Chapter 2 Parallel Programming Concepts

Contents of this chapter

- Conditions of parallelism - Various dependencies
- Bernstein's conditions
- Hardware and Software parallelism
- Effect of Program and/or Data Partitioning on Processing time

- Program flow mechanisms

- Network Architectures, Properties and Routing + **MINS**

- Performance Analysis - Effect of Granularity on the Time performance in multiprocessor systems

- Parallel Programming Paradigms:
  - Shared Memory & Message Passing
  - Supportive architectures: Multi-Core & Multi-threaded architectures
  - Problem solving using MPI & Shared-memory models

Reference: Lots of material available via internet;

Specifically:

Kai Hwang & Naraesh J, Advanced Computer Architecture-Parallelism, Scalability, programmability" any edition is fine.

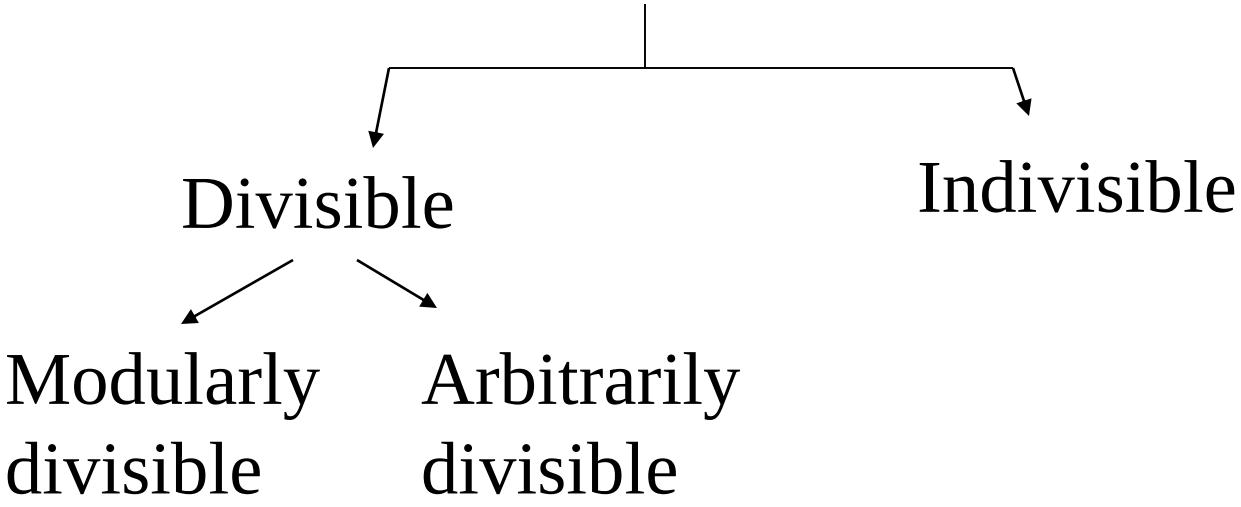
Chapter 2 – relevant contents

+ Research literature

Parallelism - Program Data

- **This chapter deals with program parallelism and some aspects of data parallelism**

Data Parallelism : DATA



Indivisible: These are types of tasks that cannot be partitioned into sub-tasks. These have to processed in their entirety. Thus, a task in its entire form must be assigned to a processor in the system.

Gantt charts are used to represent the assignment of tasks onto the processors.

Note: The tasks could range from micro-instructions to a high level task and the resources could be a processor or an execution unit inside a processor

Modularly divisible: These tasks can be partitioned to a limited extent, however, inter-dependencies must be taken care while assigning them to different processors.

Graph representation is generally used. The nodes of the graph represent sub-tasks (with or without any weights) and the edges represent the dependencies (with or without weights)

To some extent (depending on the problem size) Gantt charts can be used here too.

Arbitrarily divisible: After partitioning the data, each segment can be independently processed on any node.

There is no data dependency between the segments.

Thus, using a parallel processor set-up, we can partition the data into equal sizes and can be processed on these nodes. There are no communication delays.

However, using a loosely coupled multiprocessor system (distributed system), need to account for the non-zero communication delays and heterogeneous nature of the system.

Divisible Load theory (DLT) proposes various strategies to schedule the divisible data onto the distributed system.

Objective: Minimize the total processing time of the data subject to the communication and computation delays.

Issues: Fault-tolerance, reliability, modelling the communication and computation delays, topology

Reference: *V. Bharadwaj, et. Al (1996), "Scheduling Divisible Loads in Parallel and Distributed Systems", IEEE Computer Society, Los Alamitos, CA, USA.*

Conditions of Parallelism

Attributes of parallelism - computational granularity, Time and space complexities, communication latencies, scheduling policies, load balancing, etc.

Data dependencies

- Dependence graph - precedence relations
- Five types are identified: *Flow dependence*, *Anti-dependence*, *Output dependence*, *I/O dependence*, *Unknown dependence*

Flow dependence: A statement S2 is FD on statement S1, if an execution path exists from S1 to S2 and if at least one output of S1 feeds in as input to S2. FD is denoted as **S1 -> S2**.

Anti-dependence: Statement S2 is anti-dependent on S1 if S2 follows S1 in program order and if the output of S2 overlaps the input of S1. This is denoted as a crossed directed arrow between S1 and S2, as **S1 -/-> S2**.

Output dependence: Two statements are OD, if they produce the same output variable. A circle on the directed arrow denotes this relation as, **S1-o-> S2**.

I/O dependence: Read and write are I/O statements; I/O dependence occurs when same file is referenced by both I/O statements.

Unknown dependence: Dependence relationships cannot be determined in the following situations:

- indirect addressing
- subscript does not contain loop index variable
- a variable appears more than once with subscripts having different coefficients of the loop variable (X_{mj})
- subscript is non-linear in the loop index variable

Refer to Example 2.1 on page 52. This example verifies all the dependencies defined earlier - *A must read exercise!*

Control Dependence: When the order of the execution cannot be determined before run time, such a situation arises. Conditional statements like **IF** will not be resolved until run time.

```
Do 20 I=1,N          control independent loop
      A(I) = C(I)
      IF (A(I).LT.0) A(I)=1
20 Continue
```

```
Do 10 I=1,N          control dependent loop
        IF (A(I-1).EQ.0) A(I)=0
10 Continue
```

Control dependence often prohibits parallelism from being exploited. To circumvent this problem, compiler techniques must be developed.

Resource dependence: This is concerned with the conflicts in using shared resources, such as integer units, floating point units, registers, memory areas, etc.

- ALU dependence, Storage dependence

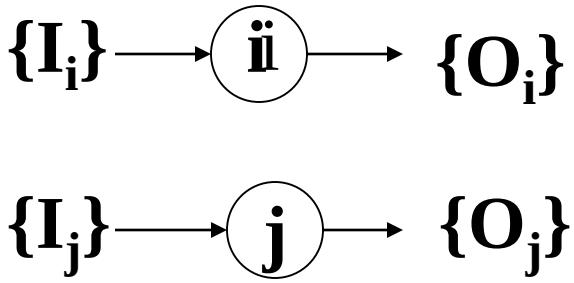
Bernstein's Conditions (1966)

These conditions are used to test whether two processes can be executed in parallel or not.

I_i: Set of input variables of a process P_i
O_i: Set of output variables

Two processes can execute in parallel, denoted as $P_i \parallel P_j$, if

$I_i \cap O_j = \phi$
$I_j \cap O_i = \phi$
$O_i \cap O_j = \phi$



These three conditions are referred to as *Bernstein's* conditions.

In terms of data dependencies, Bernstien's conditions mean that the two processes must be *flow-independent, anti-independent, output independent* for parallel execution

In general, a set of processes $P_1,\ldots,P_k$, can be executed in parallel, *if and only if* Bernstein's conditions are satisfied pairwise, i.e., for all $i \neq j$, P_i *and* P_j *must be executed in parallel.*

Refer to an example on page 55 - Example 2.2 - Detection of parallelism in a program using Bernstein's condition.

Trace the example carefully to understand figures 2.2 (a) and (b) on page 56

In general, violations of any one or more of the 3n(n-1)/2 Bernstein's conditions among n processes prohibits parallelism collectively or partially.

In general, data dependency prohibits parallelism.

