The Plural Architecture

Shared Memory Many-core with Hardware Scheduling

Ran Ginosar

Technion, Israel

and RAMONchips

Oct 2017

Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Validation
- Plural programming examples
- ManyFlow for the Plural architecture

many-cores

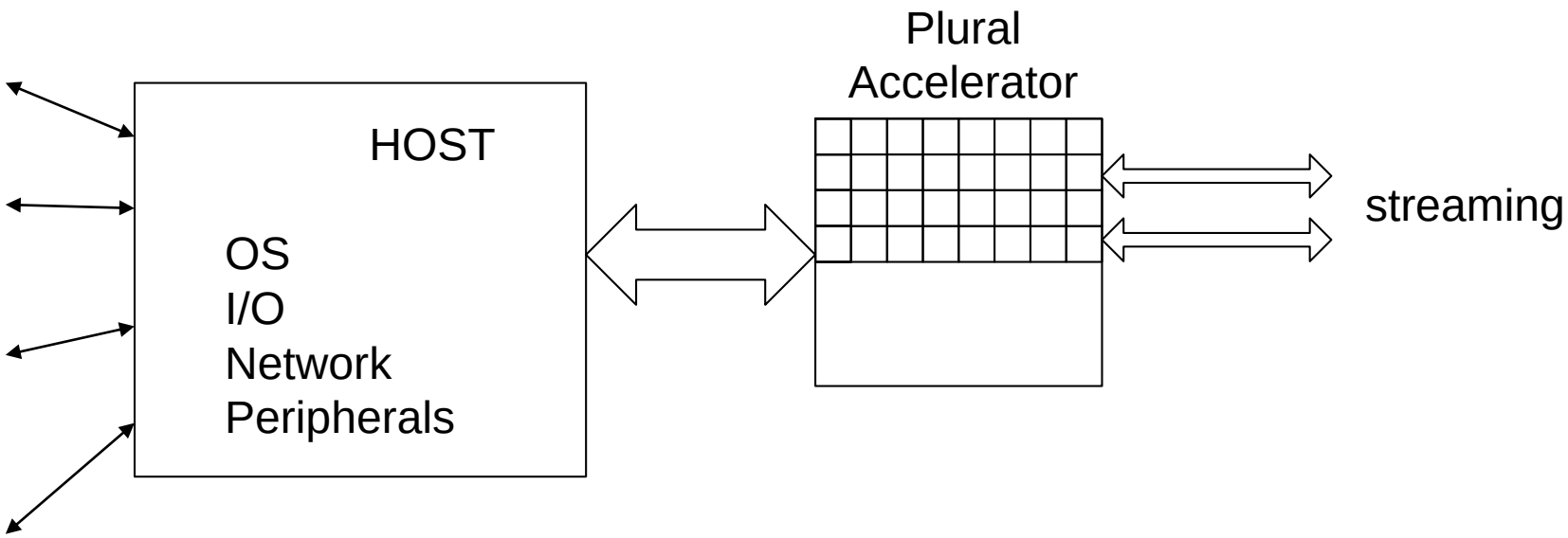
- Many-core is:
  - a single chip
  - with many (how many?) cores and on-chip memory
  - running one (parallel) program at a time, solving one problem
  - an accelerator
- Many-core is NOT:
  - Not a “normal” multi-core
  - Not running an OS
- Contending many-core architectures
  - Shared memory (the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

Why manycores? Scaling

- We know of VLSI scaling
  - Going forward with technology
  - See VLSI course
- Another scaling: More cores using same technology
  - Example: Start with one core
    - Voltage V_1
    - Frequency ≈ performance = f_1
    - Power $P_1 \approx C V_1^2$
  - Move to 2 cores
    - Voltage $V_2 = 0.8 V_1$
    - Frequency $f_2 = 0.8 f_1$
    - Performance = $2 f_2 = 1.6 f_1$
    - Power $P_2 = 2C V_2^2 = 2C (0.8 V_1)^2 = 2 \times 0.64 C V_1^2 = 1.3 P_1$
  - Conclusions
    - Increasing frequency is dead
    - Naïve parallelism is naïve
    - Manycore scaling can scale

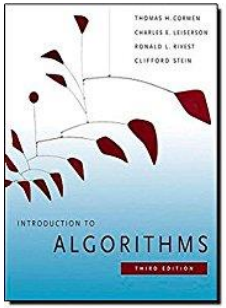
Context

- Plural:
  - homogeneous acceleration
  - for heterogeneous systems



One (parallel) program ?

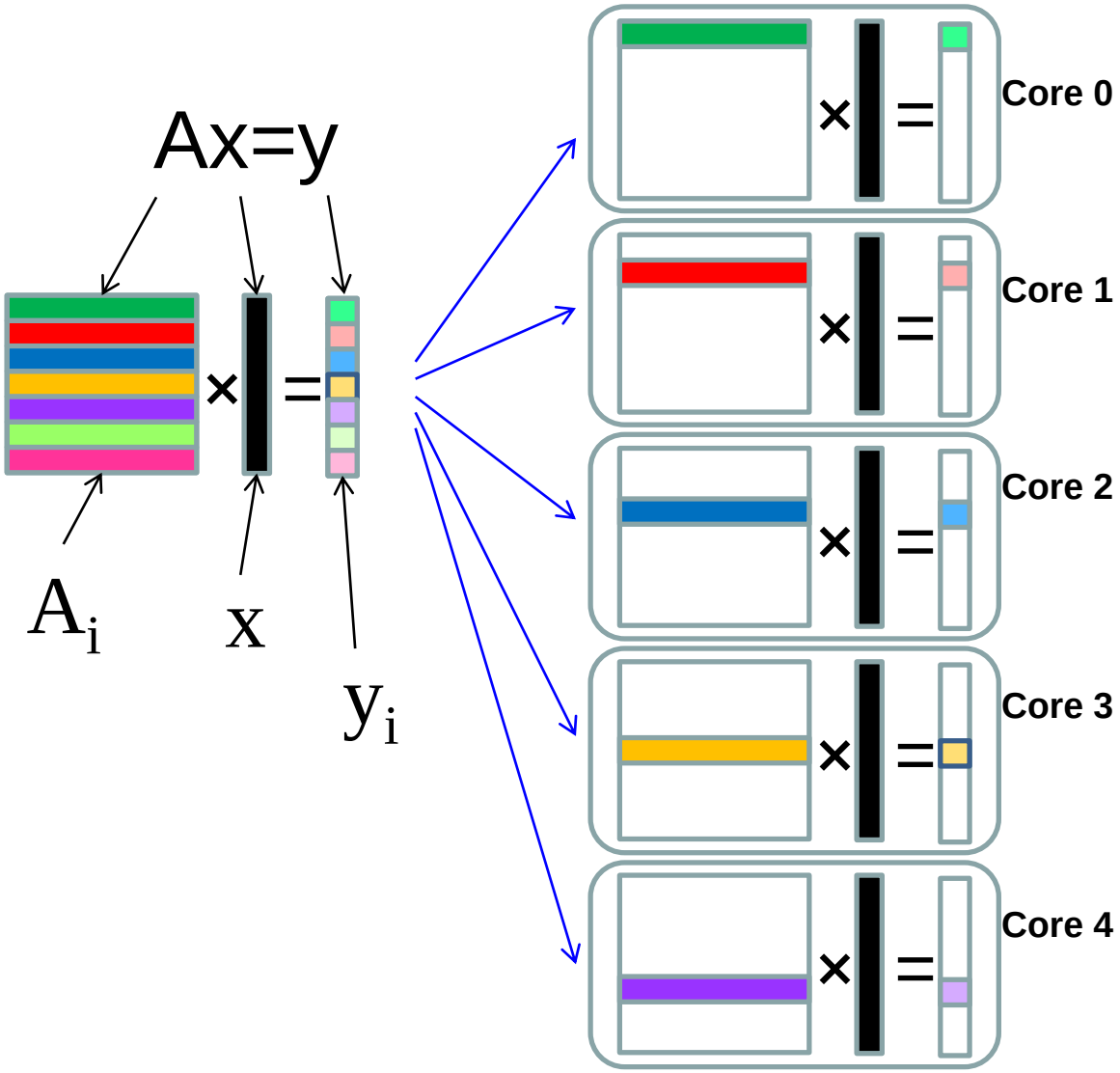
- Best formal approach to parallel programming is the PRAM model
- Manages
  - all cores as a single shared resource
  - all memory as a single shared resource
- and more…



Cormen, Leiserson, Rivest, Stein, Introduction to algorithms, 2009

Joseph F. JaJa, Introduction to Parallel Algorithms, 1992

PRAM matrix-vector multiply



The PRAM algorithm

i is row 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)

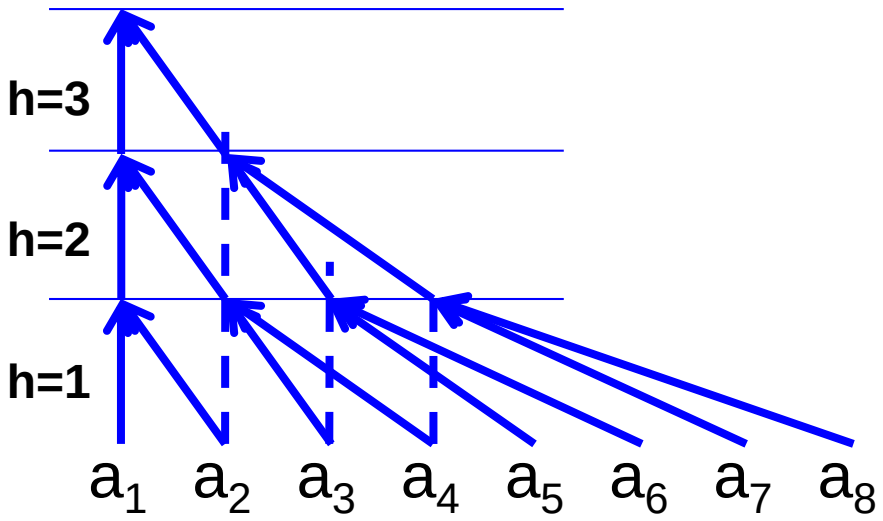
PRAM logarithmic sum

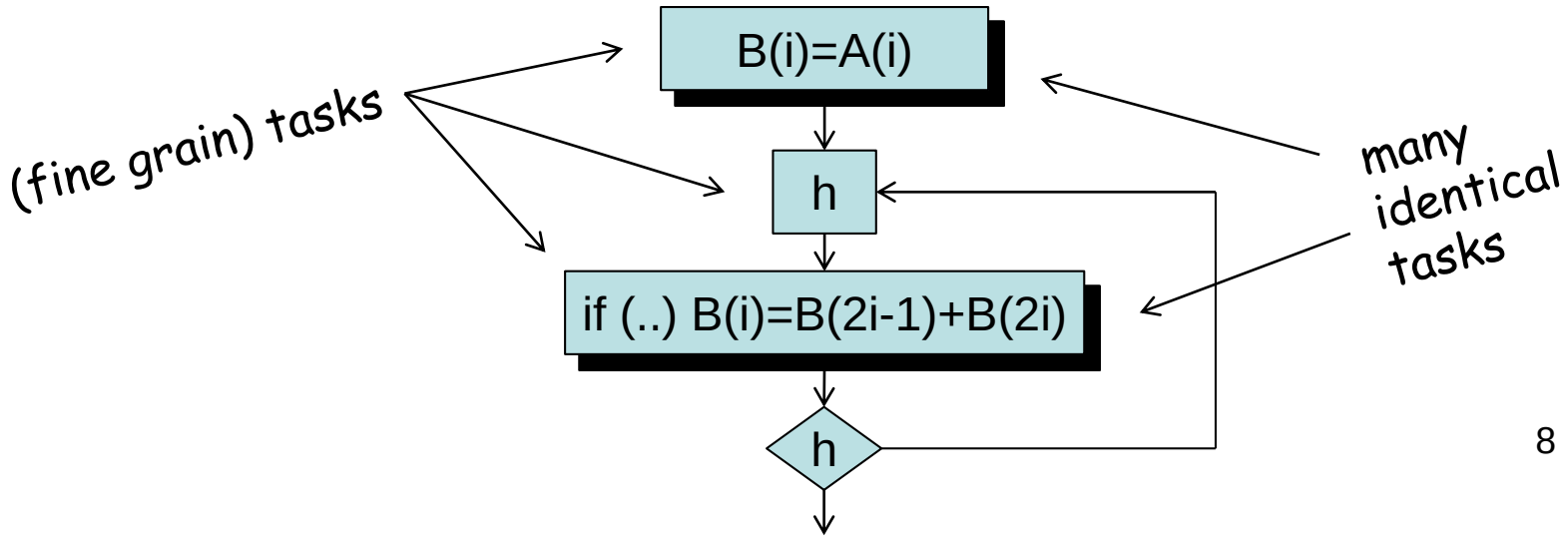
The PRAM algorithm

// Sum vector A(*)

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

// B(1) holds the sum





Advantages of PRAM-like programming

- Simpler program
  - Flat memory model
  - Same data structures as in serial code
  - No code for finding and moving the data
  - Easier programming, lower energy, higher performance
  - Scalable to higher number of cores

Advantages of PRAM-like programming

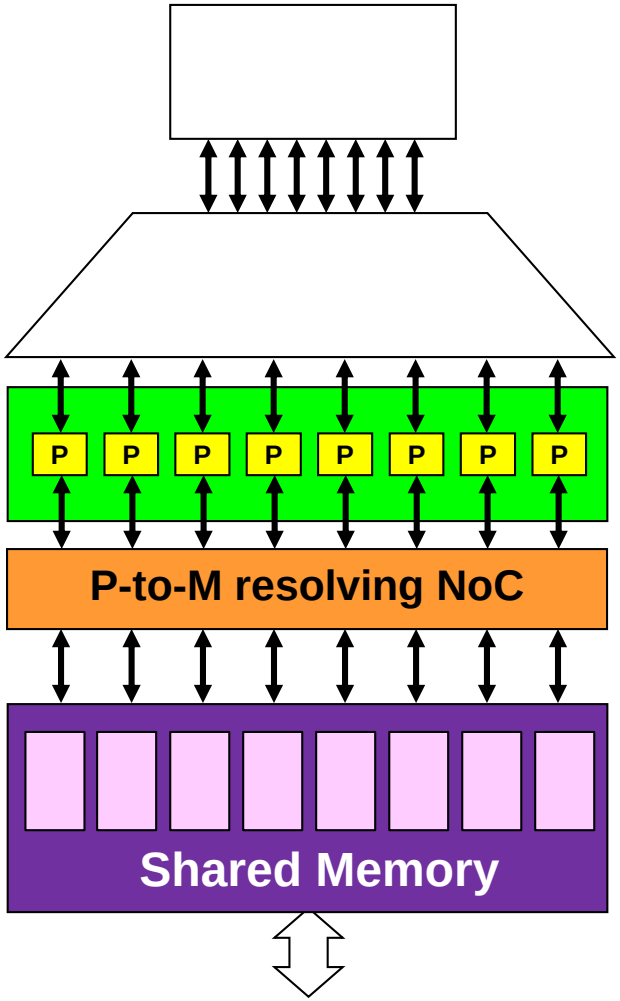
- Same-node Scalability
  - Easy to define high levels of parallelism
  - Scalable to more cores running slower at lower voltage
    - on same technology node
  - Example: same-node-scaling from N to 2N cores
    same-node-scaling of Vdd and f by α = 0.8, … ,0.5

	N cores, Vdd, f	2N cores, αVdd, αf	α = 0.8	α = 0.7	α = 0.6	α = 0.5
Perf	$P(N) = Nf$	$2N\alpha f = 2\alpha P(N)$	$P(N) \cdot 1.6$	$P(N) \cdot 1.4$	$P(N) \cdot 1.2$	$P(N)$
Time	$T(N) = \frac{W}{Nf}$	$\frac{W}{2N\alpha f} = \frac{T(N)}{2\alpha}$	$\frac{T(N)}{1.6}$	$\frac{T(N)}{1.4}$	$\frac{T(N)}{1.2}$	$T(N)$
Power	$PW(N) = NCV^2 f$	$2NC\alpha^2 V^2 \alpha f = 2\alpha^3 PW(N)$	$PW(N)$	$PW(N) \cdot 0.7$	$PW(N) \cdot 0.4$	$PW(N) \cdot 0.25$
Energy	$E(N) = P(N)T(N) = WCV^2$	$2\alpha^3 PW(N) \cdot T(N)/2\alpha = \alpha^2 E(N)$	$E(N) \cdot 0.64$	$E(N) \cdot 0.5$	$E(N) \cdot 0.36$	$E(N) \cdot 0.25$
Perf / Pwr	$PPR(N) = 1/CV^2$	$\frac{PPR(N)}{\alpha^2}$	$PPR(N) \cdot 1.5$	$PPR(N) \cdot 2$	$PPR(N) \cdot 2.8$	$PPR(N) \cdot 4$

Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Validation
- Plural programming examples
- ManyFlow for the Plural architecture

The Plural Architecture: Part I



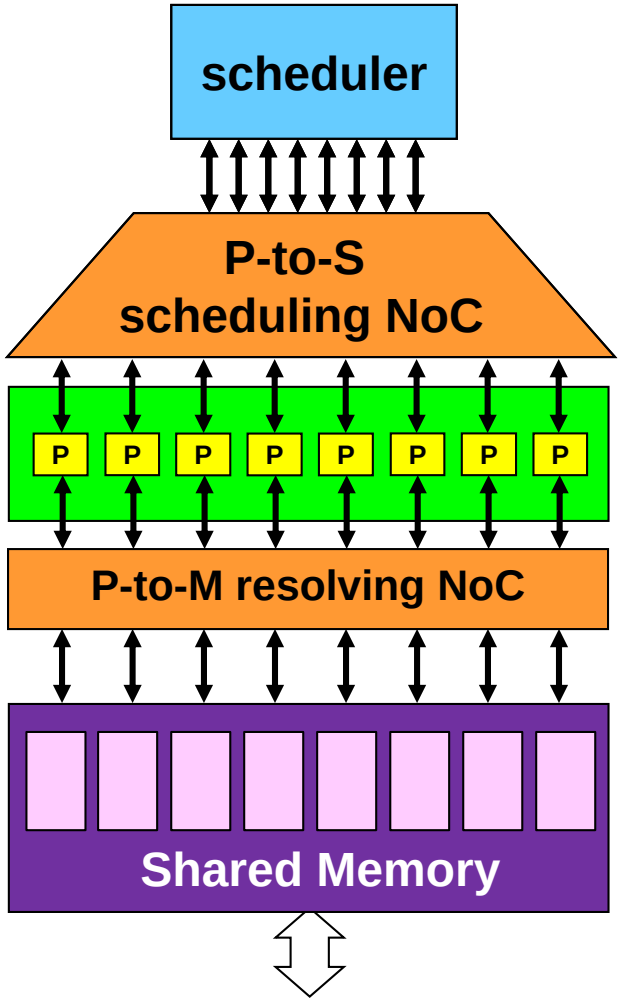
Many small processor cores
Small private memories (stack, L1 caches)

