Class Schedule

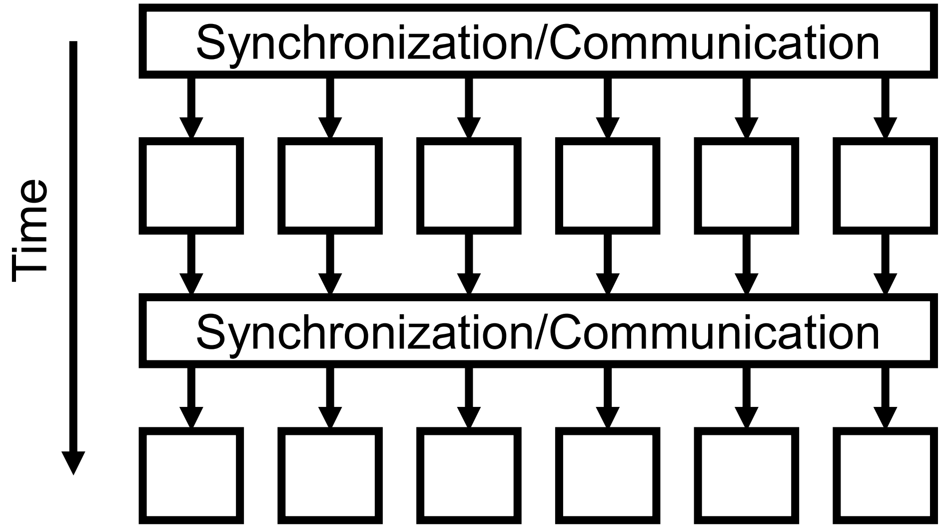
- Review Nov. 7, 2018
- Nov. 12, Regular class, Exam Monday, Nov. 14, 2018
- The week of Nov. 18, no class, individual prepare term project
- Term project presentation: Monday Nov. 25 during and after class hour., Tuesday, Nov. 26 (back up)
- Term report due: Nov. 26

Application Structure

- Frequently used patterns for parallel applications:
  - Single Program Multiple Data – SPMD (Domain Decomposition)
  - Embarrassingly Parallel
  - Master / Slave
  - Work Pool
  - Divide and Conquer
  - Pipeline
  - Competition

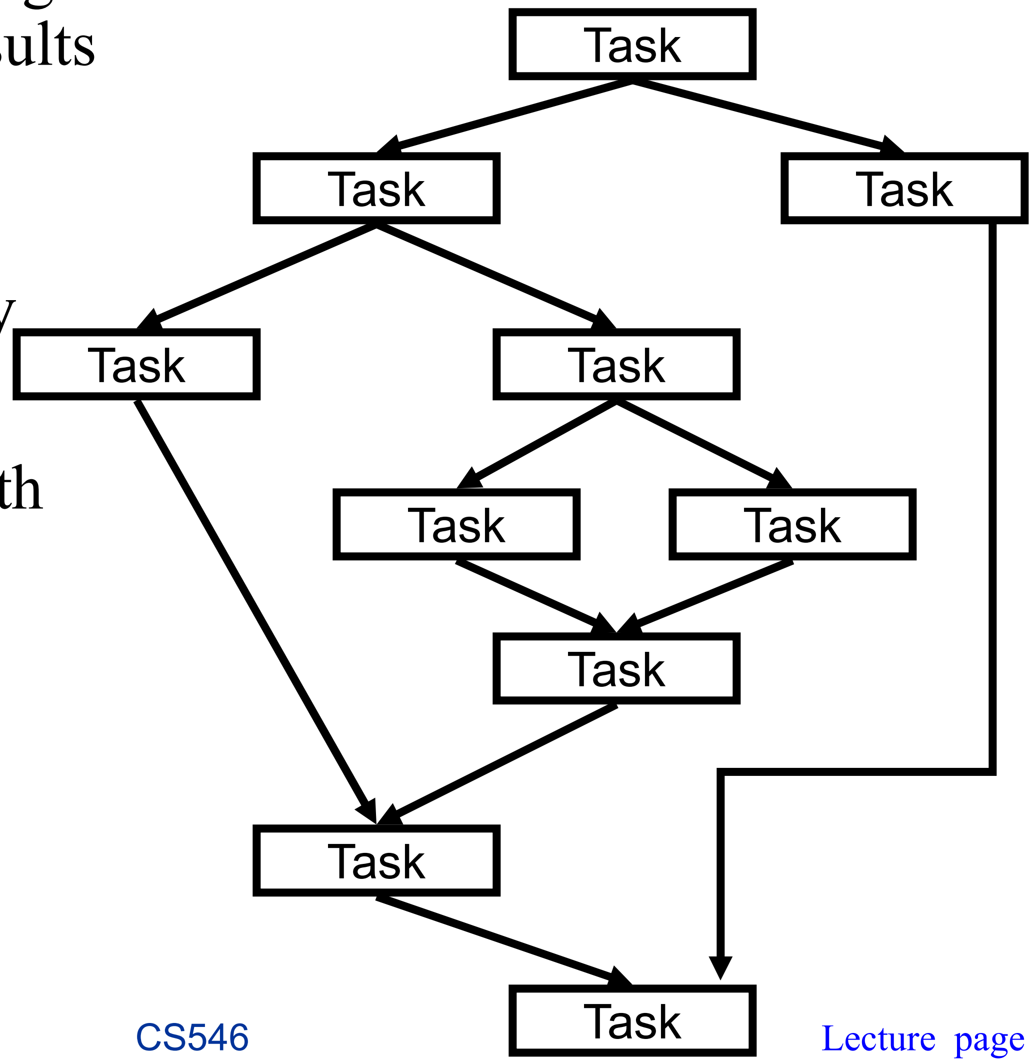
Structure: Single Program Multiple Data

- Single program is executed in a replicated fashion.
- Processes or threads execute same operations on different data.
- Loosely-synchronous: Sequence of phases of computation and communication/synchronization.

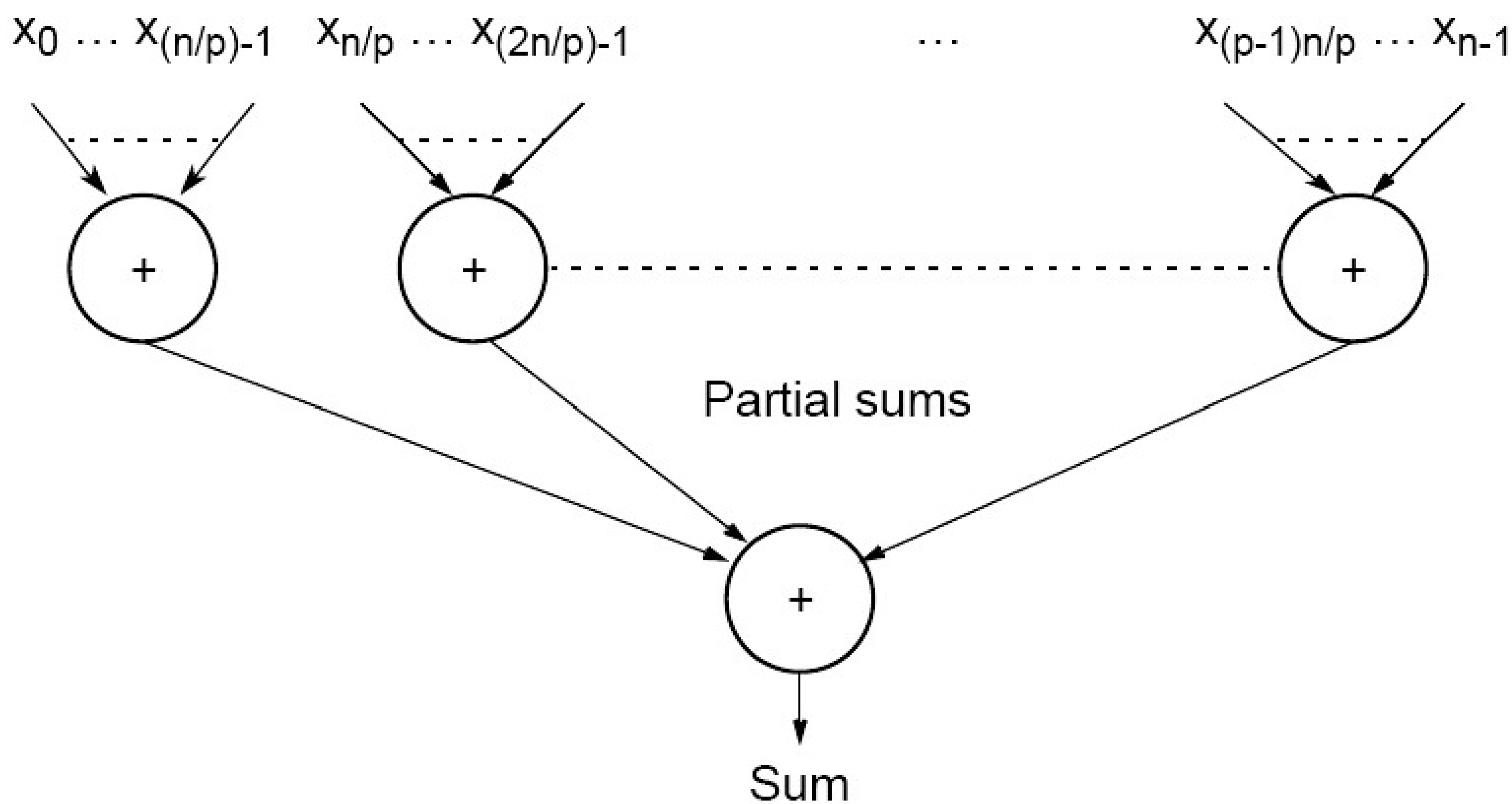


Structure: Divide and Conquer

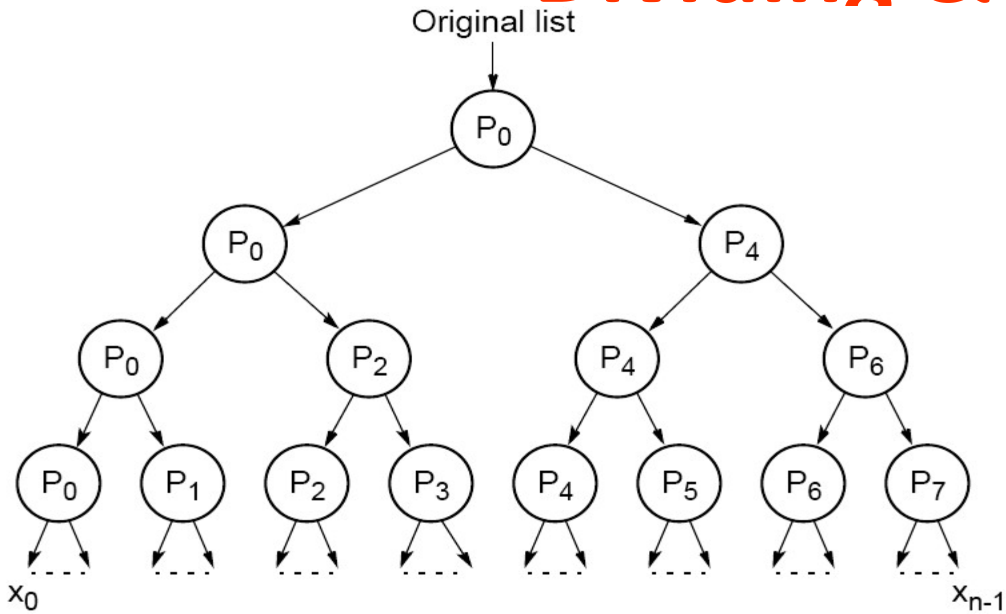
- Recursive partitioning of tasks and collection of results
- Problems:
  - o load balancing
  - o least granularity
- Used on systems with background load

