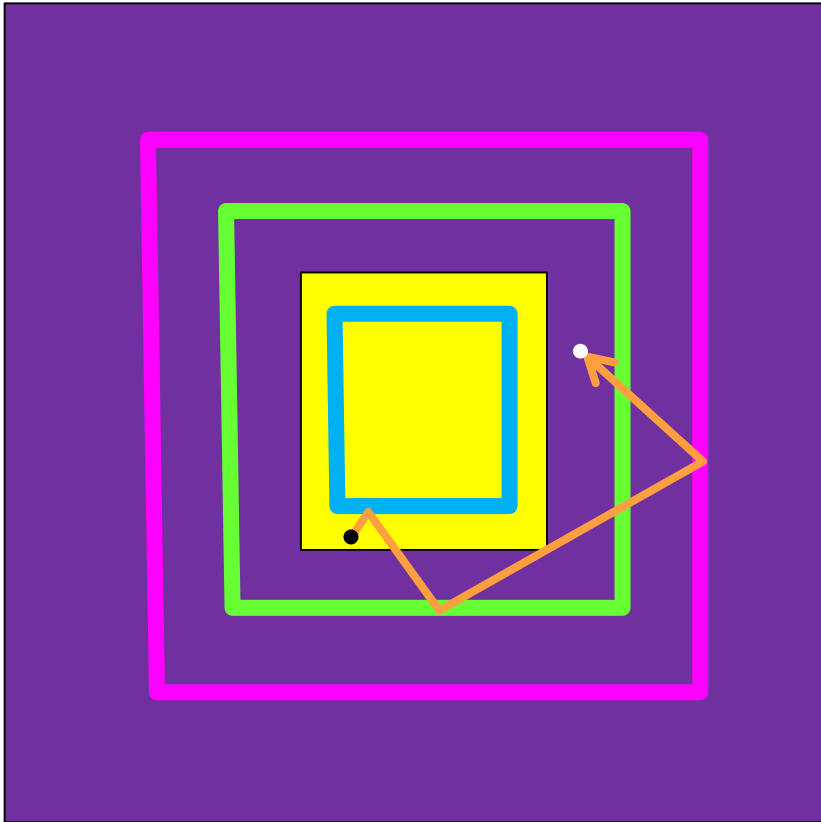


- Full bi-partite connectivity required
- But full cross-bar not required: minimize conflicts and allow stalls/re-starts

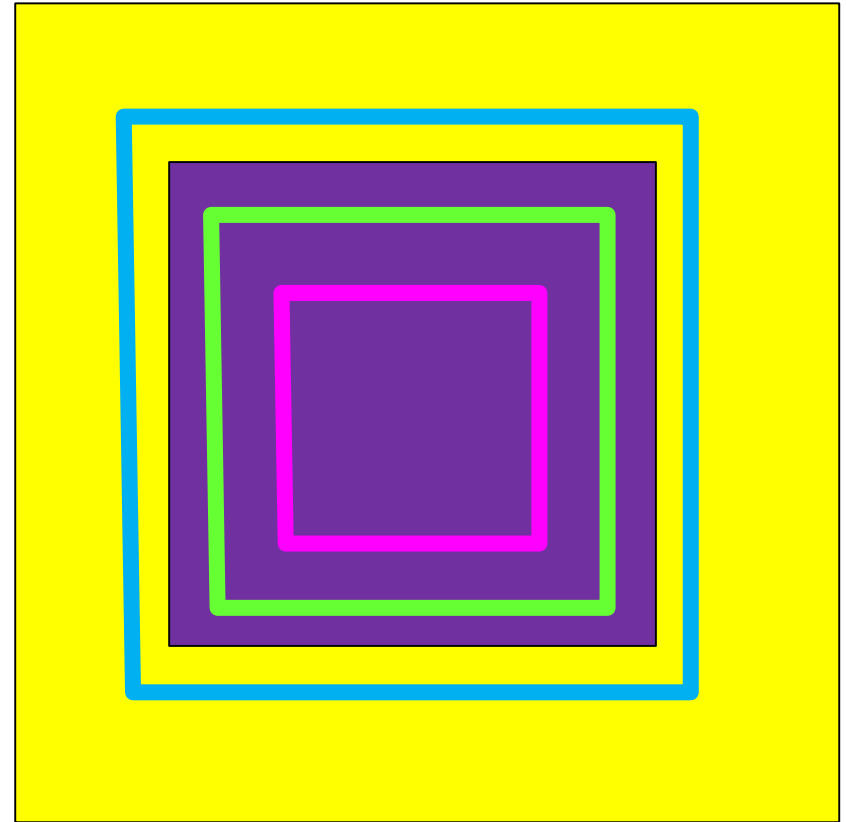
Logarithmic multistage interconnection network



Floorplans



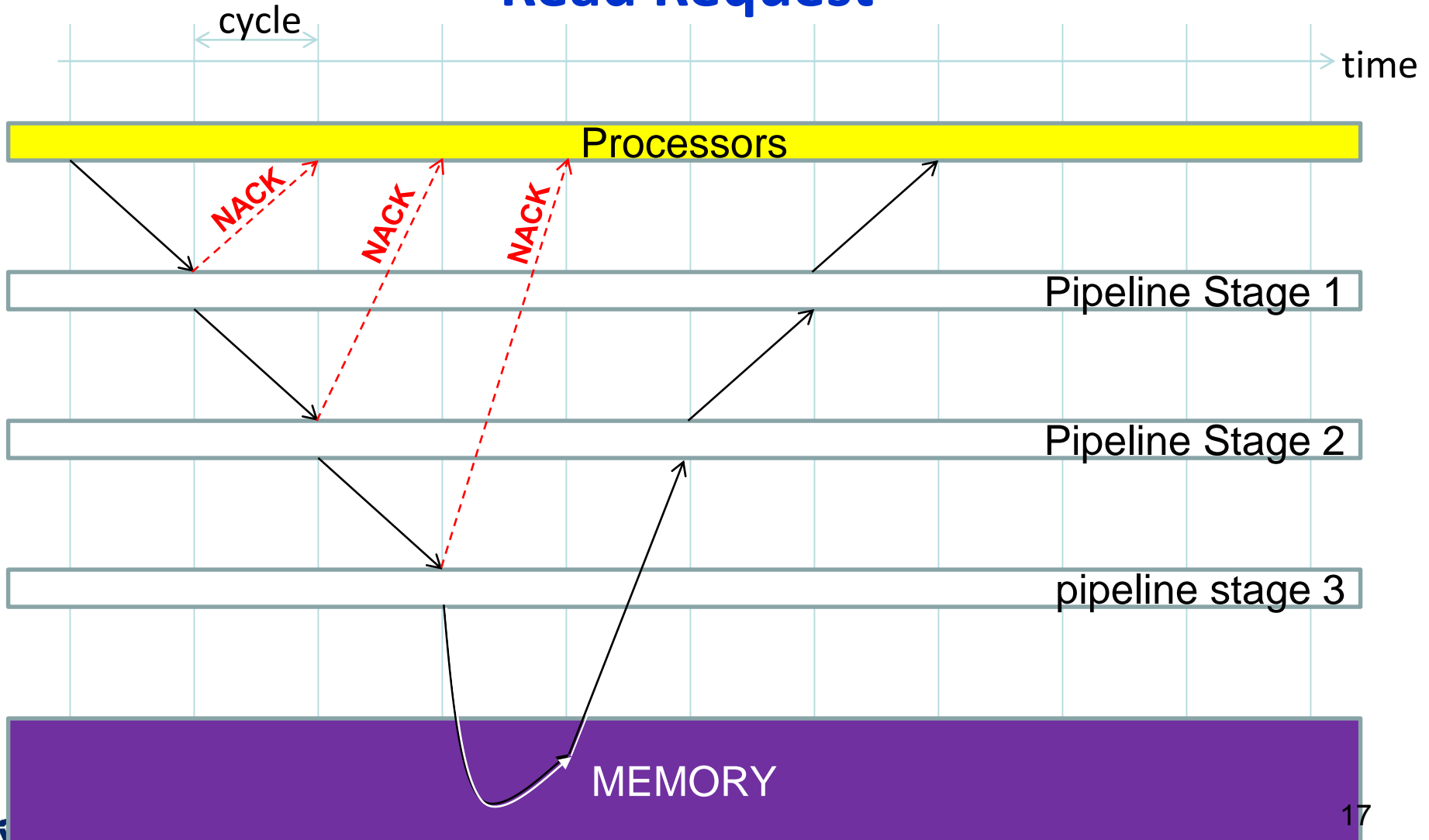
and an example of one route



The dual floor plan. Why?

access sequence: **fixed latency** (when successful)

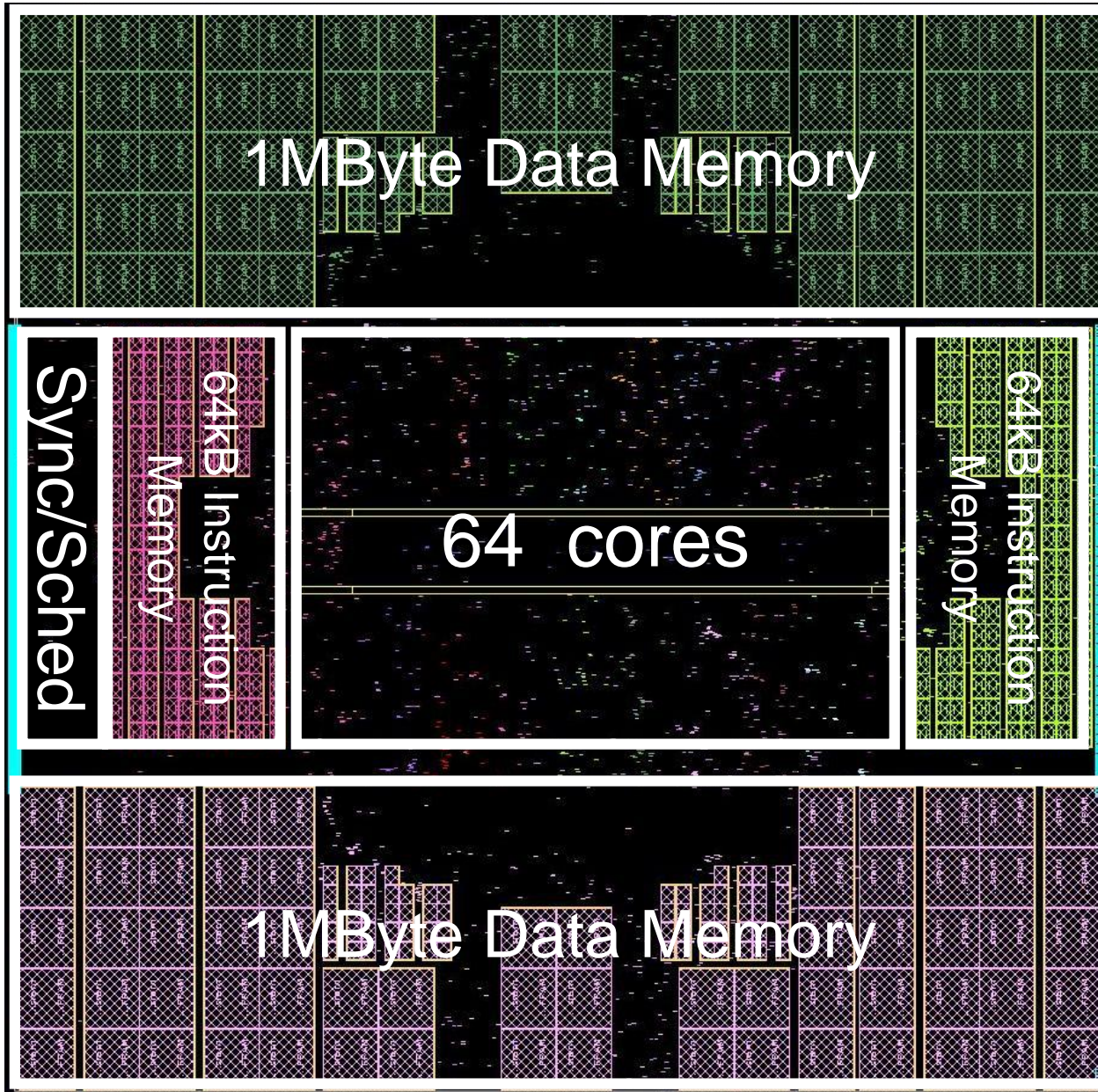
Read Request





PLURALITY

Example floorplan + layout



40nm GP

4×4mm

64 cores

16 FPU

2MB D\$
in 128 banks

128kB I\$

400 MHz

1 Watt



Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Plural programming examples
- ManyFlow for the Plural architecture
- Scaling the Plural architecture
- Mathematical model of the Plural architecture

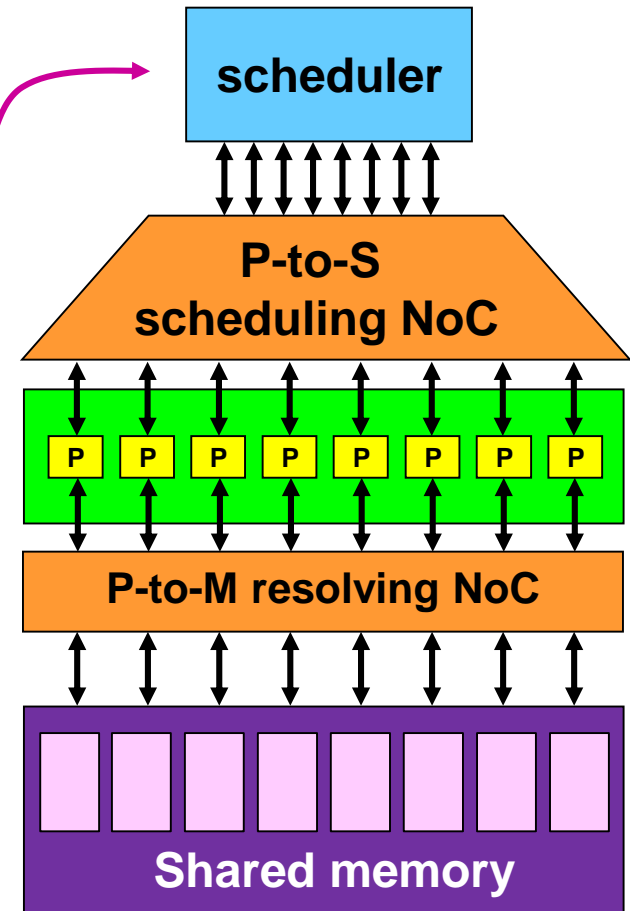


The Plural task-oriented programming model

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

Task template:

```
{ regular  
  duplicable } taskName ( instance_id )  
  
{  
    ... instance_id ....  
        // instance_id is instance number  
    .....  
}
```



Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Plural programming examples
- ManyFlow for the Plural architecture
- Scaling the Plural architecture
- Mathematical model of the Plural architecture



Fine Grain Parallelization

Convert (independent) loop iterations

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

duplicable doLargeLoop

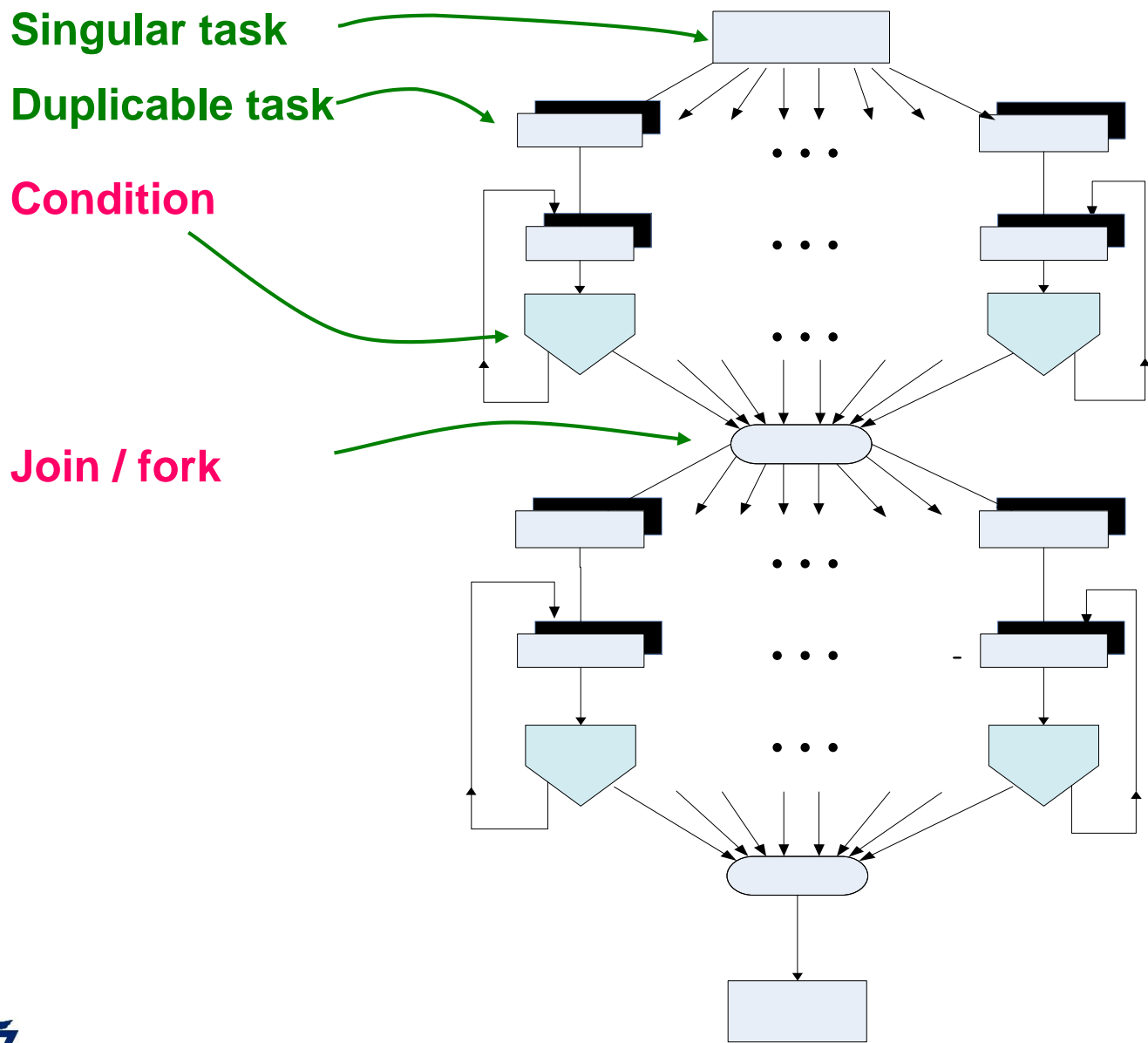
into parallel tasks

```
set_task_quota(doLargeLoop, 10000)
```