Some properties of || relation:

- In general, || relation is commutative
- || relation is not transitive
  (See example: (P1 || P5) and (P5 || P2), but P1 and P2 are not independent

The above two implies, || is not an equivalence relation

- Associativity property holds
- $I_i \cap I_j \neq \phi$ does not prevent parallelism to be expolited

Hardware Parallelism(HP)

- Defined and decided by the machine architecture
- Function of cost and performance trade-offs

How do we characterize the parallelism?

One way is by the number of instruction issues per machine cycle.

If a processor issues k instructions per machine cycle, then it is called as a *k-issue* processor.

A conventional processor takes one or two cycles for issuing a single instruction, and these are referred to as *one-issue* processor.

Examples of some earlier architectures:

1). Intel i960CA is a three-issue processor (1980s)
2). IBM RISC/System 6000 is a four-issue processor (1990)

Note: A multiprocessor system built with an ***n k-issue*** processors should be able to handle a maximum of ***nk*** threads of instructions simultaneously.

Software parallelism(SP)

This is defined entirely by the *control and data* dependence of programs.

The degree of parallelism is revealed via dependence flow graph.

SP is a function of algorithm, programming style, and compiler optimization.

Refer to an example in the scanned slides doc to understand how HP and SP are determined.

Software parallelism :

* Control parallelism
* Data parallelism

Control Parallelism: Two or more operations in parallel pipelining is an example; Handled largely by hardware

Data Parallelism: Used in MIMD and SIMD systems; parallelism is exploited in direct proportion to the amount of data available.

To solve the mismatch problem between h/w and s/w parallelism, compiler support is very much essential.

Effect of Data/Program Partitioning on Processing Time

Grain size / Granularity: Measure of the amount of computation involved in a process.

Sometimes, it is also referred to as the size of the task that can be assigned onto a a processor for execution.

The simplest measure is to count the number of instructions in a grain. Grain size also refers to the program segment that can be executed in parallel.

- Fine grain, medium grain, and coarse grain

Latency: This is a time measure of the communication overhead incurred between machine subsystems.

Examples: memory latency, synchronization latency

Levels of Parallelism

- Instruction level - *fine grain size*
- Loop level - *fine grain size*
- Procedure level - *medium grain size*
- Subprogram level - *medium/coarse grain size*
- Job level - *coarse level*

Instruction level: (***Slides 24 – 26 Reading Assignment***)

- Depending on individual programs, we can have fine grain size between 2 to 1000's in number.
- Assisted largely by compiler (implicit parallelism)

Loop level:

- Typical loop has 100's of instructions.
- If successive loop iterations are control independent, then they can be vectorized for pipeline execution or can be executed on a SIMD machine in a lock-step fashion
- Most optimized program construct to execute on a parallel or vector computer

Procedure level:

- This is at the medium grain size level; less than 2000 instr.
- SIMD is less common here
- Multitasking belongs to this category
- Cumbersome task to extract the dependencies at this level; lots of effort from the programmer is expected to restructure/organize the program

Subprogram level:

- Typical of exploiting parallelism by the algorithm designers and no compiler support is provided

Job level:

- Independent jobs on different processors
- Coarse grain size level
- Time-sharing and space sharing is exploited at this level;

Note: Fine grain provides a higher degree of parallelism however, the penalty is the higher communication overhead

Refer to Fig. 2.5 on page 62 for the hierarchy

Communication latency : This is the latency incurred with the inter-processor communication. IPC is also affected by the communication patterns besides the delay in the path.

In general, n tasks communicating with each other may require n(n-1)/2 links among them, which implies that the communication latency grows *quadratically*. This naturally sets a bound on the number of processors that can be used in the system.

- Broadcasting, multicasting are some common types

Grain packing and scheduling

- *If a program is partitioned into parallel branches, modules, microtasks or grains, and scheduled for execution, what will be the effect of its total finish(execution) time?*

- *Can we determine the optimal sizes of the concurrent grains in a computation?*

One of the early demonstrations – Kruatrachue and Lewis (KL, 1988) approach for grain packing and scheduling for parallel programming

Refer to Example 2.4 and Example 2.5; Pages 64-70

Program flow mechanisms

Network Architectures, Properties and Routing

Several computers can form a network interconnected via a medium and this interconnection can assume any specific or an arbitrary topology.

- Static and dynamic networks

Some common terminology and definitions :

1. ***Distance (u,v)*** specifies the distance between two nodes u and v in some metric that depends on the problem.

Example: The distance metric could be the shortest path between the nodes u and v

Note: In the rest of the definitions, we shall assume that the distance(u,v) is the shortest path between two nodes u and v, for the ease of understanding

2. Two paths between u and v are said to be **node-disjoint** if they have no other common nodes

3. Diameter (d) of a network is defined as,

d = max{Distance(u,v)}, where the max is determined over all pairs u,v $\in$ V

It is a measure of the network performance in terms of worst-case communication delay.

4. Degree, *deg(u)* of a node u is the number of links incident on it. (in-degree and out-degree in a directed G)

If deg(u) = δ, for all the nodes in the network, then the network graph is called as *δ-regular*.

5. For a regular network, the **cost** is defined as C = d.δ

6. ***Packing density*** of a network is defined as the ratio of the number of nodes to its cost. (nodes per unit cost)

Thus, higher the packing density, smaller the chip area required for its VLSI layout.

7. ***Node-connectivity (K)*** is the number of nodes whose removal results in a disconnected network.

Node-connectivity is a measure of fault-tolerance.

8. *f-fault diameter* of a network is defined as its worst-case diameter by removing at most ***f*** nodes

<u>Remarks</u>: You can verify the above-mentioned quantities for some standard graphs - mesh, hypercube, ring, etc

---

Multistage Interconnection Networks (MINS)

• Exclusively used to interconnect a set of processors and memory modules - Various types (CLOS, baseline, omega)

Note: This portion of the material is my own compilation. ***Follow the slides presented in the class. You can get these from Canvas!***

If time permits, we will perform *blocking probability computations*

For Exams: It is sufficient if you go through my slides for this portion of the material that is discussed in the class.

Effect of Granularity on the Time Performance in Multiprocessor Systems

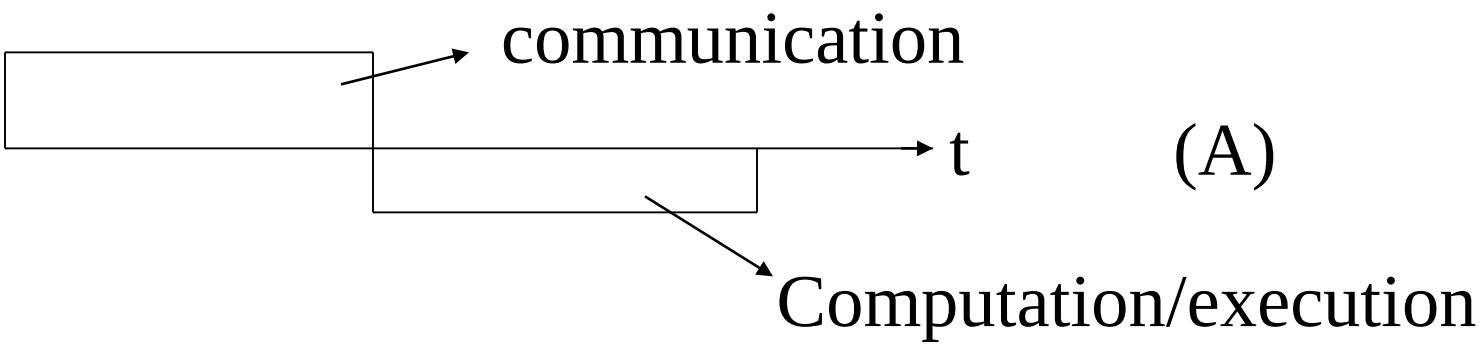
R/C is a measure of the granularity
R : length of the run time quantum **C**: Overhead associated with the communication for that quantum