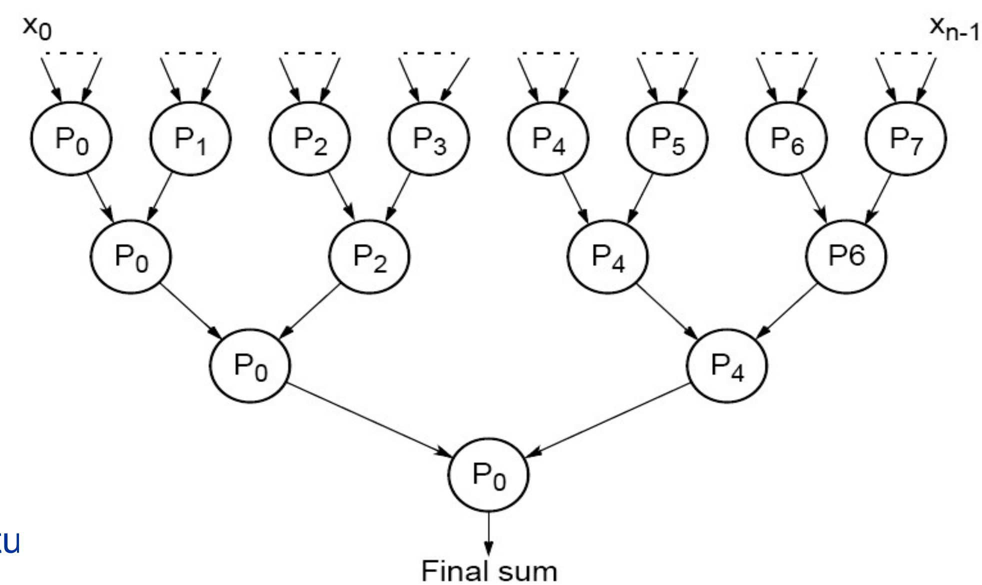


Adding Numbers



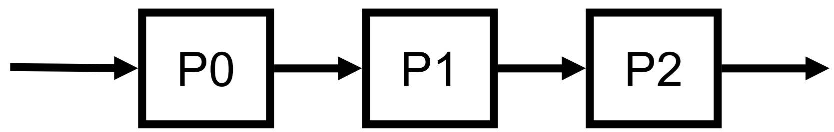
Dividing & Conquer





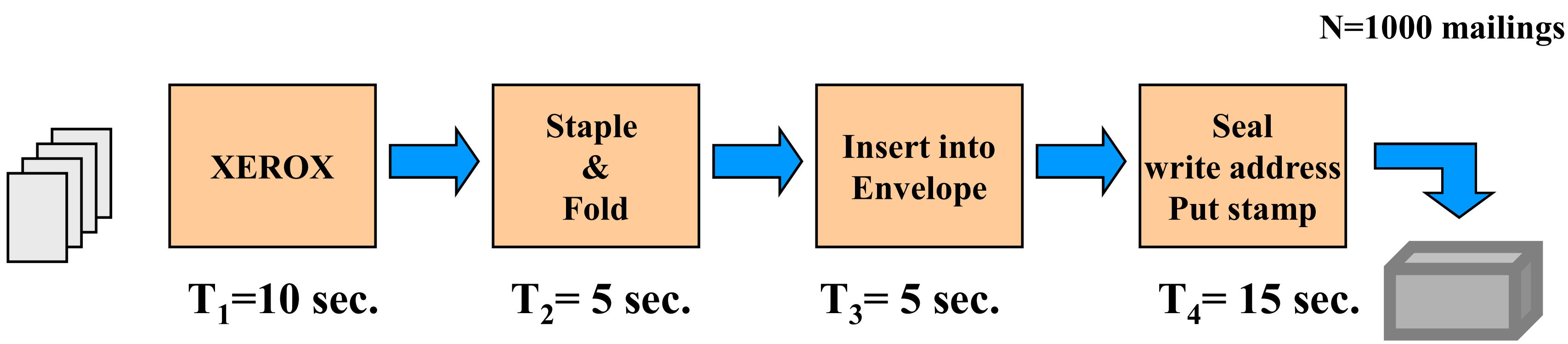
Structure: Pipeline

- Examples
  - Different functions are applied to data: functional decomposition
  - Parallel execution of functions for different data.
  - Signal and image processing
  - Groundwater flow, flow of pollutants, visualization
  - Almost no example of high parallelism



Pipelining: Example

We would like to prepare and mail 1000 envelopes each containing a document of 4 pages to members of an association.



- At what intervals, do we see a new envelope prepared for mailing?

  $\text{Max}(\mathbf{T_1}, \mathbf{T_2}, \ldots, \mathbf{T_k}) = \mathbf{T_{max}} = 15$ sec.

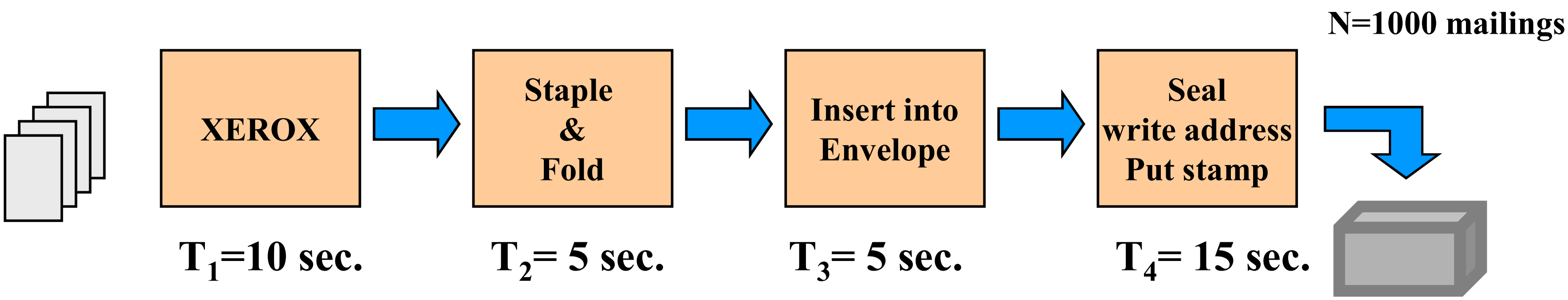
- What is the total time to get N envelopes prepared?

  $\text{Time} = \text{Cold\_Start\_Time} + \mathbf{T_{max}} * (N-1) \cong N * \mathbf{T_{max}}$

- What is the total time we would have spent if pipelining is not used?

  $N * \Sigma_i \mathbf{T_i}$

Pipelining: Example (contd.)



How much speedup do we get?

Speedup = $T_{seq}/T_{pipe} = [\ N^*\Sigma_i\ \mathbf{T_i}\ ] /\ N^*\ \mathbf{T_{max}} = \Sigma_i\ \mathbf{T_i}\ /\ \mathbf{T_{max}}$

Speedup = 35/15

If you can not do much about the completion time for one task (i.e. $\Sigma_i\ \mathbf{T_i}$); what can you do to maximize the speedup?

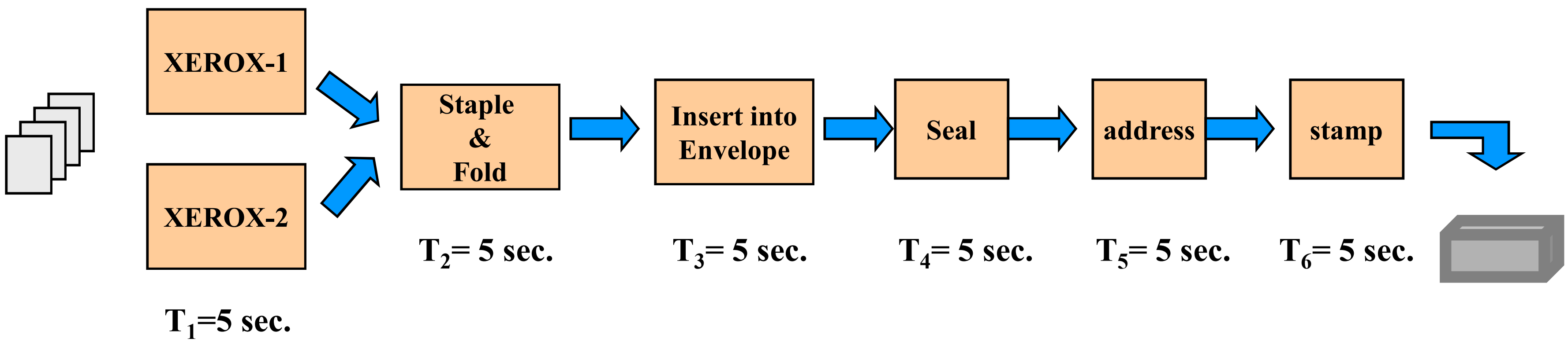
(i) Create as many stations (stages) as possible

and

(ii) Try to balance the load at each station, i.e. $\mathbf{T_1 = T_2 = \ldots = T_k}$

Pipelining: Example (contd.)

One possible configuration to maximize speedup:



- At what intervals, do we see a new envelope prepared for mailing?

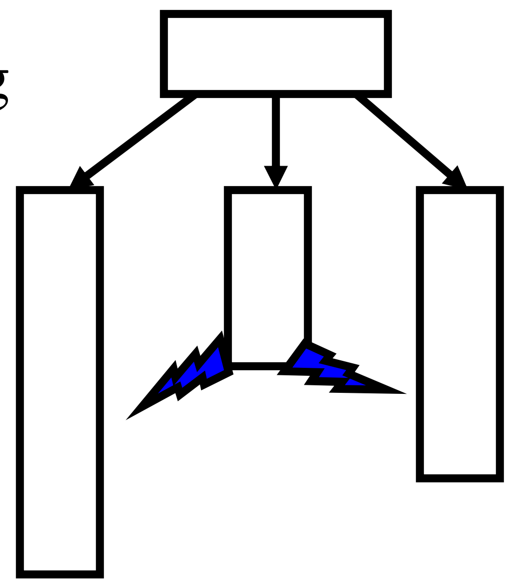
  $\text{Max}(T_1, T_2, \ldots, T_k) = T_{max} = 5$ sec.

- What is the speedup now?

  Speedup = 30/5 = 6 = number of stages in the pipeline !