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



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

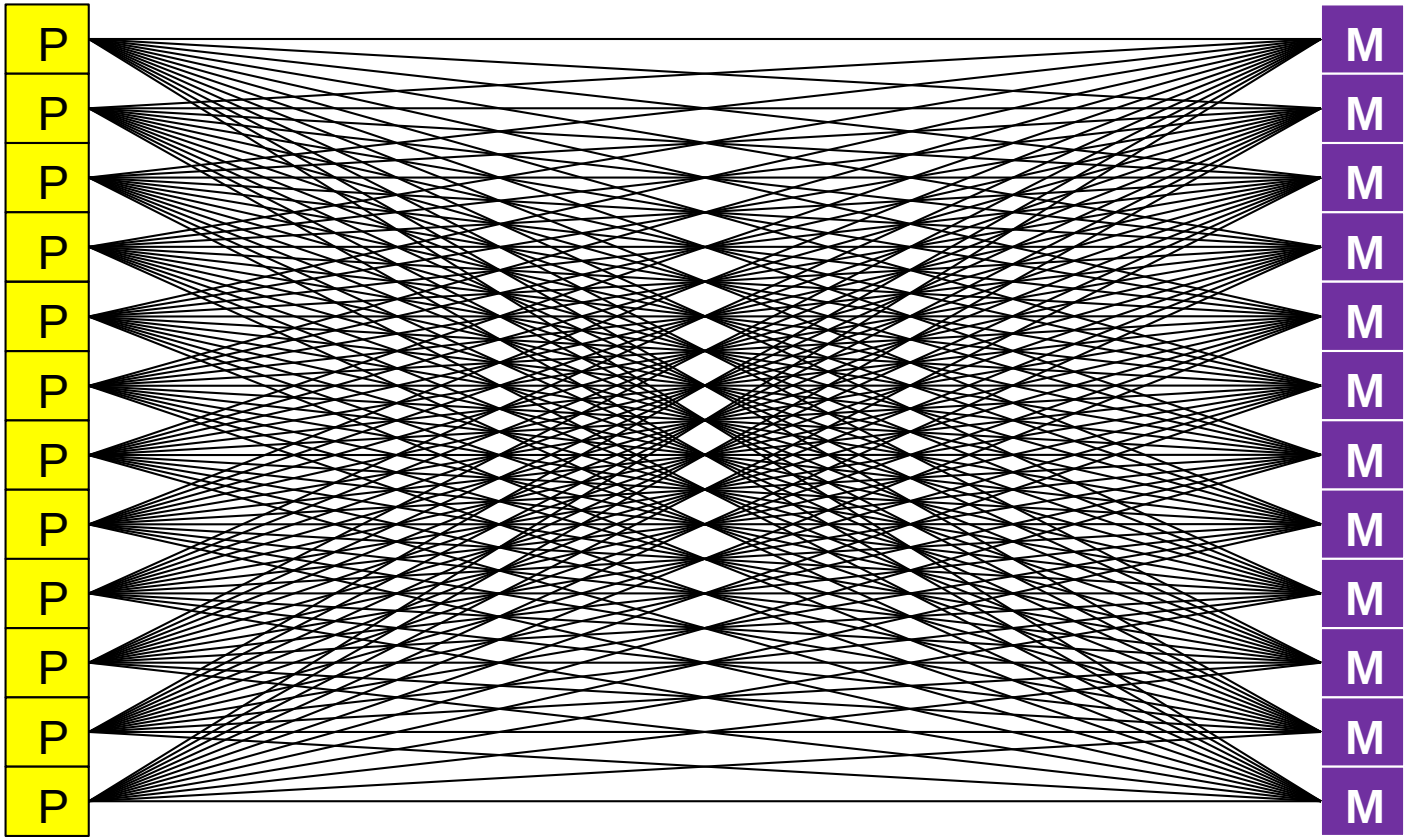
SHARED memory, many banks
~Equi-distant from cores (2-3 cycles)

“Anti-local” address interleaving
Negligible conflicts

Outline

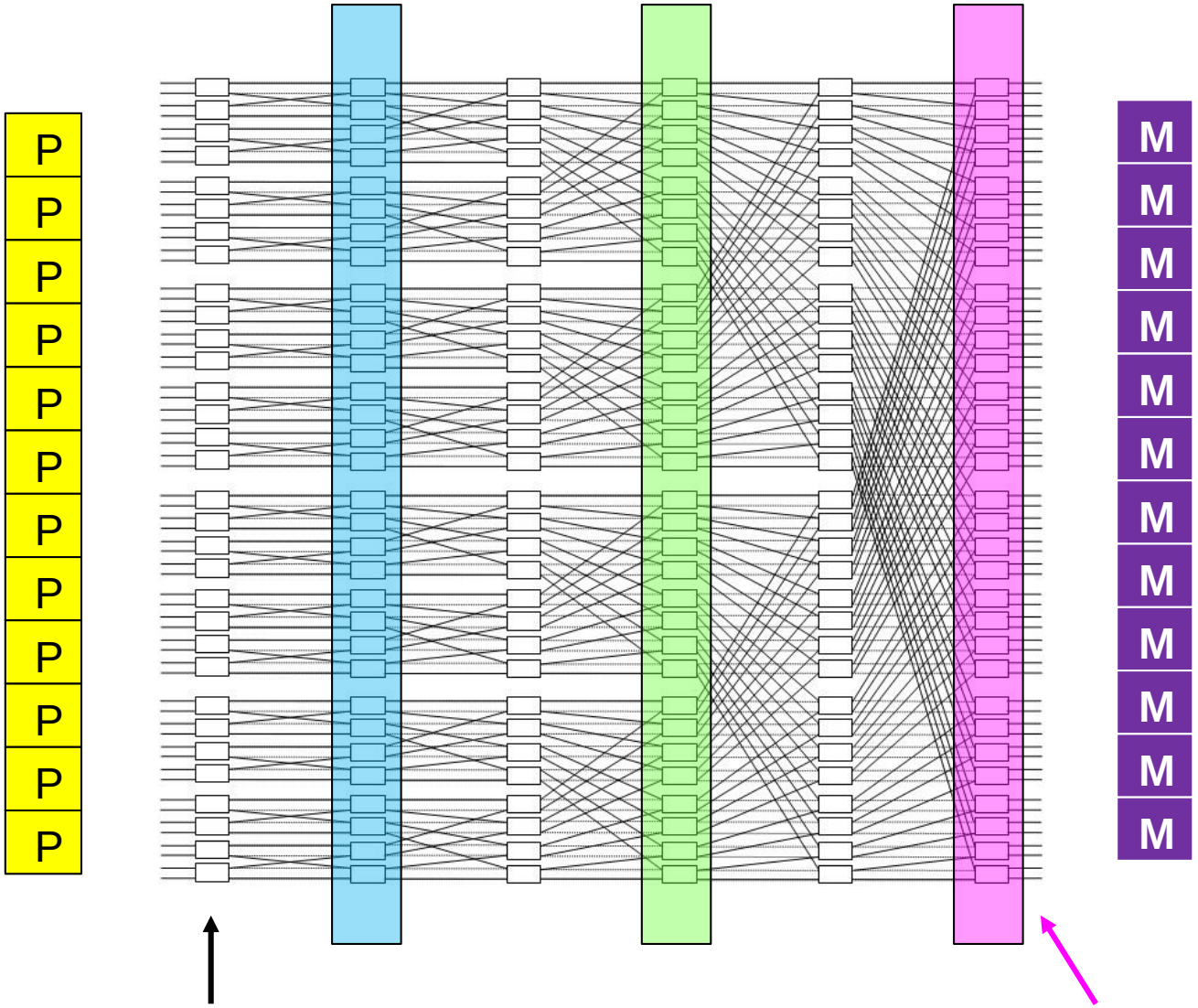
- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Validation
- Plural programming examples
- ManyFlow for the Plural architecture

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

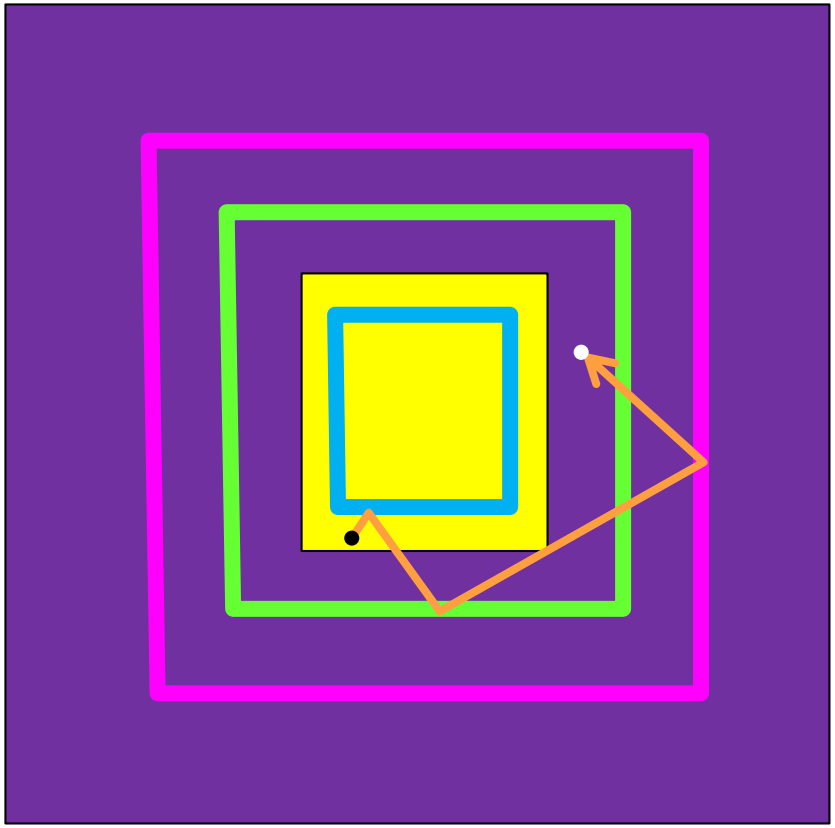


- Full bi-partite connectivity required
- But full cross-bar not required: minimize conflicts and allow stalls/re-starts

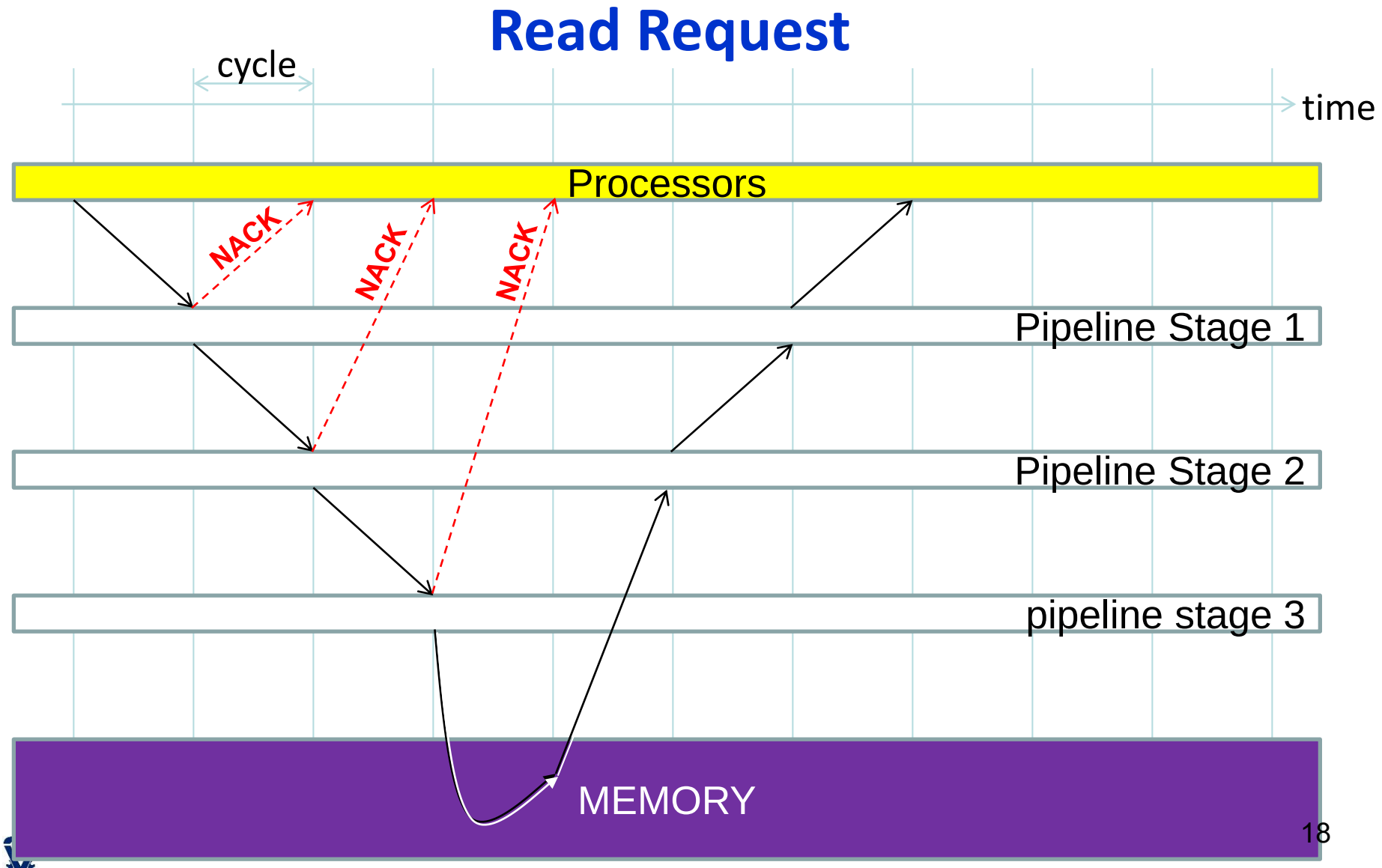
Logarithmic multistage interconnection network



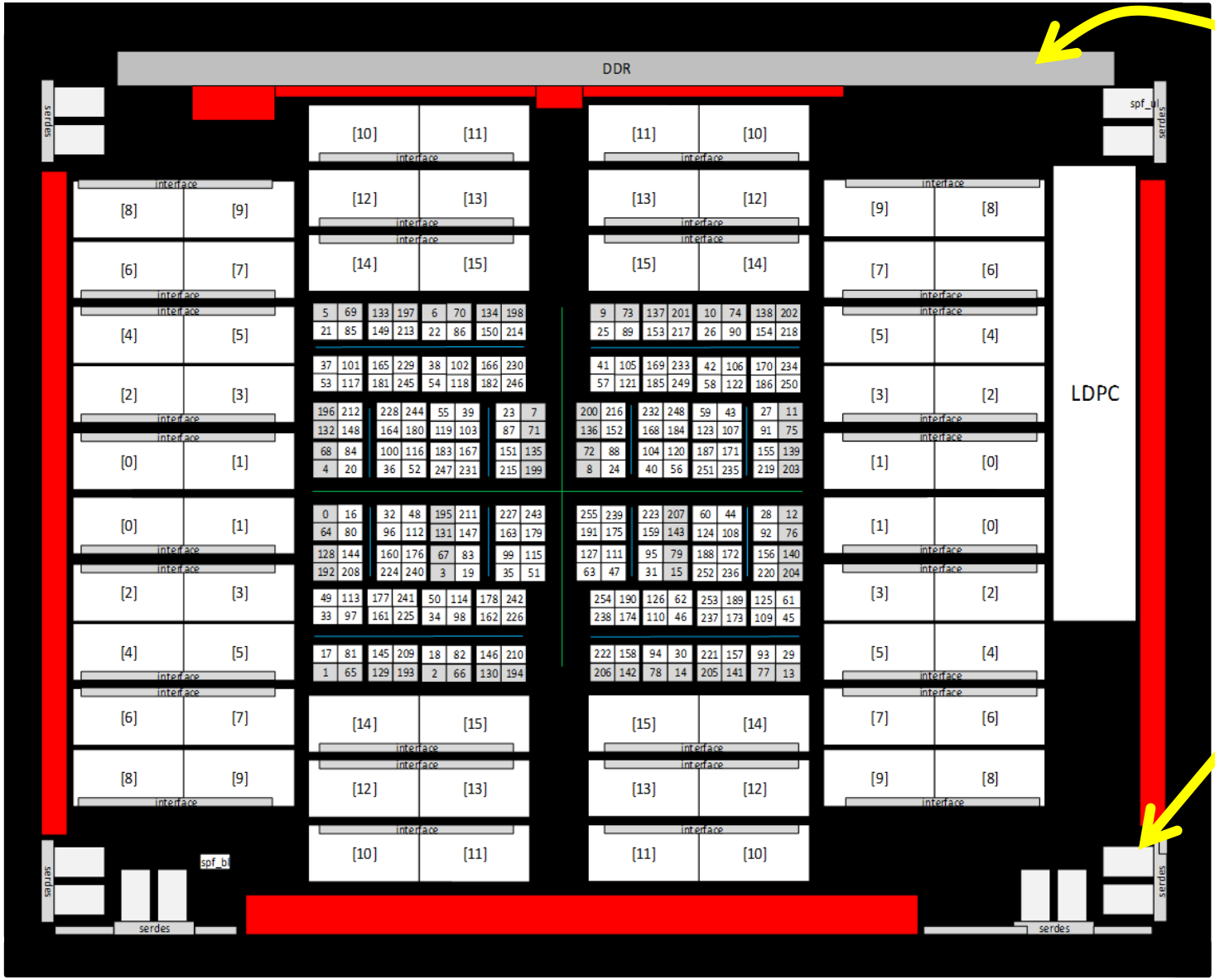
Floor plan and route to shared memory



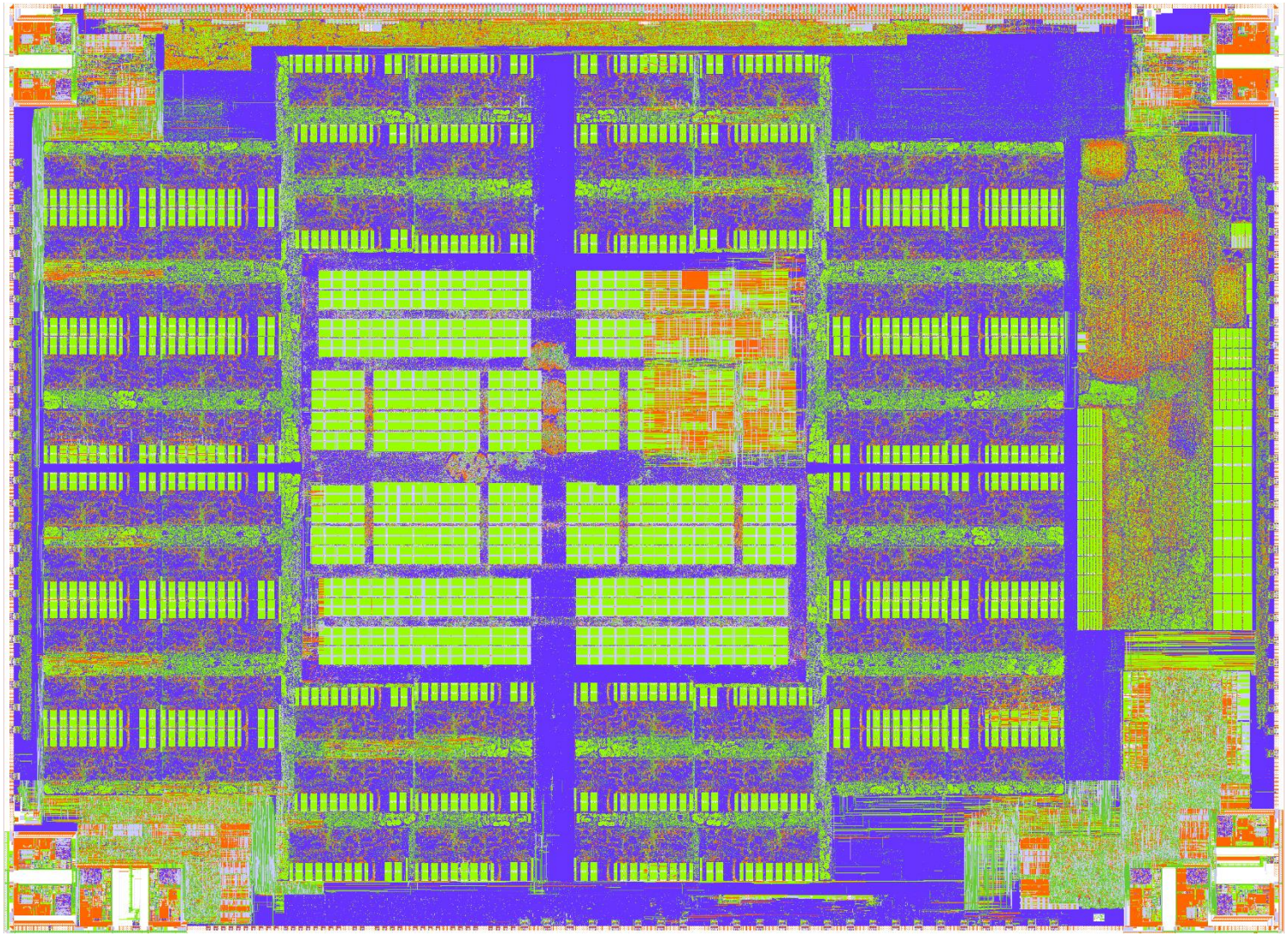
access sequence: **fixed latency** (when successful)