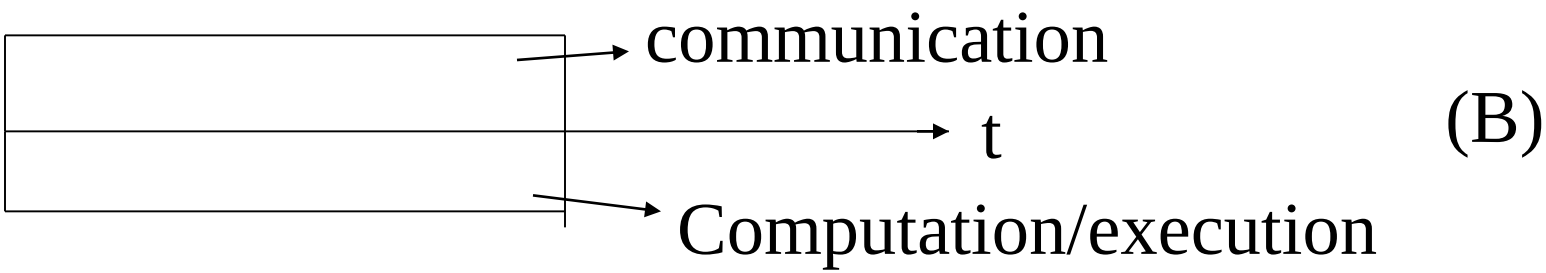
Coarse grain : **R/C** is high Fine grain : **R/C** is low

We will present two different models to analyze the performance of the multiprocessor systems including the granularity of the tasks

Assumptions : (1). Each task executes in R units of time
(2). Each task communicates with every other task with an overhead of C units, whenever the communicating tasks reside on different processors

Communication and Computation can be in a time overlapped fashion. We will discuss on (A). Non-overlapped case and (B). Overlapped case. We consider a two processor system first.





(A). Non-Overlapped Case: The performance of this case can be described by the following equation.

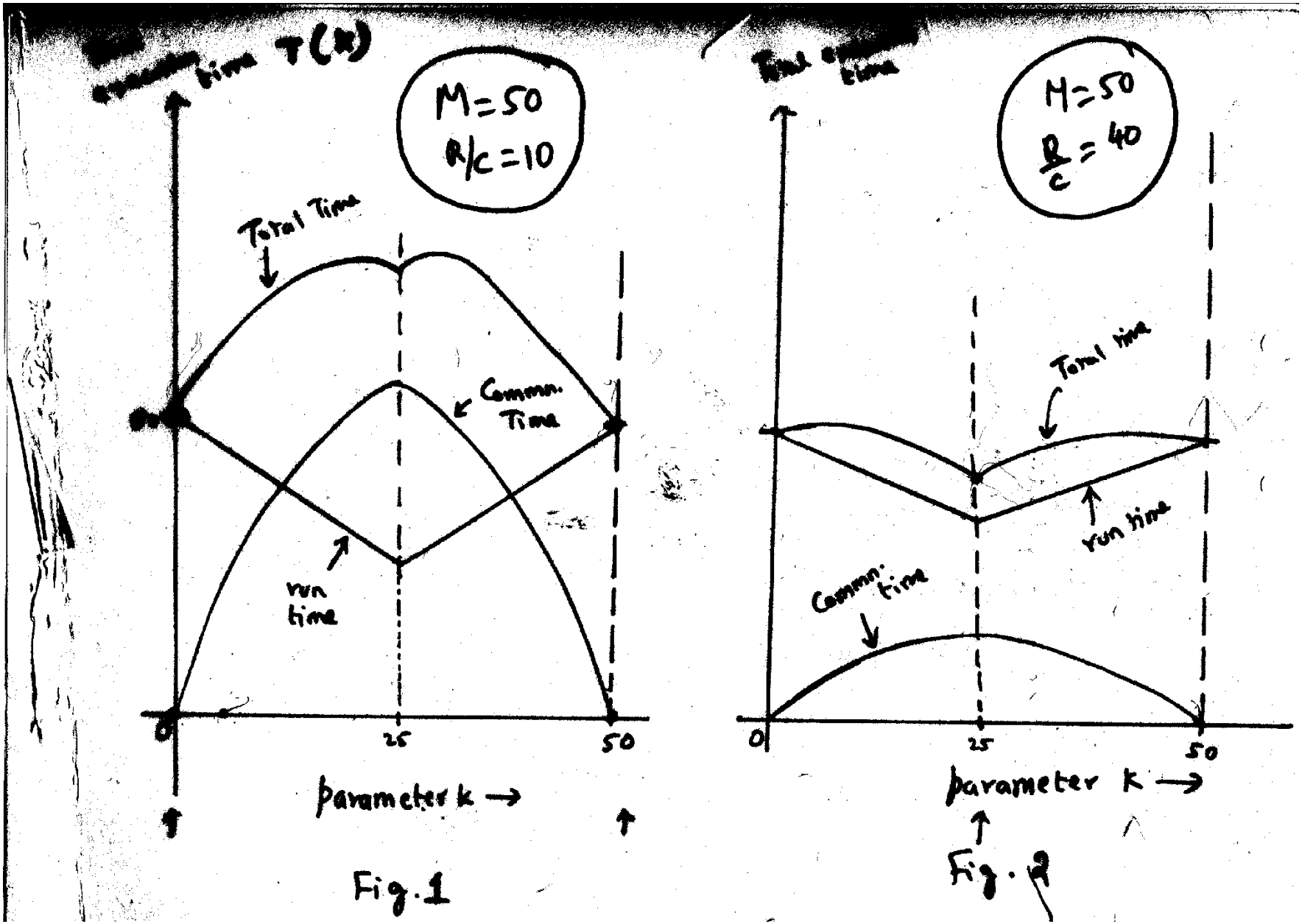
Execution Time : $T(k) = R\ \mathbf{Max}\{M-k,k\} + C\ (M-k)k \quad (1)$

Equation (1) describes the total time to complete M tasks on a system with two processors. The first term describes the computation time when **k** tasks are scheduled on one processor and the rest of the (**M-k**) are scheduled on a different processor. Note that the first term is a linear function of **k**. The second term describes the effect of communication between the processors. Note that this is a quadratic term due to the fact that every task communicates with every other task, i.e., accounting the number of pair-wise communications.

It is interesting to investigate on the behavior of the execution time as a function of **k** and also to determine an **optimal k** that gives the minimum execution time. Solution to this problem can be obtained by analyzing the time performance as shown in the figures.

From the figures, we observe the following. In Fig. 1 (next slide) we see that the *communication overheads* are much more severe than in Fig.2.

First term in (1) is linear function of **k**, and is symmetrical about the point **k=M/2**. When this linear term is added to the quadratic term, the resulting curve has a minimum at **M/2** in Fig. 2 and has a minimum at **k=0** (k=M)in Fig. 1.



What we see here is that, whenever **R/C < M/2**, the minimum occurs at the value **k=0**, implies that it is *optimal* to schedule all the M tasks on a single processor. On the other hand, whenever **R/C >= M/2**, the minimum occurs at **k=M/2**, implies that assigning equal loads to the processors is *optimal*.

The same line of argument can be applied for the ***N*** processors case as follows.

When there are N processors in the system, we have the following equation describing the behavior of the execution time.

Execution time $T(k) = R \max_{i} \{k[i]\} + (C/2) \left(\sum_{i} k[i](M-k[i])\right)$ (2)

where the maximization is over all i=1,…,M. Simplifying, we have,

$$T(k) = R \max_i(k[i]) + (C/2)\,(M^2 - \sum_i k[i]k[i]) \qquad (3)$$

where we allow only distinct pair-wise communications between tasks

Note that the first term accounts for the maximum computation time among all the N processors.

The second term accounts for a distinct pair-wise communication interaction among the tasks, each contributing an overhead of C units.

We observe that the same rationale for the case of two processors also hold in this case.

What is more interesting is the case when **R/C** value is more than **M/2**. Note that the case when **R/C < M/2** is trivial and needs a single processor for optimality. Now, considering the difference in the total execution times with N processors and a single processor, for **R/C >= M**/2 and for very large **N**, we have,

$$T_{diff} = [\ R*M/N + C*M^2/2 - C*M^2/(2*N)\ ] - R*M \qquad (4)$$

We have assumed that M is an integral multiple of **N**. Now, to solve for the threshold value of R/C, we equate (4) to zero, and obtain, **R/C = M/2**. This shows that, if **R/C >= M/2**, then an even distribution of tasks to as many processors available will fetch optimal finish time. On the other hand, if **R/C < M/2**, no matter how large **N** is, scheduling all the **M** tasks on a single processor will fetch the optimal finish time.