Structure: Competition

- Evaluation of multiple solution strategies in parallel.
- It might be unknown which strategy is successful or which one is the fastest.
- With k processors, k strategies can be tested. If one of the additional strategies - not tested in the sequential program - is very fast, the speedup can be more than k (Superlinear speedup)
- Random search, speculative computing

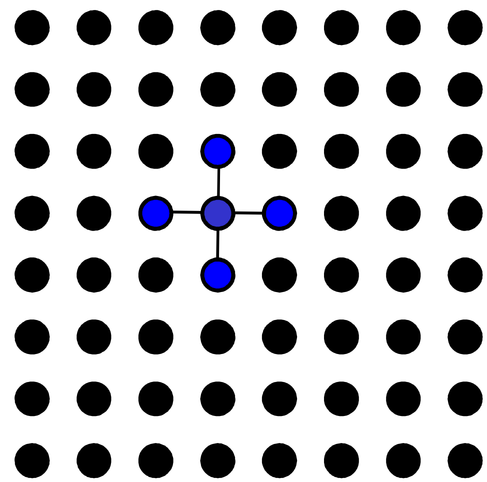


Application Structure: Application

- Performance Analysis
  - Pipeline
  - Based on the parameters to make decision
- System Support
  - MapReduce
- Optimization

Example: Equation Solver Kernel

- Solver kernel for a simple partial differential equation.
- Finite difference method
- Grid $(n+2) \times (n+2)$
- Fixed boundaries
- Interior points are recomputed
  - Mean value of five points in stencil
  - In place computation
  - Gauss-Seidel method
    - New values of upper and right point
    - Old values of lower and left point
  - Termination if difference between old and new value is below threshold for all points



```
A[i,j]=0.2*(A[i,j]+
  A[i-1,j]+A[i,j-1]+
  A[i,j+1]+A[i+1,j]
```

Sequential Code

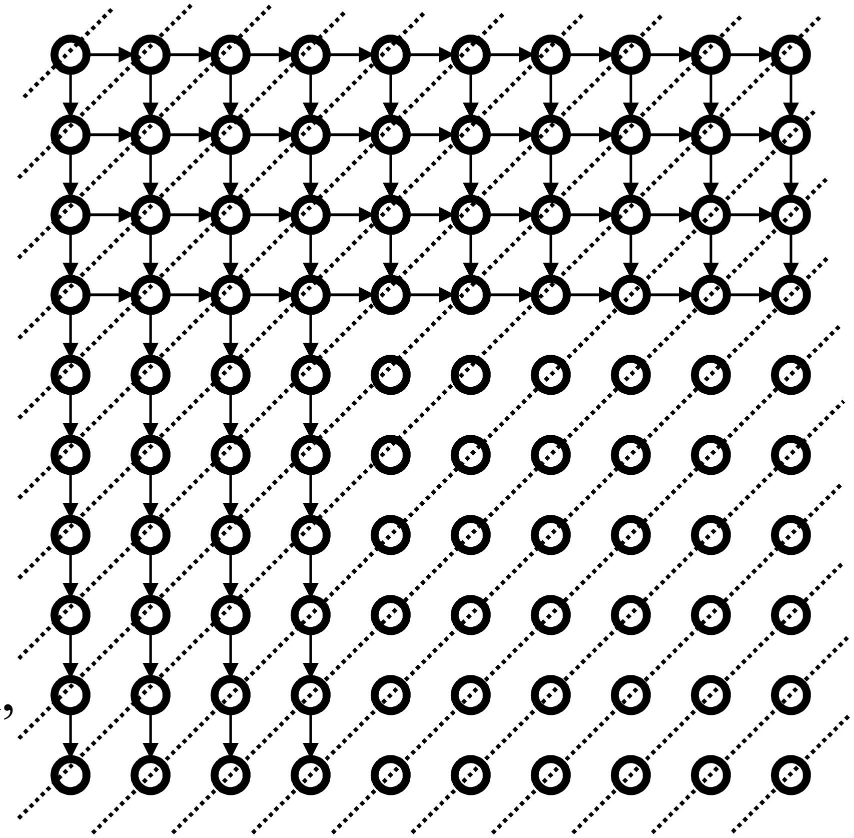
```
int n;                  /*size of matrix: (n + 2-by-n + 2)*/
float: **A, diff = 0;

main ()
begin
read(n);                /*read input parameter: matrix size*/
A = malloc (a 2-d array of size n + 2 by n + 2 doubles);
initialize(A);          /*initialize the matrix A somehow*/
Solve (A);              /*call the routine to solve equation*/

end main
```

Routine SOLVE

```
procedure Solve (A)          /*solve the equation system'/
float **A;                    /*A is an (n + 2)-by-(n + 2) array*/
begin
int i, j, done = 0;
float diff = 0, temp;
while (!done) do             /*outermost loop over sweeps*/
  diff = 0;                  /*initialize maximum difference to 0*/
  for i=1 to n do             /*sweep over nonborder points of grid*/
    for j=1 to n do
      temp = A[i,j];         /*save old value of element*/
      A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                A[i,j+1] + A[i+1,j]); /*compute average*/
      diff += abs(A[i,j] - temp);
    end for
  end for
  if (diff/(n*n) < TOL) then done = 1;
end while
end procedure
```

Dependences in Gauss-Seidel

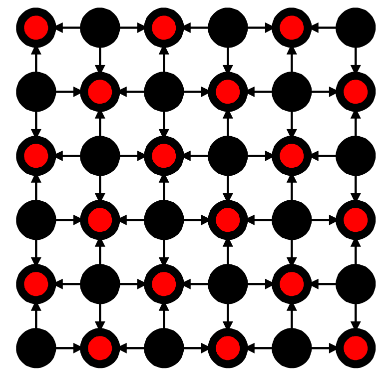
- Dependences prohibit row or column wise parallelization
- Point-wise synchronization
- Parallel execution along anti-diagonals
  - Proportional to n
  - Frequent synchronization, once per anti-diagonal
  - Load imbalance for short anti-diagonals



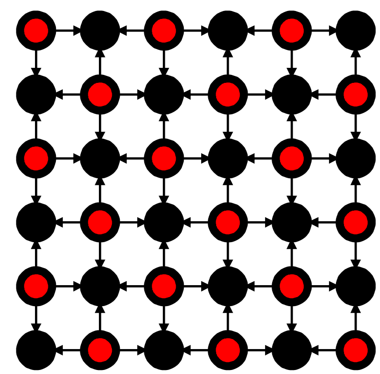
Relaxing Ordering Constraints

- Jacobi iterations
  - Full sweep with old values
  - Much slower convergence→more iterations
  - N^2 parallel tasks in each iteration
- Red-Black iterations:
  - Checkerboard like coloring scheme
  - Two phases
    - Computation of red points with old black values
    - Computation of black points with new red values
  - Faster convergence than Jacobi but more iterations than Gauss-Seidel
  - Each phase with $n^2/2$ parallel tasks

Red phase



Black phase



Message Passing Programming (1/7)

- Processes have private address spaces
- Values are communicated via send/receive operations
- Writing local code alike to thread programming

Message Passing Programming (2/7)

```
Int pid, n, nprocs;              /*process id and number of processes*/
Float **myA;

Main()
Begin
  nprocs=get_proc_num();         /*number of started processes*/
  pid=get_my_rank();             /*pid of the executing process*/
  if (pid==0) read(n);           /*master only*/
  broadcast(0,n,sizeof(int),SIZE); /*communicate to all others*/
  solve();                       /*start computation*/
End
```

Message Passing Programming (3/7)

```
procedure solve()
begin
int I,j, pid, n1=n/nprocs, done=0;
float temp, tempdiff, mydiff=0;
myA=malloc(n/nprocs+2 by n+2);          /*allocate my rows+ghost rows*/
initialize (myA);                       /*intialize my rows*/

while (!done) do
  mydiff=0;
  if (pid!=0) then
    send(&myA[1,0],n*sizeof(float),pid-1,ROW)
  if (pid!=nprocs-1) then
    send(&myA[n1,0],n*sizeof(float),pid+1,ROW)
  if (pid!=0) then
    receive(&myA[0,0],n*sizeof(float),pid-1,ROW)
  if (pid!=nprocs-1) then
    receive(myA[n1+1,0],n*sizeof(float),pid+1,ROW)
```

Message Passing Programming (4/7)

```
for i=1 to n1 do            /*sweep over nonborder points of grid*/
    for j=1 to n do
      temp = myA[i,j];      /*save old value of element*/
      myA[i,jl = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
               myA[i,j+1] + myA[i+1,j]); /*compute average*/
      mydiff += abs(myA[i,j] - temp);
    end for
  end for
```

Message Passing Programming (5/7)

```
if (pid !=0) then                          /*send local diffs to P0*/
   send (mydiff,sizeof(float),0,DIFF)
   receive(done,sizeof(int),0,DONE)        /*receive done flag from P0*/
else
   for i=1 to nprocs-1 do
     receive(tempdiff,sizeof(float),*,DIFF) /*receive local diffs*/
     mydiff += tempdiff;                   /*accumulate local diffs*/
   endfor
   if (mydiff/(n*n)<TOL) then  done=1 /*check condition*/
   for i=1 to nprocs-1 do
     send(done,sizeof(int),i,done);        /*send done flag to other procs*/
   end for
endif

endwhile
end procedure
```

Message Passing Programming (6/7)

```
reduce(0,mydiff,sizeof(float),ADD)
if (pid ==0) then
  if (mydiff/(n*n)<TOL) then done=1;
  endif
endif
broadcast(0,done,sizeof(int));
endwhile
end procedure
```

Message Passing Programming (7/7)

Syntax	Function
Send(src_addr, size, dest, tag)	Send size bytes starting at src_addr to the dest process, with tag identifier.
Receive(buffer_addr, size, src, tag)	Recieve a message with the tag identifier from the src process, and put size bytes of it into buffer starting at buffer_addr.
Reduce(root_pid, buffer, length, oper)	Compute global value from local values in buffer of the given length of all processes with operation oper. The global value is delivered to root process.
Broadcast(root_pid, buffer, length, tag)	The root process sends the value in buffer to the other processes with the tag identifier.

The Parallel Partition LU (PPT) Algorithm

Step 1. Allocate $A_i, d^{(i)}$ and elements $a_{im}, c_{(i+1)m-1}$ to the *ith* node, where $0 \le i \le p-1$.

Step 2. Use the *LU* decomposition method to solve

$$A_i[\tilde{x}^{(i)}, v^{(i)}, w^{(i)}] = [d^{(i)}, a_{im}e_0, c_{(i+1)m-1}e_{m-1}]$$

Step 3. Send $\tilde{x}_0^{(i)}, \tilde{x}_{m-1}^{(i)}, v_0^{(i)}, v_{m-1}^{(i)}, w_0^{(i)}, w_{m-1}^{(i)}$ from the *ith* node to the other nodes $0 \le i \le p-1$.