Floor plan: 64 DSP cores (24KB each) & 4MB shared memory take 320 mm² on 65nm



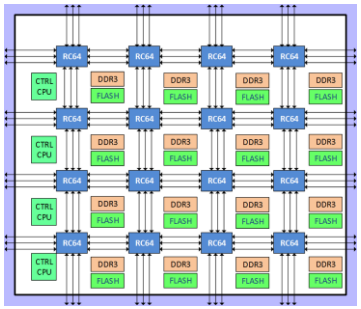
RC64



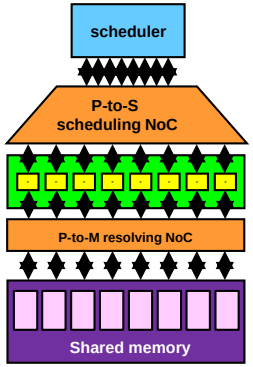
Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Validation
- Plural programming examples
- ManyFlow for the Plural architecture

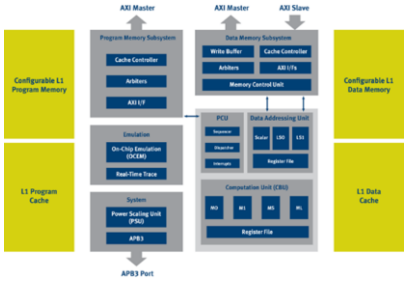
Three levels of “parallel” programming



- Multiple Plural chips
  - Distributed computing (message passing)
  - OR: shared memory



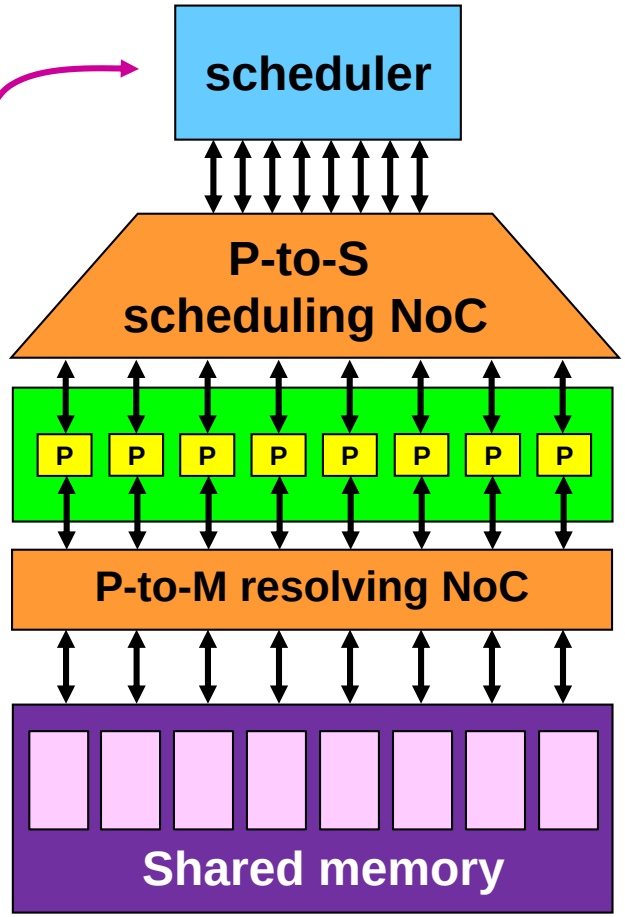
- The Plural chip
  - 64 cores, shared memory



- A high performance DSP core
  - VLIW + SIMD

The Plural task-oriented programming model

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



Task template:

regular
duplicable taskName **(** instance_id **)**

{

... instance_id **....**

// instance_id is instance number

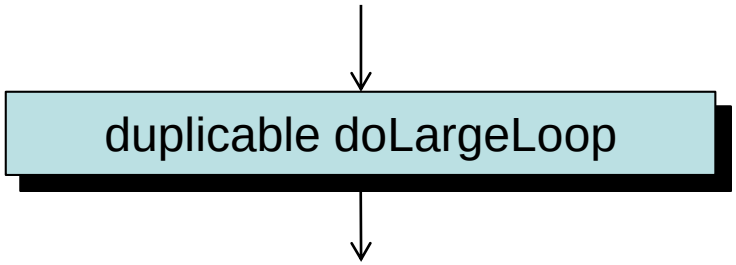
.....

}

Fine Grain Parallelization

Convert (independent) loop iterations

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

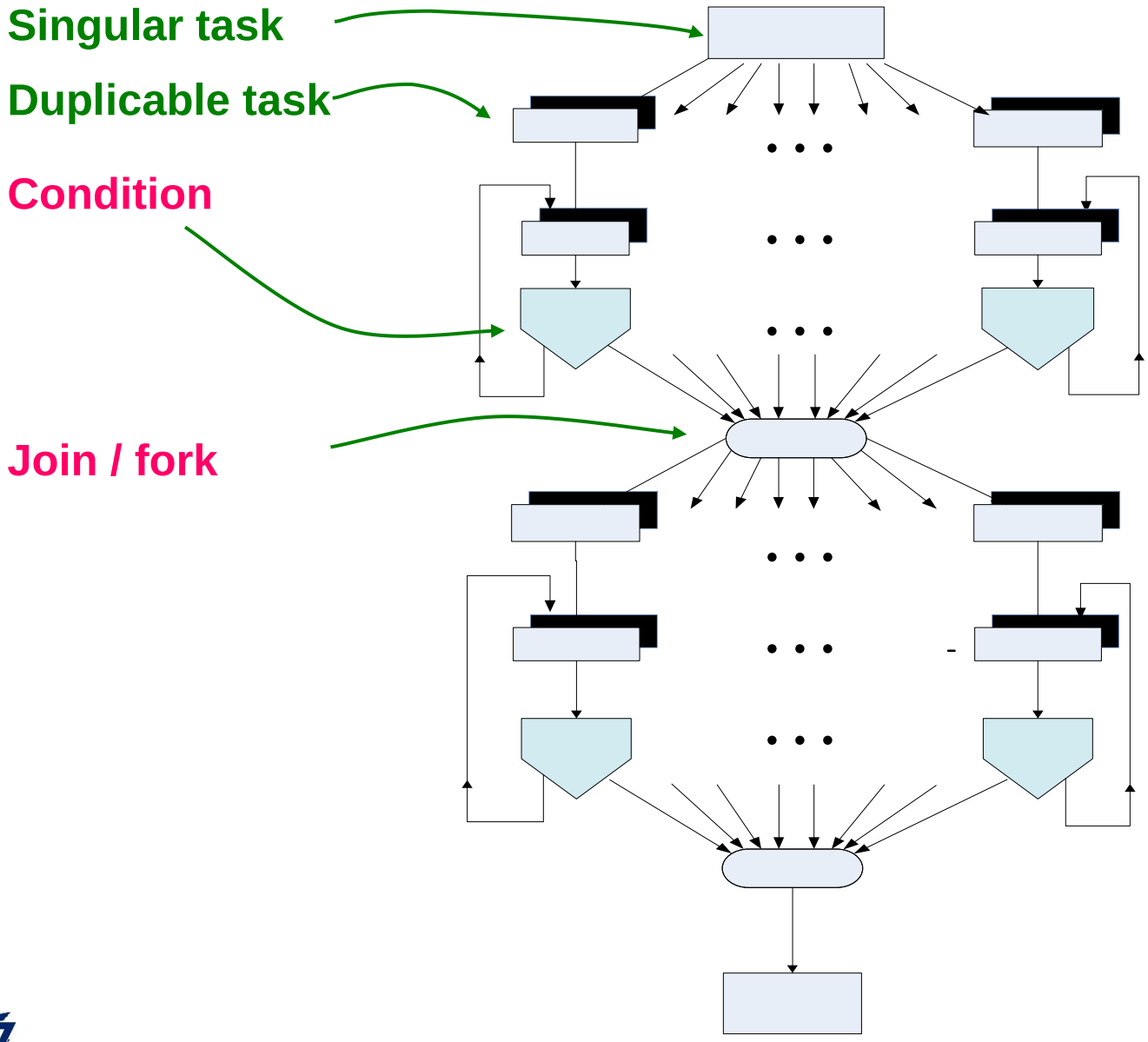


into parallel tasks

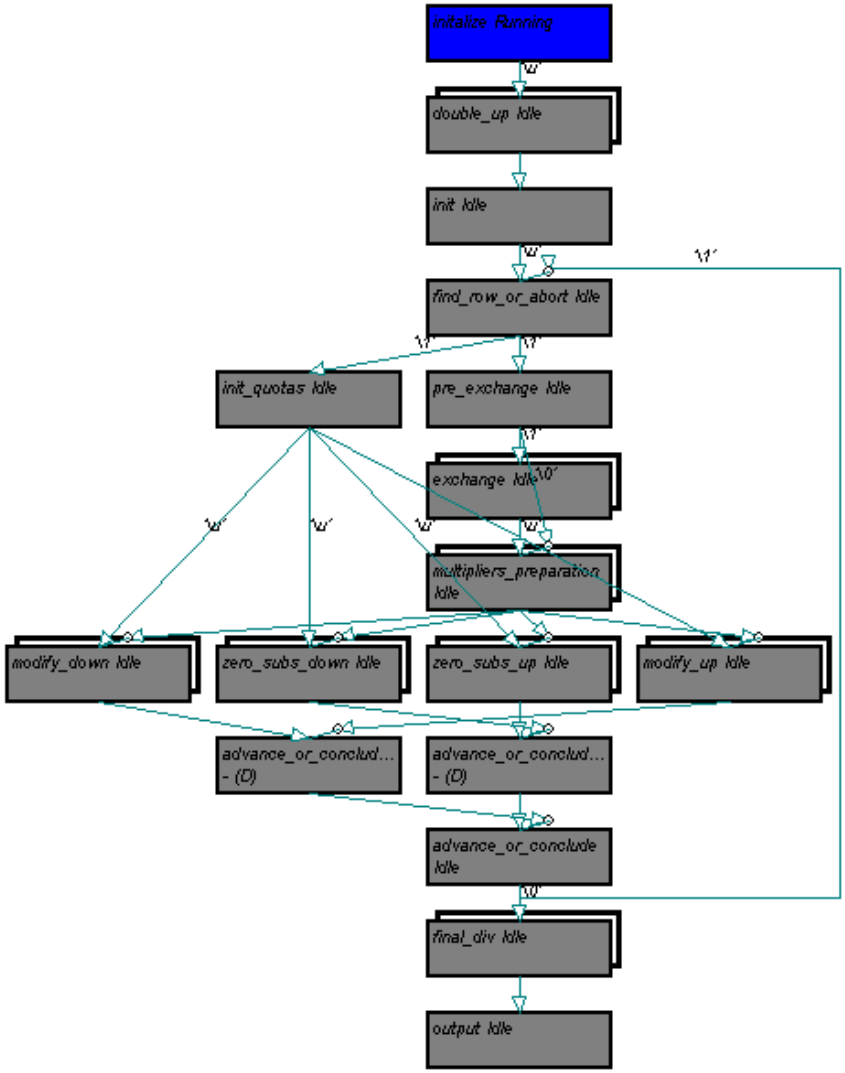
```
set_task_quota(doLargeLoop, 10000)
```

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

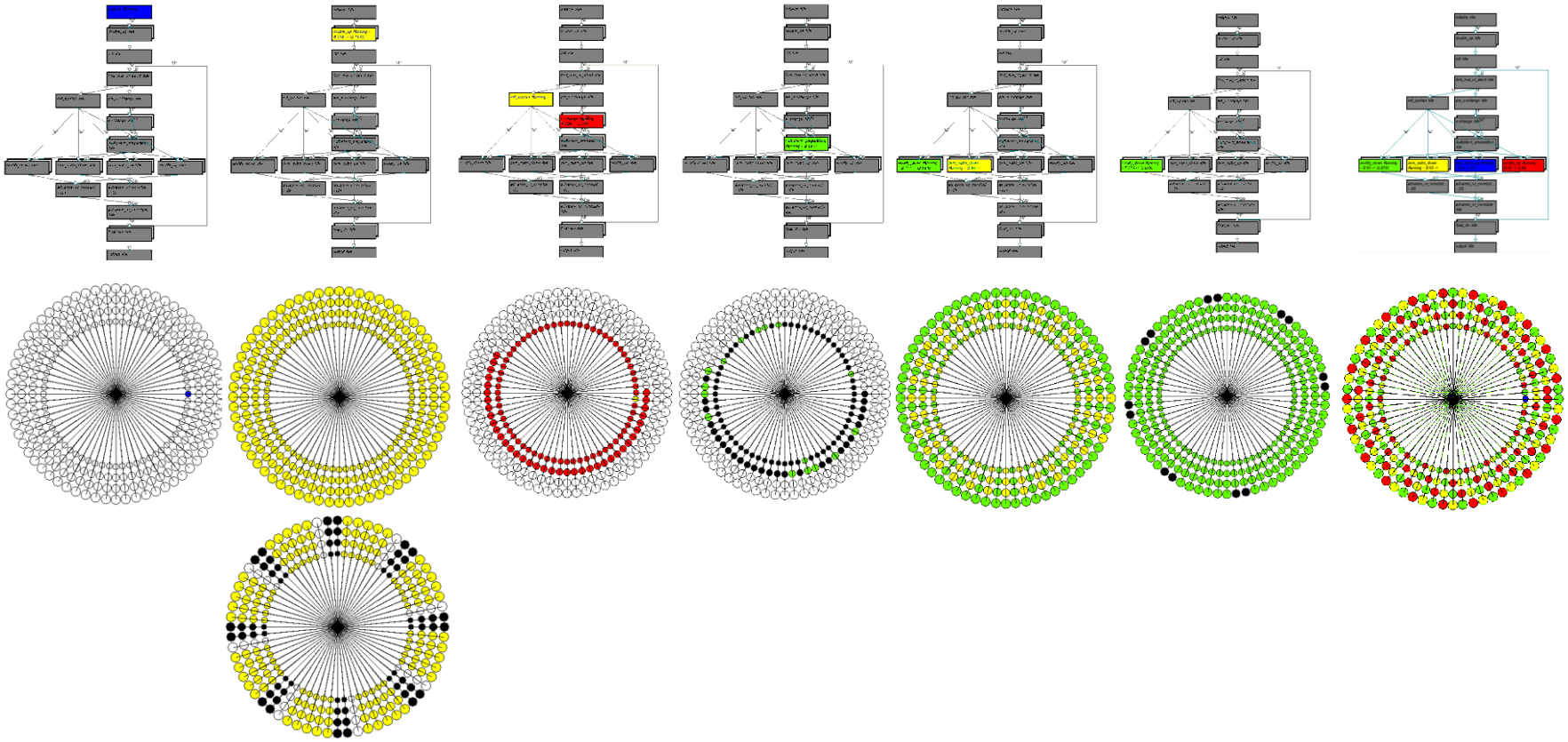
Task graph example (2D FFT)



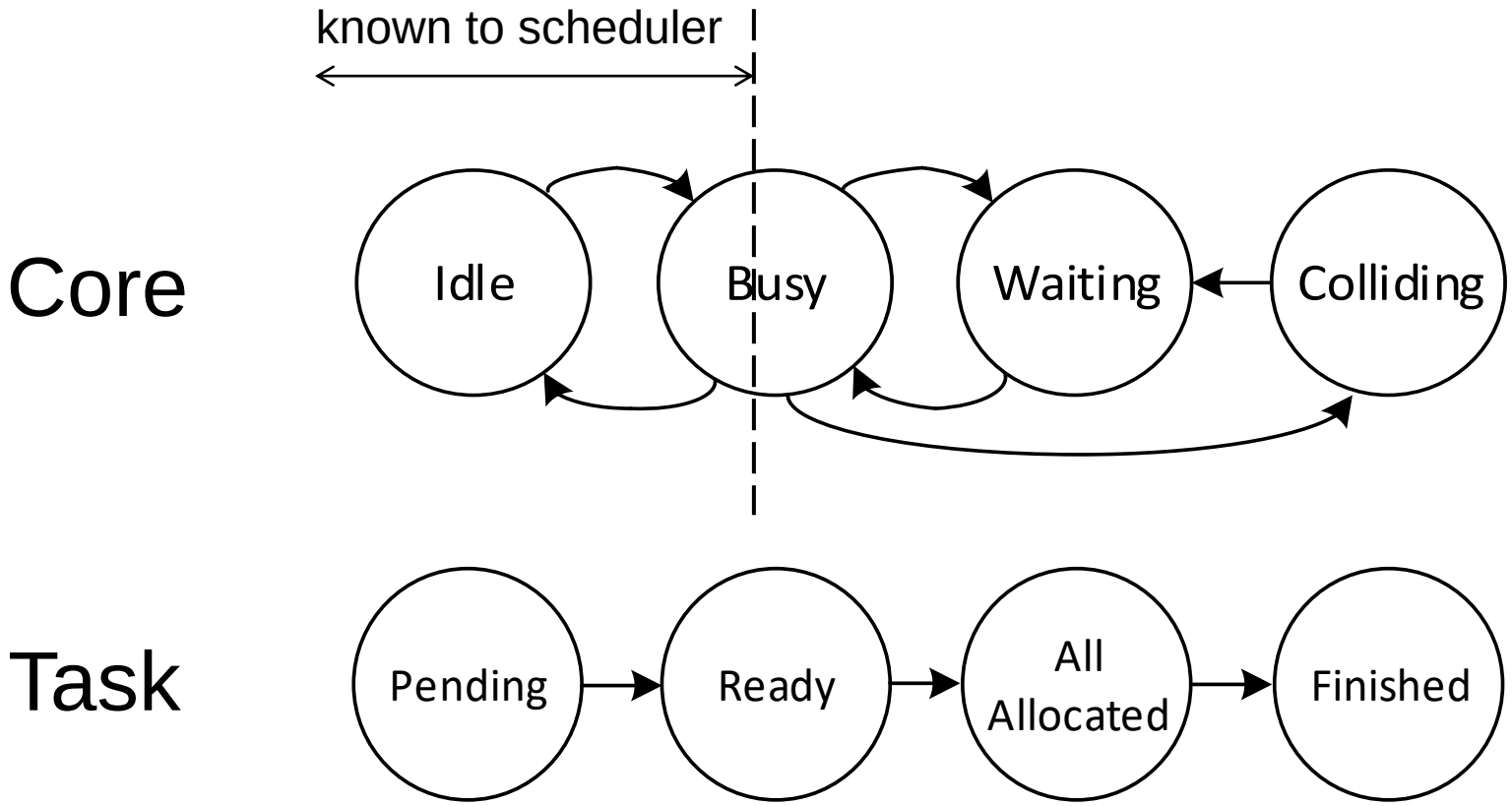
Another task graph (linear solver)