As the performance is usually determined by the ***speed-up*** factor, in our case, we have,

Divide the Nr & Dr by C & bring N to the Nr;

$$\underline{Speed\text{-}up} = R*M / (RM/N + C*M^2/2 - CM^2/2N)$$
$$= (RN/C) / (R/C + M(N-1)/2) \qquad (5)$$

(Hint : Use the definition of speed-up and the fact that N is very large)

Discussion

Now, if **R/C >> M*(N-1)/2**, then we achieve a speed-up proportional to N. This means that we need **M** and **N** to be as small as possible for **R/C** to be too large. Suppose, if we need more parallelism with a large **N**, the second term in the denominator is larger than the first term, the speed-up is proportional to **R/(C*M),** which does not depend on the number of processors. Hence, as **N** increases, the speed-up asymptotically approaches this value. At this point, each processor added to the system is of no benefit!

Thus, the so far analysis has demonstrated the effect of granularity and the overhead factor on the time performance. It shows the importance of choosing the right granularity which minimizes the total 'cost' of the system. One can use a variety of models to analyze the multiprocessor performance. We will now consider the case when we have a time overlapped computation and communication operations.

(B). Overlapped Case : This is an *optimistic* model in which the communication phase is completely masked by the computation. To realize this, we have the communication term to be overlapped with the computation term as much as possible. We describe such a behavior as,

$$\text{Execution Time} = \mathbf{Max}_{i} \{ R \max(k[i]), (C/2)\ k[i](M-k[i]) \} \qquad (6)$$

Time-overlapped model

•Analyze the performance for a two-processor system for the overlapped case

• Suppose if have a model in which the cost of communication is proportional to the number of processors, and not to the number of tasks assigned remotely. How will be the performance?

Execution time = R Max { k_i} + C N

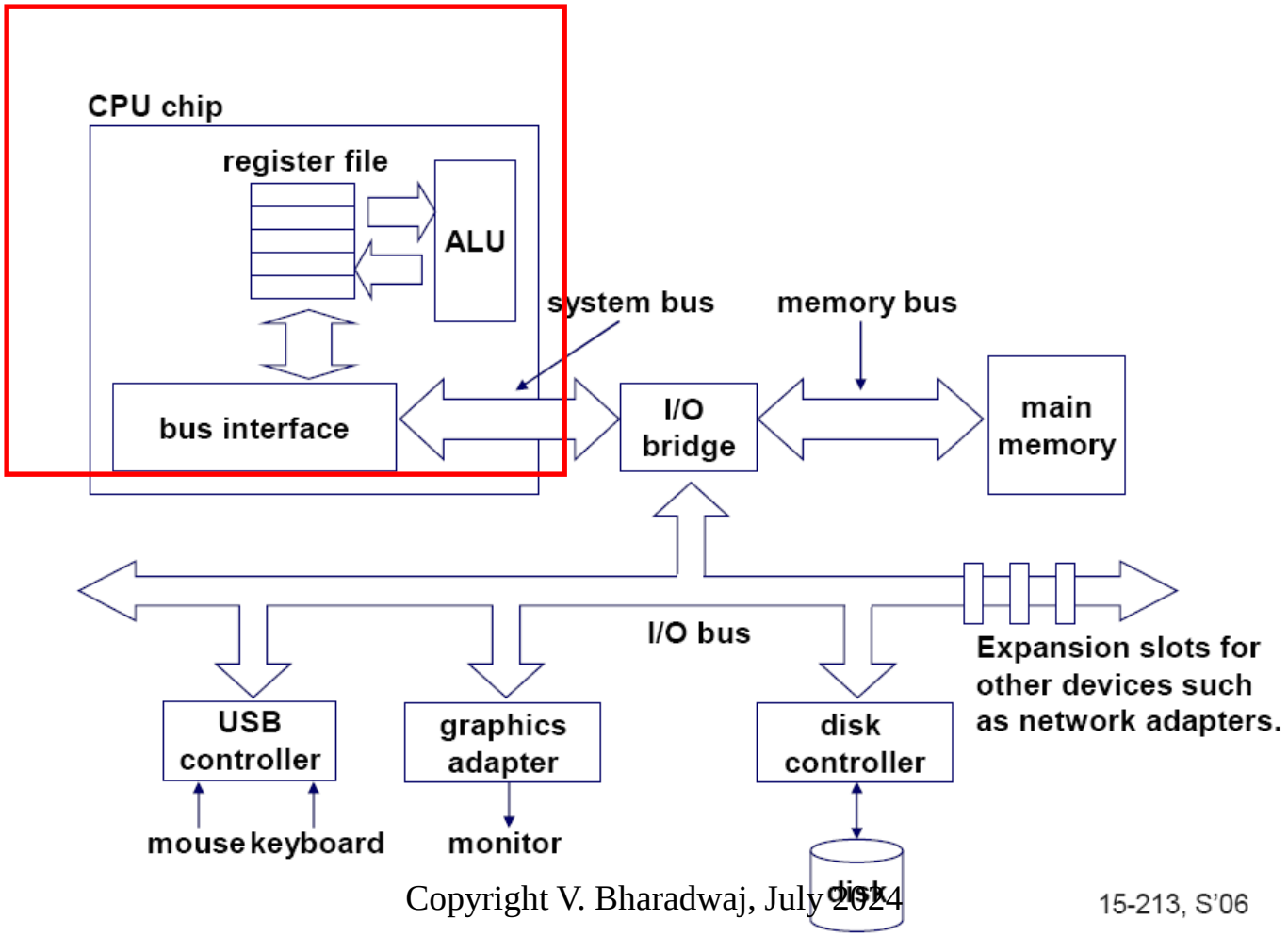
Describe the performance when more processors are added to the system.

Parallel Programming Paradigms

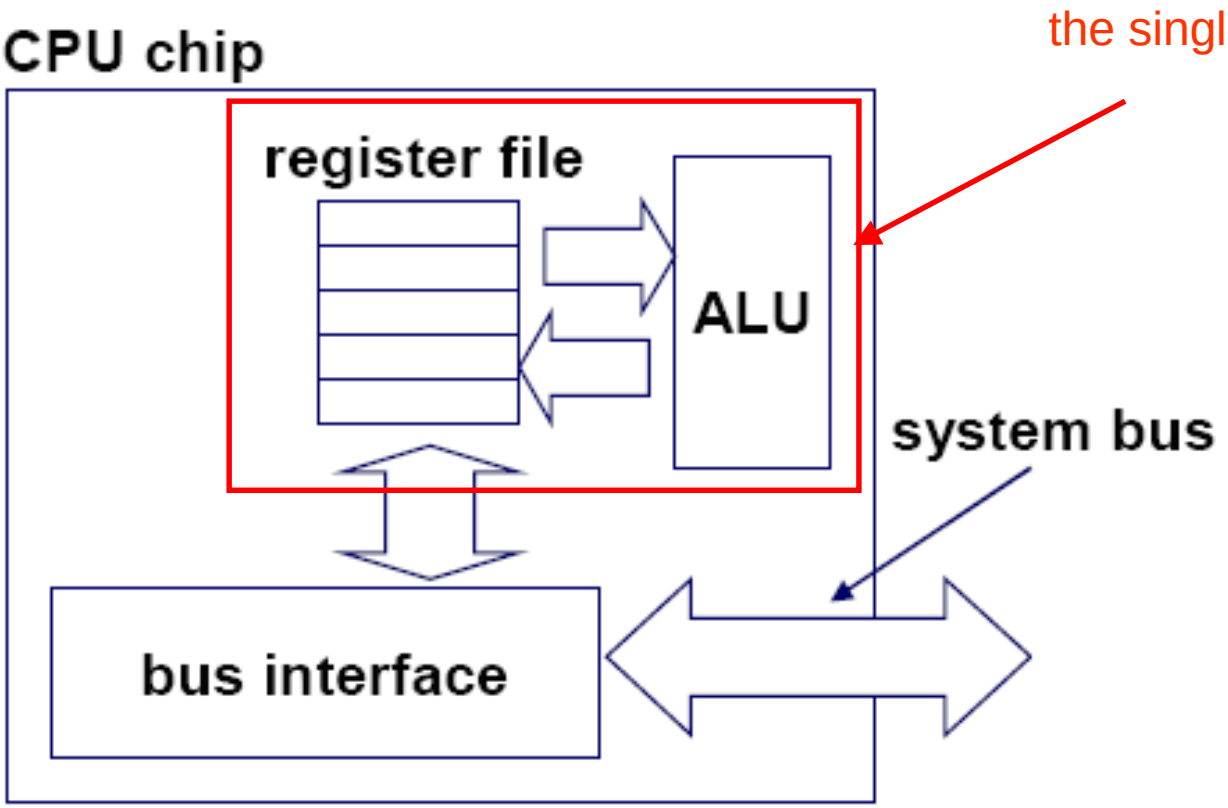
Recall Concurrency vs Parallelism from Chap 1