Step 4. Use the LU method to solve Zy = h on all nodes

Step 5. Compute in parallel on p processors

$$\Delta x^{(i)} = [v^{(i)}, w^{(i)}] \begin{bmatrix} y_{2i-1} \\ y_{2i} \end{bmatrix}$$

$$x^{(i)} = \tilde{x}^{(i)} - \Delta x^{(i)}$$

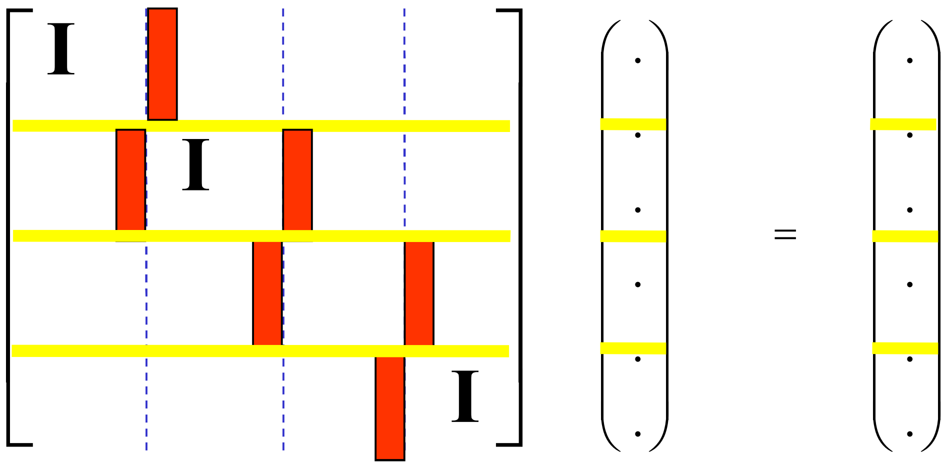
The Solving process

1. Solve the subsystems in parallel
2. Solve the reduced system
3. Modification

$$\tilde{\mathbf{A}}^{-1}\mathbf{A}\mathbf{x} = \tilde{\mathbf{A}}^{-1}\mathbf{d}$$

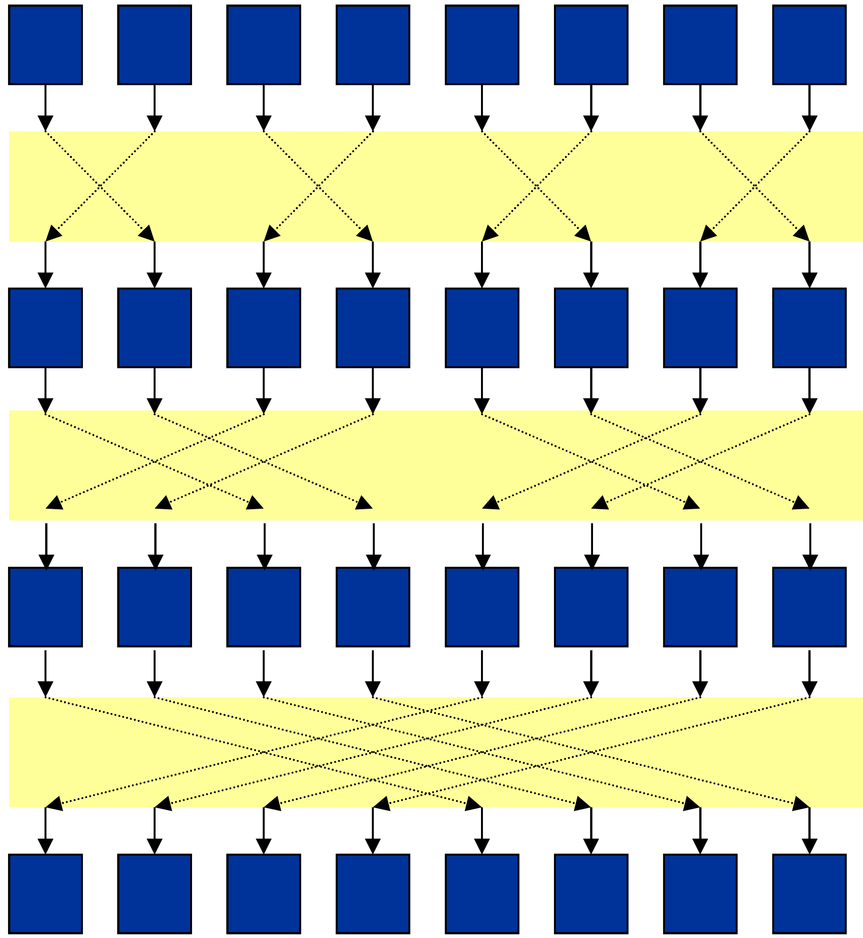
$$\left[\begin{array}{c:c:c:c} A_0^{-1} & & & \\ \hdashline & A_1^{-1} & & \\ \hdashline & & & \\ \hdashline & & & A_{p-1}^{-1} \end{array}\right]\left[\begin{array}{c:c:c:c} A_0 & * & & \\ \hdashline * & A_1 & * & \\ \hdashline & * & & \\ \hdashline & & * & A_{p-1} \end{array}\right]\begin{pmatrix} x_0 \\ x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_{n-1} \end{pmatrix} = \left[\begin{array}{c:c:c:c} A_0^{-1} & & & \\ \hdashline & A_1^{-1} & & \\ \hdashline & & & \\ \hdashline & & & A_{p-1}^{-1} \end{array}\right]\begin{pmatrix} d_0 \\ d_1 \\ \cdot \\ \cdot \\ \cdot \\ d_{n-1} \end{pmatrix}$$

The Reduced System (*Zy*=*h*)



Needs global communication

All-to-All Total Data Exchange



X. Sun (IIT)

Summary Parallel Programming

- Steps in the parallelization process
  - Decomposition, assignment, orchestration, mapping
- Variety of programming models for SM and DM systems
  - Different parallel languages and APIs
  - Trend towards standards (OpenMP, MPI, Posix)

Summary Parallel Programming Models

- Global vs local programming
  - Global: directive-based, data parallel
  - Local: thread-based, remote memory access, message passing
- Data vs functional parallelism
  - Data parallelism only: data parallel
  - Both: all the others
- Parallelism
  - High: remote memory access, message passing, thread-based
  - Low:directive-based, high level data parallel

Figure 4-1 A Dog

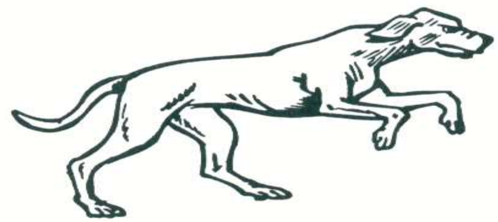


Figure 4-2 A Pack of Dogs

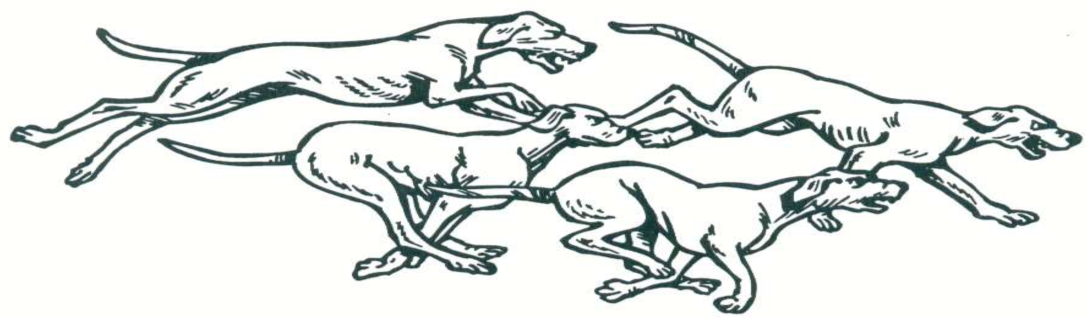
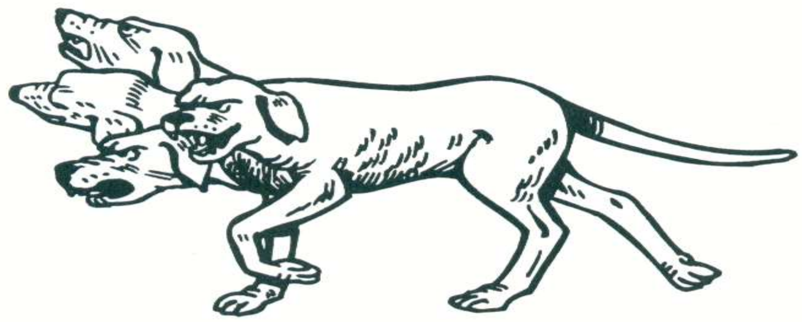


Figure 4-3 A Savage Multiheaded Pooch



Concurrent-AMAT: step to optimization

- The traditional AMAT(Average Memory Access Time) :

$$\text{AMAT} = \text{HitCycle} + \text{MR} \times \text{AMP}$$

- MR is the miss rate of cache accesses; and AMP is the average miss penalty

- **Concurrent-AMAT (C-AMAT)**:

$$\text{C-AMAT} = \text{HitCycle}/C_H + p\text{MR} \times p\text{AMP}/C_M = 1/\text{APC}$$

- C_H is the hit concurrency; C_M is the *pure* miss concurrency
- pMR and pAMP are *pure* miss rate and average *pure* miss penalty
- A pure miss is a miss containing at least one cycle which does not have any hit activity

X.-H. Sun and D. Wang, "Concurrent Average Memory Access Time", in ***IEEE Computers***, vol. 47, no. 5, pp. 74-80,May 2014.(IIT Technical Report, IIT/CS-SCS-2012-05)

Recursive in Memory Hierarchy

- AMAT is recursive
  - AMAT = $HitCycle_1 + MR_1 \times AMP_1$

  Where $AMP_1 = (HitCycle_2 + MR_2 \times AMP_2)$

- C-AMAT is also recursive

$$C\text{-}AMAT_1 = \frac{H_1}{C_{H_1}} + MR_1 \times \kappa_1 \times C\text{-}AMAT_2$$