Linear Solver: Simulation snap-shots



Cores and Tasks



Hardware Scheduler: Under the hood

core #

	state	task #	Instance #	…
0				
1				
2				

⋮

task #

	total instances	dependencies	state	# already allocated	…
0					
1					
2					

⋮

task graph

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 graph
  - 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

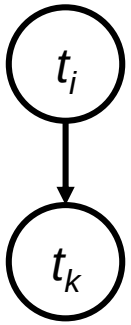
- 2 types of tasks:
  - Regular task (Executes once)
  - Duplicable task
    - Many independent concurrent instances
    - Identified/dispatch: entry point, instance number
- Conditions on tasks checked by scheduler
- Tasks are not functions
  - No arguments, no inputs, no outputs
  - Share data only 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 P-to-M NoC

Outline

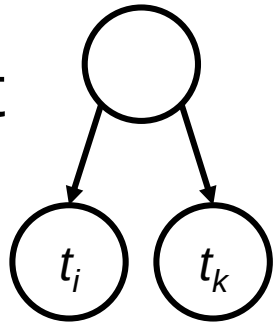
- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Validation
- Plural programming examples
- ManyFlow for the Plural architecture

Concurrency in shared memory manycore

- Non-preemptive execution
- Task graph defines tasks and dependencies
- Task graph executed by scheduler
- ∃ path $t_i \rightarrow t_k \Rightarrow t_i$, t_k are non-concurrent
  - Execution of t_i must complete before start of execution of t_k



- Otherwise, t_i, t_k are concurrent
  May execute simultaneously or at any order



- Task graph must be decomposable into *concurrent sets*

(verifiable) Shared Memory Access Rules

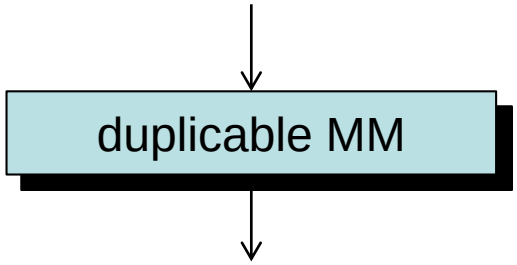
1. Predictable Addressing
   - Shared memory address derivable at compile time
   - No data-dependent shared memory addresses
   - Predictable *malloc()* address
2. Exclusive Write (EW)
   - Task t_i writes into A
     $\Rightarrow$ *compiler can verify that*
     no concurrent task t_k allowed to access A
     (neither read nor write)
3. Concurrent Read (CR)
   - *Compiler can verify that*
     Concurrent tasks may read from same address
     but none of them may write into it

Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Validation
- Plural programming examples
- ManyFlow for the Plural architecture

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(unsigned int 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
}
```



Algorithms and their performance

- Matrix multiplication
  - RTD by Ramon Chips
- Image processing
  - RTD by TU Braunschweig, DSI, DLR, ELBIT/ELOP
    - Hyperspectral imaging
    - SAR imaging
- Modem
  - RTD by Ramon Chips

Matrix Multiplication on RC64

$$C = A \times B$$

$$C_{i,j} = \sum_{m} A_{i,m} \times B_{m,j}$$

- Each result element $C_{i,j}$ is computed by a task
  - For N×N matrices, N×N tasks (regardless of #cores)
- Later, each task computes an entire row of C
  - Only N tasks

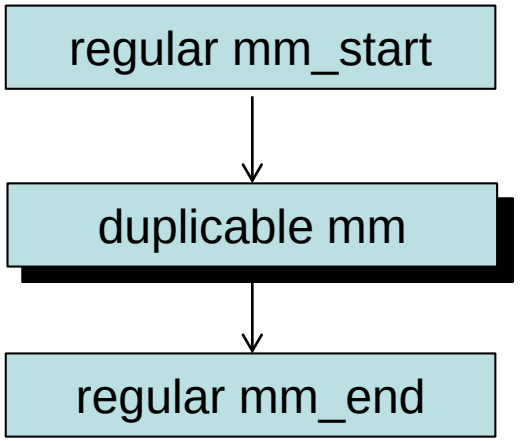
Matrix Multiplication on RC64

CODE (plain C)

```
#define MSIZE 100
float  A[MSIZE][MSIZE],      B[MSIZE][MSIZE],
       C[MSIZE][MSIZE];
int  mm_start()                              REGULAR
{    int i,j;
     for (i=0; i< MSIZE; i++)
          for (j=0; j< MSIZE; j++)
               { A[i][j] = 13;  B[i][j] = 9;   }
}
void  mm (unsigned int id)          DUPLICABLE
{    int  i,j,m;  float sum = 0;
     i = id % MSIZE;      j = id / MSIZE;
     for (m=0; m < MSIZE; m++)
          sum += A[i][m]*B[m][j];
     C[i][j]=sum;
}
int  mm_end ()                               REGULAR
{  printf("finished mm\n");  }
```

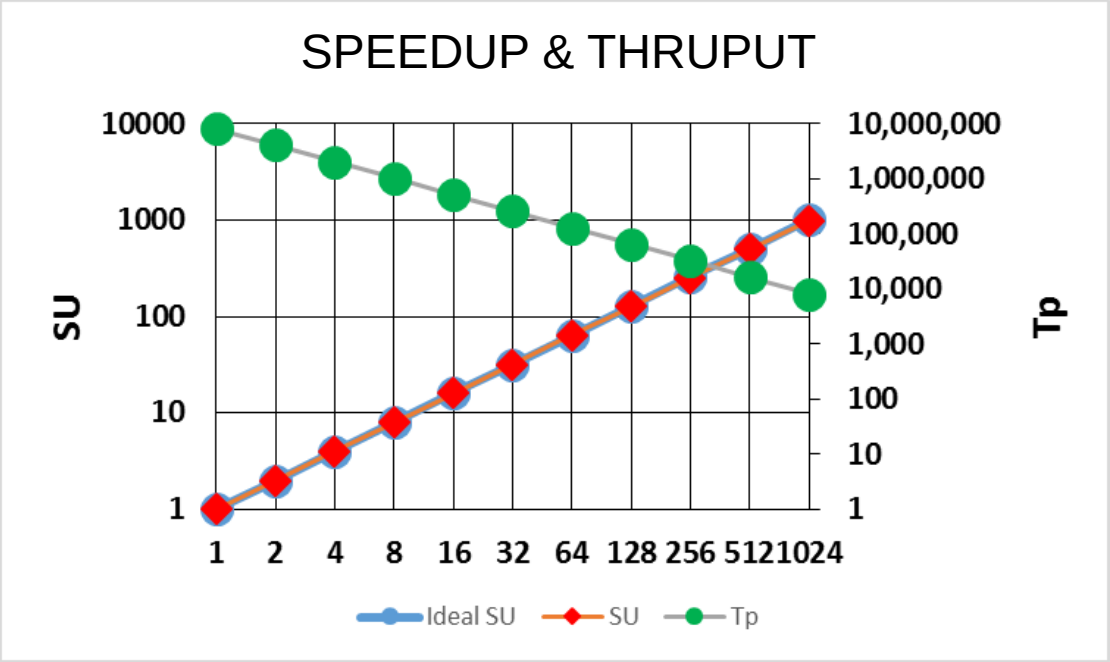
TASK GRAPH

```
#define MSIZE 100
#define MMSIZE 10000

regular       mm_start()
duplicable  mm(mm_start) MMSIZE
regular      mm_end(mm)
```



Matrix Multiplication on RC64

P	Tp	SU	Eff
1	8,190,021	1	1.00
2	4,095,021	2	1.00
4	2,047,521	4	1.00
8	1,023,771	8	1.00
16	511,896	16	1.00
32	256,368	32	1.00
64	128,604	64	1.00
128	64,722	127	0.99
256	32,781	250	0.98
512	16,401	499	0.98
1024	8,211	997	0.97



- Why is SU(1024) still less than 1024?

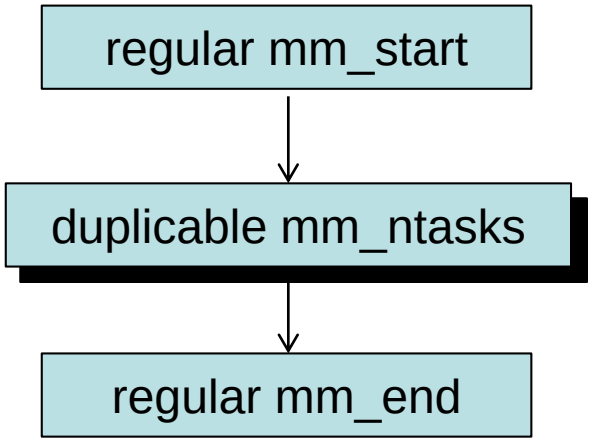
Matrix Multiplication using only N tasks

CODE (plain C)

```
#define MSIZE 100
float  A[MSIZE][MSIZE],   B[MSIZE][MSIZE],
       C[MSIZE][MSIZE];
int      mm_start ()                    REGULAR
{    int i,j;
     for (i=0; i< MSIZE; i++)
          for (j=0; j< MSIZE; j++)
               { A[i][j] = 13;  B[i][j] = 9;   }
}
void mm_ntasks (unsigned int id)          DUP
{    int  m, k; float sum = 0;
     for (k=0; k<MSIZE; k++) {
          sum = 0;
          for (m=0; m < MSIZE; m++)
               sum += A[id][m]*B[m][k];
          C[id][k]=sum;
     }
}
int     mm_end ()                         REGULAR
{    printf("finished mm with N tasks\n");  }
```

TASK GRAPH

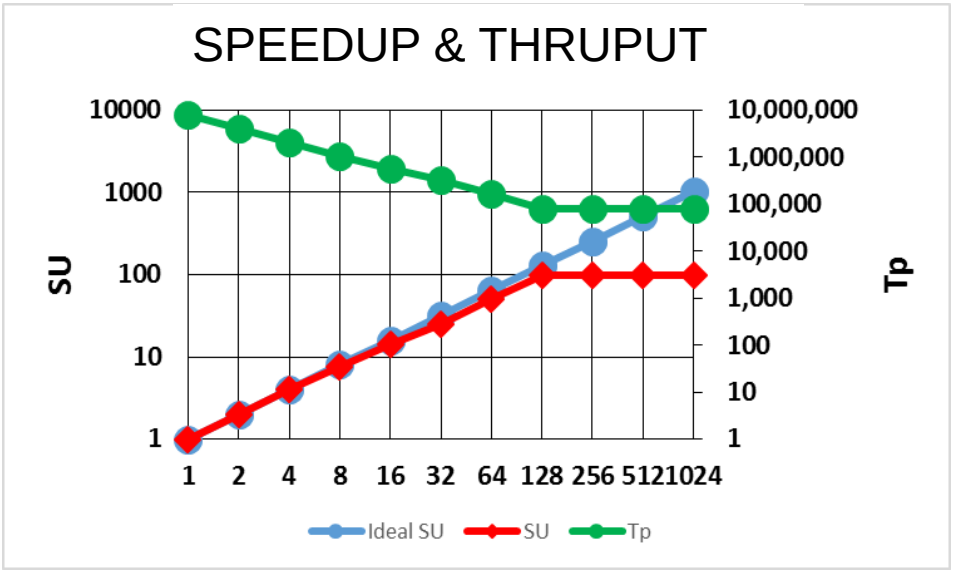
```
#define MSIZE 100