Supportive architectures (single and multi-core)

Single-core computer

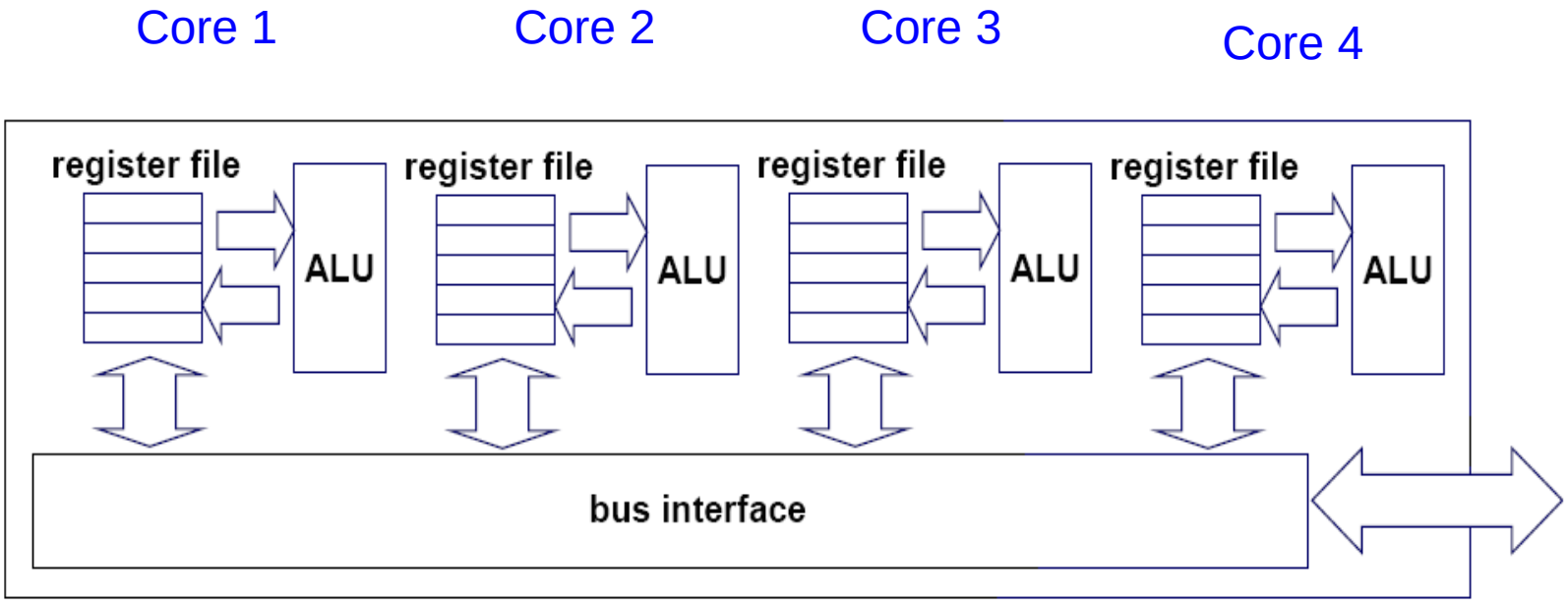


Single-core CPU chip



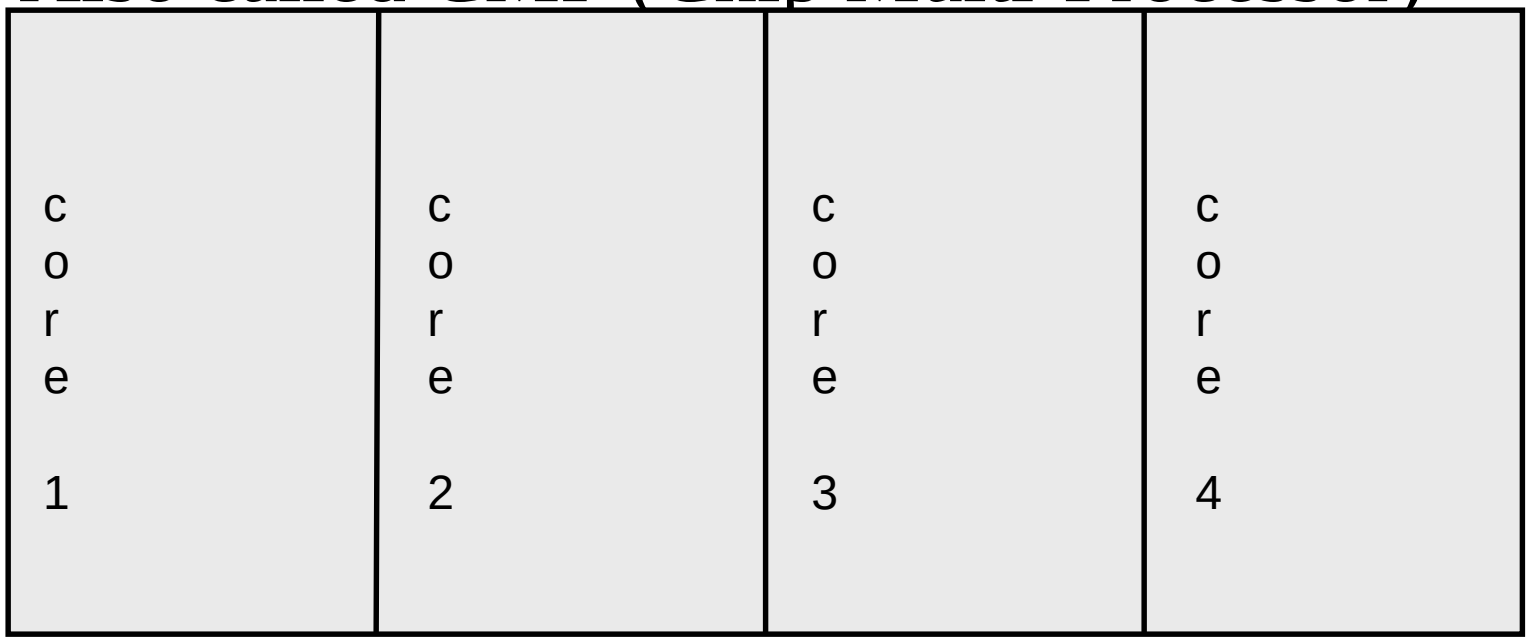
Multi-core architectures

- Replicate multiple processor cores on a single die.