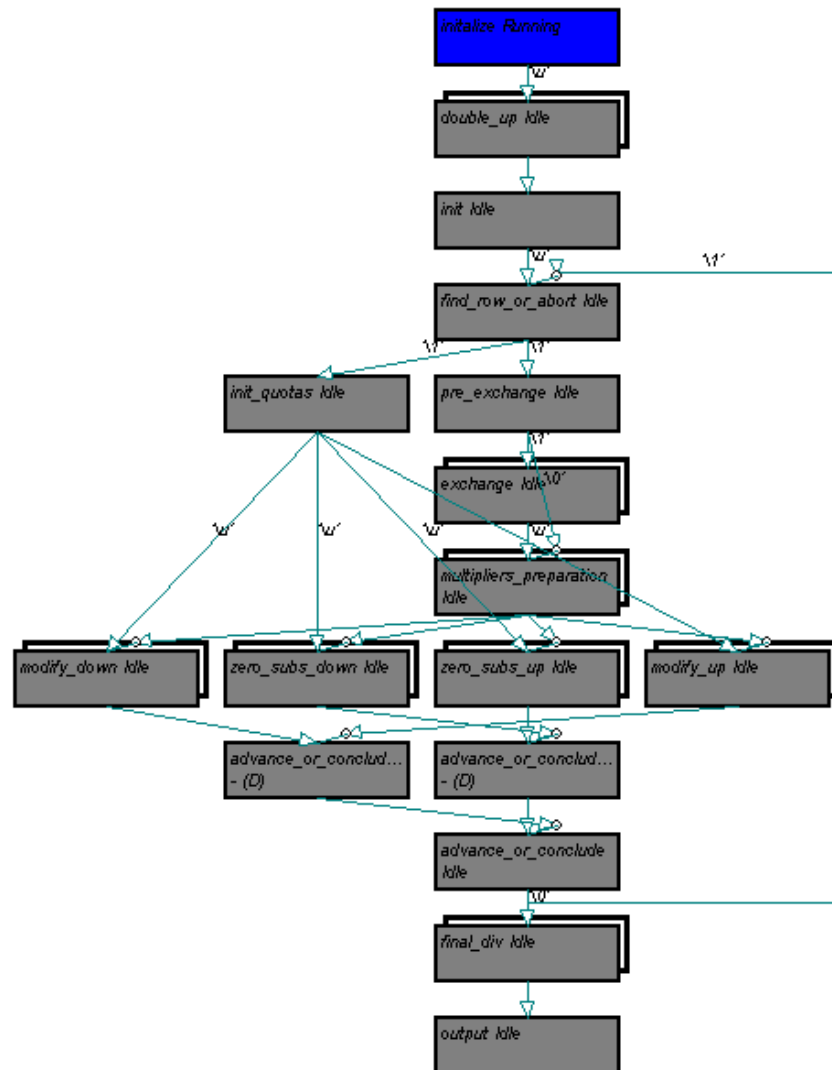
```
void doLargeLoop(id)  
{ a[id] = b[id]*c[id]; } //id is instance number
```



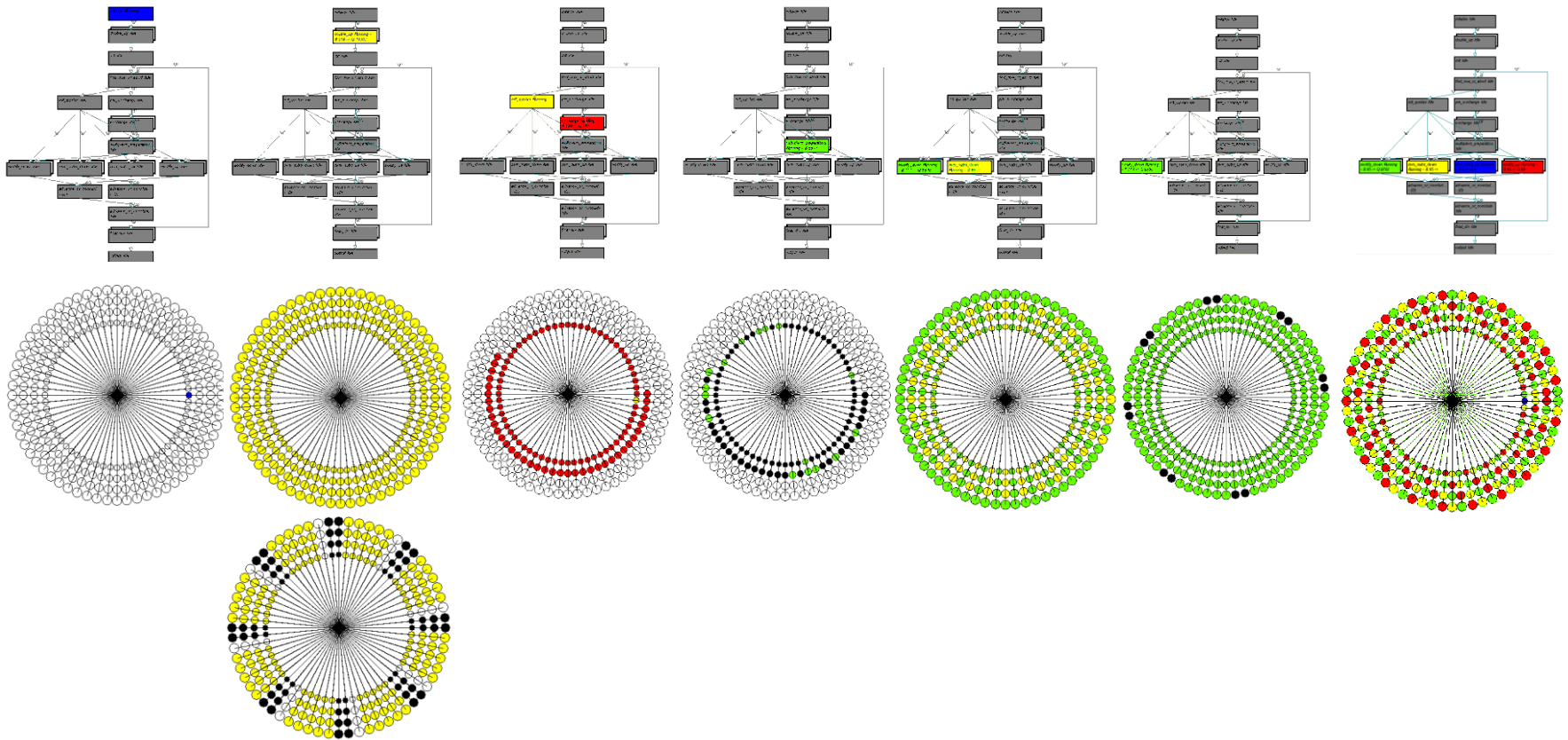
Task map example (2D FFT)



Another task map (linear solver)



Linear Solver: Simulation snap-shots



Plural Task Oriented Programming Model: 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



Plural Task Oriented Programming Model:

Task Rules 2

- 2 types of tasks:
 - Regular 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
- Conditions on tasks 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

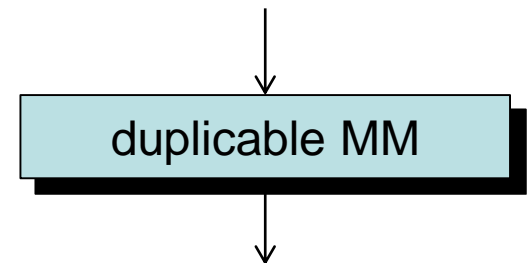


Example: Matrix Multiplication

```
set_task_quota(mm, N*N); // create N×N tasks

extern float A[],B[],C[] // A,B,C in shared mem

void mm(id) // id = instance number
{
    i = id mod N; // row number
    k = id / N; // column number
    sum = 0;
    for(m=0; m<N; m++){
        sum += A[i][m] * B[m][k]; // read row & column from
        // shared mem
    }
    C[i][k] = sum; // store result in shared mem
}
```



What if parallelism is limited ?

- So far, examples were highly parallel
- What if algorithm CANNOT be parallelized?
 - Execute many (serial) instances in parallel
 - Each instance on different data
- What if algorithm is mixture of serial / parallel segments?
 - Use ManyFlow



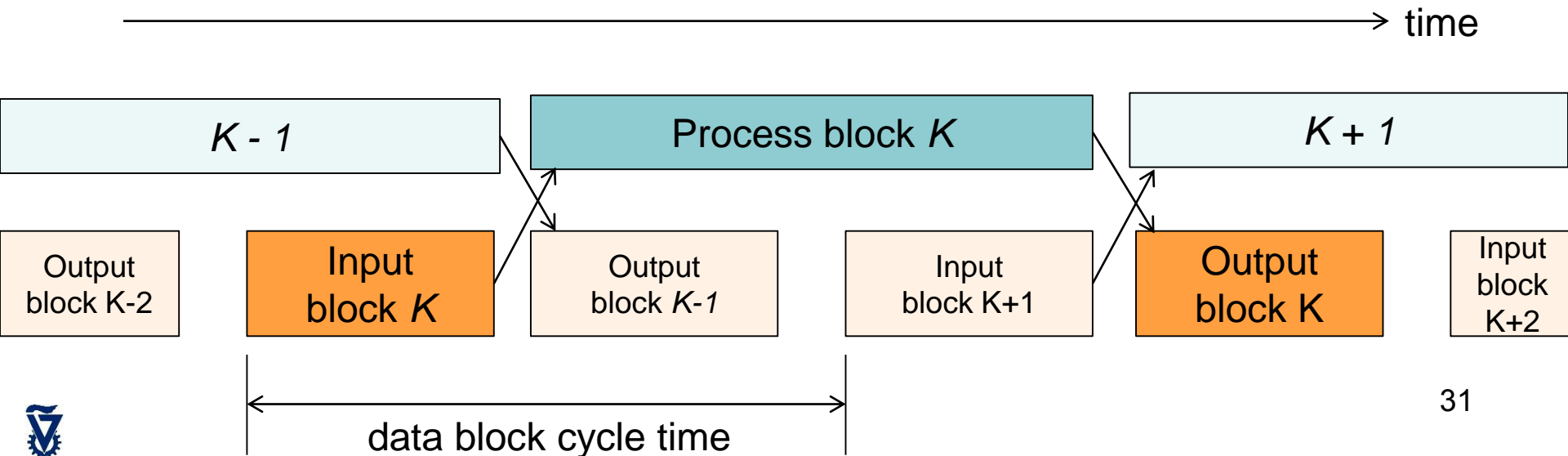
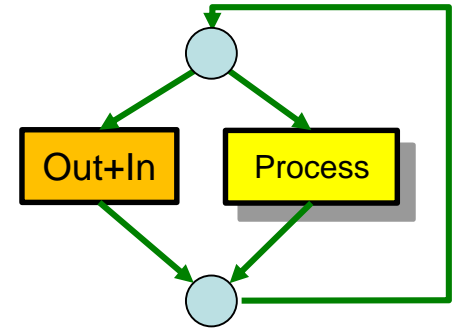
Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Plural programming examples
- ManyFlow for the Plural architecture
- Scaling the Plural architecture
- Mathematical model of the Plural architecture



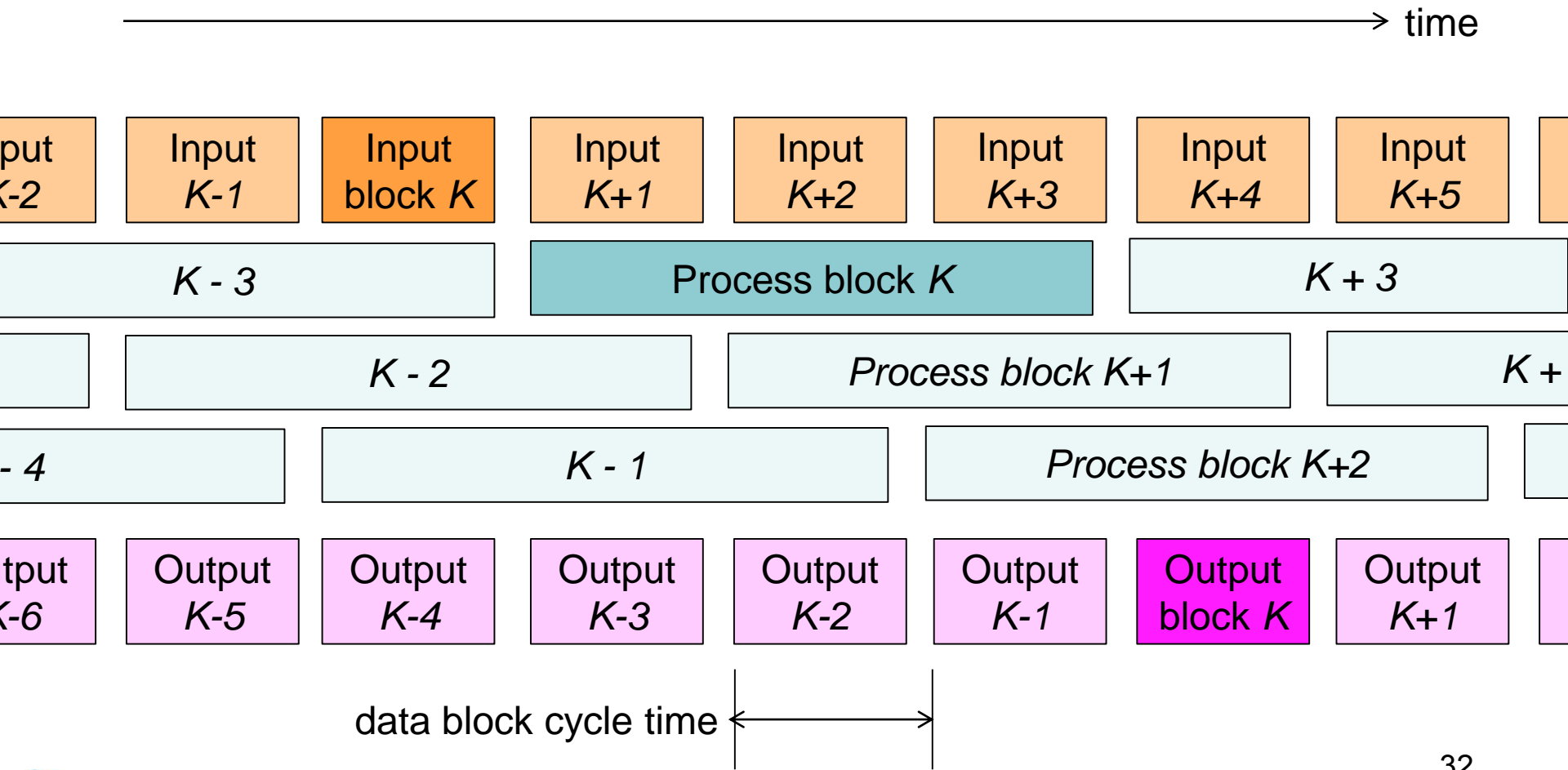
Stream Processing

- Data arrives in a sequence of blocks
- In parallel:
 - Process current block (K)
 - Output results of previous block ($K-1$)
 - Input next block ($K+1$)