regular mm_start()
duplicable  mm_ntasks(mm_start) MSIZE
regular  mm_end(mm_ntasks)
```

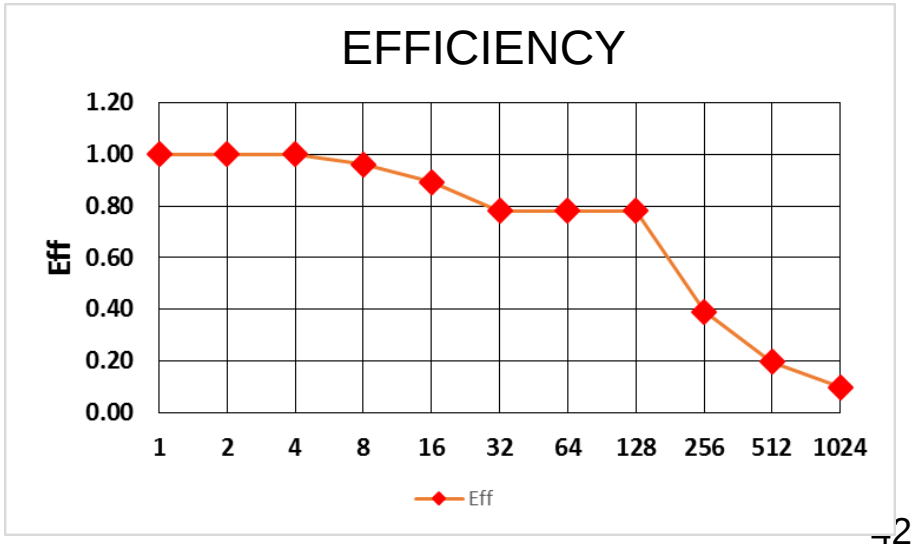


Matrix Multiplication using only N tasks

P	Tp	SU	Eff
1	8,140,021	1	1.00
2	4,070,021	2	1.00
4	2,035,021	4	1.00
8	1,058,221	8	0.96
16	569,821	14	0.89
32	325,621	25	0.78
64	162,821	50	0.78
128	81,421	100	0.78
256	81,421	100	0.39
512	81,421	100	0.20
1024	81,421	100	0.10

- What went wrong ?

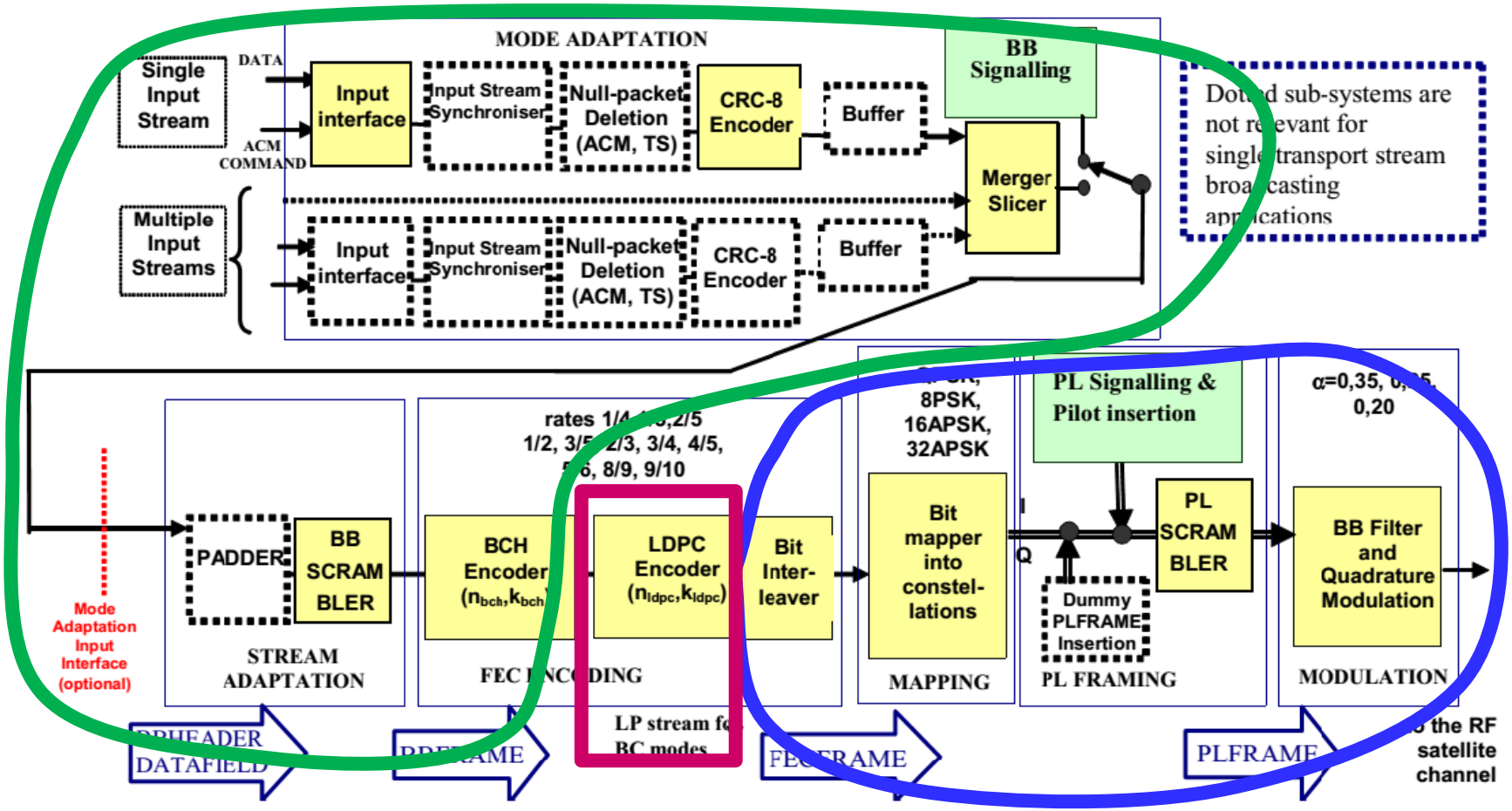




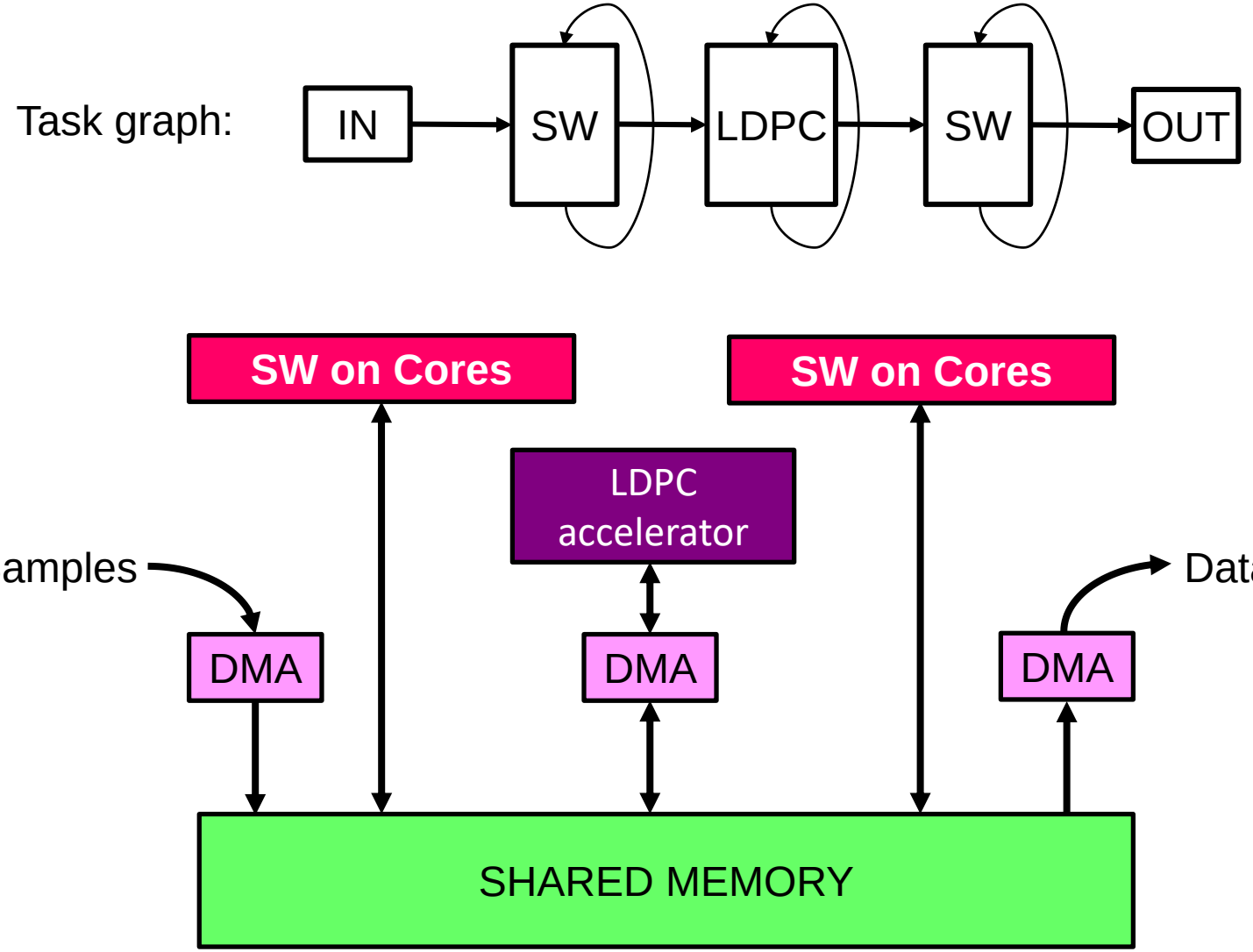
DVB-S2x MODEM on RC64

- Developed at Ramon Chips
- DVB-S2 is critical to modern communication satellites

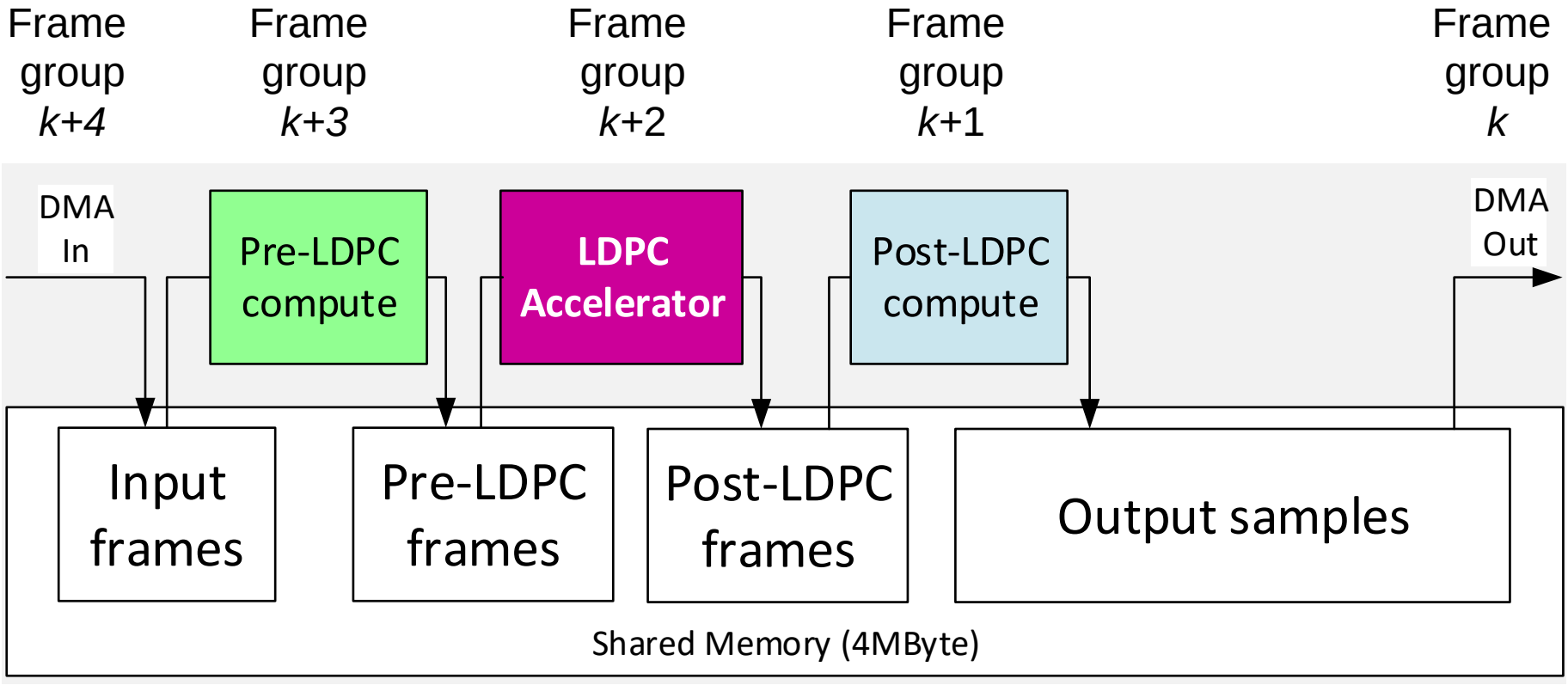
DVB-S2 transmit—three stages



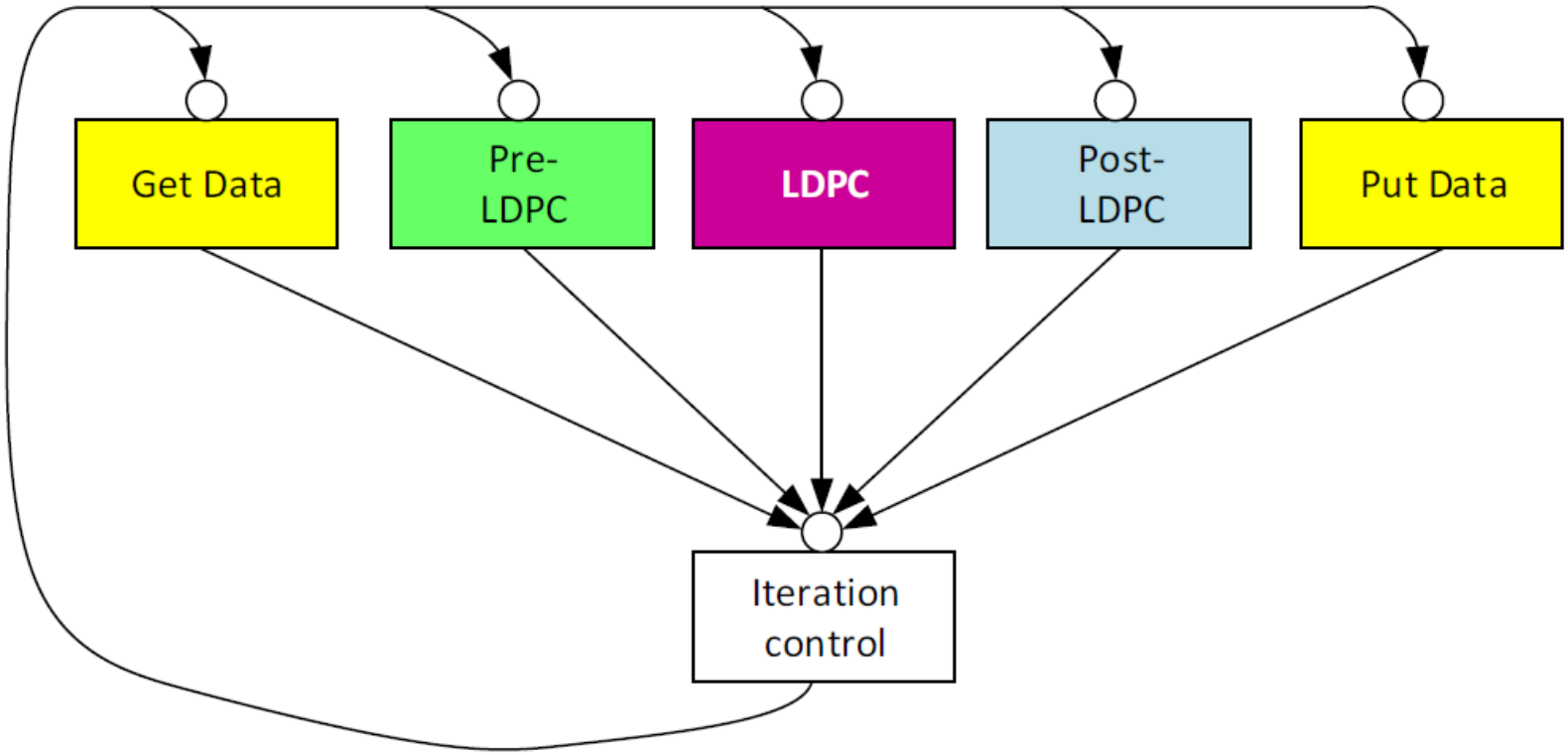
Many-Flow: DVB-S2x Modem



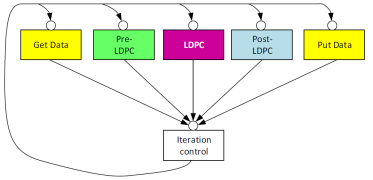
Many-Flow: software pipeline

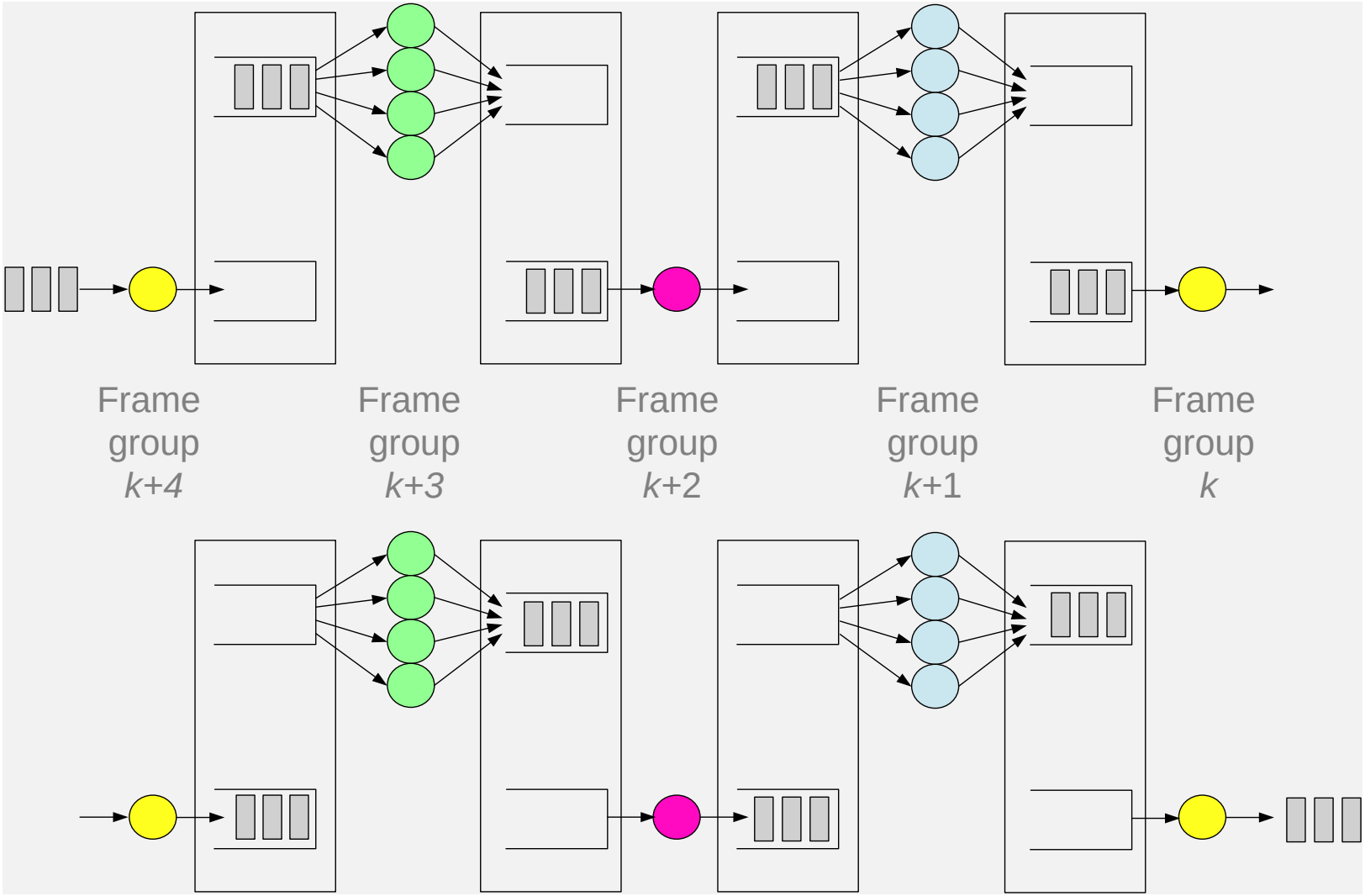


DVB-S2x task graph

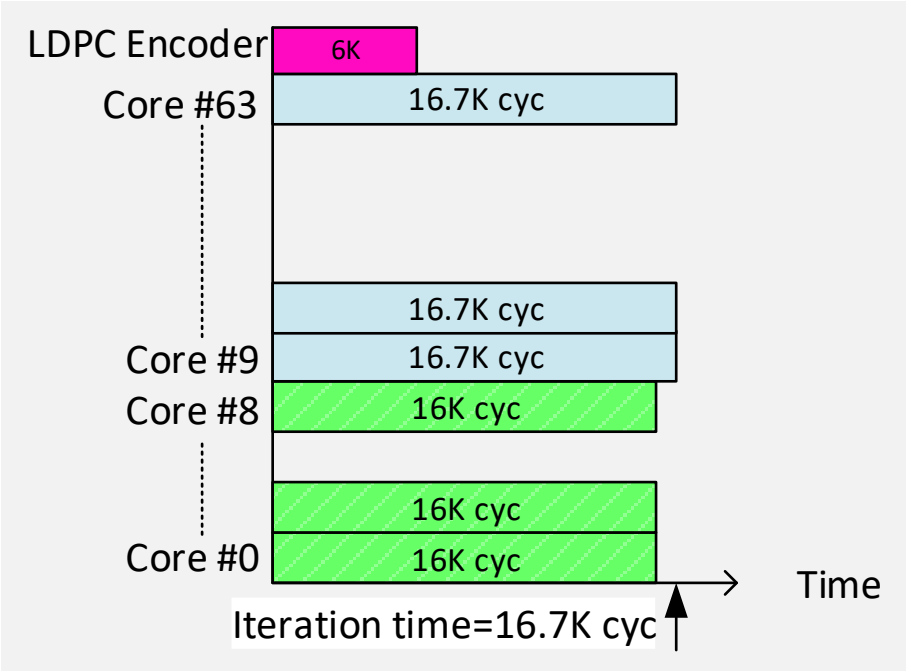


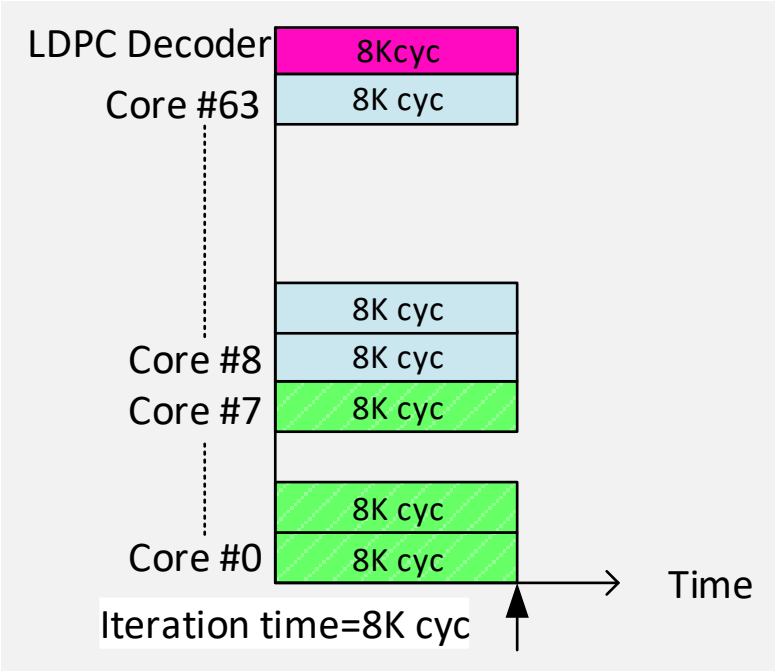
Many-Flow: Double buffering





Performance of DVB-S2 on RC64





Tx 2.3 Gb/s

Rx 1.0 Gb/s

SW development flow: MATLAB to RC64

1. MATLAB float, unrestricted (also SIMULINK)
2. MATLAB float, restricted memory size and I/O
3. MATLAB fixed point 16-bit
   - Insert DSP library functions
   - Create Golden model
4. Convert to C
   - Sequential code on laptop
   - Bit-exact comparison to Golden model
5. Parallelize for RC64 many-core. Create task graph