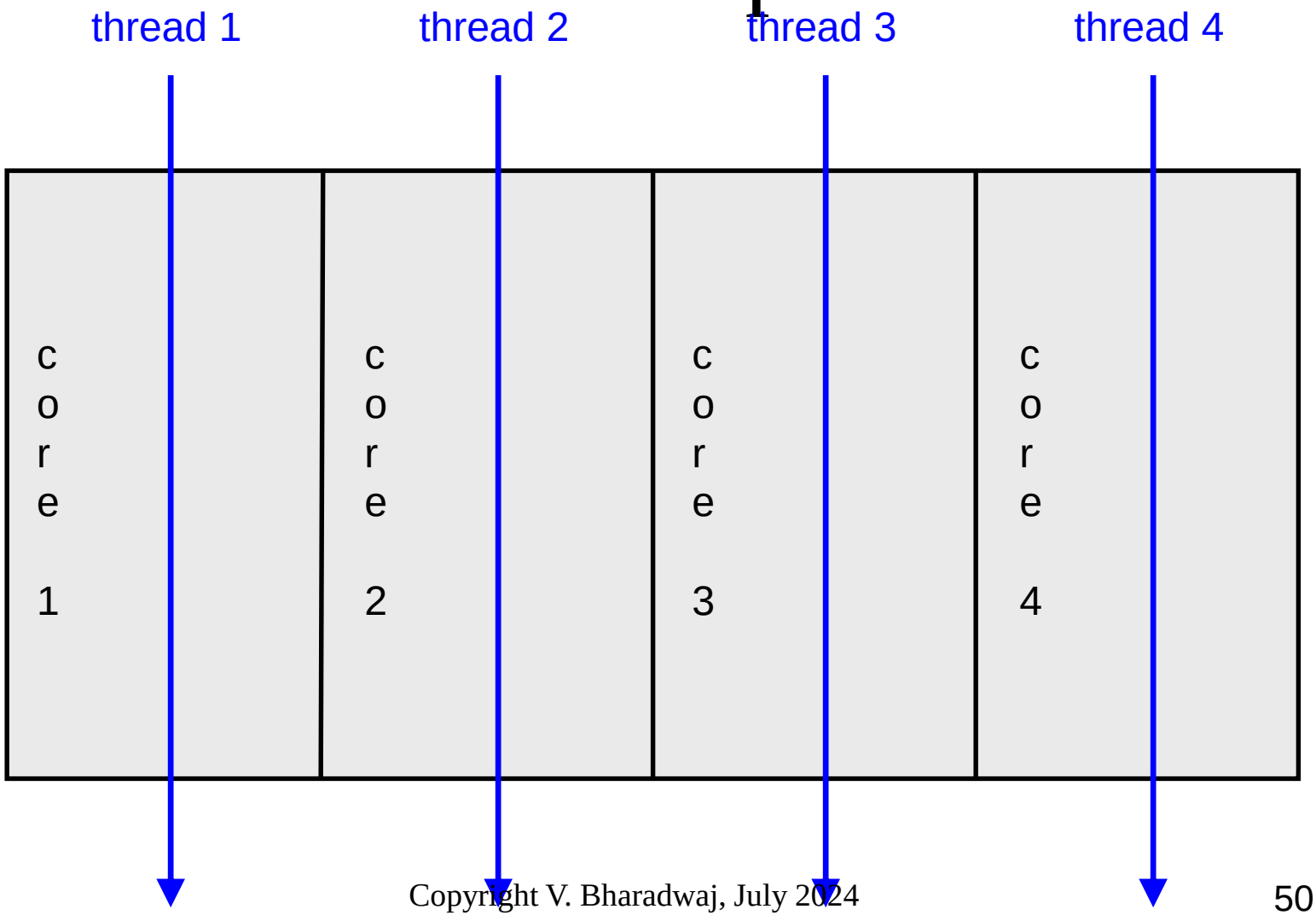


Multi-core CPU chip

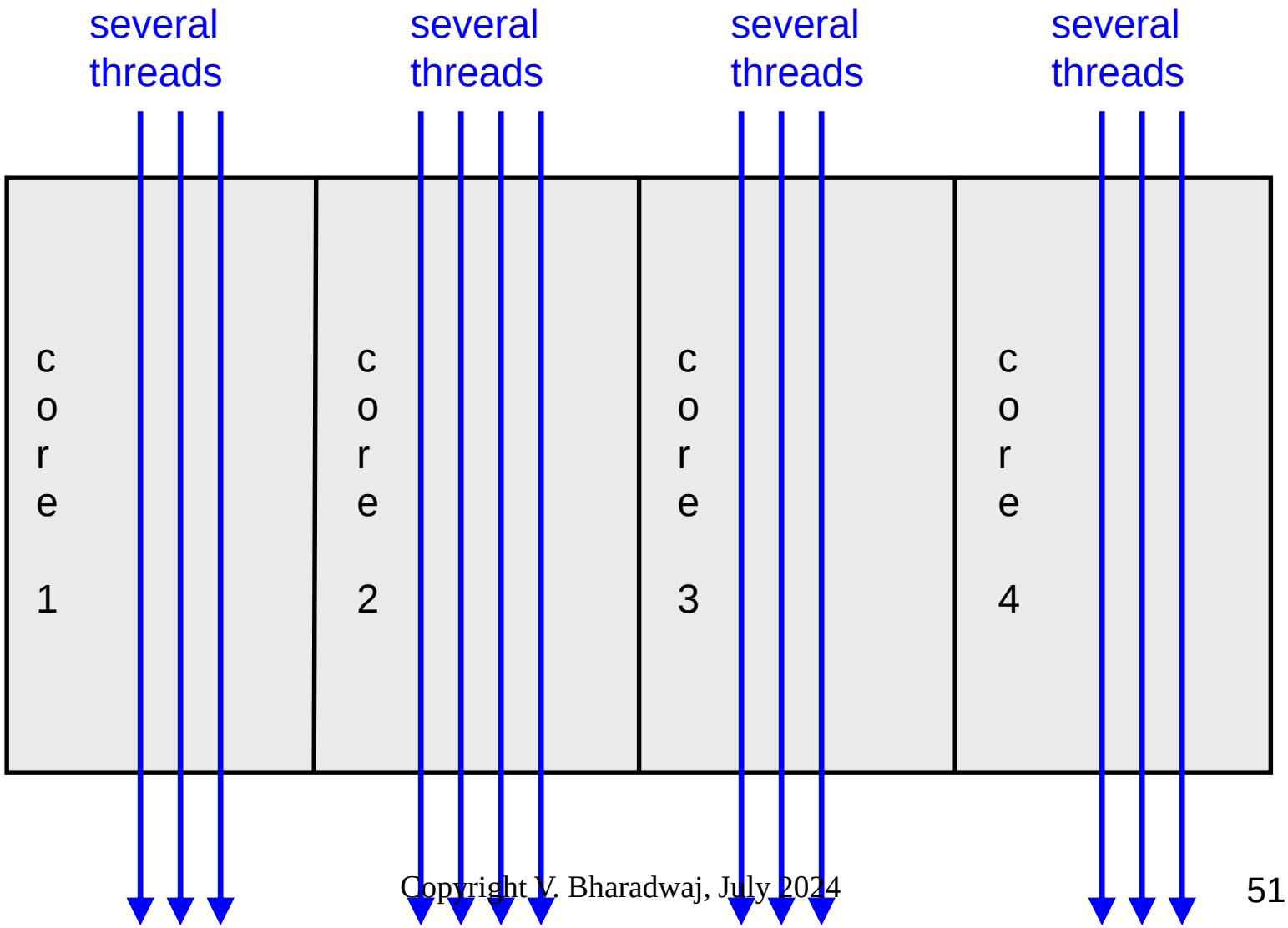
- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



Cores run in parallel



Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction with the Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today: Windows, Linux, Mac OS X, …

Multi-core - Instruction-level parallelism

- Parallelism at the machine-instruction level
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

Thread-level parallelism (TLP)

- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures: explicitly exploiting TLP

Multiprocessor memory types

- Shared memory:
  In this model, there is one (large) common shared memory for all processors

- Distributed memory:
  In this model, each processor has its own (small) local memory, and its content is not replicated anywhere else

Multi-core processor is a special kind of a multiprocessor:

All processors are on the same chip

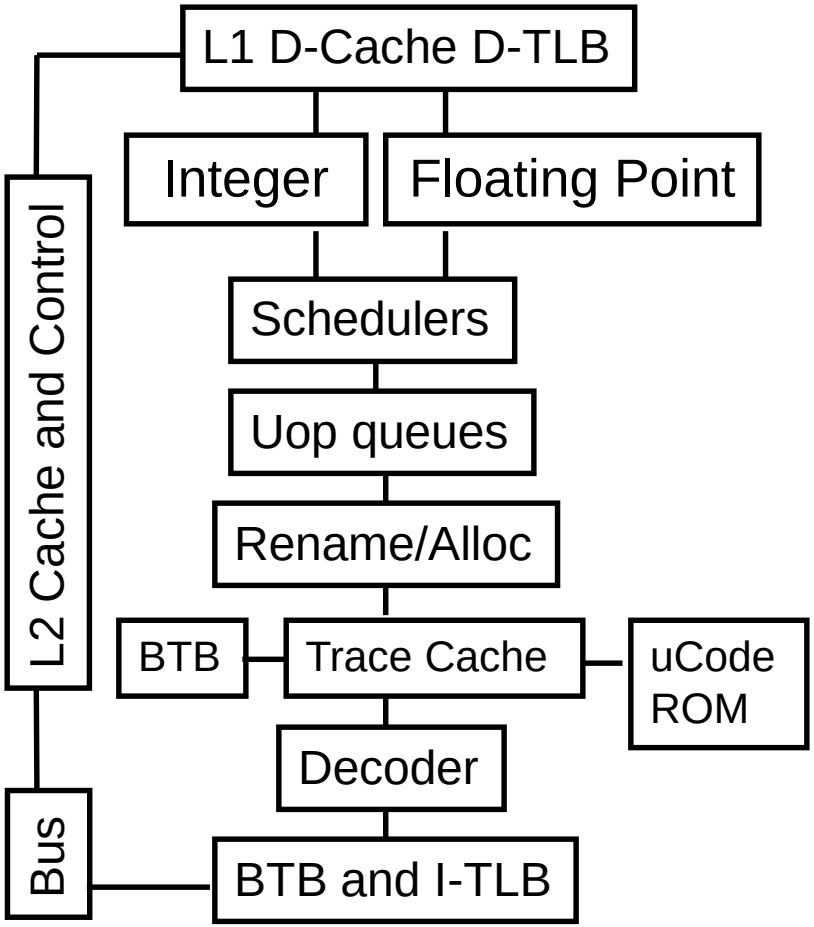
- Multi-core processors are MIMD:
  Different cores execute different threads (Multiple Instructions), operating on different parts of memory (Multiple Data).