PIPELINED stream processing

- For faster data & slower processing

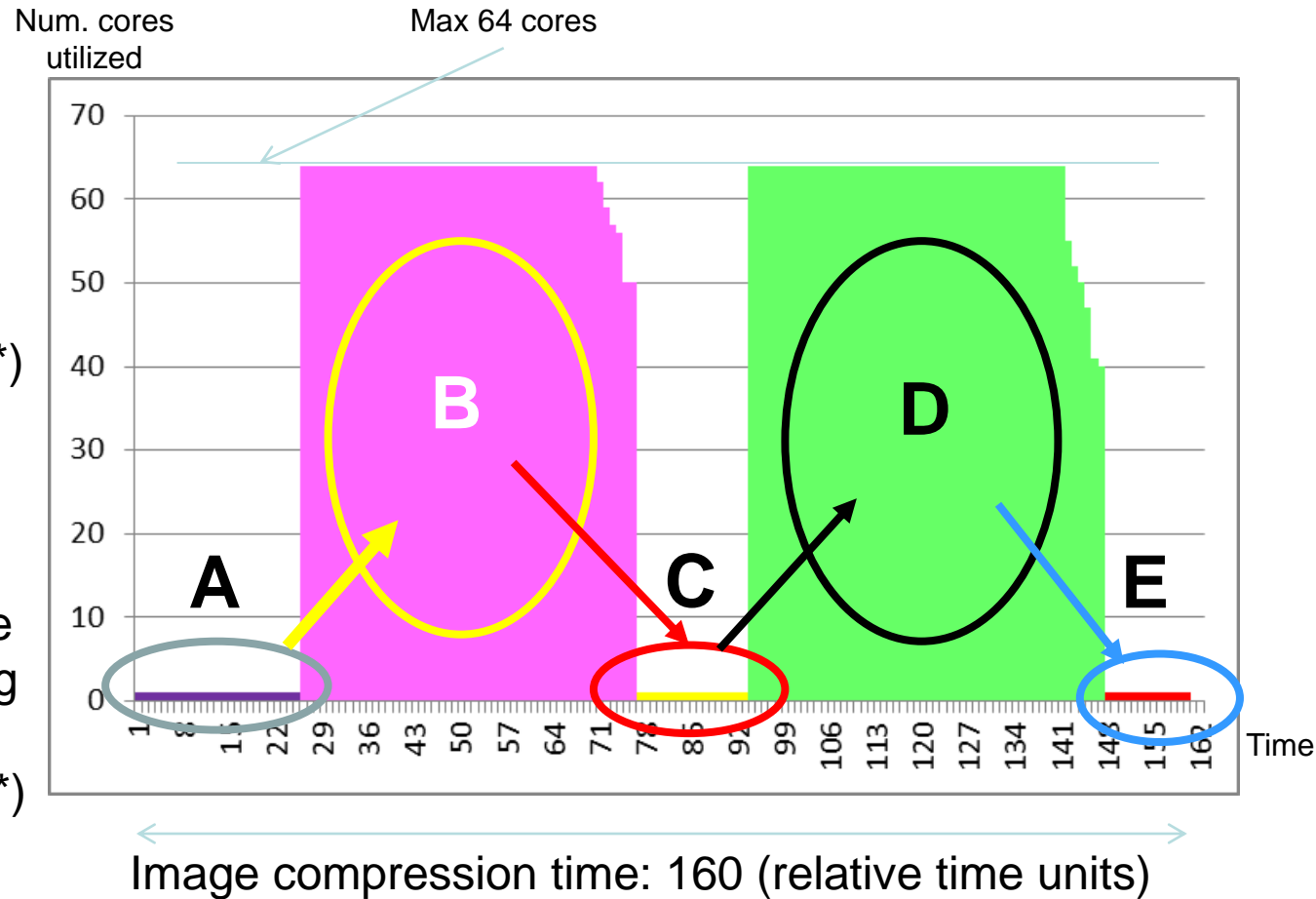
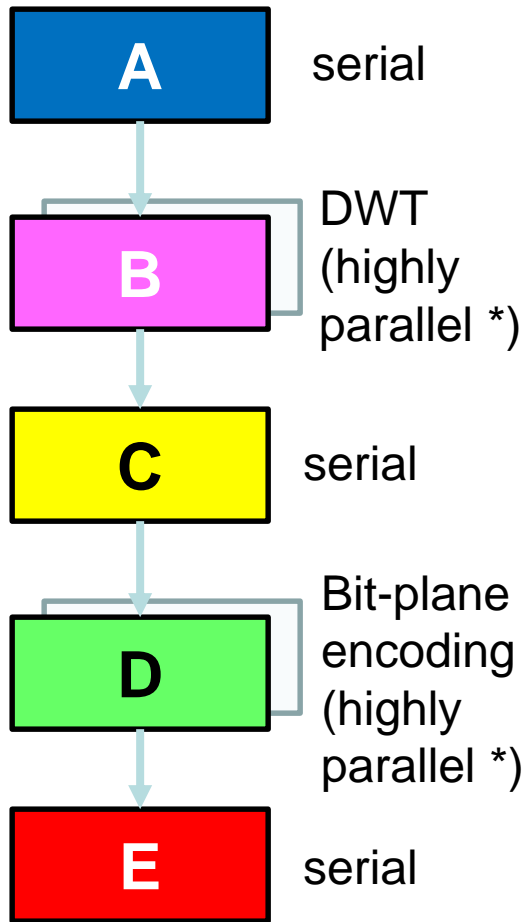


PIPELINED stream processing: **ManyFlow**

- Parallel execution of pipelined stream processing on the shared-memory manycore Plural architectures
- Flexible, dynamic, out-of-order, task-oriented execution

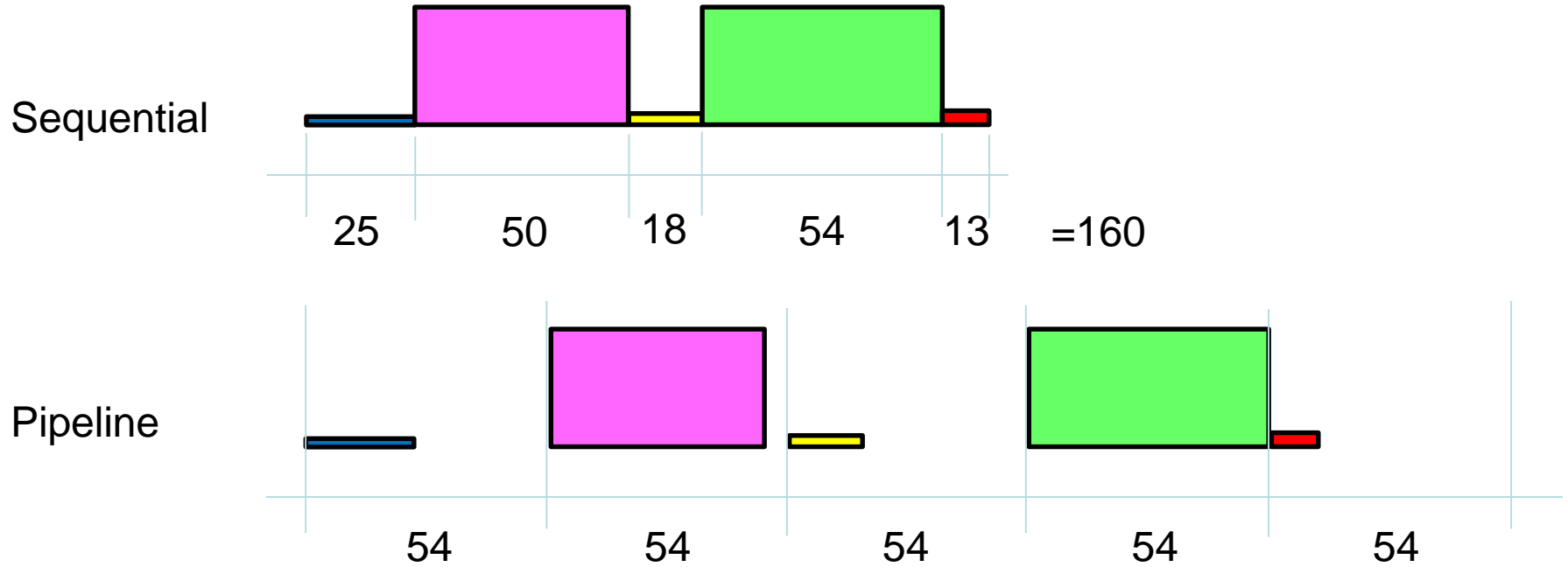


Example: A DWT image compression algorithm

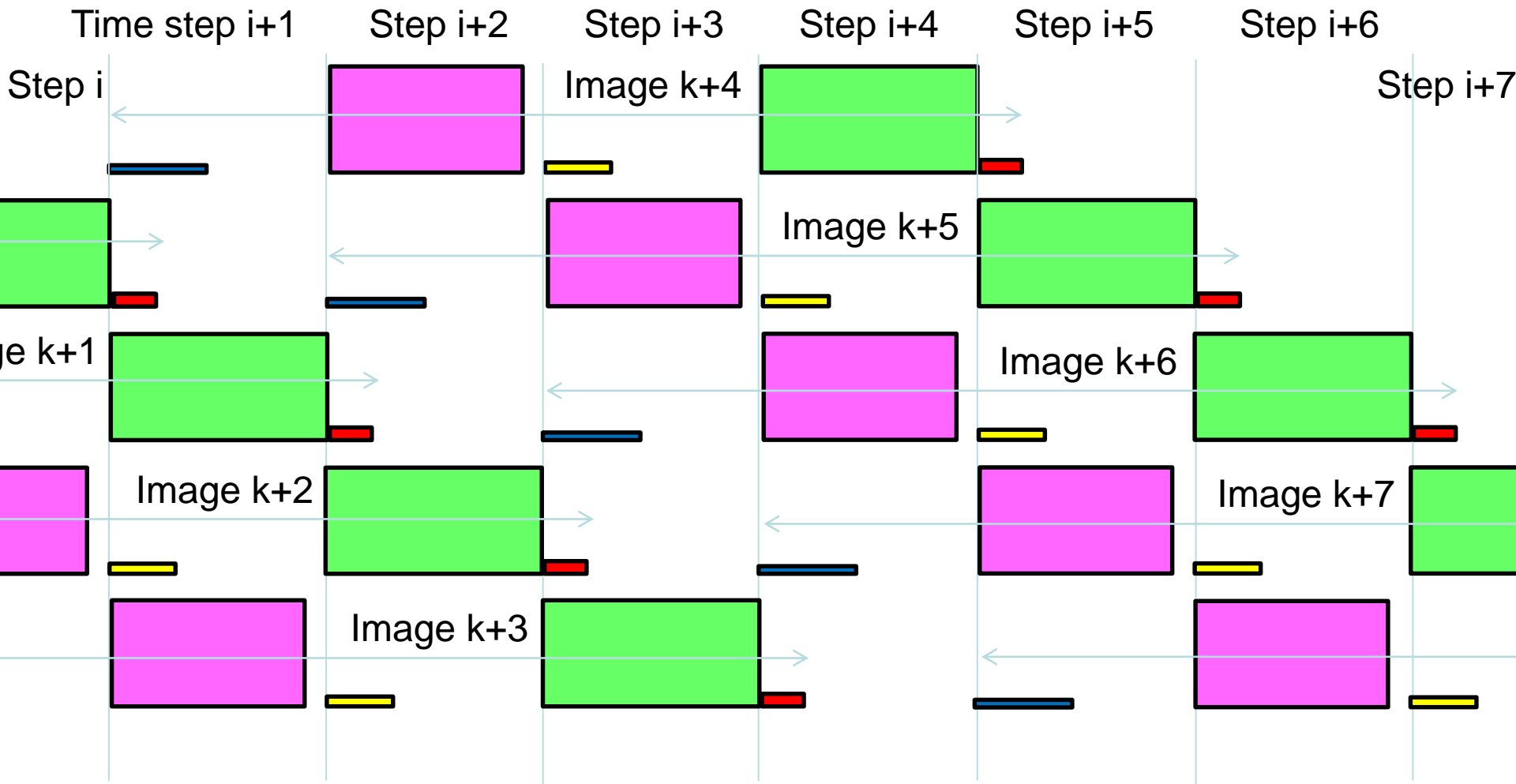


Low utilization: only 65%

Speed it up with a pipeline?



Hardware-like Pipeline



Needs 5 stages: two with 64 cores each, three with one core each (total 131 cores)

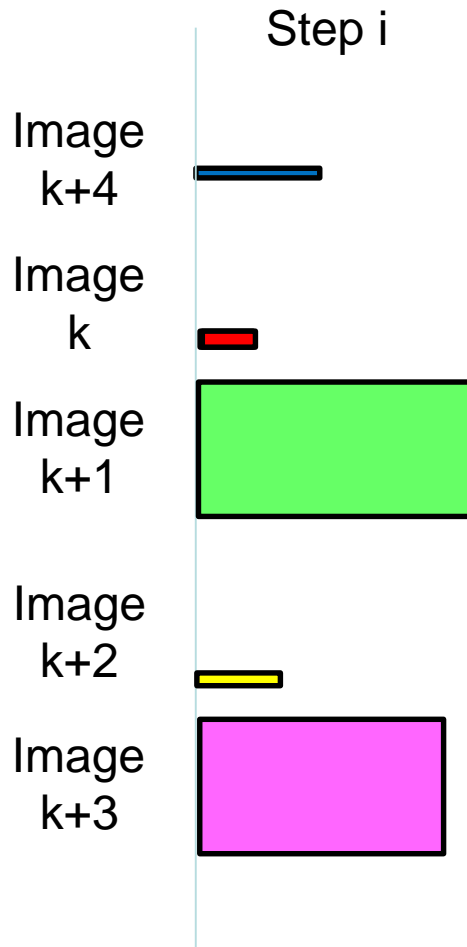
If only 64 cores, time / step = $64 \times 2 + 25 = 153$ (how? What is the utilization?)

36

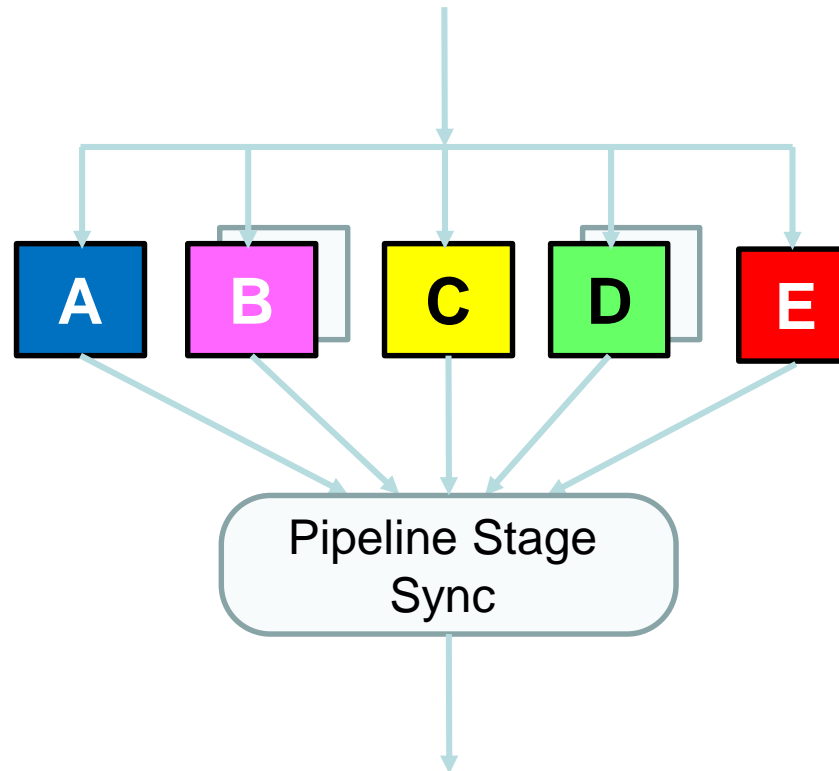


Hard to program, inefficient, inflexible, fixed task per core. Need to store 5 images

Parallel / pipelined “ManyFlow”



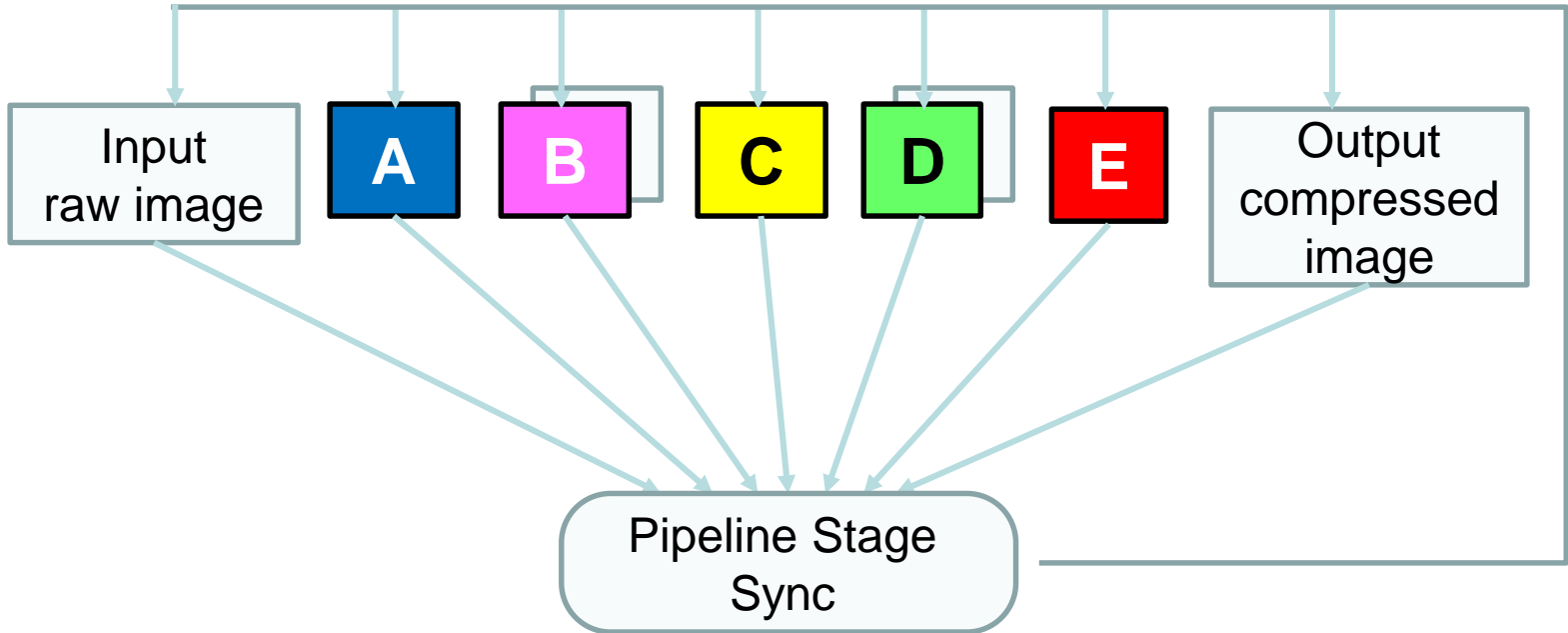
All 5 stages are independent (order does not matter)
→ Can run concurrently
→ Scheduler will dispatch most efficiently



Still need to store 5 images (and their temporary storage)



Parallel / pipelined “ManyFlow”



Task map for continuous execution
Includes two more pipe stages, for I/O of images

Now need to store 7 images (and their temporary storage)