Where

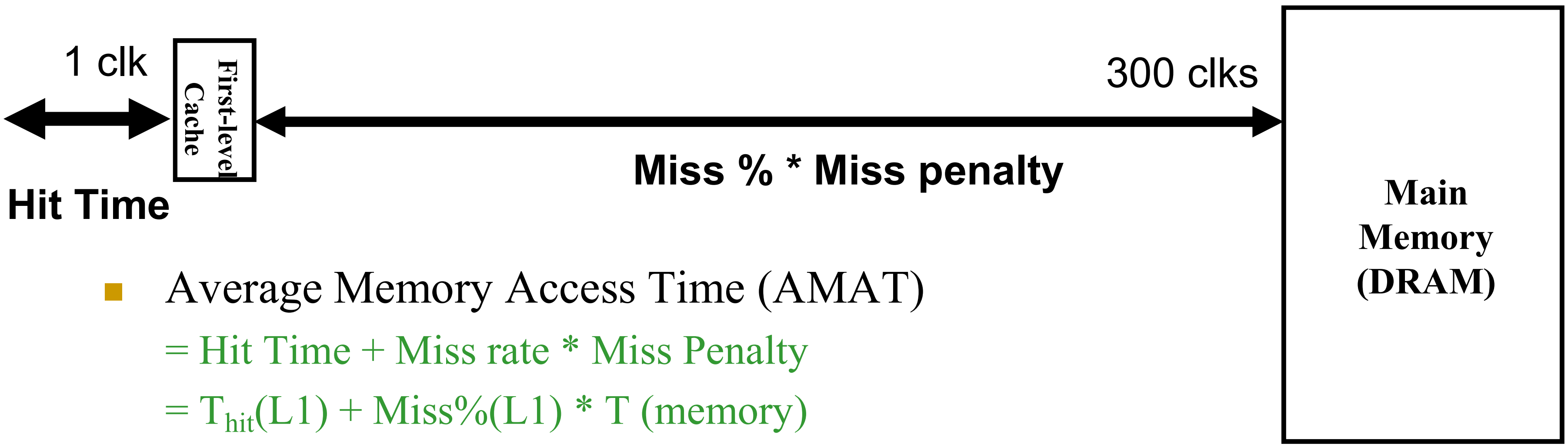
$$C\text{-}AMAT_2 = \frac{H_2}{C_{H_2}} + pMR_2 \times \frac{pAMP_2}{C_{M_2}}$$

$$\kappa_1 = \frac{pMR_1}{MR_1} \times \frac{pAMP_1}{AMP_1} \times \frac{C_{m_1}}{C_{M_1}}$$

With Clear Physical Meaning

X.-H. Sun, “Concurrent-AMAT: a mathematical model for Big Data access,” HPC-Magazine, May 12, 2014

Memory Hierarchy Performance

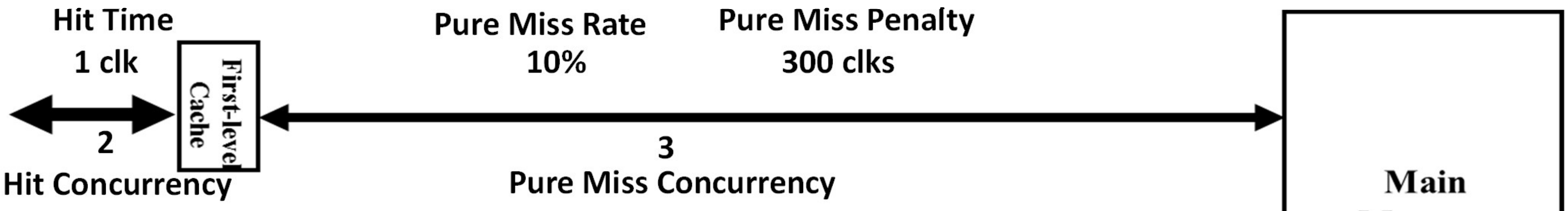


- Average Memory Access Time (AMAT)

  = Hit Time + Miss rate * Miss Penalty

  = T_{hit}(L1) + Miss%(L1) * T (memory)
- Example:
  - Cache Hit = 1 cycle
  - Miss rate = 10%
  - Miss penalty = 300 cycles
  - AMAT = 1 + 300*10% = 31 cycles
- To further improve it?

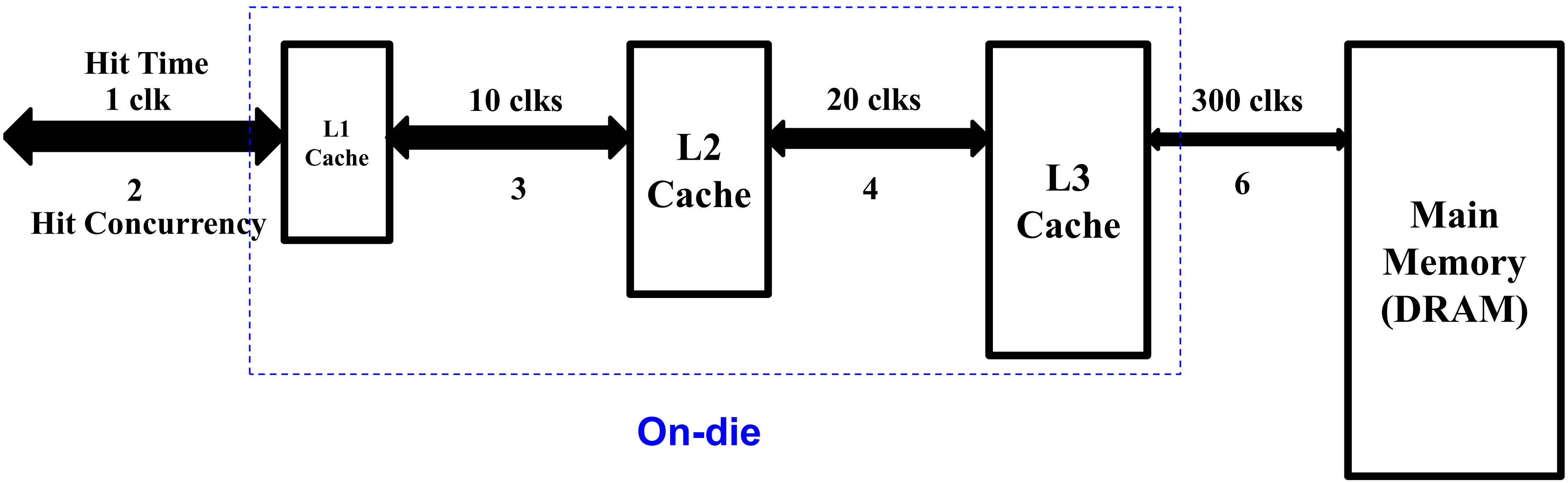
Example of calculating C-AMAT



Concurrent Average Memory Access Time (C-AMAT)
= Hit Time/ Hit Concurrency + Pure Miss Rate * (Pure Miss Penalty/ Pure Miss Concurrency)

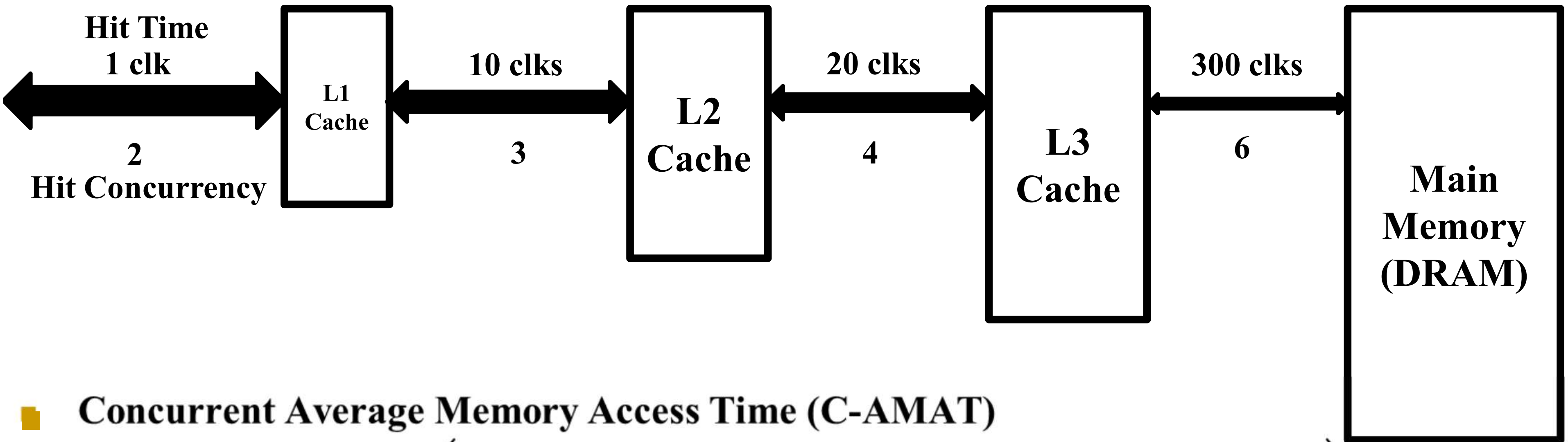
- Example:
  - Hit Time = 1 cycle
  - Hit Concurrency = 2
  - Pure Miss Rate = 10%
  - Pure Miss Penalty = 300 cycles
  - Miss Concurrency = 3
  - C-AMAT = 1/2 + 10%*(300/3) = 0.5 +10 = 10.5 cycles

Data Access Time: AMAT



- Average Memory Access Time (AMAT)
  $= T_{hit}(L1) + Miss\%(L1)^* \ (T_{hit}(L2) + Miss\%(L2)^* \ (T_{hit}(L3) + Miss\%(L3)^*T(memory)\ )\ )$
- Example: (Latency as shown above)
  - Miss rate: L1=10%, L2=5%, L3=1% (*Be careful miss rate definition*)
  - AMAT
    = 2.115

Data Access Time: C-AMAT



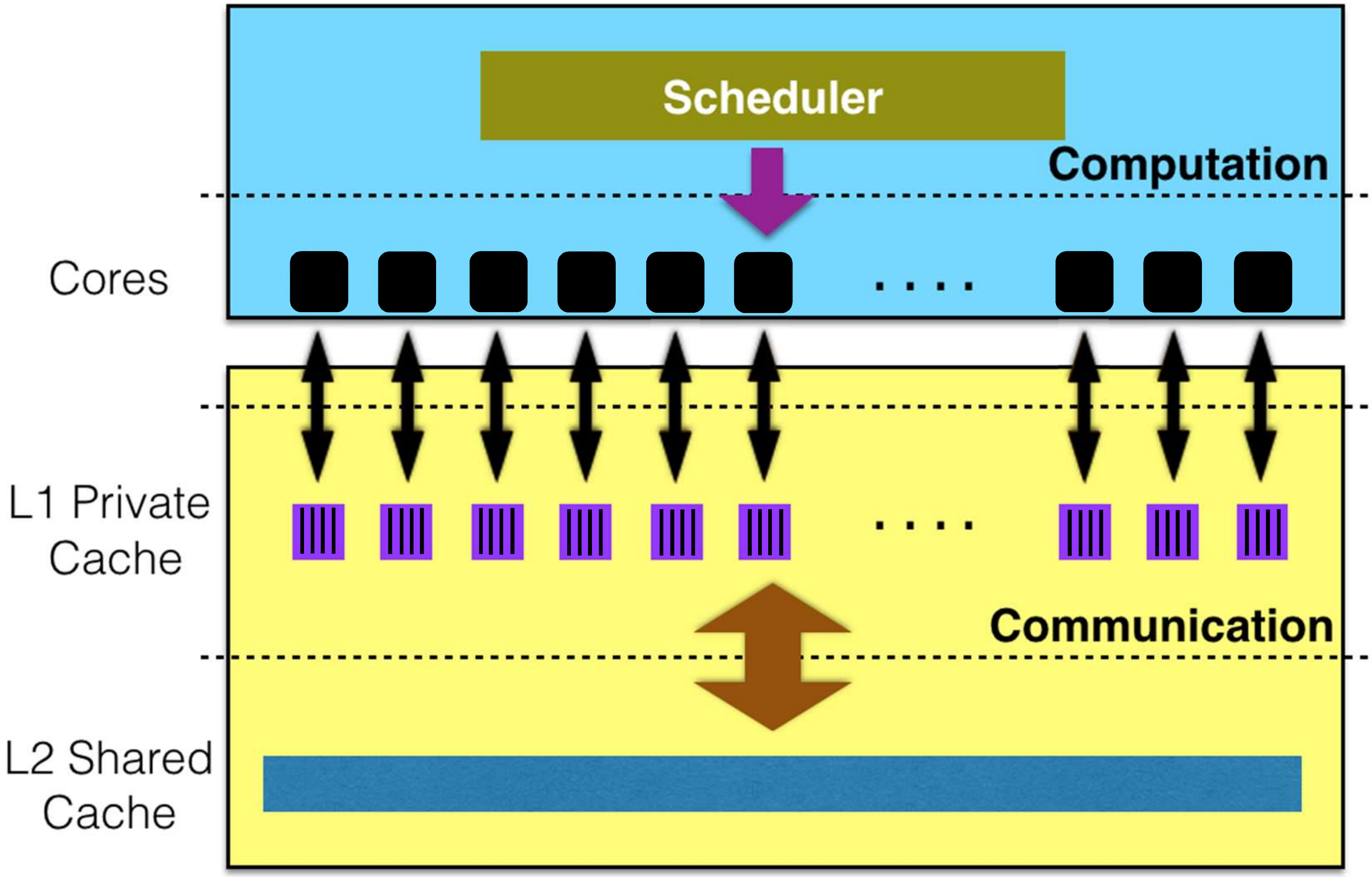
- **Concurrent Average Memory Access Time (C-AMAT)**