   - Simulate using “many-task emulator” on laptop
   - Bit-exact comparison to Golden model
6. Transfer to RC64
   - Execute on hardware, or
   - Simulate using cycle-accurate RC64 simulator
   - Bit-exact comparison to Golden model

What if parallelism is limited ?

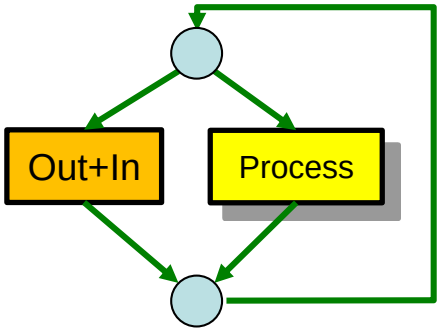
- So far, examples were highly parallel
- What if algorithm CANNOT be parallelized?
  - Execute many (serial) instances in parallel
  - Each instance on different data
- What if algorithm is mixture of serial / parallel segments?
  - Use ManyFlow

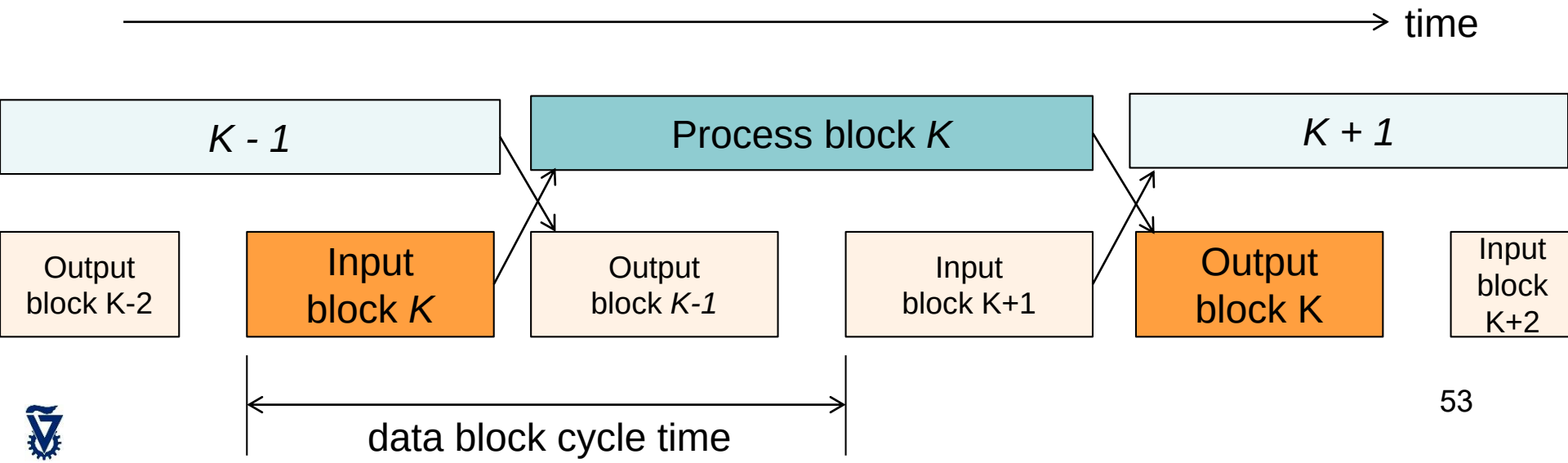
Outline

- Motivation: Programming model
- Plural architecture
- Plural implementation
- Plural programming model
- Validation
- Plural programming examples
- ManyFlow for the Plural architecture

Stream Processing

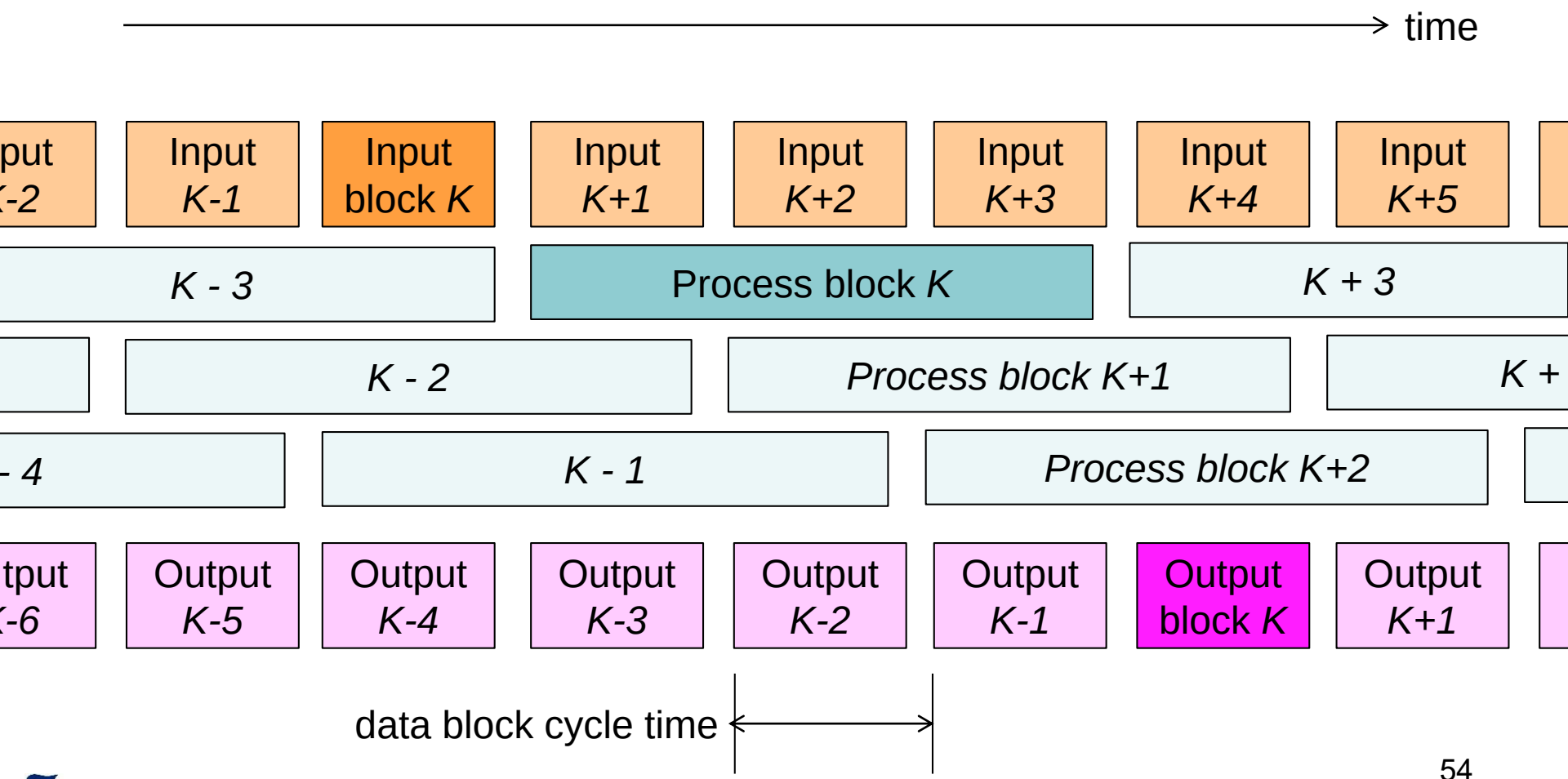
- Data arrives in a sequence of blocks
- In parallel:
  - Process current block (*K*)
  - Output results of previous block (*K-1*)
  - Input next block (*K+1*)





PIPELINED stream processing

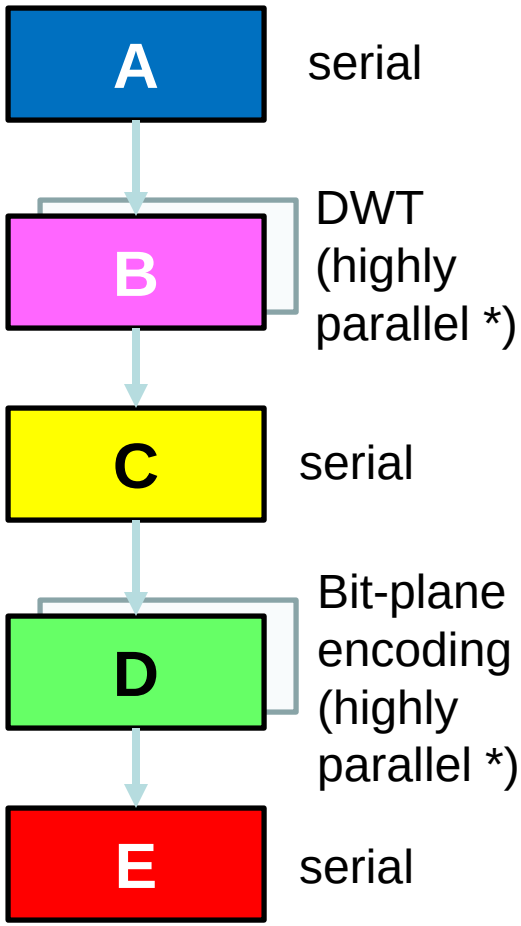
- For faster data & slower processing



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



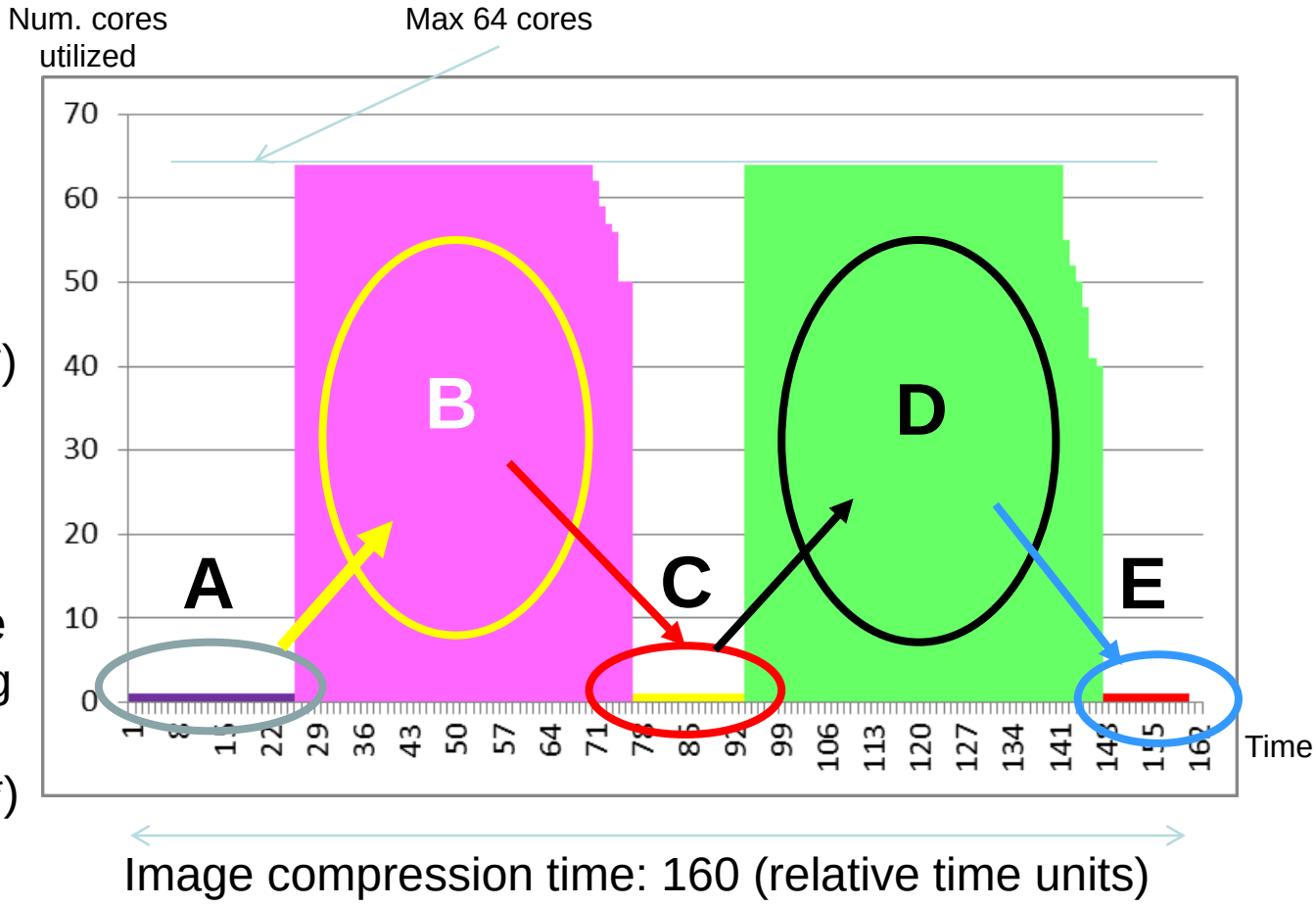
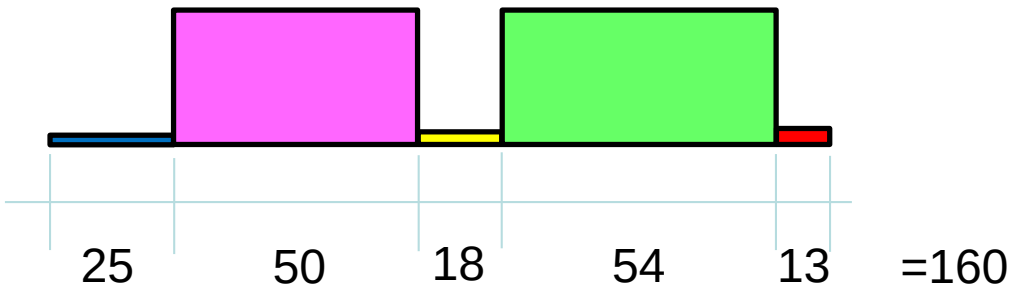


Image compression time: 160 (relative time units)

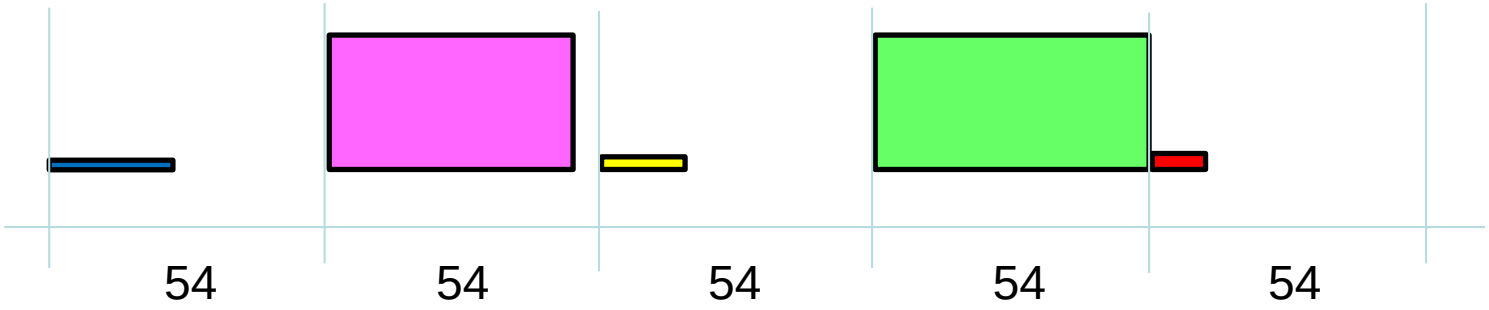
Low utilization: only 65%