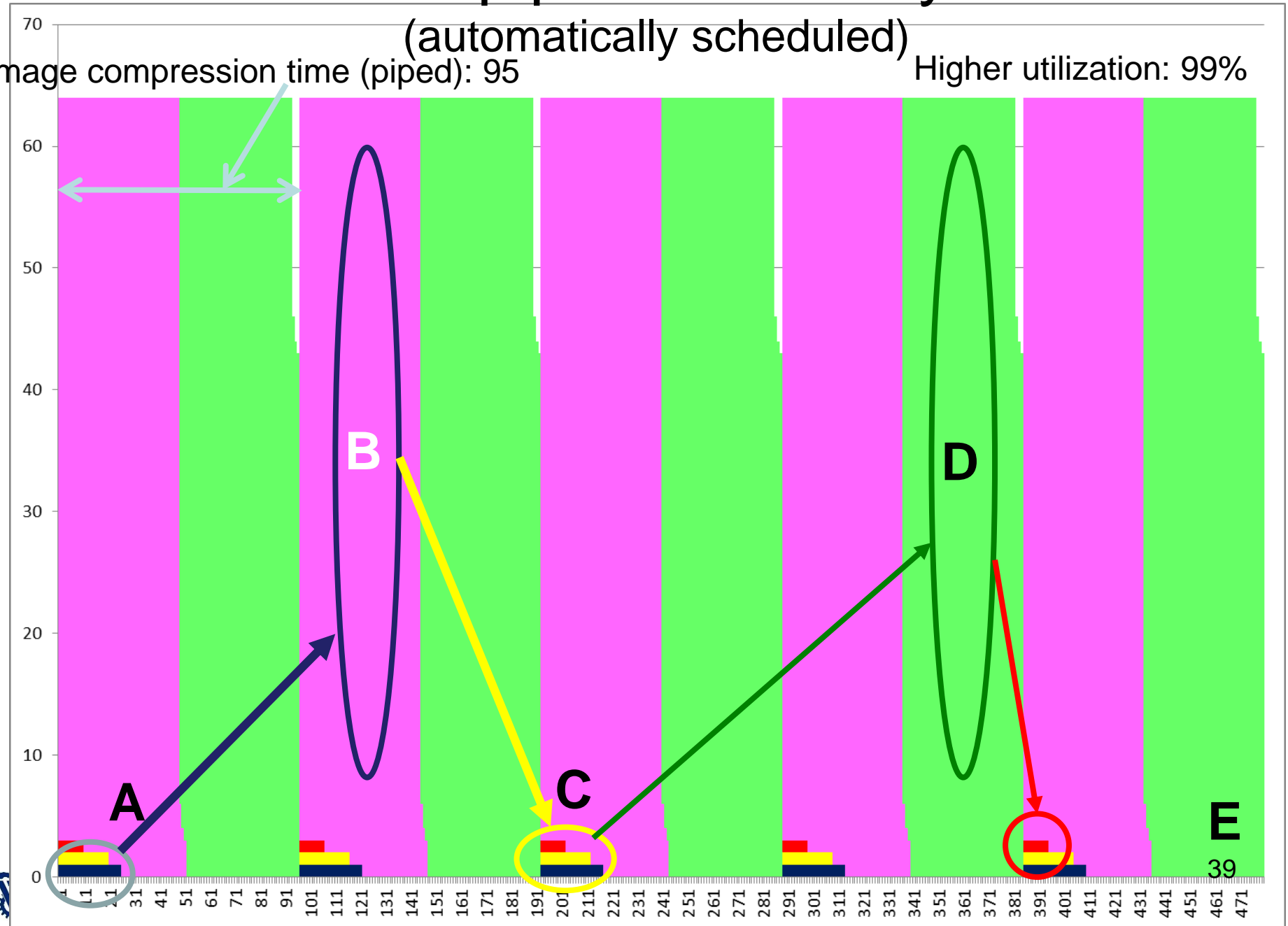


Parallel / pipelined "ManyFlow"

(automatically scheduled)

Image compression time (piped): 95

Higher utilization: 99%



The code

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 1000
int round_counter = 0;

void program_start() {
    set_task_quota(BB,N);
    set_task_quota (DD,N);
}

void AA (void) { set_task_runtime(25); }
void BB (void) { set_task_runtime(3); }
void CC (void) { set_task_runtime(20); }
void DD (void) { set_task_runtime(3); }
void EE (void) { set_task_runtime(10); }

int task_manager(void) {
    round_counter++;
    if (round_counter < 5)
        return(0);
    else
        return(1);
}