$$= \frac{H_1}{C_{H_1}} + MR_1 \times \kappa_1 \times \left(\frac{H_2}{C_{H_2}} + MR_2 \times \kappa_2 \times \left(\frac{H_3}{C_{H_3}} + MR_3 \times \kappa_3 \times \frac{H_{Mem}}{C_{H_{Mem}}} \right) \right)$$

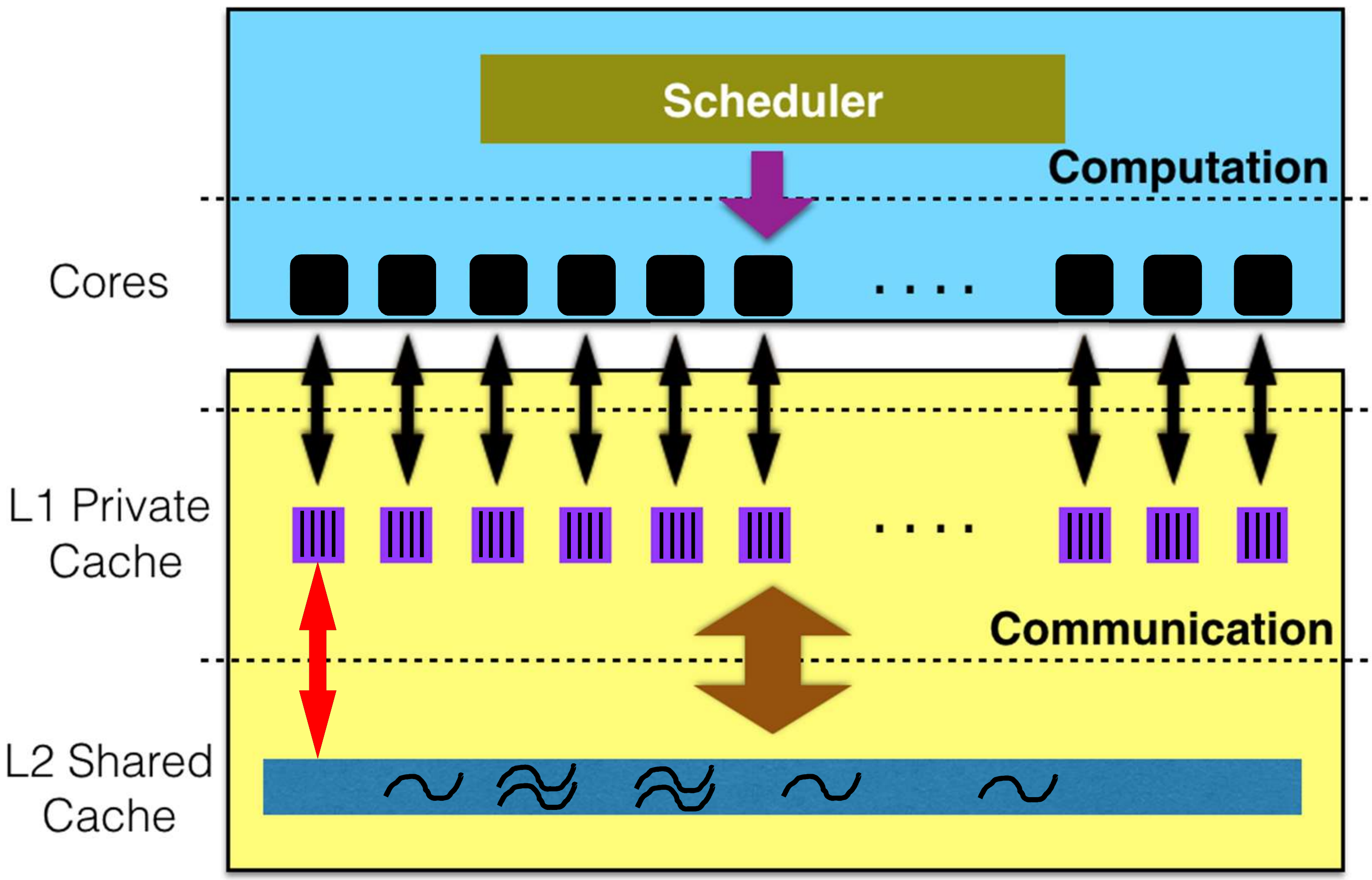
- **Example**
  - Miss Rate: L1=10%, L2=5%, L3=1%
  - κ: L1=0.56, L2=0.6, L3=0.8
  - C-AMAT≈0.696

pMR, pAMP, AMP, C_M, C_m: L1=7%, 10, 10, 5, 4
L2=3%, 60, 40, 9, 6
L3=0.8%, 400, 300, 16, 12

C-AMAT in Multi-Core Environments



C-AMAT in Multi-Core Environments



What Does C-AMAT Say?

- C-AMAT is an extension of AMAT to consider concurrency
  - The same as AMAT, if no concurrency present
- C-AMAT introduces the **Pure Miss** concept:
  - Only pure miss causes performance penalty
- **High locality may hurt performance**
  - High locality may lead to pure miss
- **Two contributions of concurrency**
  - Increase bandwidth
  - Latency hiding (overlapping)

What Does C-AMAT Say?

- C-AMAT also contains the overlapping factor

$$C\text{-}AMAT_1 = \frac{H_1}{C_{H_1}} + MR_1 \times \kappa_1 \times C\text{-}AMAT_2$$

- **Balance** locality, concurrency and Overlapping with C-AMAT
- A good explanation for why
  Optimal ≠ Optimal Locality + Optimal Concurrence
- C-AMAT uniquely integrates the **joint impact** of locality concurrency, and overlapping for optimization
- Overlapping in connection locality and concurrency is a new issue of research

Impact of C-AMAT

- New dimensions for optimization: **concurrency, overlapping and balancing**

$$C\text{-}AMAT_1 = \frac{H_1}{C_{H_1}} + MR_1 \times \kappa_1 \times C\text{-}AMAT_2$$

- Can apply at **each layer** of a memory hierarchy
- Existing mechanisms are readily to be extended
  - Every AMAT based optimization has a corresponding C-AMAT extension to include concurrency
- Concurrency as bandwidth increaser and **penalty reducer (overlapping)**: Accurate measure the concurrency contribution

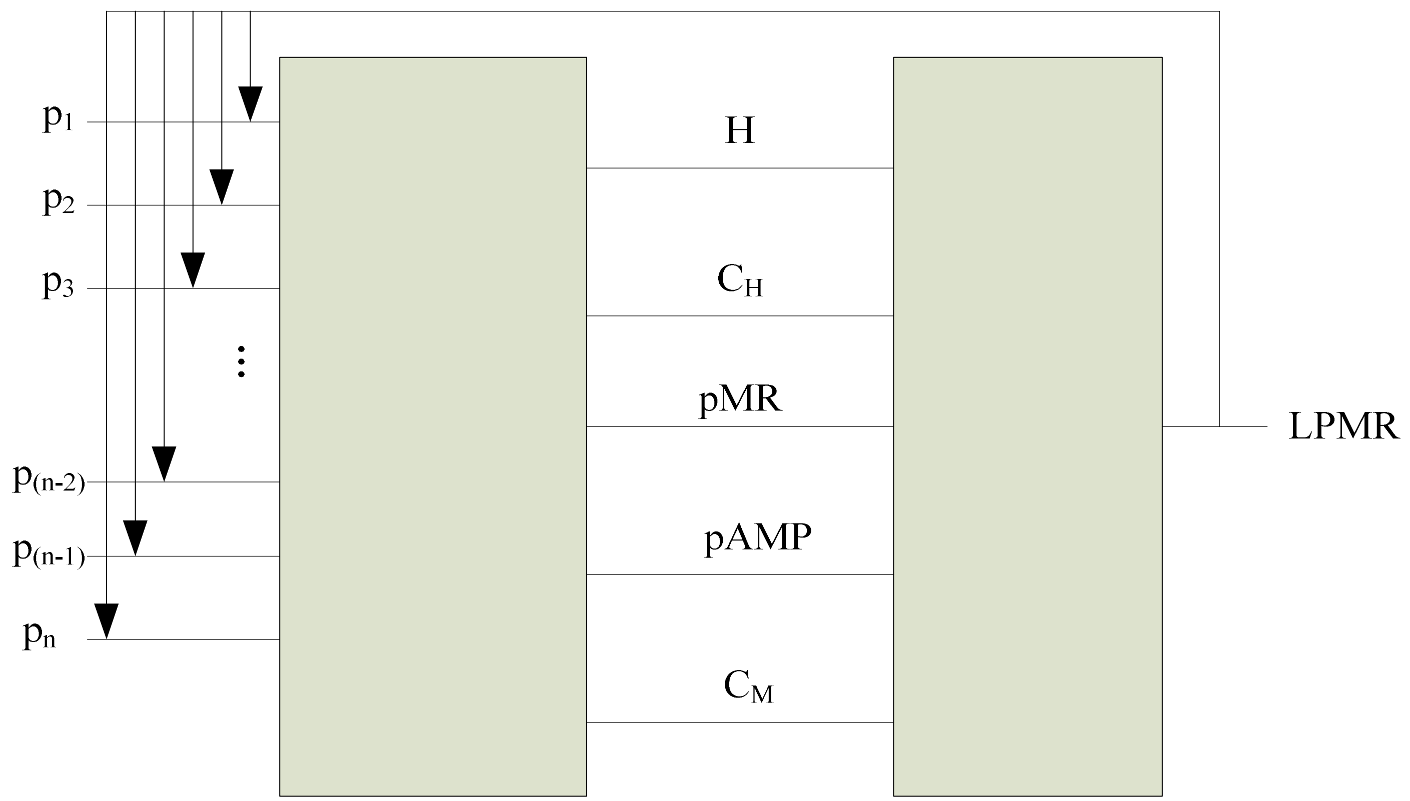
$$\kappa_1 = \frac{pMR_1}{MR_1} \times \frac{pAMP_1}{AMP_1} \times \frac{C_{m_1}}{C_{M_1}}$$