Speed it up with a pipeline?

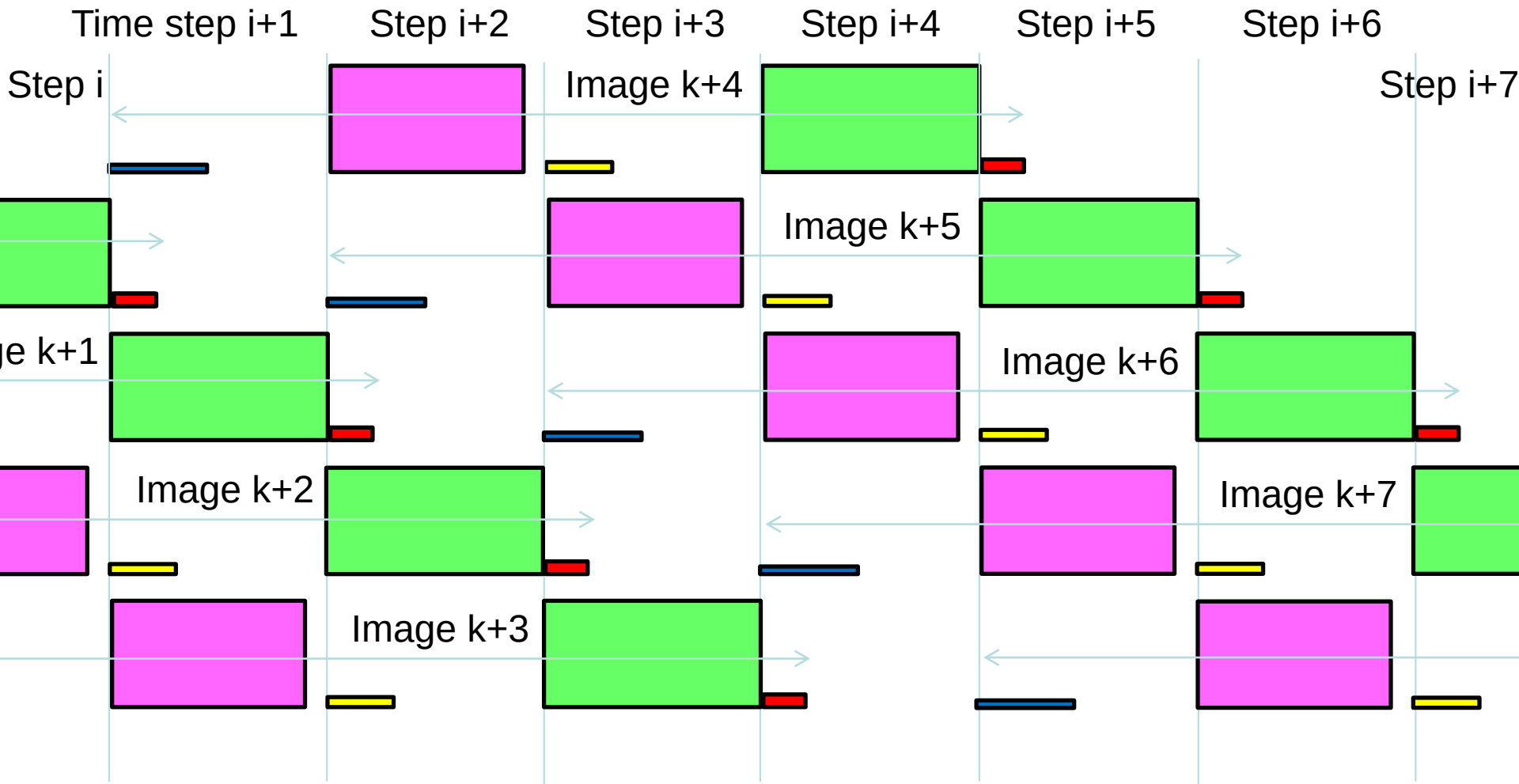
Sequential



Pipeline



Hardware-like Pipeline

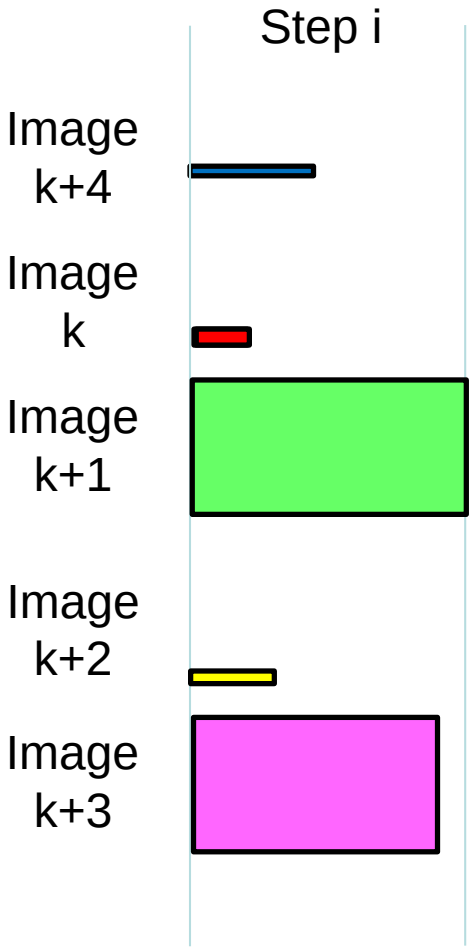


Needs 5 stages: two with 64 cores each, three with one core each (total 131 cores)

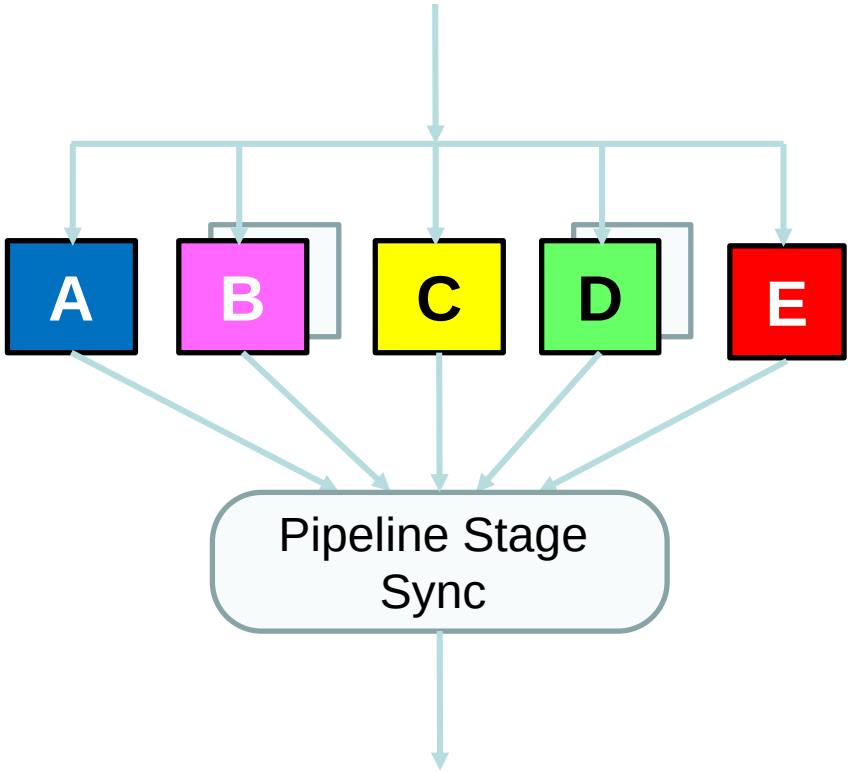
If only 64 cores, time / step = 64x2 + 25 = 153 (how ? What is the utilization?)

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

Parallel / pipelined “ManyFlow”

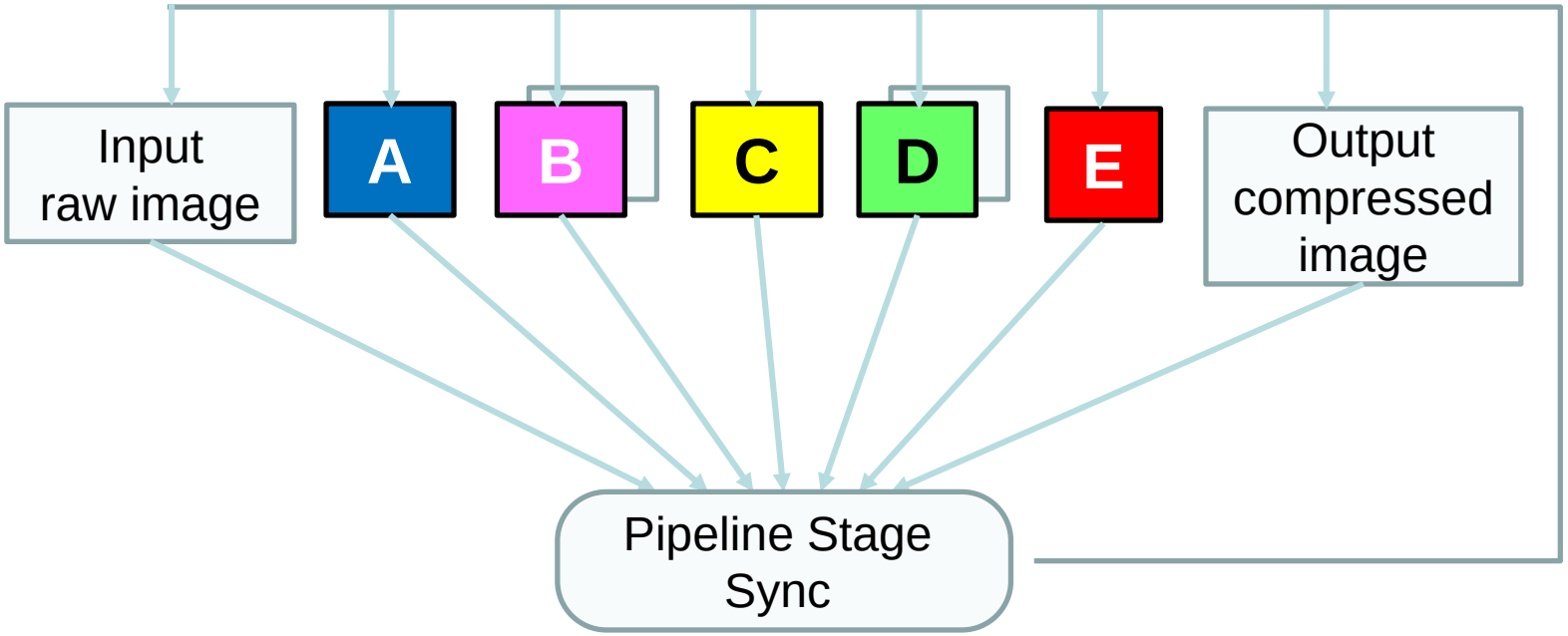


All 5 stages are independent (order does not matter)
→ Can run concurrently
→ Scheduler will dispatch most efficiently



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

Parallel / pipelined “ManyFlow”

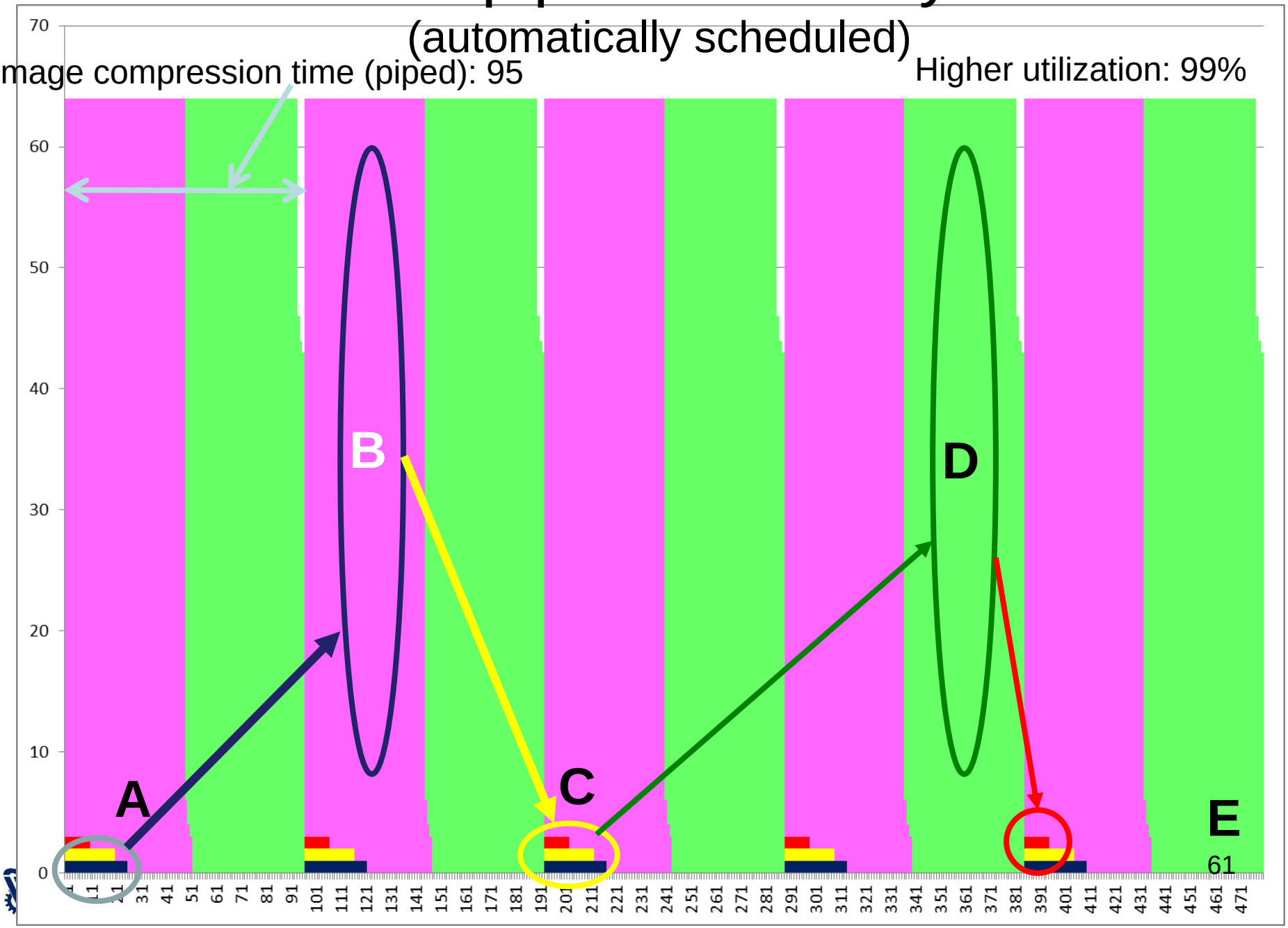


Task graph for continuous execution
Includes two more pipe stages, for I/O of images

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

Parallel / pipelined “ManyFlow”

(automatically scheduled)



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 graph

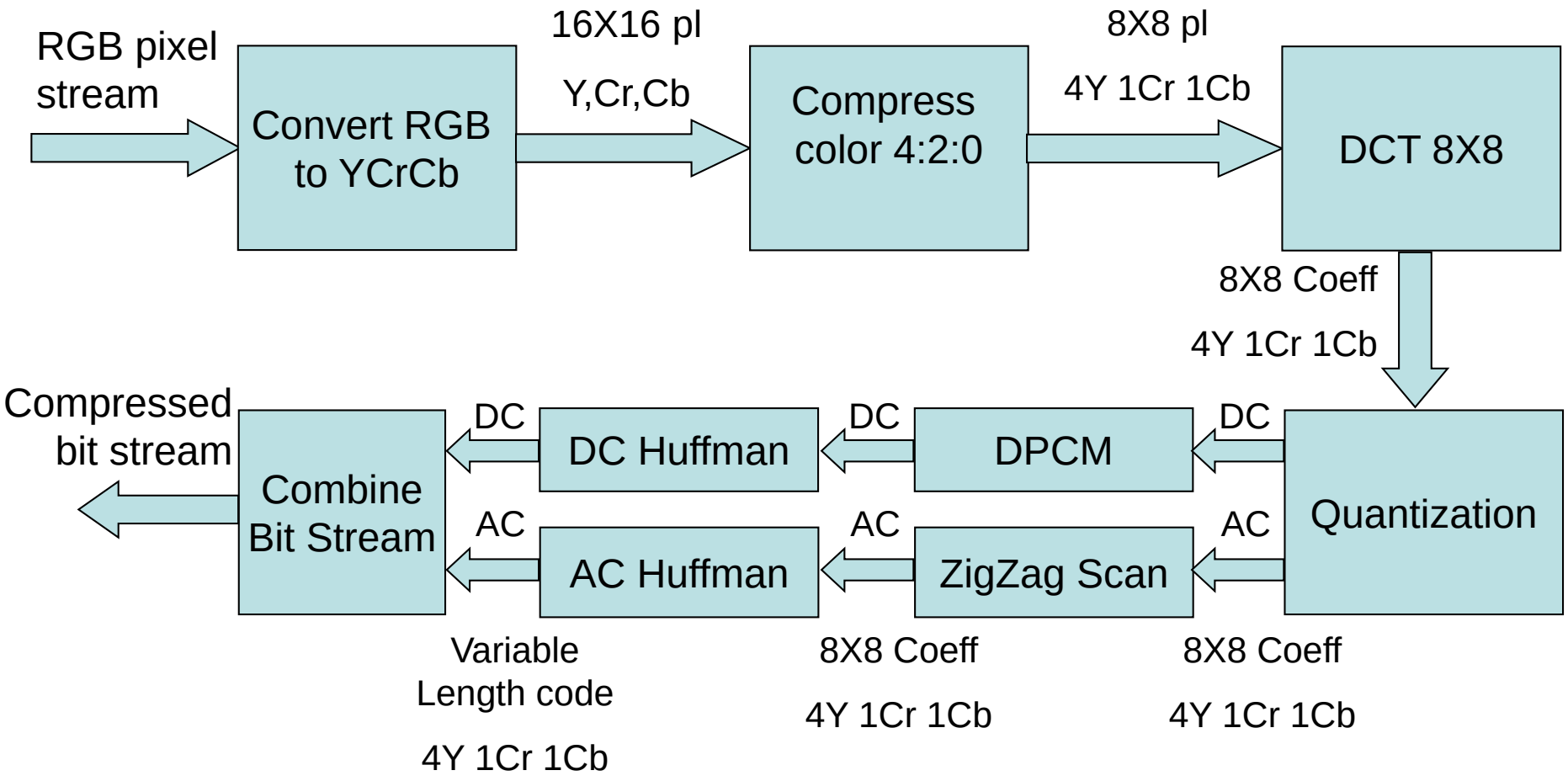
```
regular  task      program_start()
regular  task      AA       (program_start || task_manager==0)
regular  task      CC       (program_start || task_manager==0)
regular  task      EE       (program_start || task_manager==0)
duplicable  task BB       (program_start || task_manager==0)
duplicable  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)