void program_end(void) { }
```

TASK MAP

```
regular task    program_start()
regular task    AA      (program_start || task_manager==0)
regular task    CC      (program_start || task_manager==0)
regular task    EE      (program_start || task_manager==0)
duplicable task BB      (program_start || task_manager==0)
duplicable task DD      (program_start || task_manager==0)
regular task    task_manager (AA && BB && CC && DD && EE)
regular task    program_end (task_manager==1)
```

(for simplicity, real task code replaced by indication of duration)

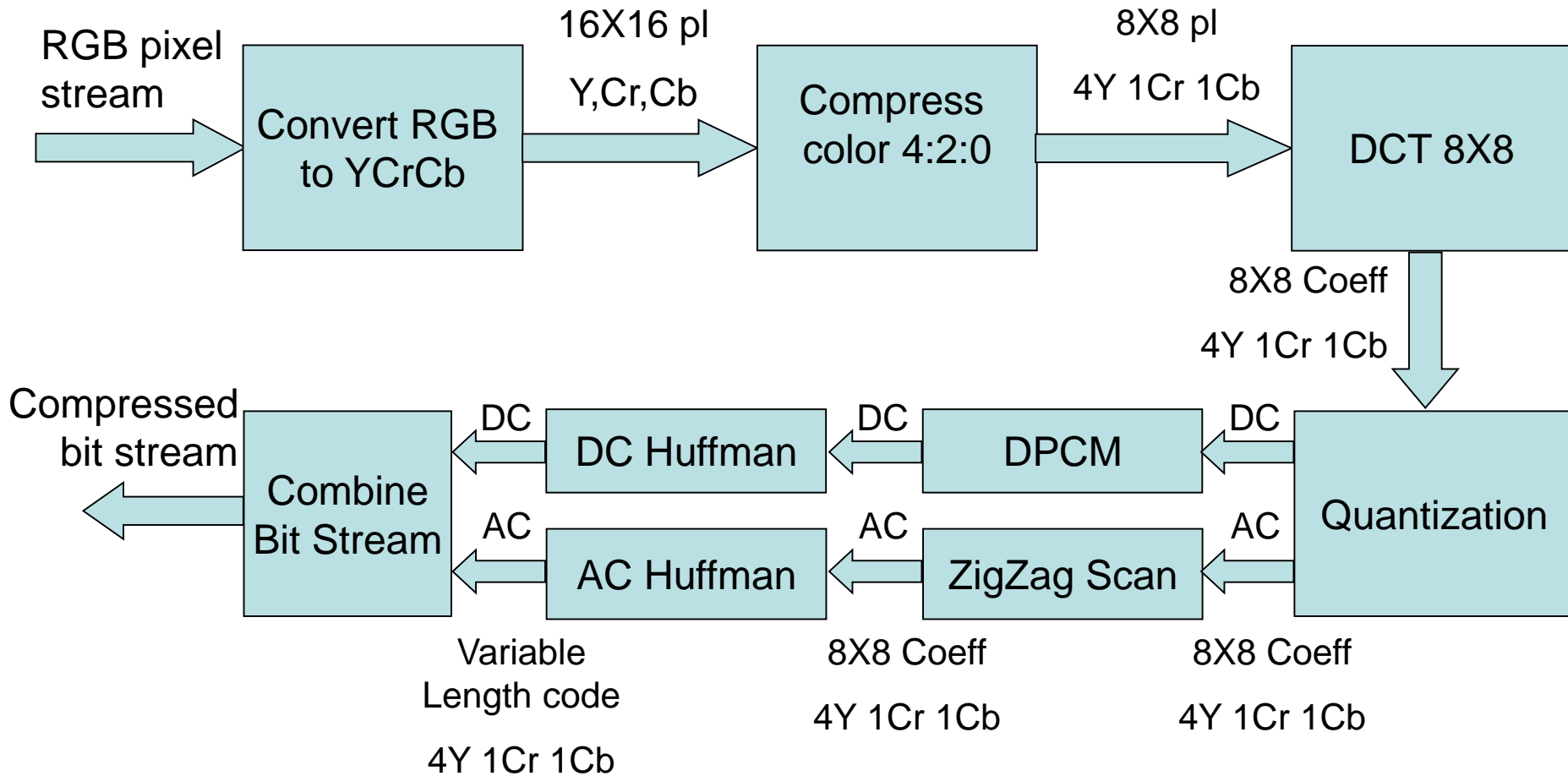


Challenges

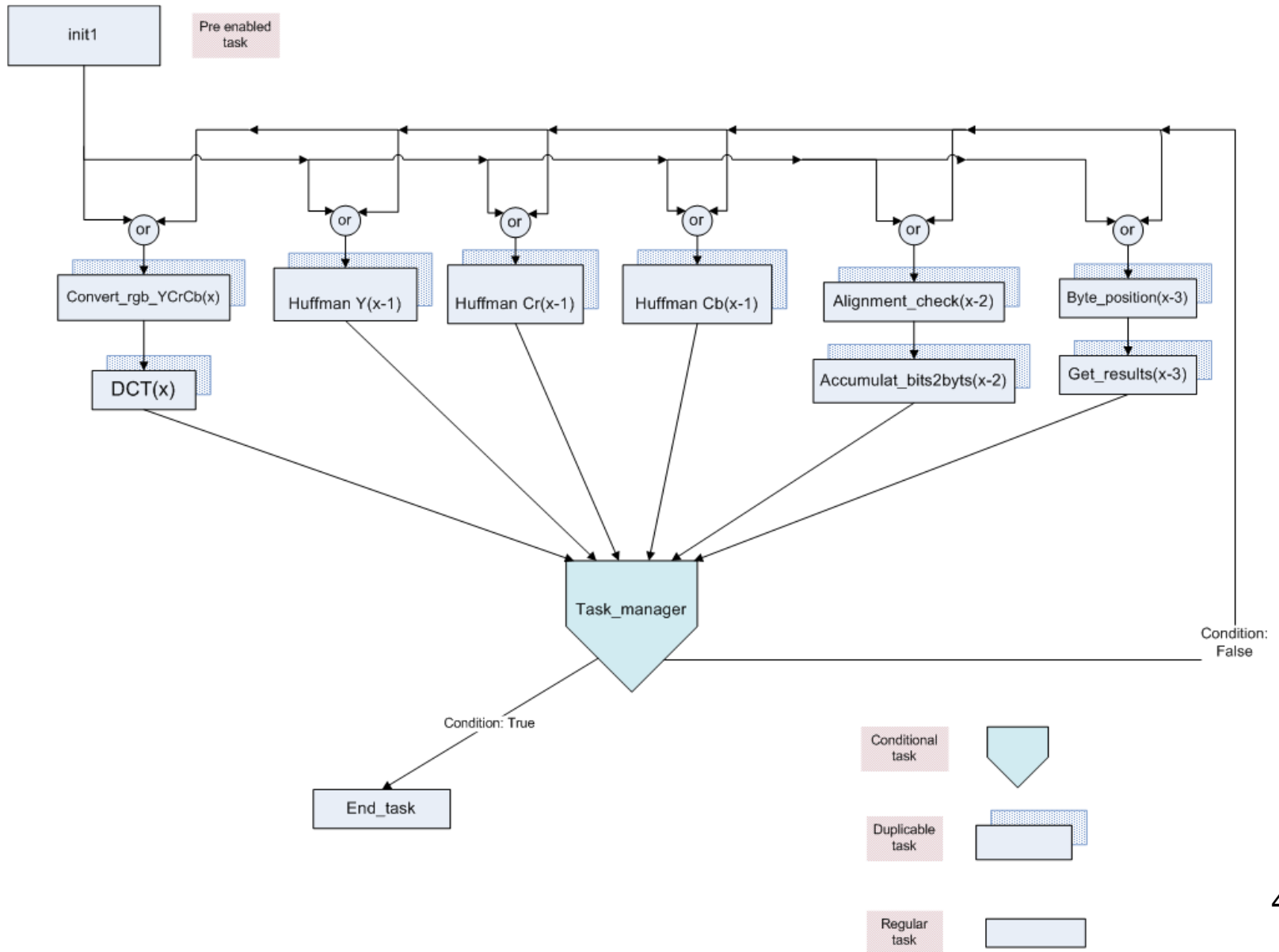
- What if on-chip memory is limited?
 - Input & output to/from same area
 - Process smaller data blocks
 - Decompose algorithm to fewer steps
 - Beware of combining serial and parallel code segments in same pipe stage
 - Stages may be serial, highly parallel, or limited parallel



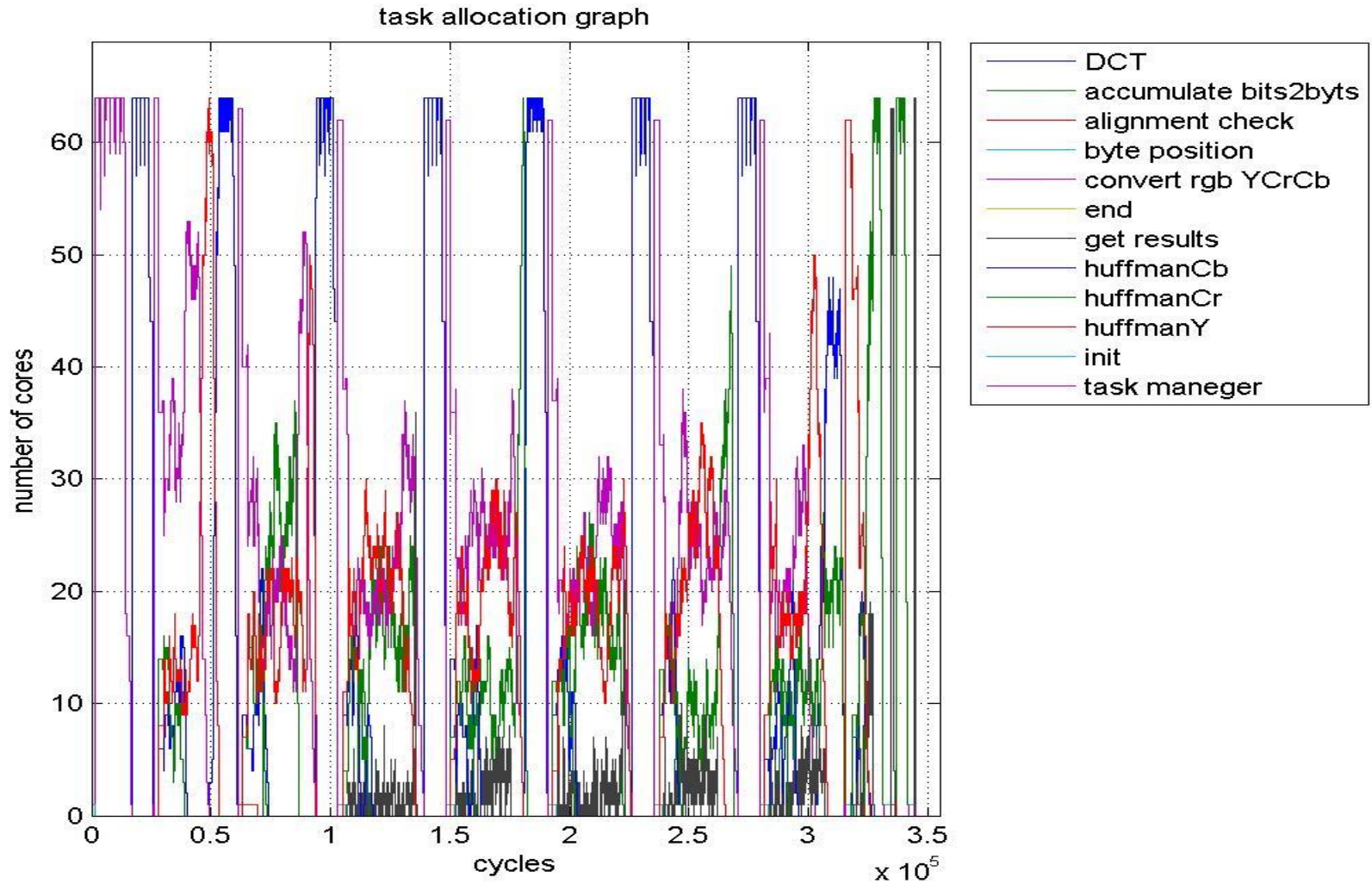
Example: JPEG compression algorithm using ManyFlow



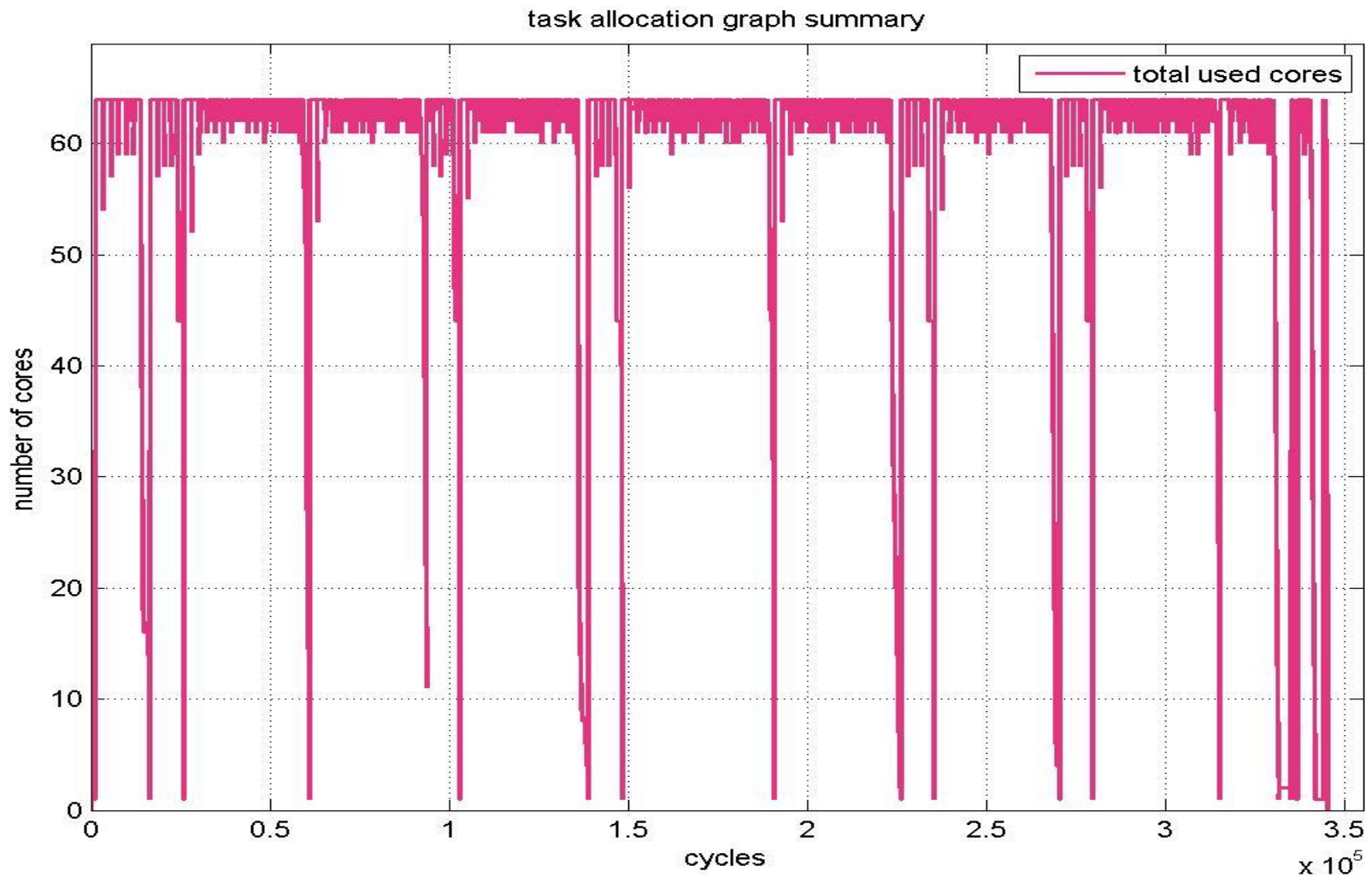
JPEG compression: ManyFlow



JPEG compression: Task Allocation



JPEG compression: Most cores active



Example: JPEG2000 Encoder

Image: 1K × 1K 8b pixels

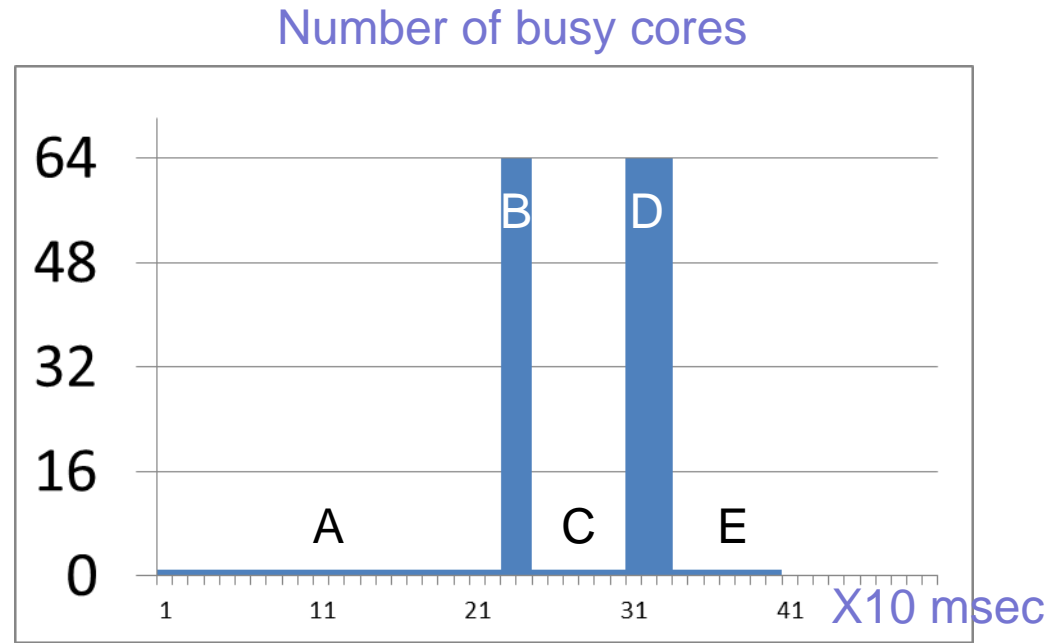
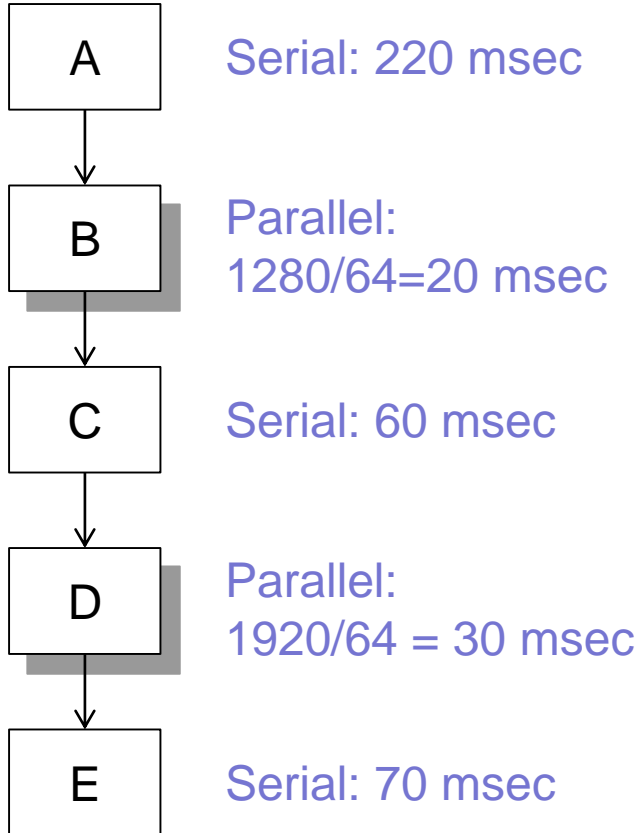
Core frequency $F_1 = 250$ MHz

Serial time $T_1 = 3.55$ sec

Parallel time $T_{64} = 400$ msec

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

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

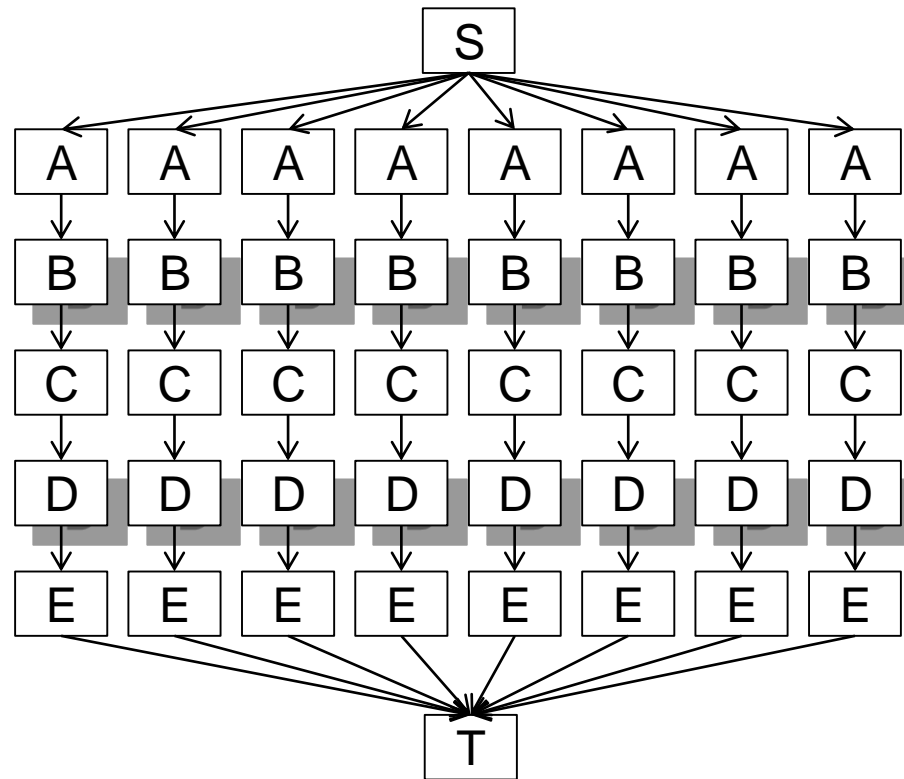


Parallel fraction $f=95\%$



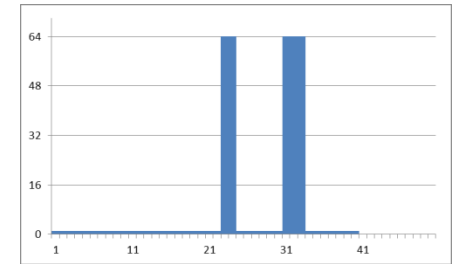
Non-ManyFlow RIGID Multi-Job Scheduling

- Run multiple serial sections in parallel
- Run a single parallel section at a time