Misunderstanding of Memory Performance

Optimal = Optimal Locality

Optimal = Optimal Locality + Optimal Concurrence

Practical: Parameters can be measured in hardware



Feedback-based optimization on scheduling and on reconfigurable architecture

Challenge in Practice

- C-AMAT is general and powerful, but its measurement is environment (hardware) dependent
- Need a good understanding to conduct C-AMAT analysis in Multiccore, GPU, GPGPU, FPGA ASIC, etc. environments
- GPU is a special case of C-AMAT matching

Memory stall time: *the performance we care*

Traditional AMAT model

$$\text{CPU-time} = IC \times (CPI_{exe} + \underline{f_{mem} \times AMAT}) \times \text{Cycle-time}$$

Memory stall time

New C-AMAT model

$$\text{CPU-time} = IC \times (CPI_{exe} + \underline{f_{mem} \times \text{C-AMAT} \times (1 - overlapRatio_{c\text{-}m})}) \times \text{cycle-time}$$

Memory stall time

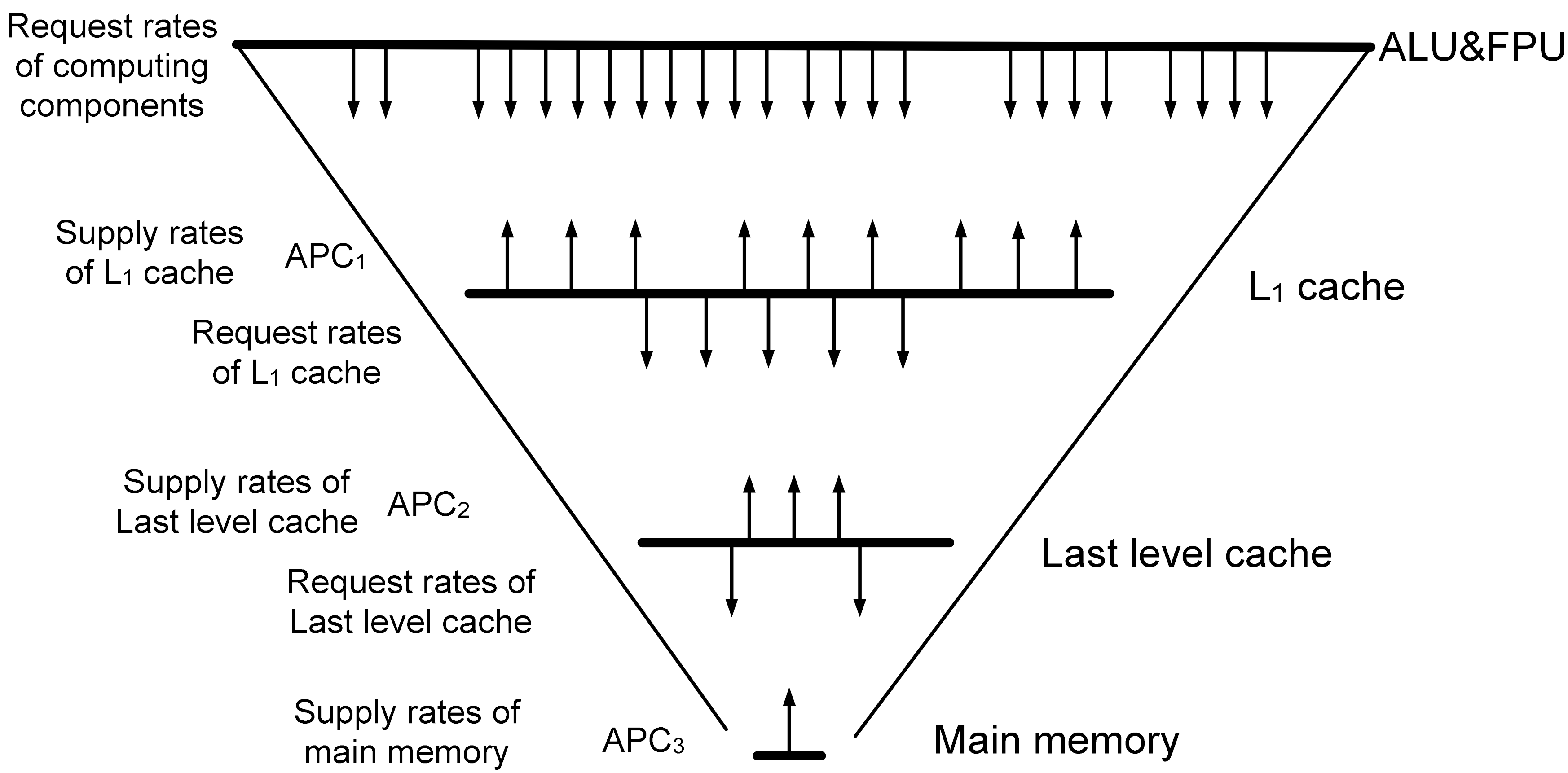
$$\text{CPU-time} = IC \times (CPI_{exe} + \underline{f_{mem} \times \frac{pMR \times pAMP}{C_M}}) \times \text{Cycle-time}$$

Memory stall time

Only pure miss will cause processor stall, and the penalty is formulated here

Y. Liu and X.-H. Sun, "*Reevaluating Data Stall Time with the Consideration of Data Access Concurrency*," Journal of Computer Science and Technology (JCST), March, 2015

Application: Layered Performance Matching



Yu-Hang Liu, Xian-He Sun, "LPM: Concurrency-driven Layered Performance Matching," in ICPP2015, Beijing, China, Sept. 2015.

Idea: Match the Request with Supply

$$LPMR(ALU\,\&\,FPU, L_1) = \frac{Request\ rate\ from\ ALU\,\&FPU}{Supply\ rate\ by\ L_1\ cache}$$

$$LPMR(L_1, LLC) = \frac{Request\ rate\ from\ L_1\ cache}{Supply\ rate\ by\ LLC}$$

$$LPMR(LLC, MM) = \frac{Request\ rate\ from\ LLC}{Supply\ rate\ by\ main\ memory}$$

- Match at **each memory layer**
- Adjust the supply performance with concurrency

Quantify Mismatching: with C-AMAT

$$LPMR_1 = \frac{IPC_{exe} \times f_{mem}}{APC_1}$$

$$LPMR_2 = \frac{IPC_{exe} \times f_{mem} \times MR_1}{APC_2}$$

$$LPMR_3 = \frac{IPC_{exe} \times f_{mem} \times MR_1 \times MR_2}{APC_3}$$

- C-AMAT measures the request and supply at each layer
- C-AMAT can increase supply with effective concurrency
- Mismatch ratio directly determines memory stall time

C-AMAT in Action

New C-AMAT model

$$CPU\text{-}time = IC \times (CPI_{exe} + \underline{f_{mem} \times \frac{pMR \times pAMP}{C_M}}) \times Cycle\text{-}time$$

Memory stall time

Only pure miss will cause processor stall, and the penalty is formulated here

The Relation of LPMR and Stall time

$$CPU\text{-}time = IC \times CPI_{exe} \times (1 + \underline{\kappa_1 \times LPMR_2}) \times Cycle\text{-}time$$

Memory stall time

Y. Liu and X.-H. Sun, "Reevaluating Data Stall Time with the Consideration of Data Access Concurrency," Journal of Computer Science and Technology (JCST), March, 2015

The LPM Model

We get the relation between data stall time and LPMR1

$$Data - stall - time = \ CPI_{exe} \ \times \ (1 - overlapRatio_{c-m}) \times LP$$

We get the relation between data stall time and LPMR2

$$Data - stall - time = CPI_{exe} \times \kappa_1 \times LPMR_2$$

Because our final goal is to minimize data stall time, the two equations above. provide the baseline for LPM optimizations

The Threshold Requirement

$$overlapRatio_{c-m} = \frac{H_1/C_{H_1}}{(H_1/C_{H_1} + pAMP_1/C_{M_1})}$$

$$LPMR_1 \leq \frac{\Delta\%}{1 - overlapRatio_{c-m}}$$

$$LPMR_2 \leq \frac{\Delta\%}{\kappa_1}$$

A Matching Example (increase performance)

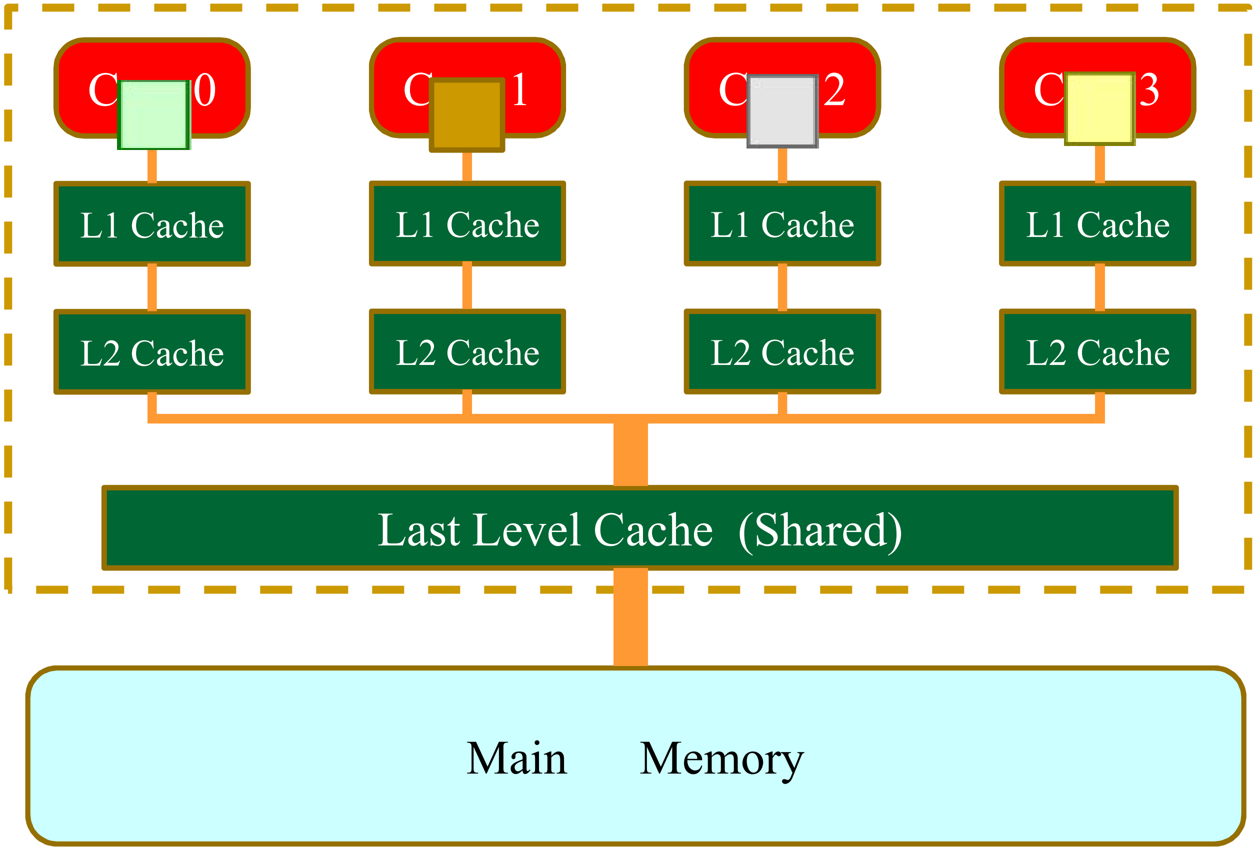
$$LPMR = \frac{IPC_{exe} \times f_{mem}}{APC} \qquad (1)$$