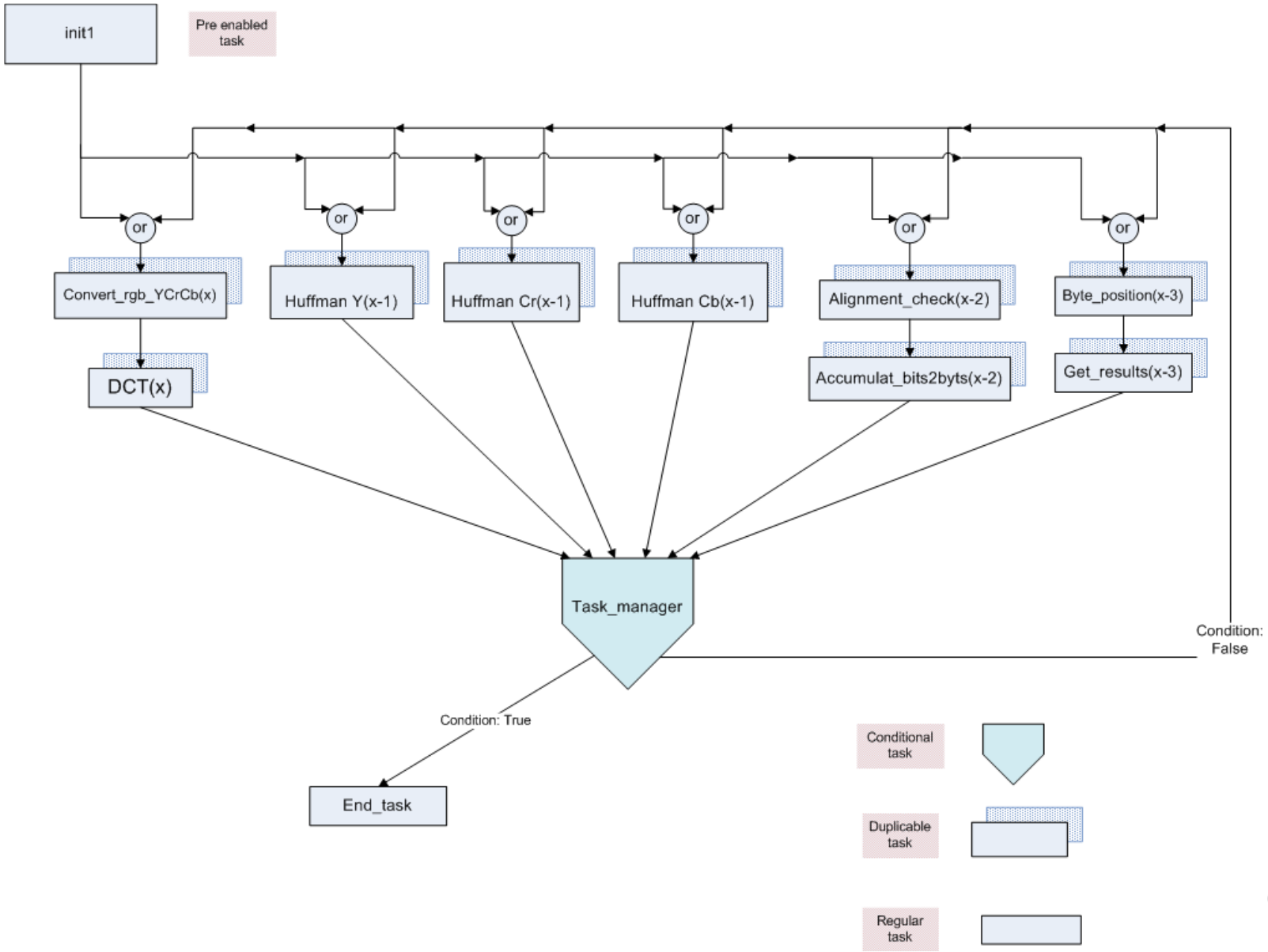
Challenges

- 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

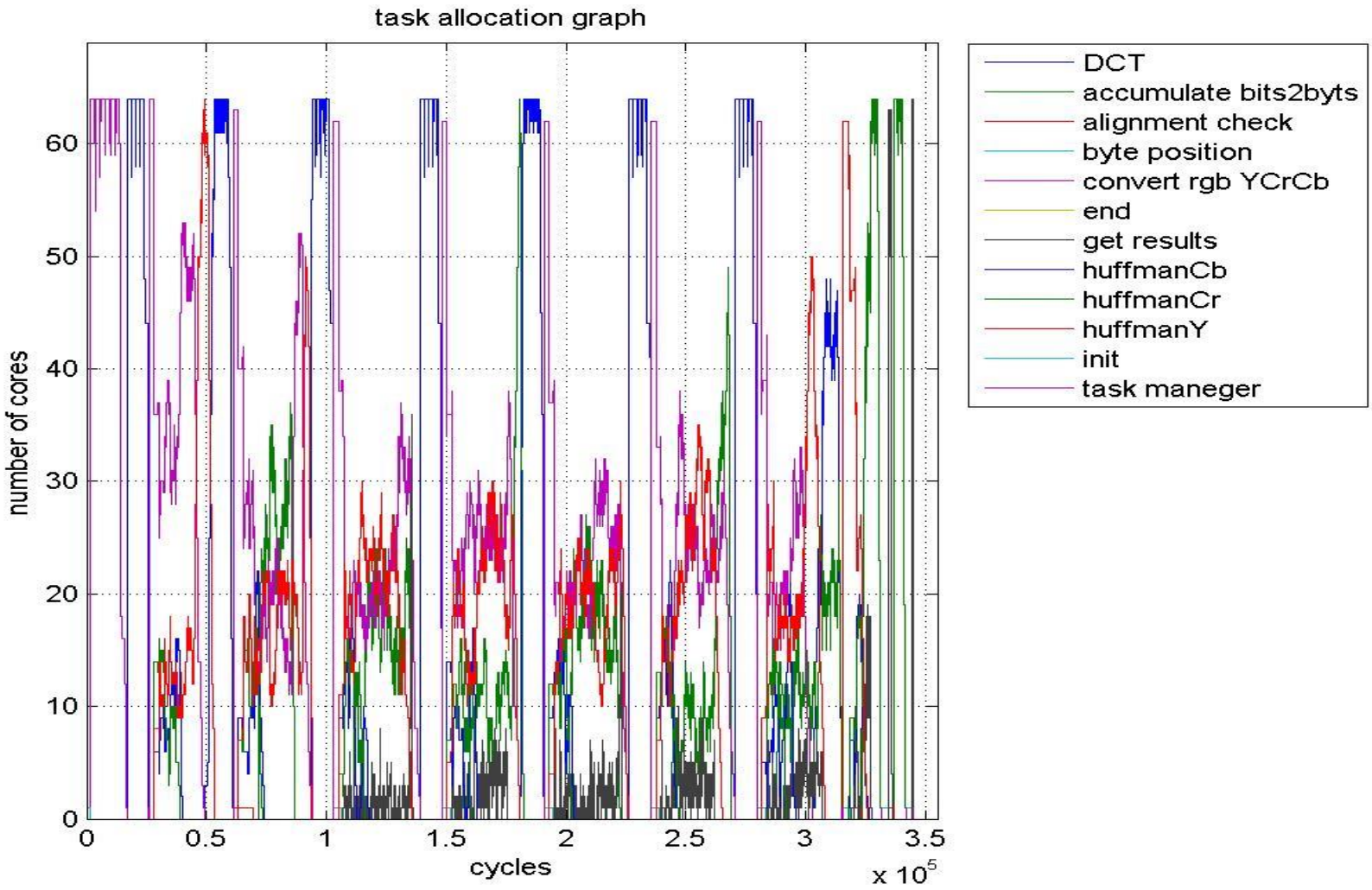
Example: JPEG compression algorithm using ManyFlow



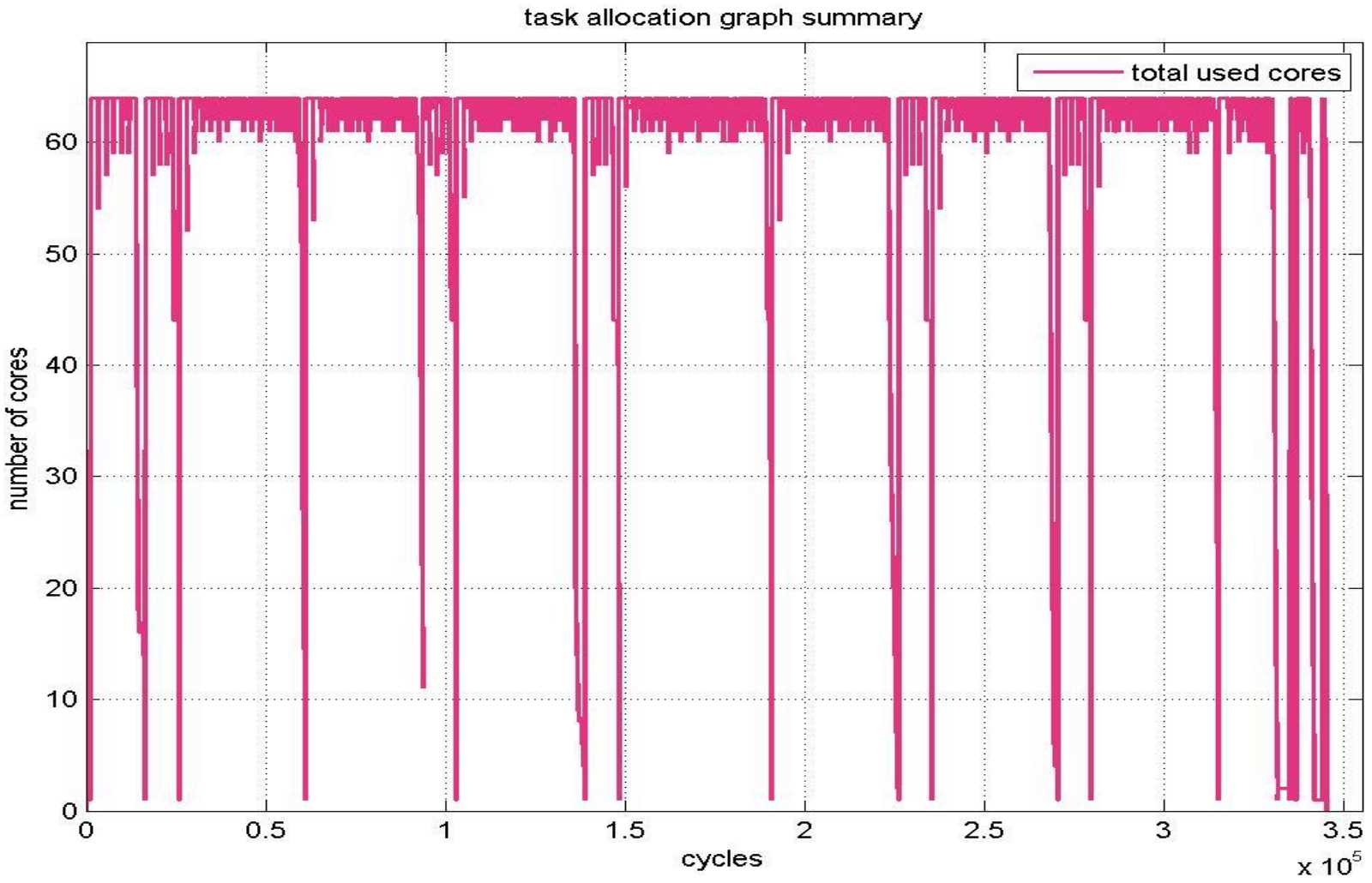
JPEG compression: ManyFlow



JPEG compression: Task Allocation

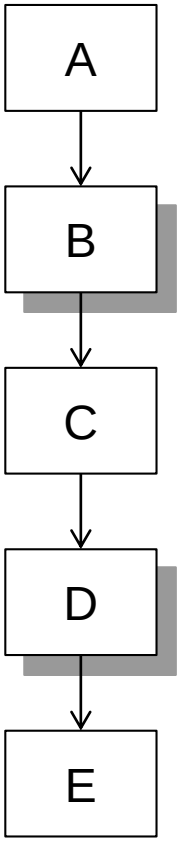


JPEG compression: Most cores active



Example: JPEG2000 Encoder

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



Serial: 220 msec

Parallel:
1280/64=20 msec

Serial: 60 msec

Parallel:
1920/64 = 30 msec

Serial: 70 msec

Core frequency $F_1 = 250$ MHz

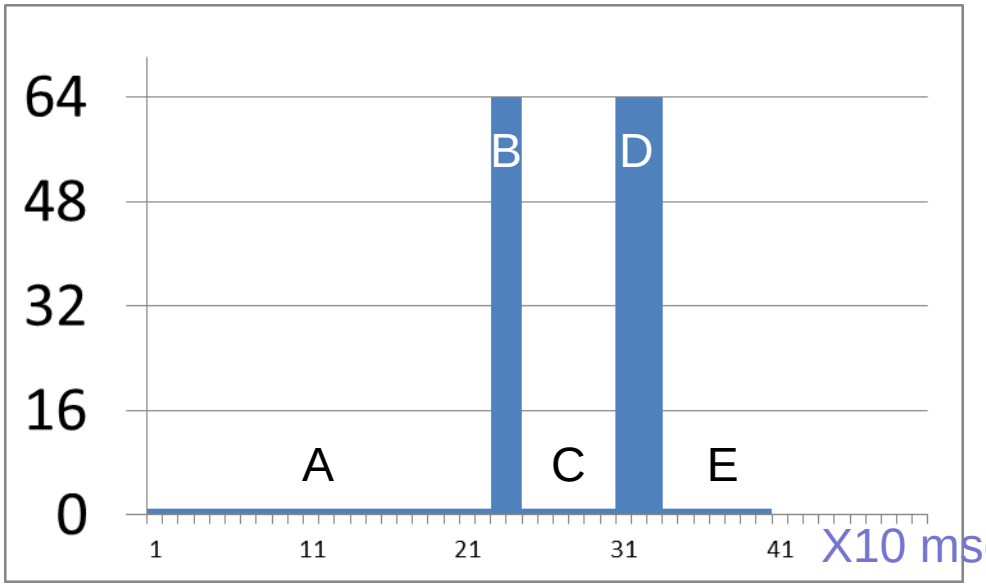
Serial time $T_1 = 3.55$ sec

Parallel time $T_{64} = 400$ msec

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

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

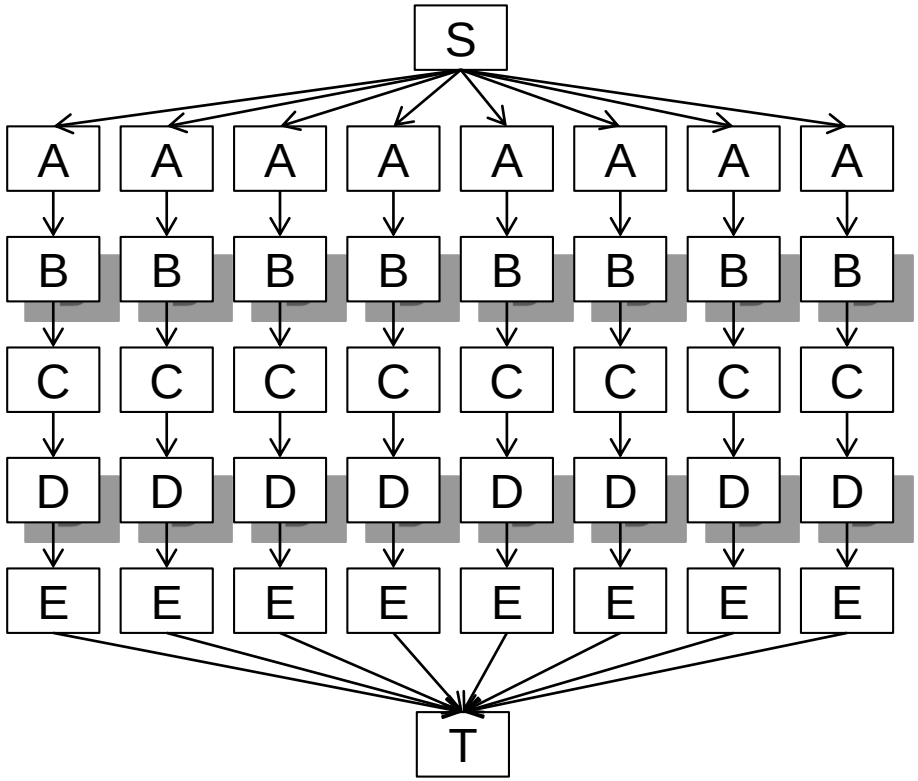
Number of busy cores



Parallel fraction f=95%

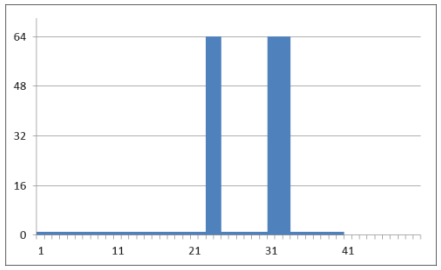
Non-ManyFlow RIGID Multi-Job Scheduling

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



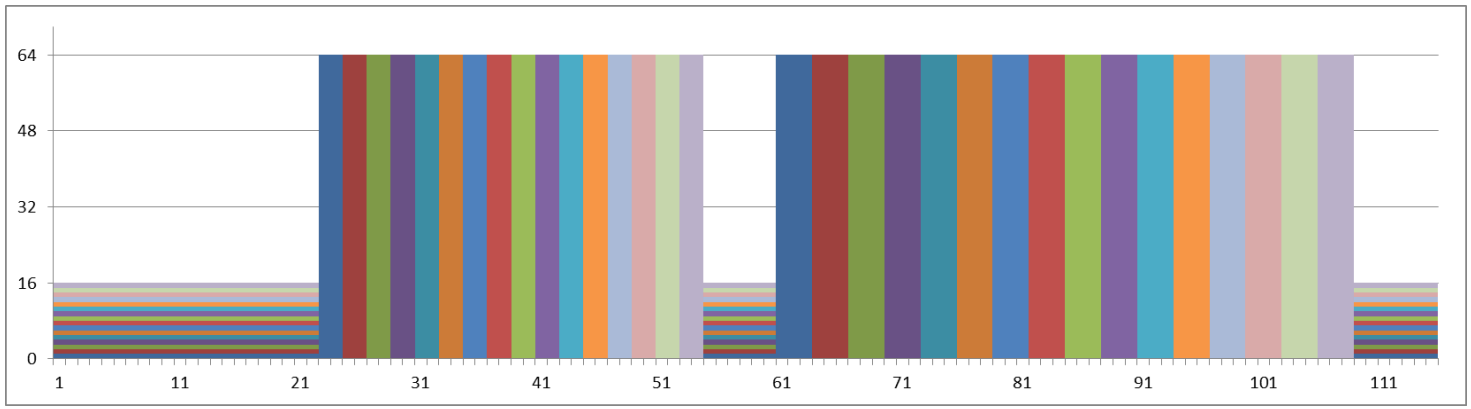
Non-ManyFlow RIGID Multi-Job Scheduling

- 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,…
- 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 $T_P(J) = T_{PS} + T_{PP}$

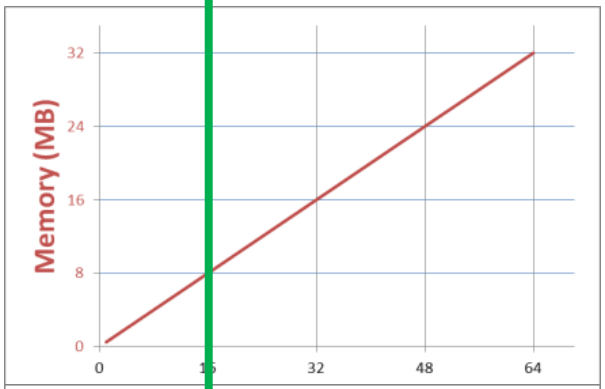


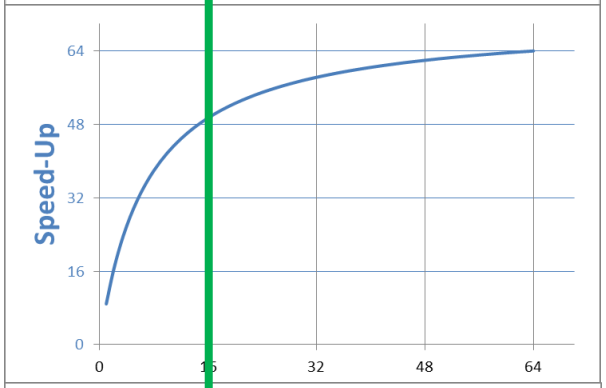
JPEG2000, J=1, f=95%

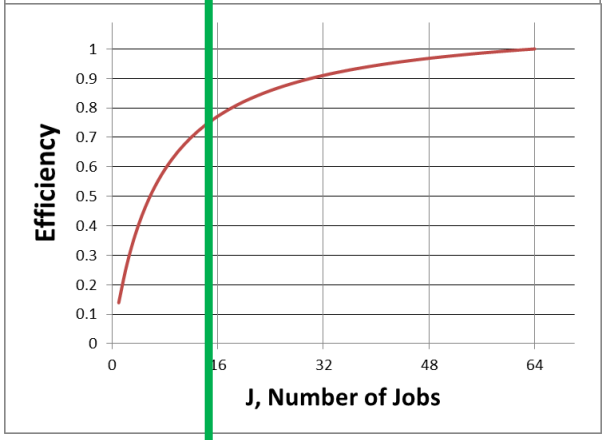
J=16



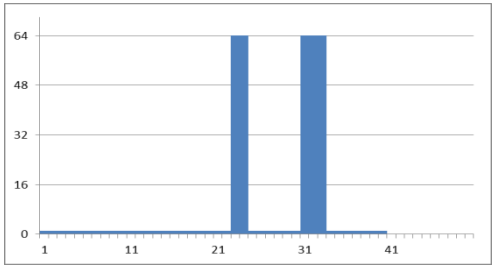
Non-ManyFlow RIGID Multi-Job Scheduling





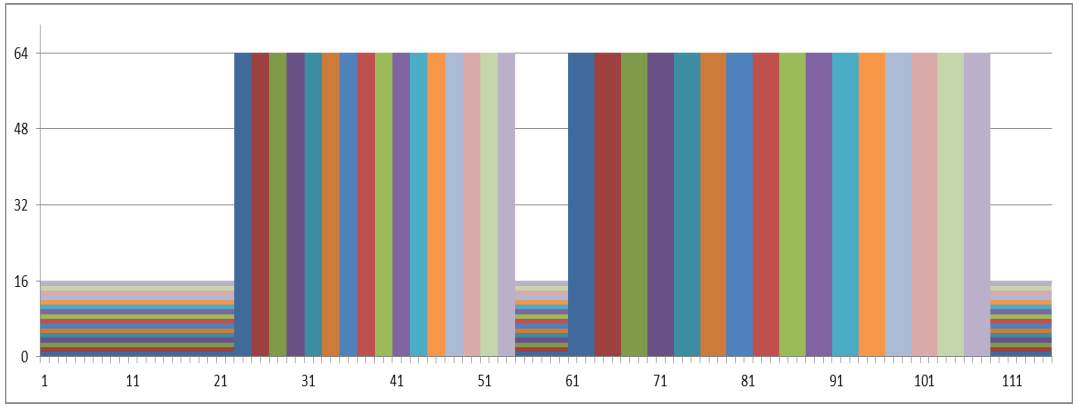


- Memory-limited
- 8MB (¼ max memory) enables:
  - J=16 jobs
  - Speed-up 50 (cf. 9)
  - 0.8 efficiency (cf. 0.14)
    - ***ManyFlow works better !***



JPEG2000, J=1

J=16



Advantages of the Plural Architecture

- 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
- Any core can do any task equally well on short notice
  - scales well
- Programming model:
  - intuitive to programmers
  - CREW verifiable
  - “easy” for automatic parallelizing compiler (?)

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