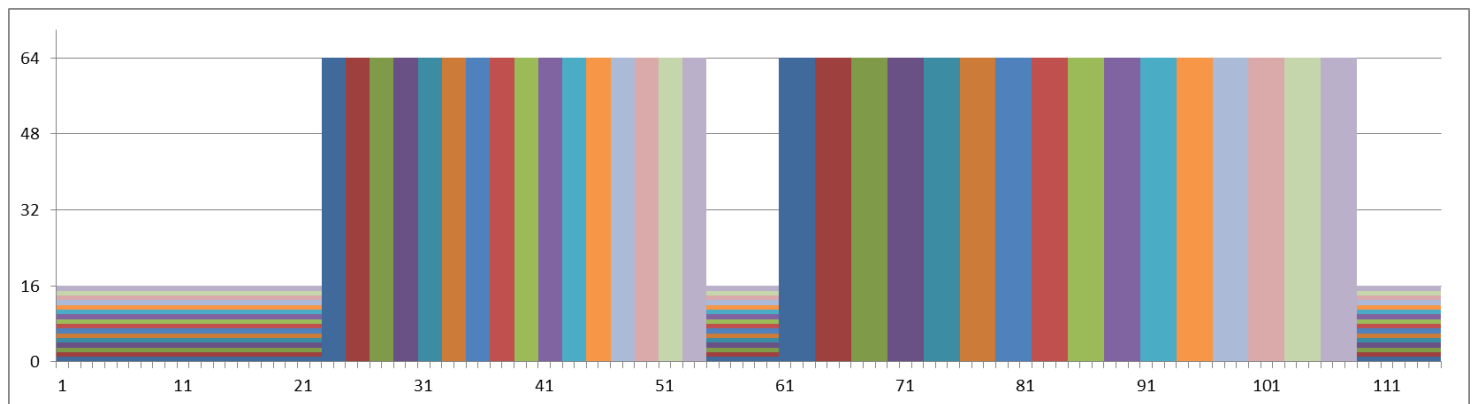
Non-ManyFlow RIGID Multi-Job Scheduling

- 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 $J=1,2,\dots$
- 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 $T_P(J) = T_{PS} + T_{PP}$



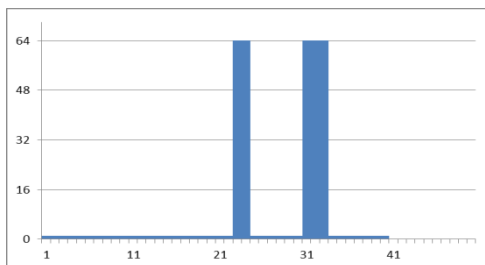
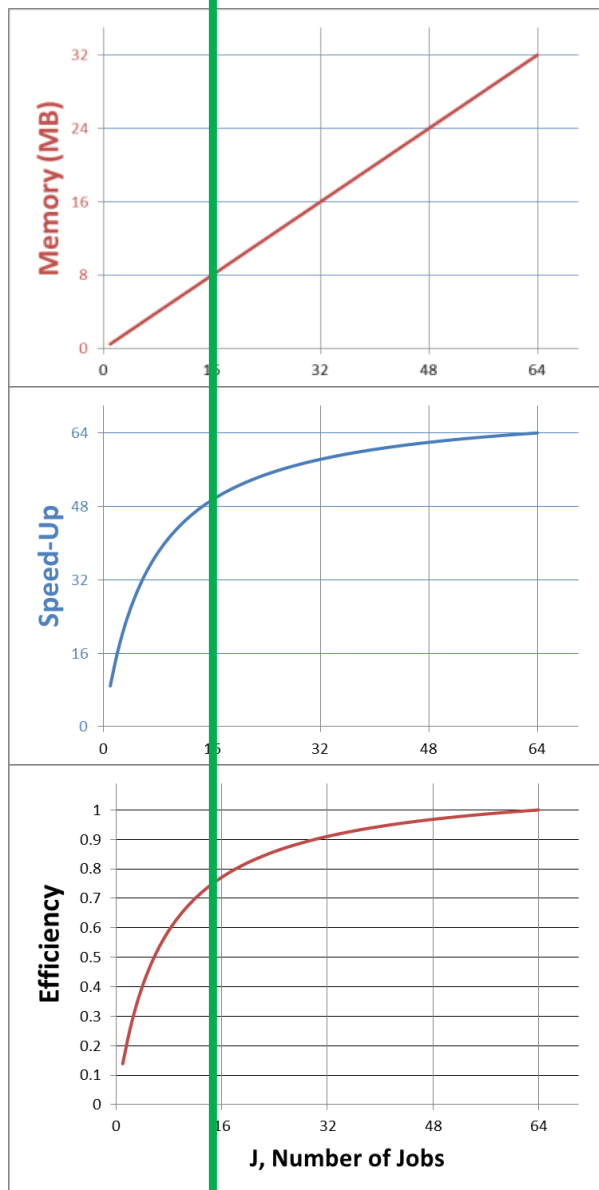
JPEG2000, J=1, $f=95\%$

J=16



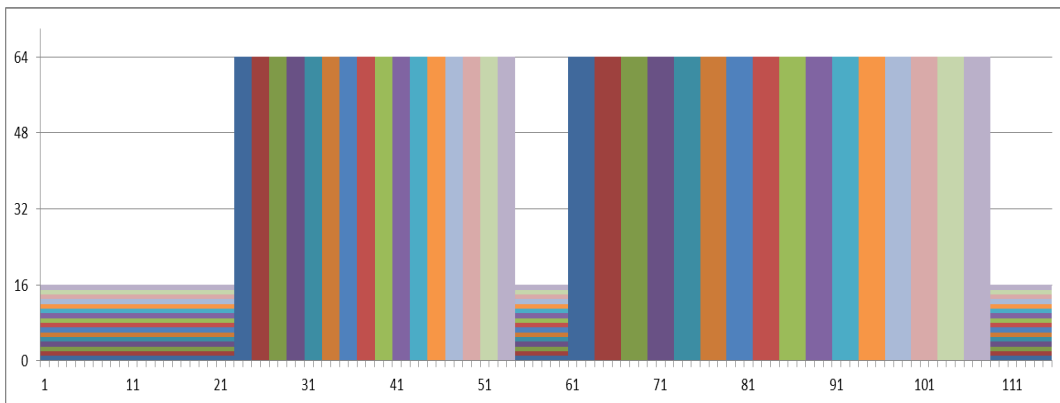
Non-ManyFlow RIGID Multi-Job Scheduling

- Memory-limited
- 8MB ($\frac{1}{4}$ max memory) enables:
 - J=16 jobs
 - Speed-up 50 (cf. 9)
 - 0.8 efficiency (cf. 0.14)
 - *ManyFlow works better !*



JPEG2000, J=1

J=16

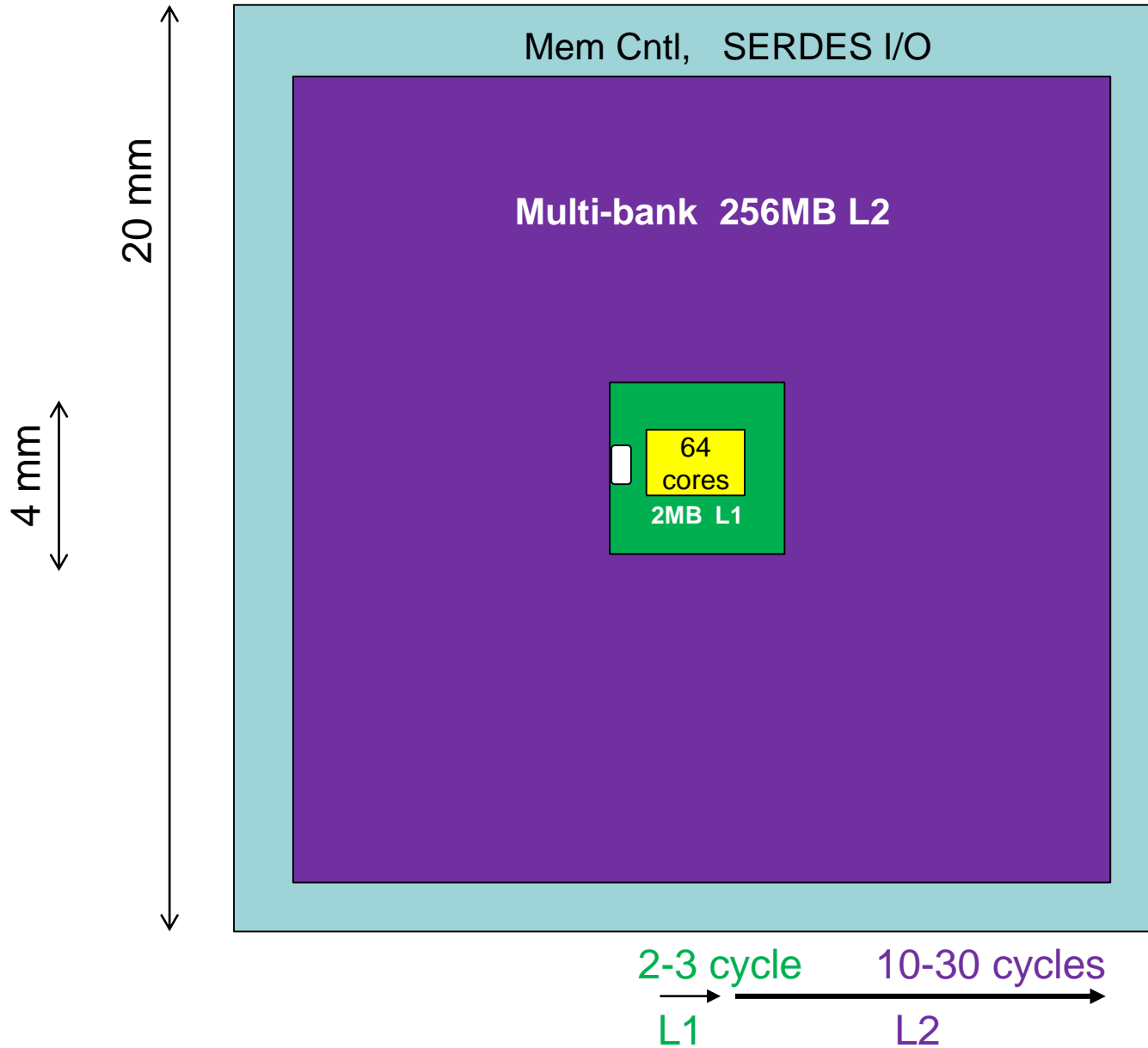


Outline

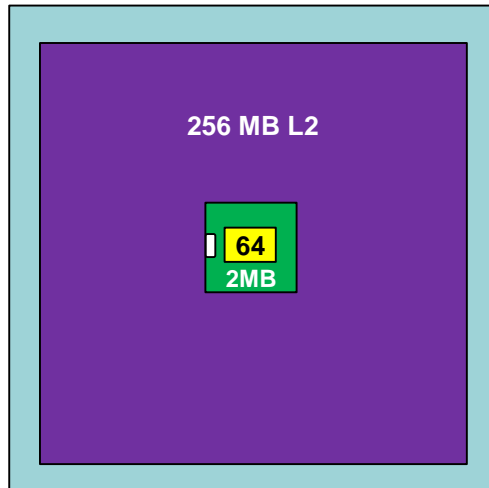
- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Plural programming examples
- ManyFlow for the Plural architecture
- Scaling the Plural architecture
- Mathematical model of the Plural architecture



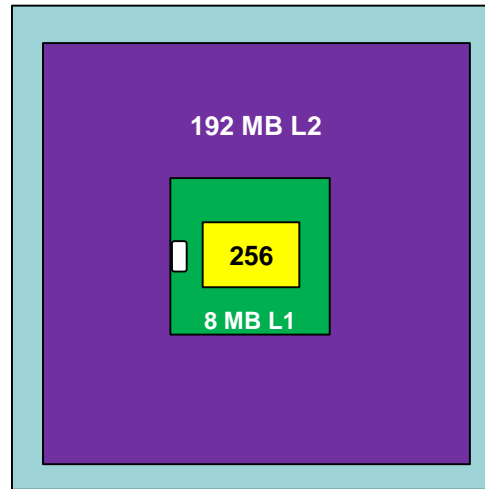
Possible Full-Chip Plan



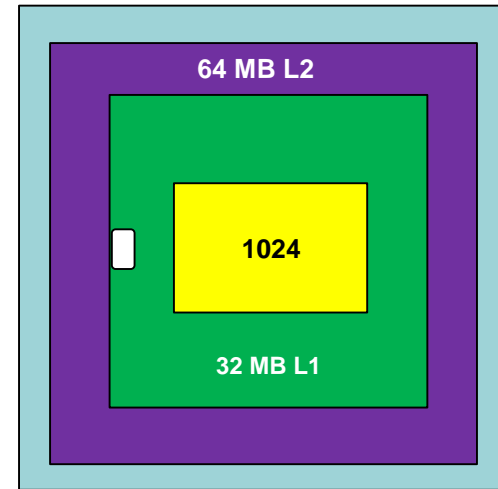
But does it scale (more processors)?



64 Cores
2 MB shared L1
256 MB shared L2

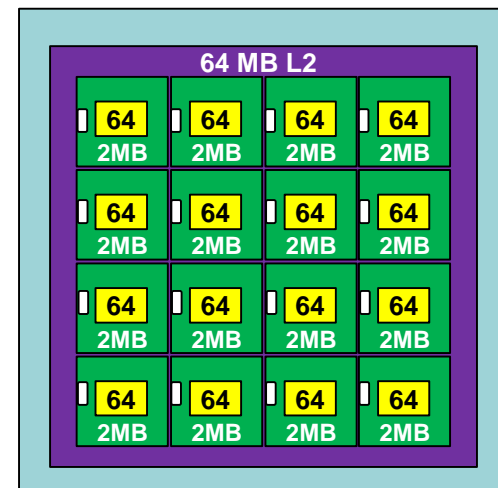
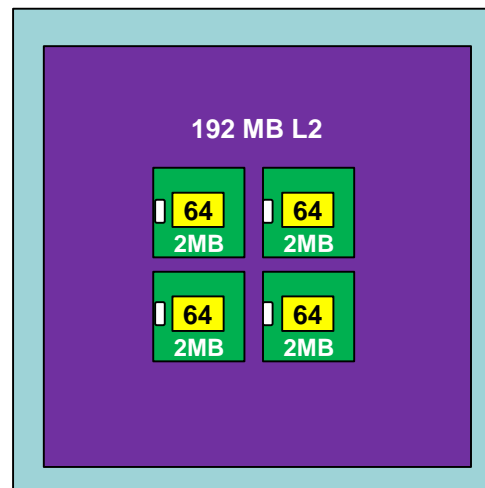


256 Cores
8 MB shared L1
192 MB shared L2



1024 Cores
32 MB shared L1
64 MB shared L2

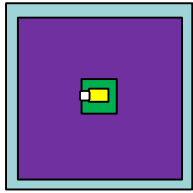
Long, high energy access to larger shared memory



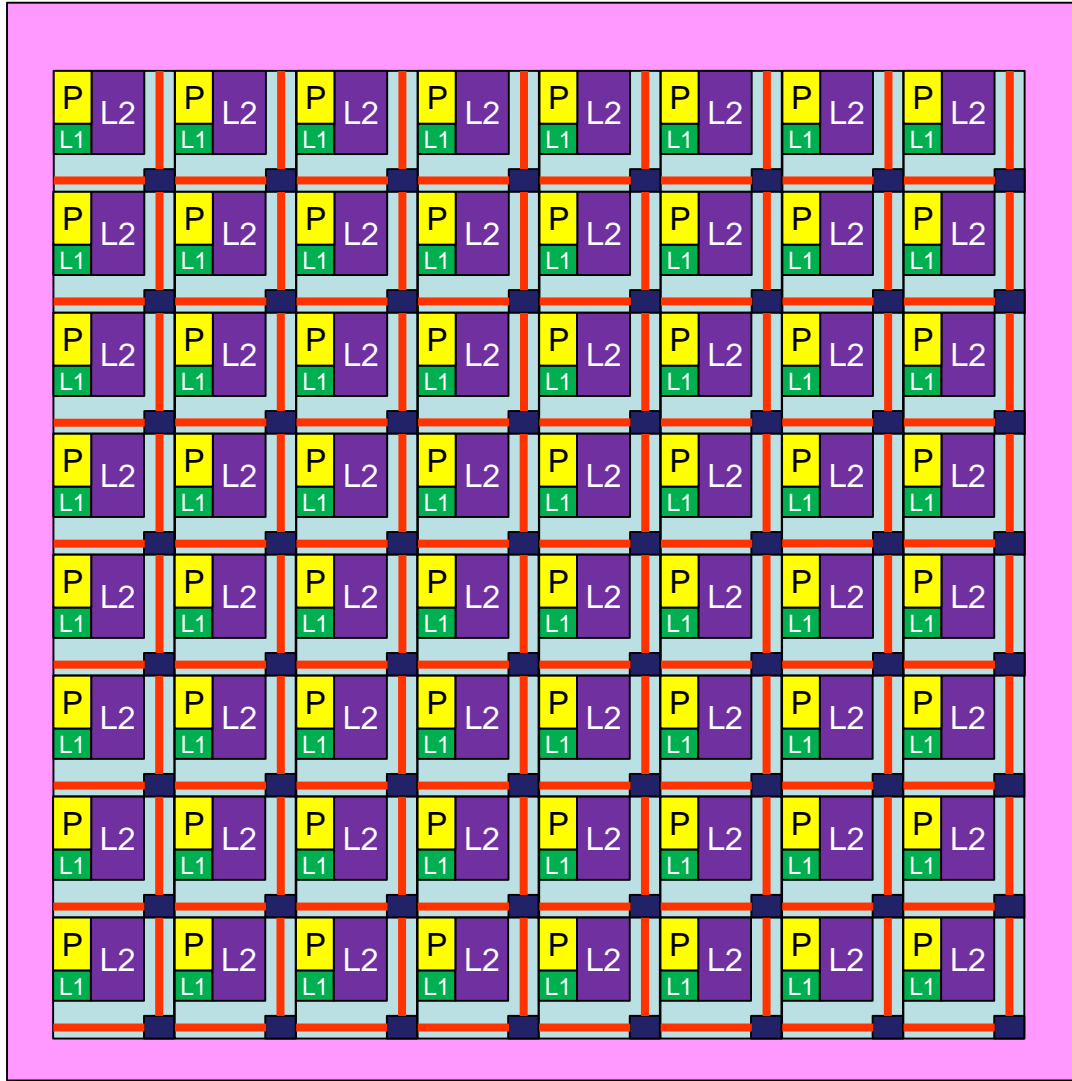
Cluster, Not a shared memory



Compare with "tiled" CMP using mesh NOC



20 mm



20×20mm

64 tiles

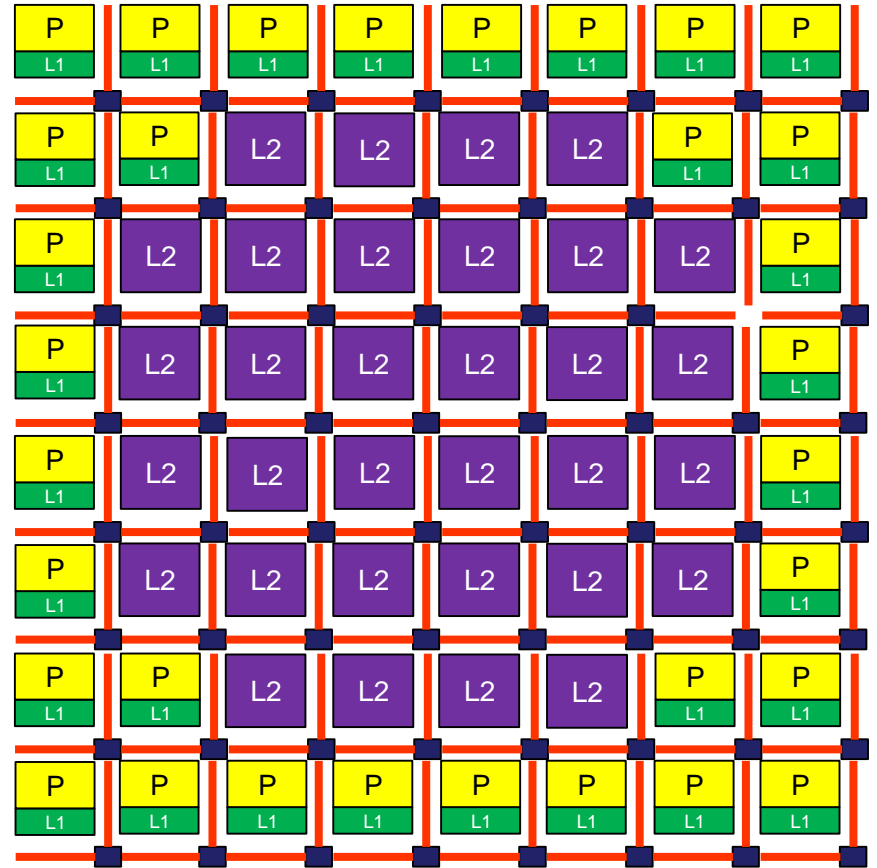
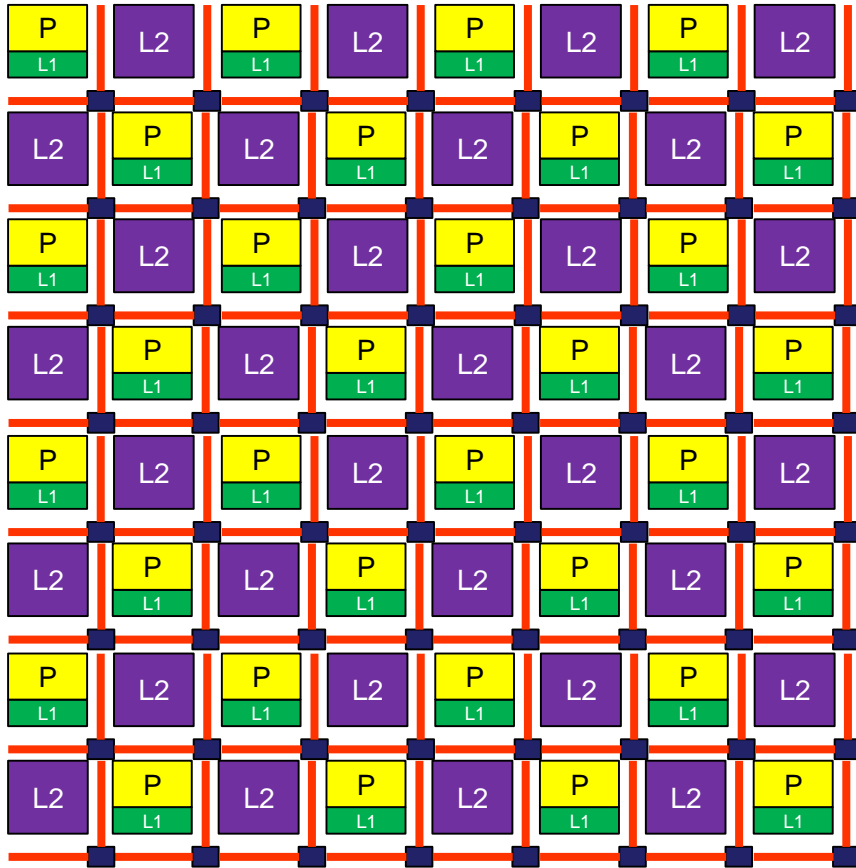
32 kB L1 x64
= 2 MB

4 MB L2 x64
= 256 MB

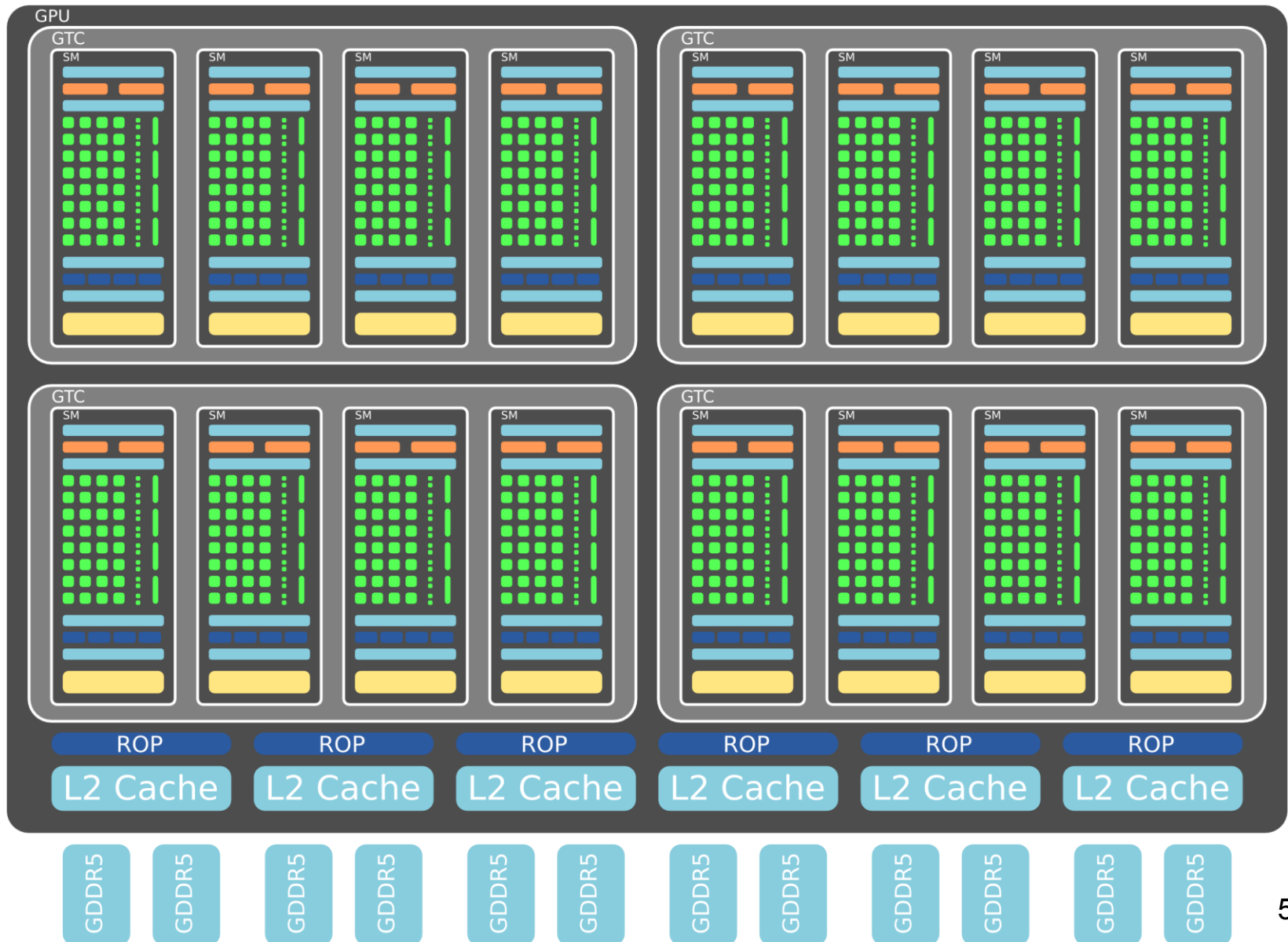
Directory:
All L2's = L3



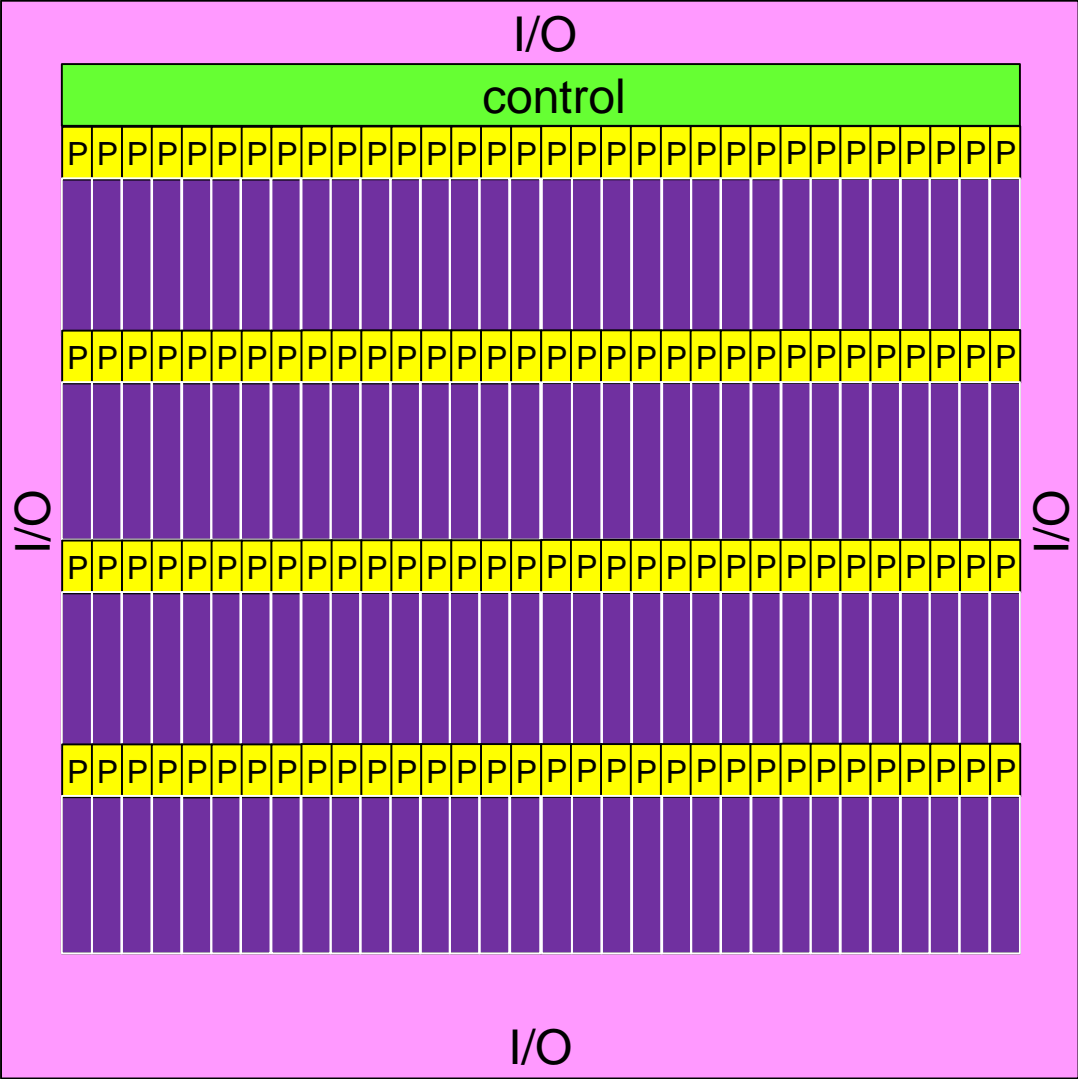
Other proposed NOC-based manycores