- **Assume**:

$$Cycle_{CPU} = 2 \text{ ns}$$
$$Cycle_{mem} = 8 \text{ ns}$$
$$f_{\text{mem}} = 20\%$$
$$IPC_{exe} = 2.5$$
$$APC = 1$$

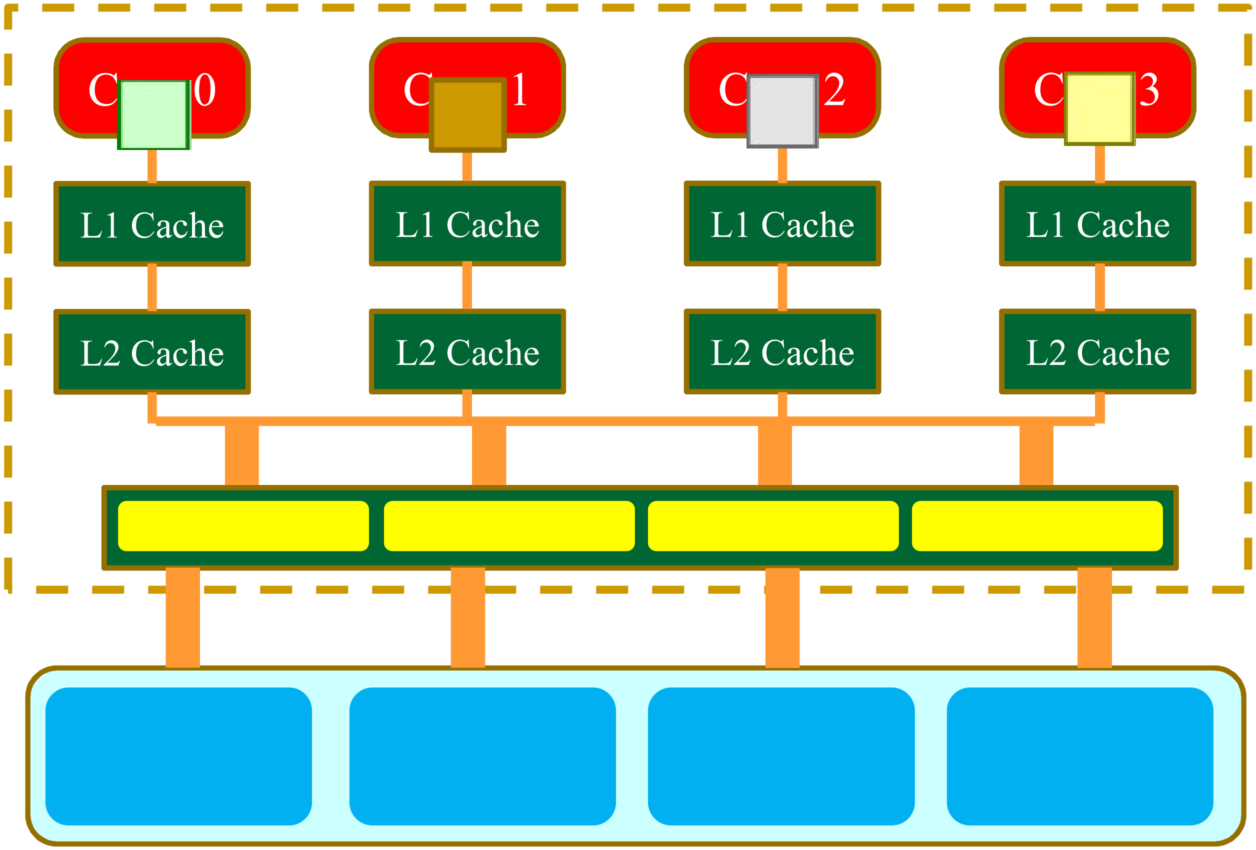
- **Then**:
  - In 8 ns, there is 1 memory cycle, and the data supply rate is:
  $$APC * N\_Cycle_{mem} = 1$$
  - In 8 ns, there is 4 cpu cycle, and the data request rate is:
  $$IPC_{exe} * N\_Cycle_{CPU} * f_{\text{mem}} = 2$$
- **So**:
  - The data supply rate does not match the data request rate. We can improve memory *concurrency* by adding memory banks or ports to increase data supply ability to adjust the data supply rate. For example, increase *APC* from 1 to 2, so:
  $$\boldsymbol{APC * N\_Cycle_{mem} = 2}$$
  - And the data supply rate matches with the data request rate.

Memory Access (Concurrency)



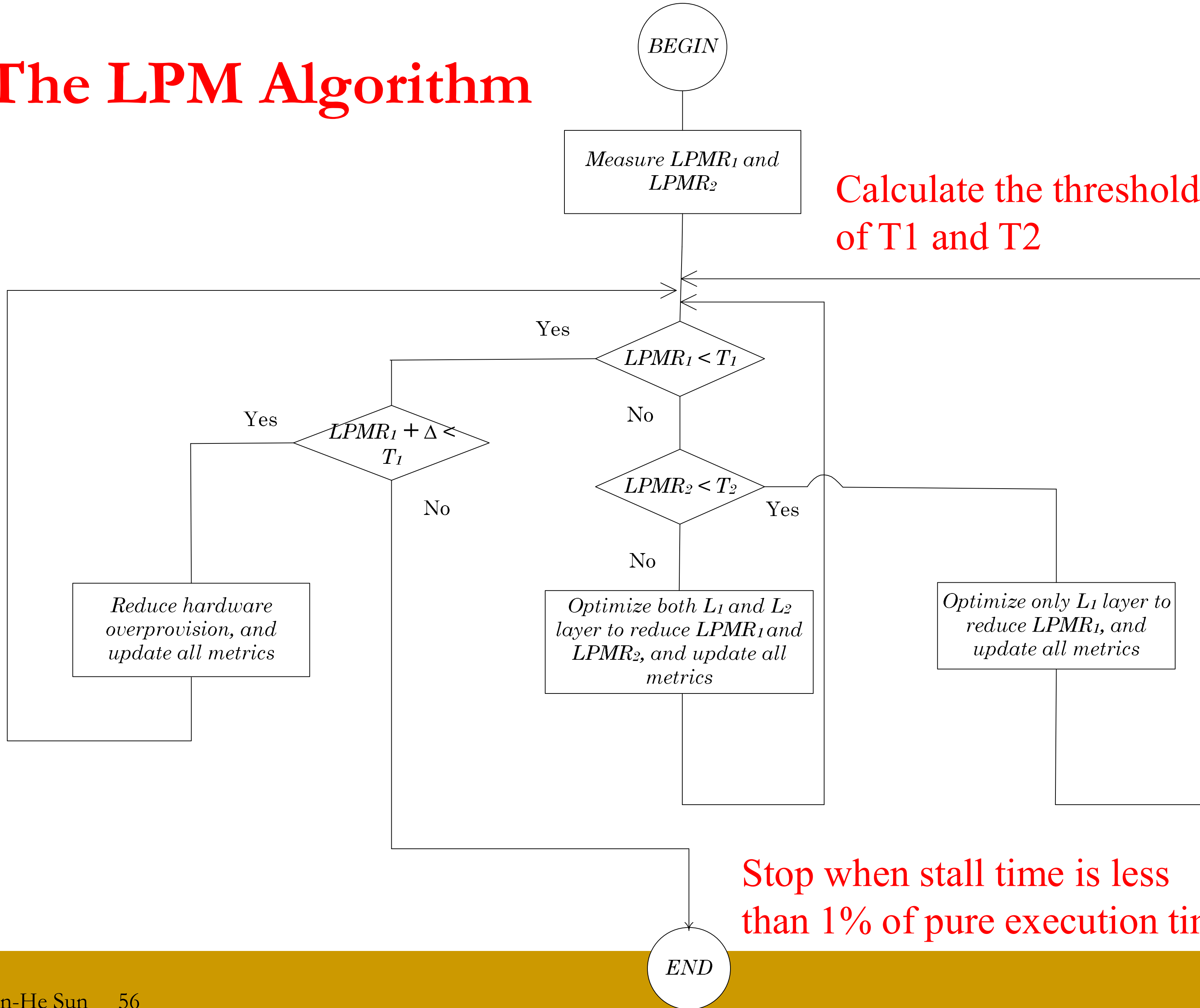
Model and algorithm

Memory Access (Concurrency)



Model and algorithm

The LPM Algorithm



Calculate the thresholds of T1 and T2

Stop when stall time is less than 1% of pure execution time

LPMR Reduction Algorithm

```
// Initially measure the metrics
For each application or thread, measure the LPMRs in a memory hierarchy
Get the threshold T_1 and T_2 according to Eq. (22) and (23)
Begin Do
  // LPM optimization loop
    // Case I when both L1 and L2 layer need an optimization
    While (LPMR_1 > T_1 and LPMR_2 > T_2) Do
        Optimizing at L1 layer and L2 layer,
        Update all the metrics (LPMR_1 , LPMR_2, T_1 and T_2)
    Until (LPMR_1 ≤ T_1 or LPMR_2 ≤ T_2)
    // Case II when only L1 layer needs an optimization
    While (LPMR_1 > T_1 and LPMR_2 ≤ T_2 ) Do
        Optimizing at L1 layer
        Update all the metrics (LPMR_1 , LPMR_2, T_1 and T_2)
    Until (LPMR_1 ≤ T_1)
    // Case III when no layer needs to optimize and overprovision may need to reduce
    // δ is a positive value
    While (LPMR_1 + δ < T_1) Do
        Reduce hardware overprovision
        Update all the metrics (LPMR_1, LPMR_2, T_1 and T_2)
    Until (LPMR_1 ≥ T_1 - δ )
    // Case IV when no layer needs to optimize and no overprovision needs to reduce
    If (T_1 ≥ LPMR_1 ≥ T_1 - δ)
        End the algorithm
    Endif
Until (End)
```

Pseudo code

Comments and Questions:

- Pipelining can be combined with concurrency for best performance: subarray
- Can you derive the general form of LPMR_i?