GPU: Yet another manycore



Another idea: SIMD



EXAMPLE

256 cores

memory banks
1 MB x256
= 256 MB



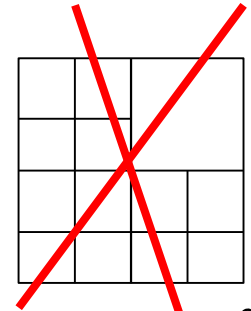
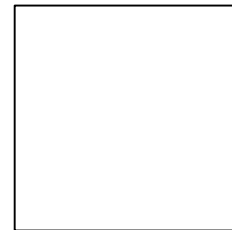
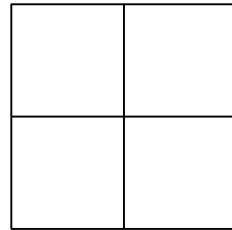
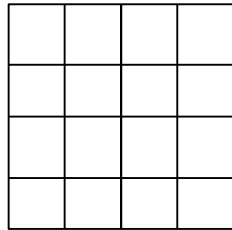
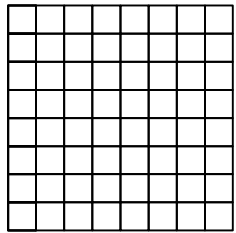
Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Plural programming examples
- ManyFlow for the Plural architecture
- Scaling the Plural architecture
- Mathematical model of the Plural architecture



The many-core research question

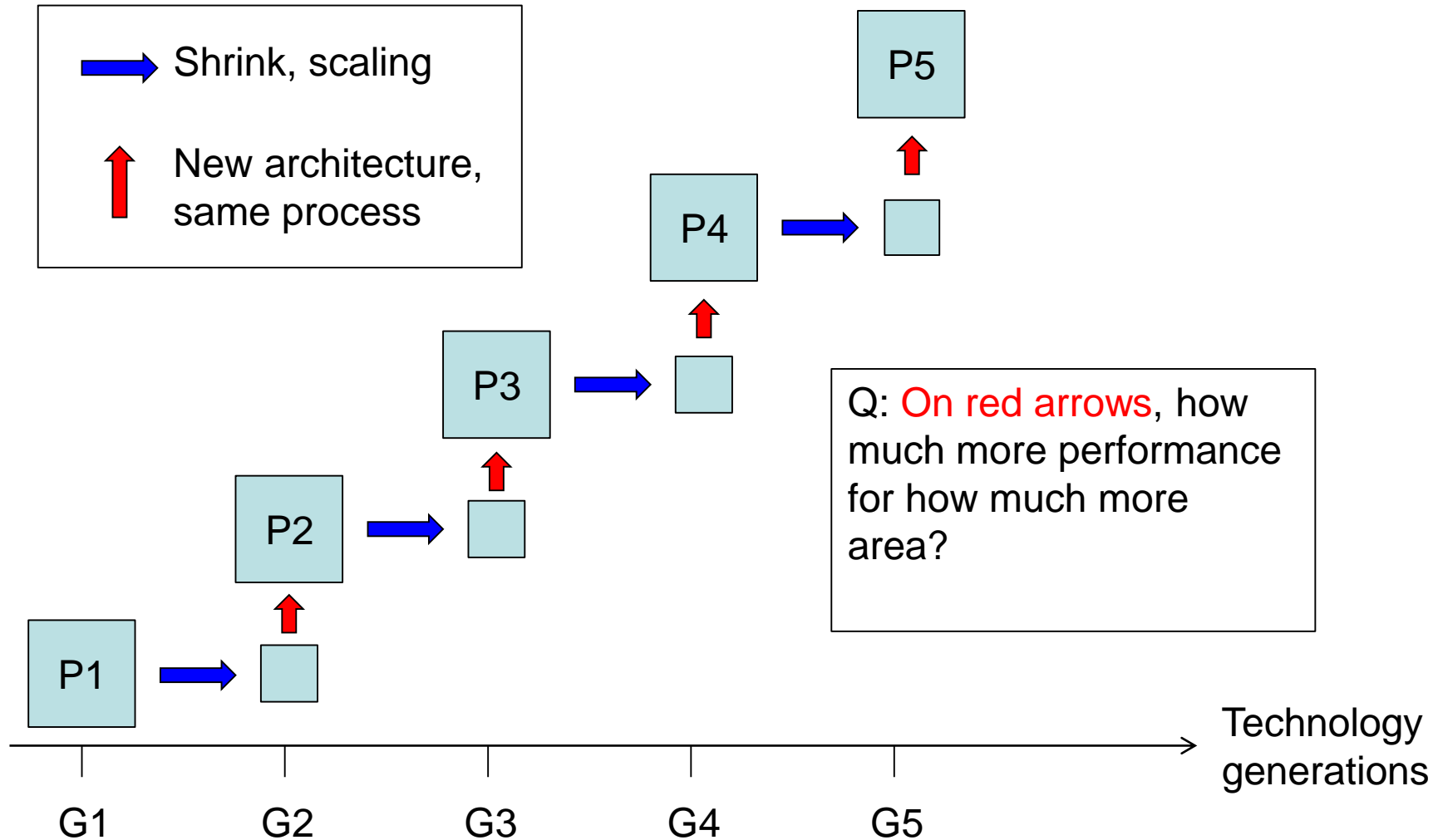
- Given fixed **area**, into how many processor cores should we divide it?



No heterogeneous many-cores in this discussion

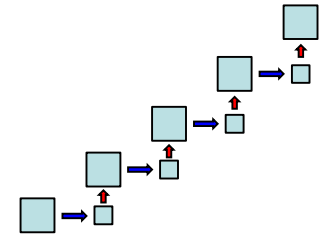
- Analysis can be based on Pollack's rule
- Other good questions (not dealt here):
 - Given fixed **power**, how many cores? which cores?
 - Given fixed **energy**, how many cores? which cores?
 - Given target performance, how many? Which?

The history at the basis of Pollack's analysis



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 $\sim \sqrt{area}$



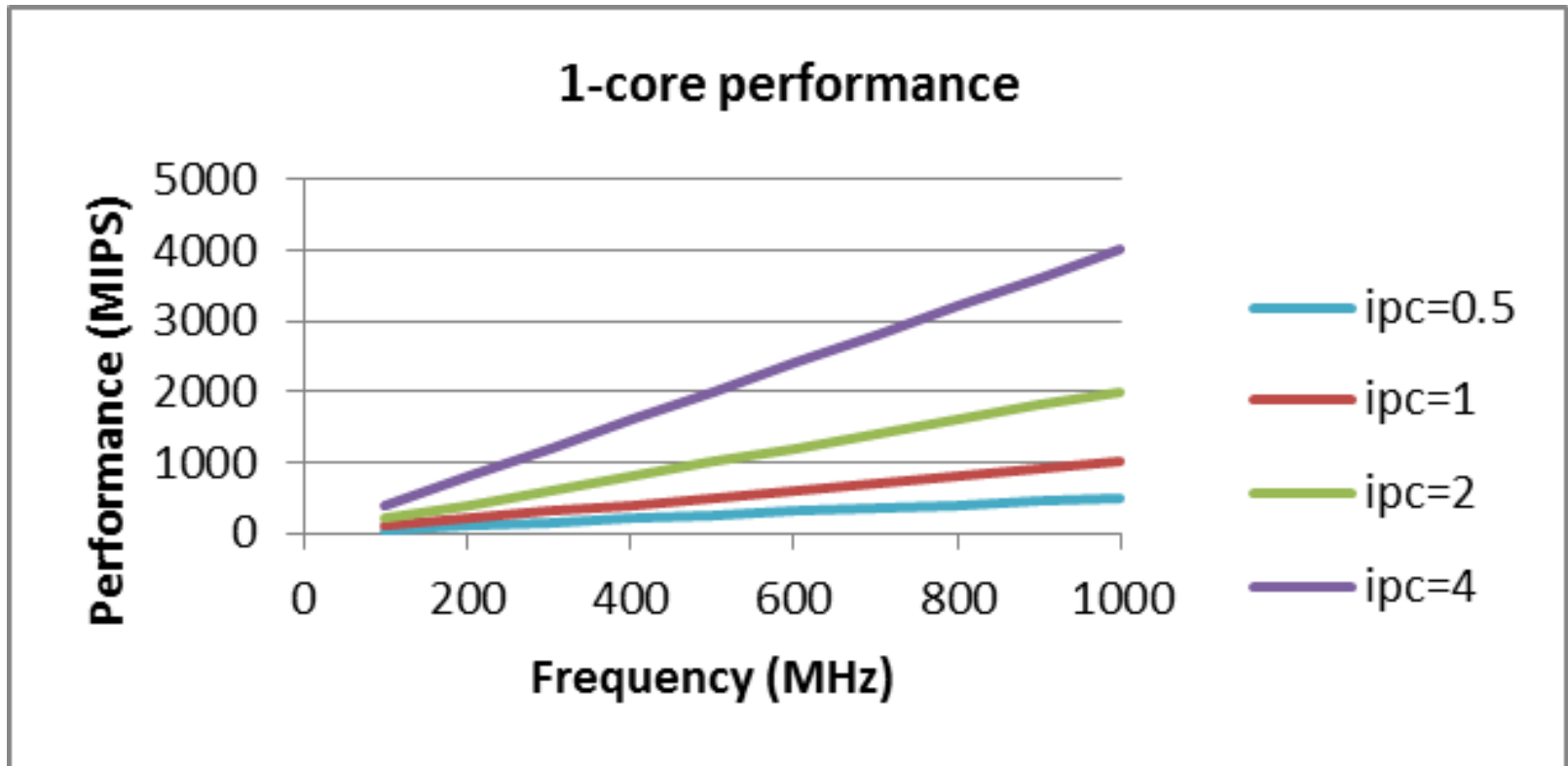
The many-core *fixed-total-area* model

- Assume fixed chip area (typically 300-500 mm²)
- Split chip area $A = A_{\text{cores}} + A_{\text{mem}}$
 - Memory size addressed by other math models
- Divide A_{cores} into m cores. How many ?
 - Area of each core: $a = \frac{A_{\text{cores}}}{m}$. Thus, $m \sim 1/a$
- [Pollack's]: core area determines core performance. Select IPC and frequency f so that:
 - Performance (core) = $IPC \times f \sim \sqrt{a}$. 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) = $a \times f \times m \sim \frac{IPC^2 f^3}{IPC^2 f^2} = f \sim \frac{1}{IPC \sqrt{m}}$

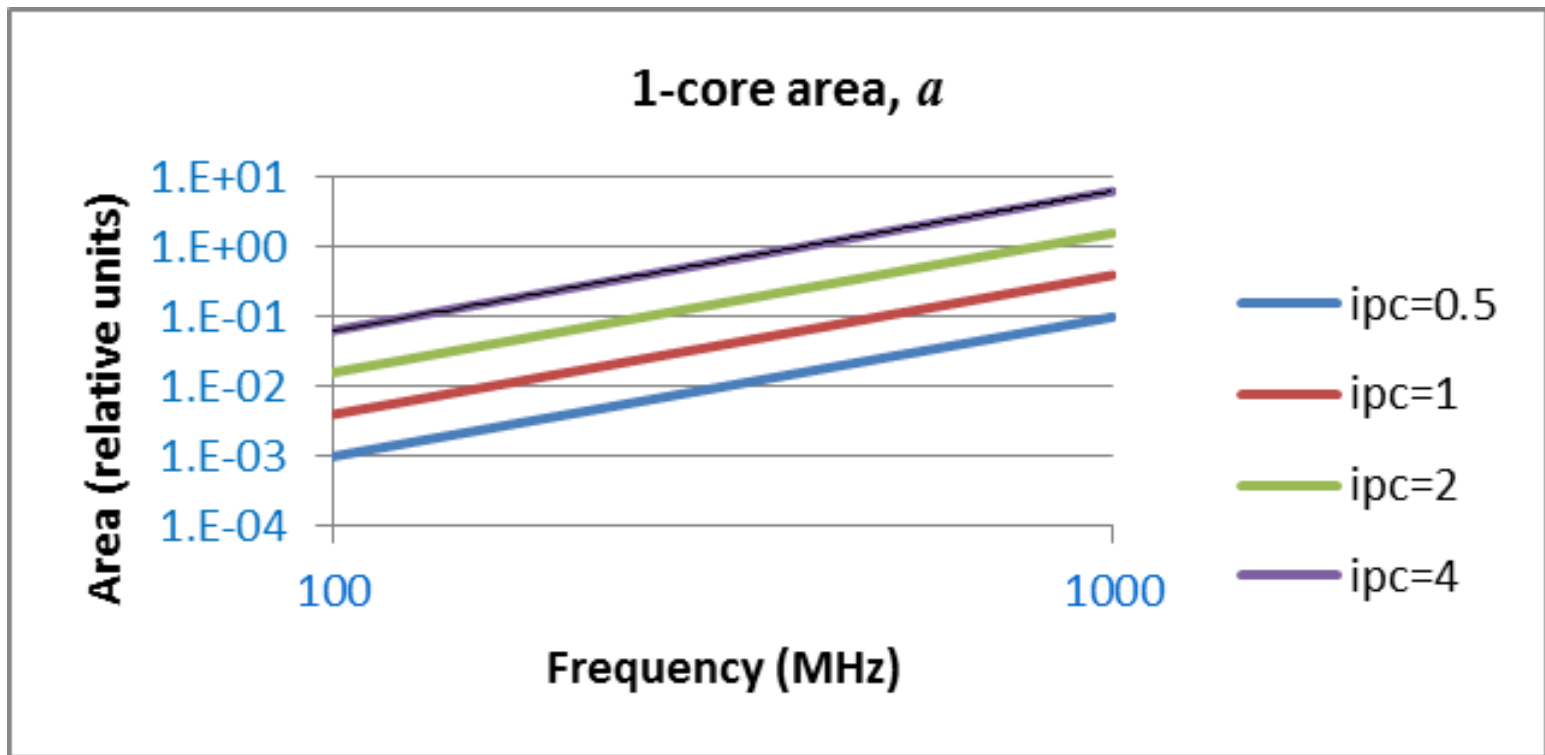
Summary: Performance $\sim \frac{1}{f} \sim \sqrt{m}$, Power $\sim \frac{1}{\sqrt{m}} \sim f$, $m \sim \frac{1}{f^2}$



$$\text{Performance (core)} = \text{IPC} \times f$$

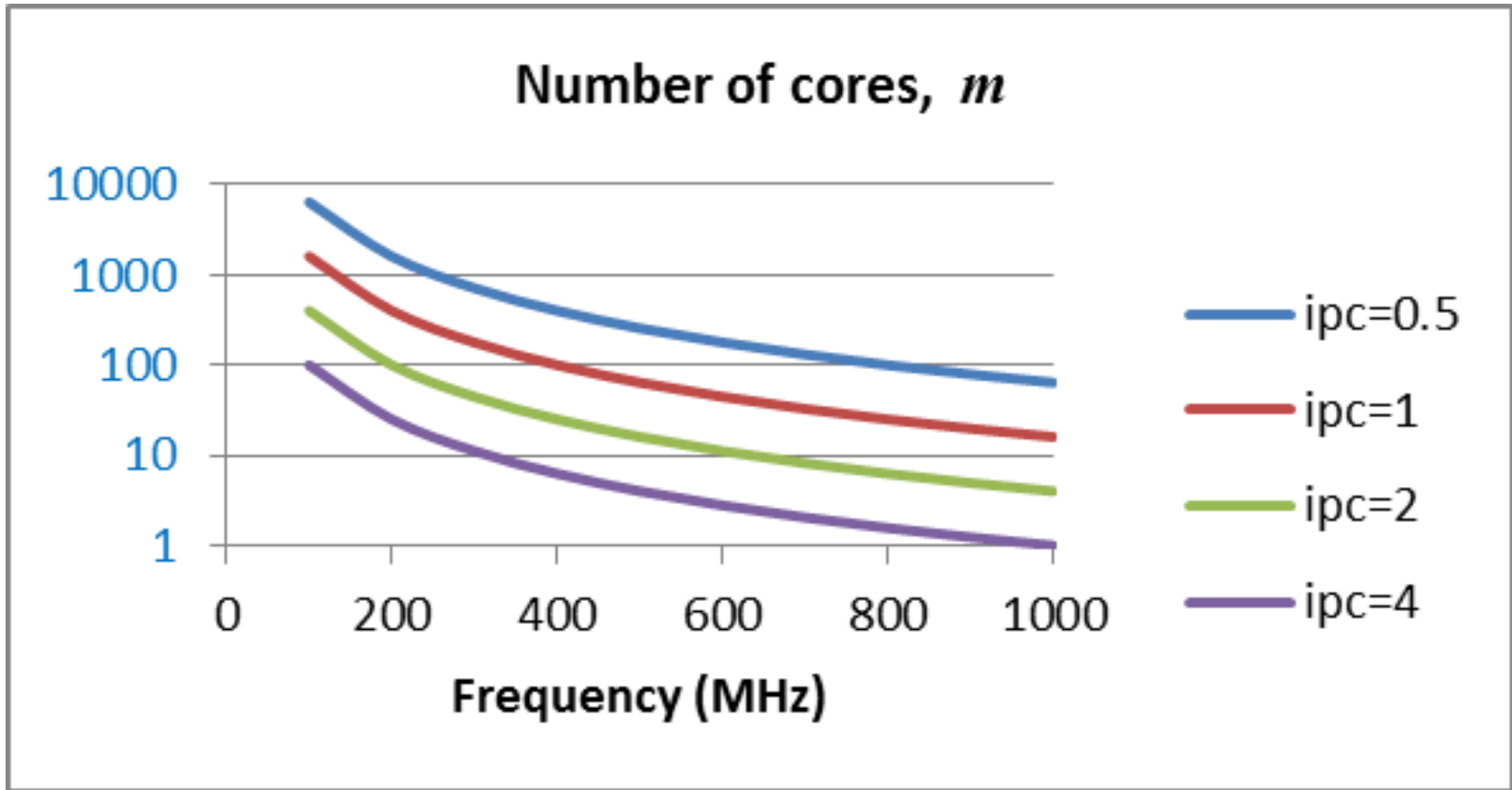


$$a \sim IPC^2 f^2$$



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

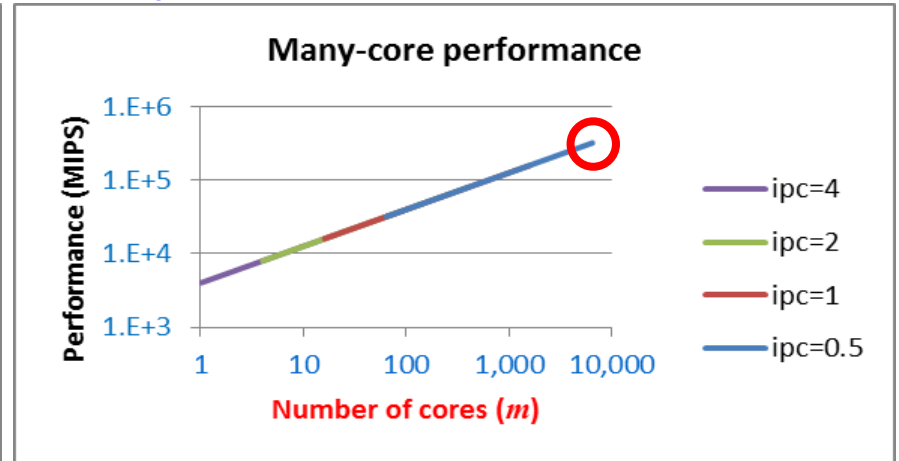
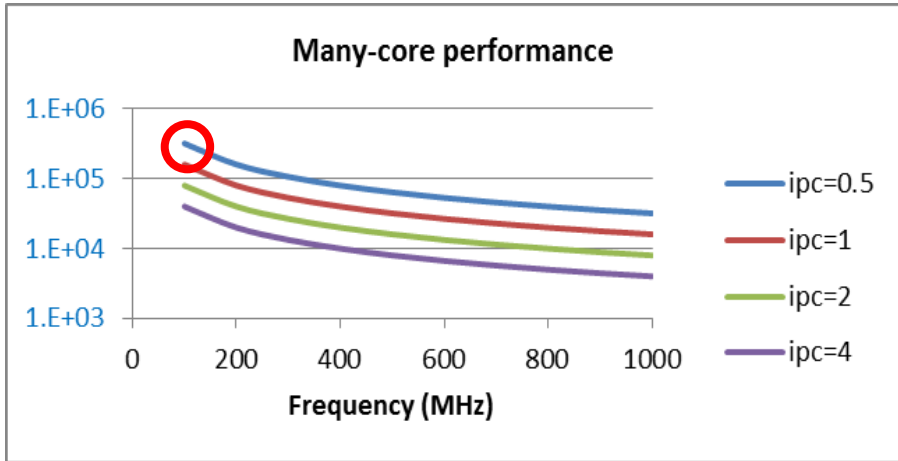
$$m \sim \frac{1}{IPC^2 f^2}$$



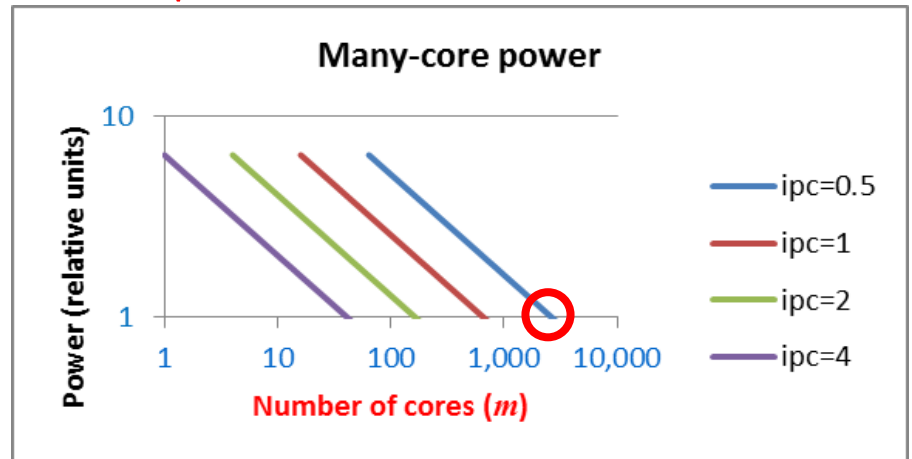
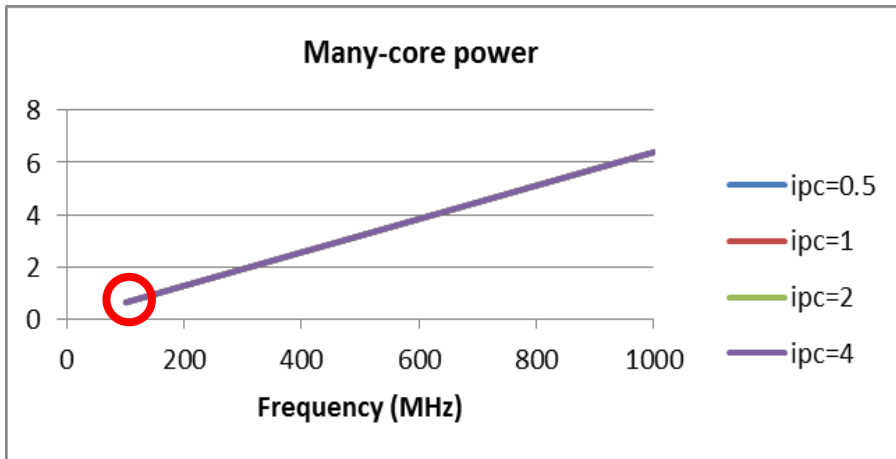
For each IPC curve, $m \sim \frac{1}{f^2}$



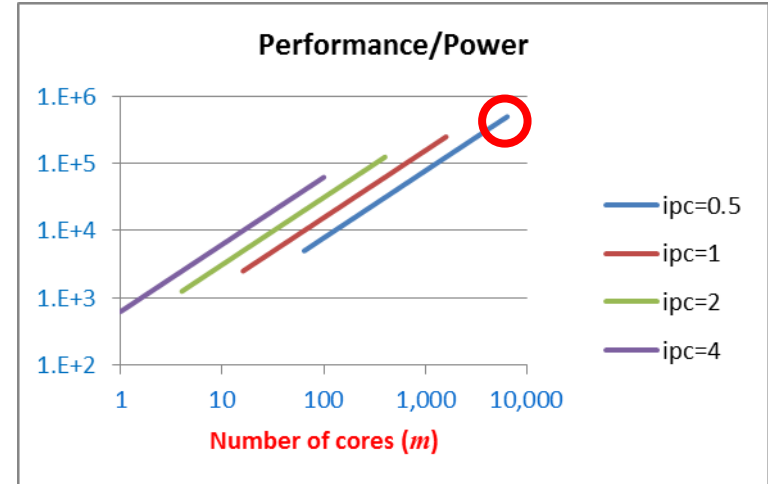
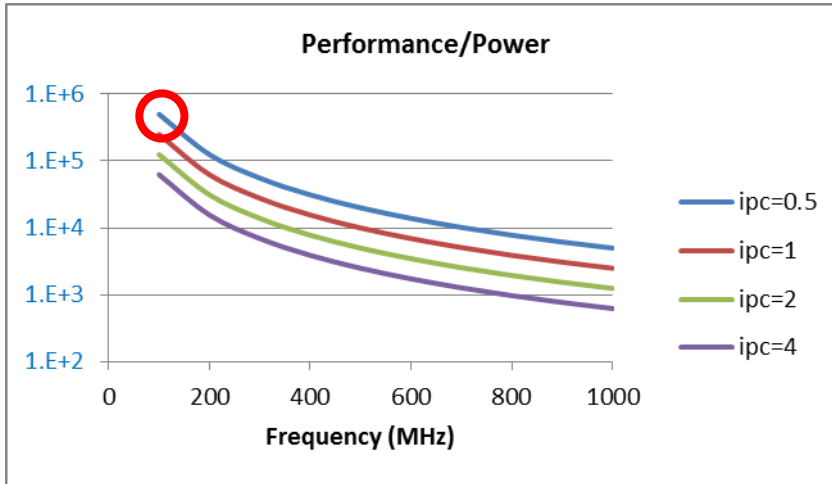
$$\text{Performance} \sim \frac{1}{f} \sim \sqrt{m}$$



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



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



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...

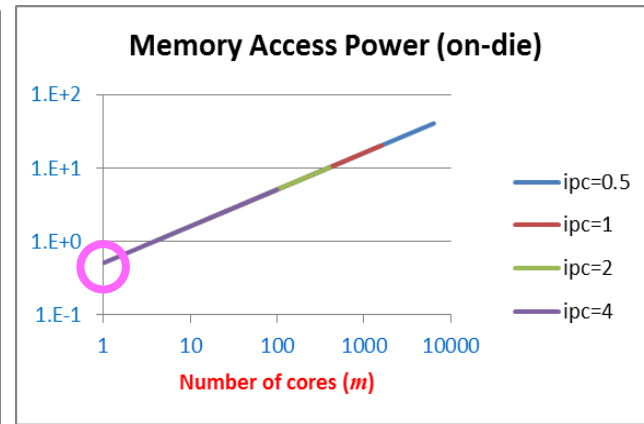
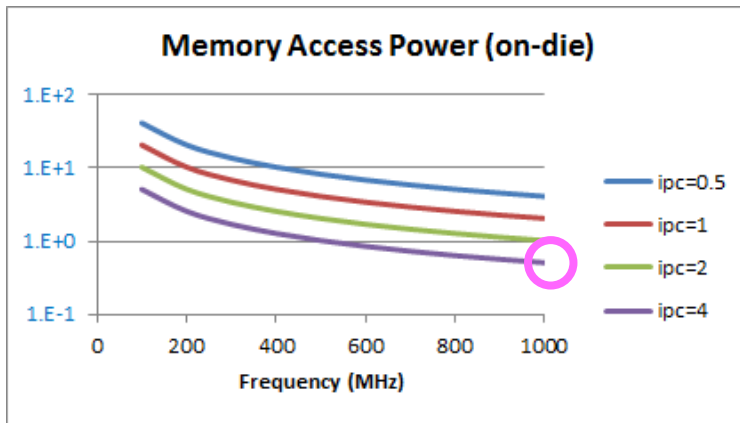


Now add memory

- 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.

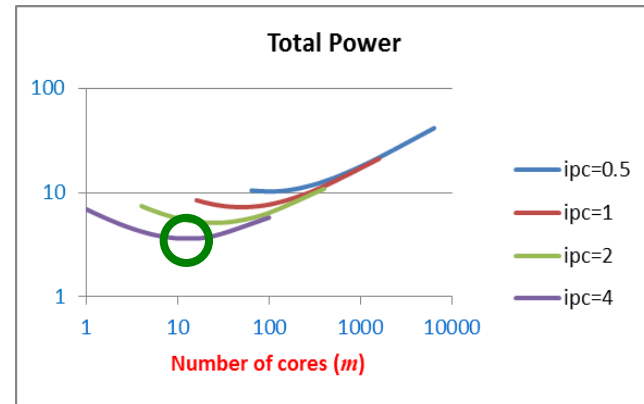
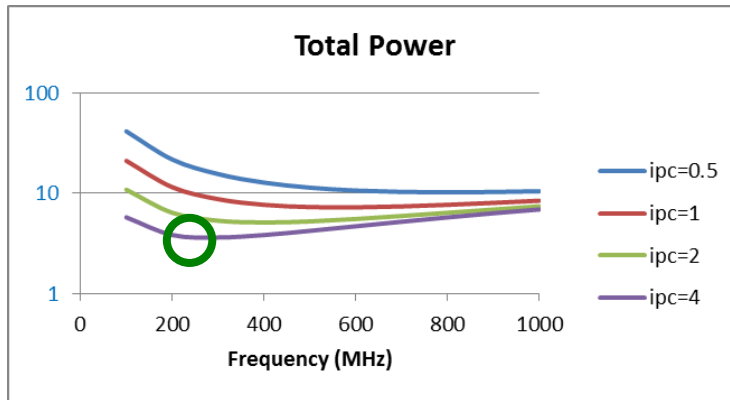


$$\frac{1}{f}$$



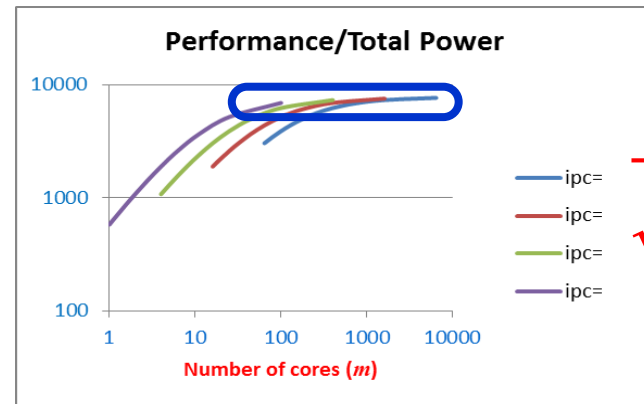
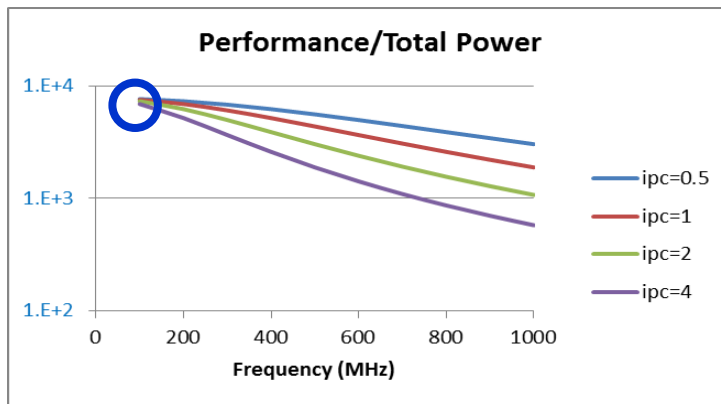
$$\sqrt{m}$$

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



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

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



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



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 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 (?)



On-going Research

- Mathematical model incl. memories
- Scaling: full chip, multiple chips
- 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
- 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