- Multi-core is a shared memory multiprocessor:
  All cores share the same memory

A technique complementary to multi-core: Simultaneous Multi-Threading (SMT)

Processor pipeline can get stalled: *Why?*

- Waiting for the result of a long floating point (or integer) operation
- Waiting for data to arrive from memory
- Other execution units wait;



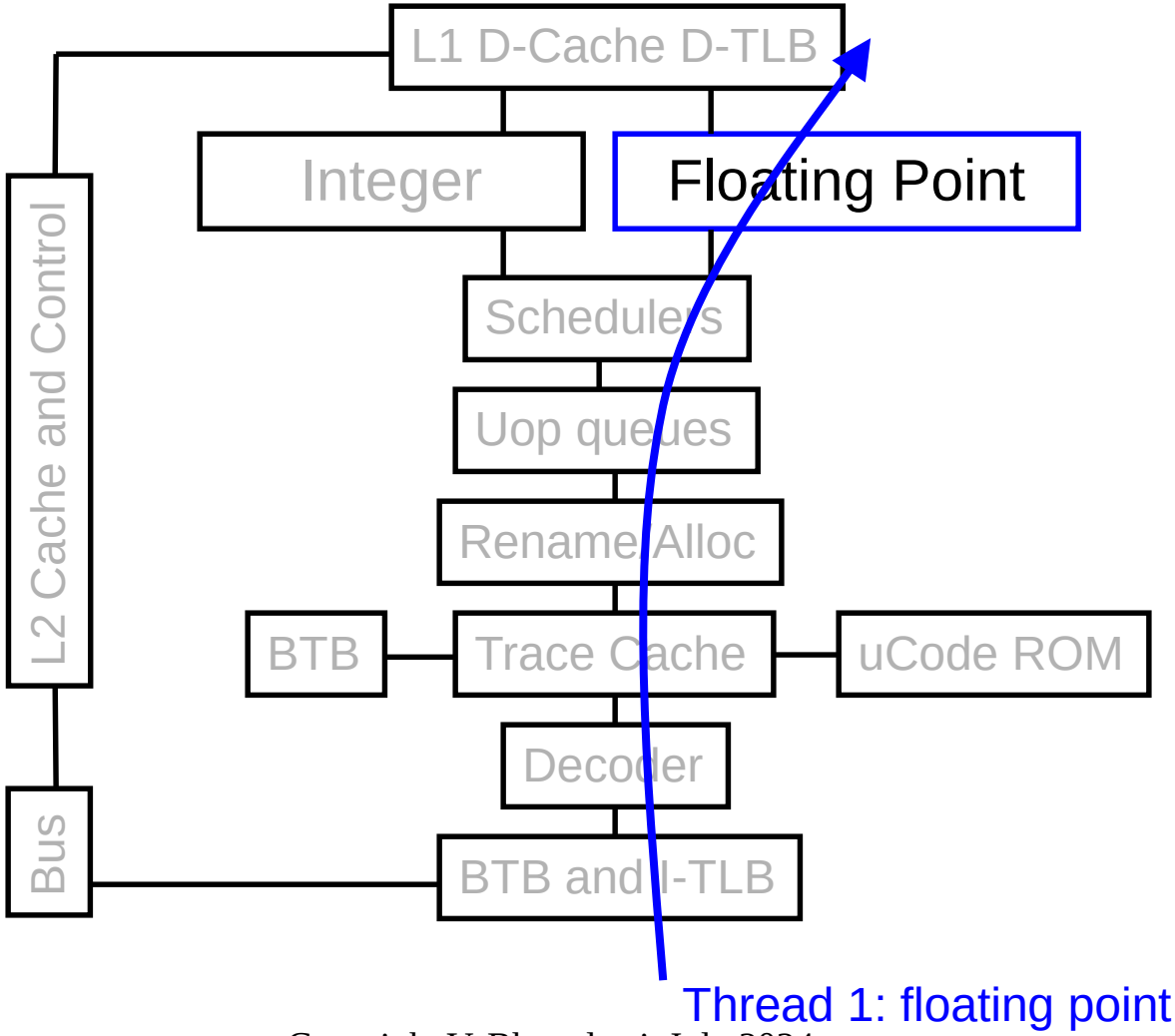
Source: Intel

Simultaneous multithreading (SMT)

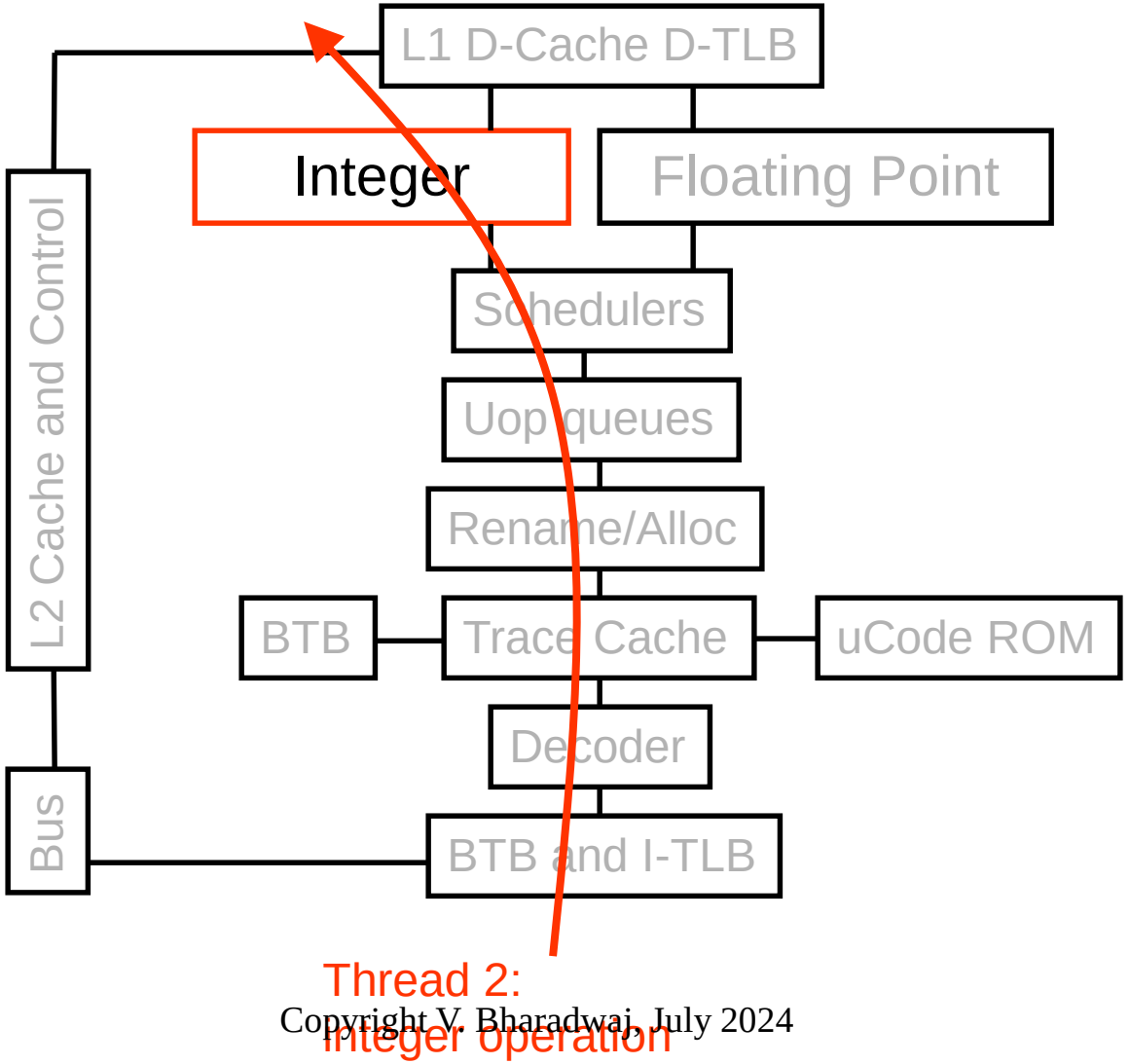
- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple "threads" on the same core

- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

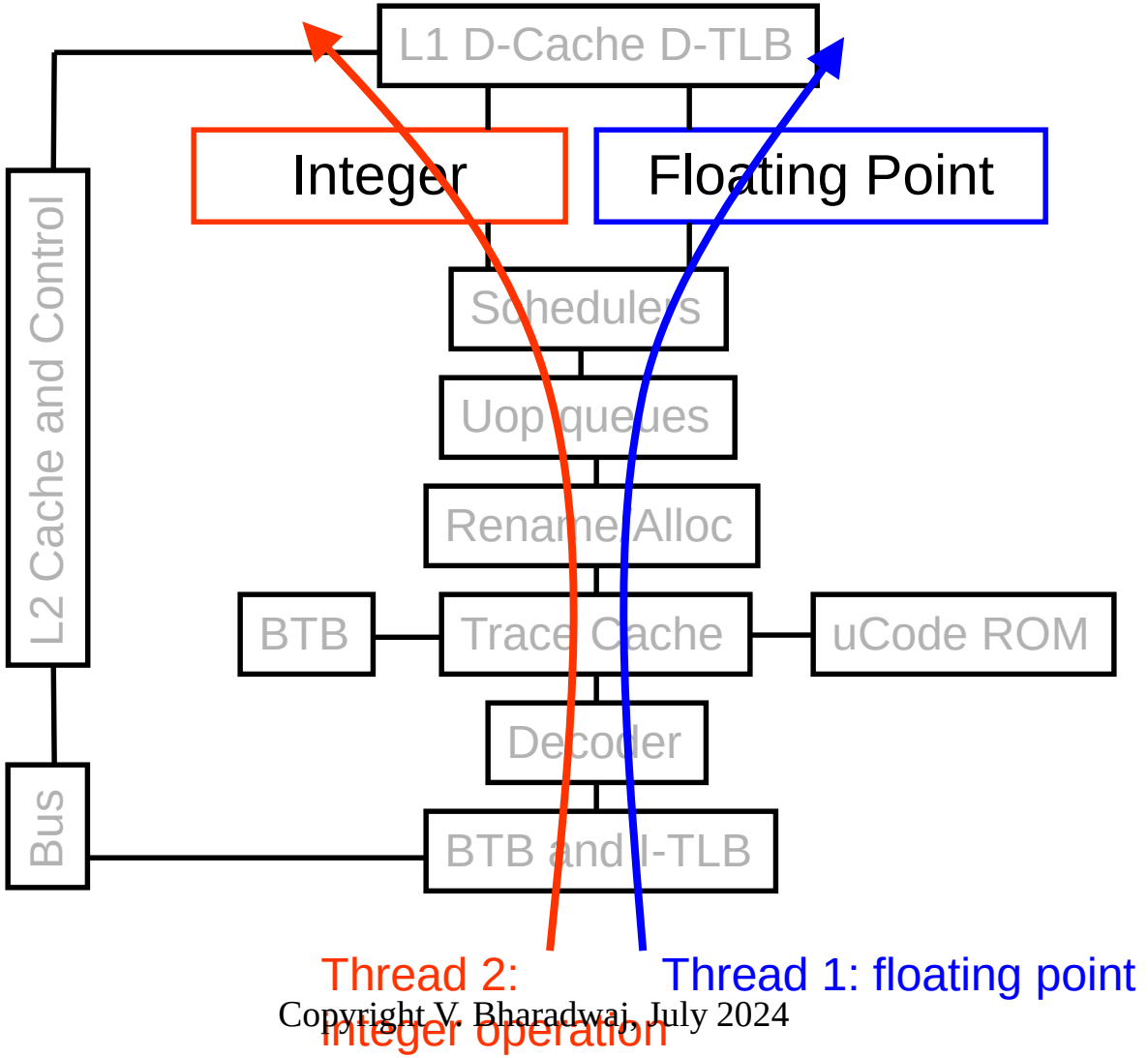
Without SMT, only a single thread can run at any given time



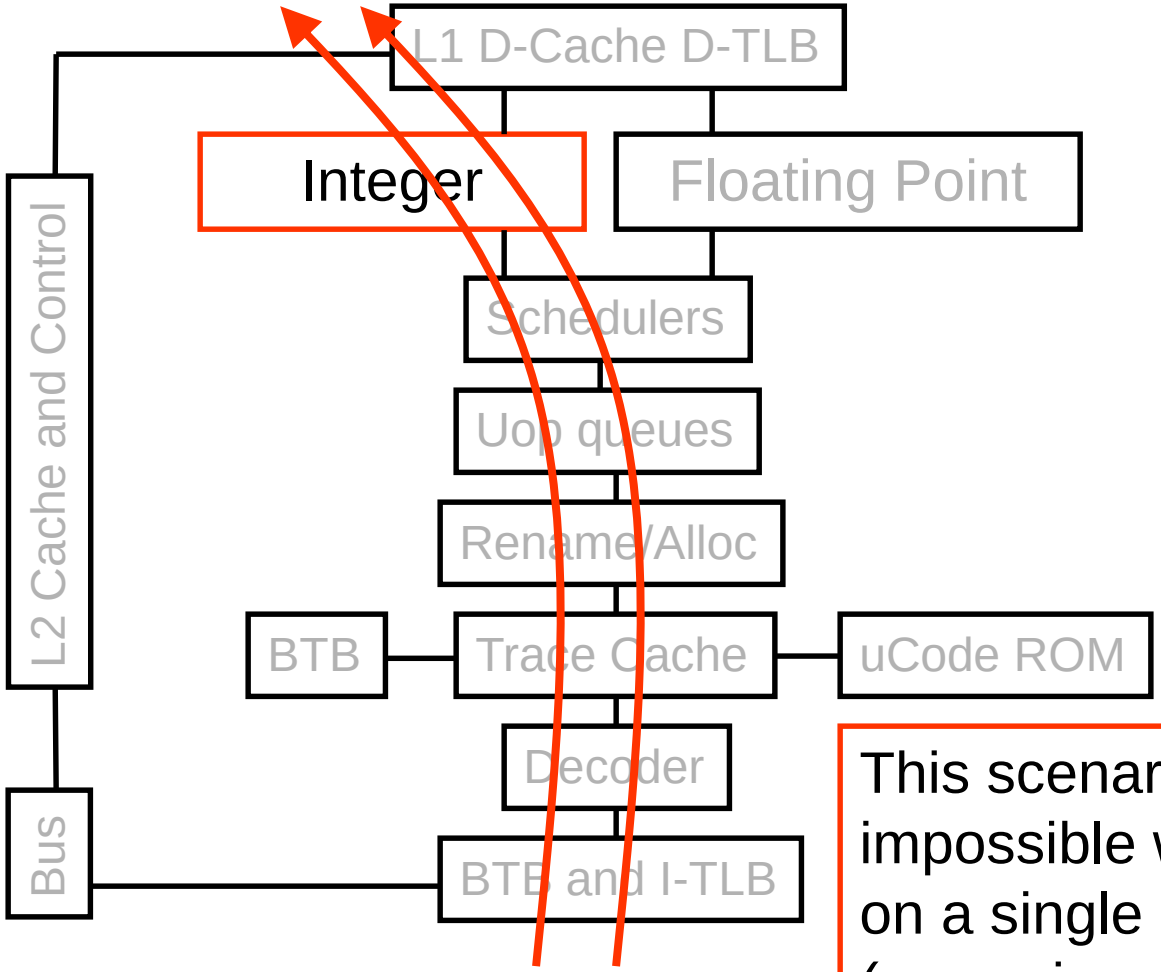
Without SMT, only a single thread can run at any given time



SMT processor: both threads can run concurrently



But can't simultaneously use the same functional unit

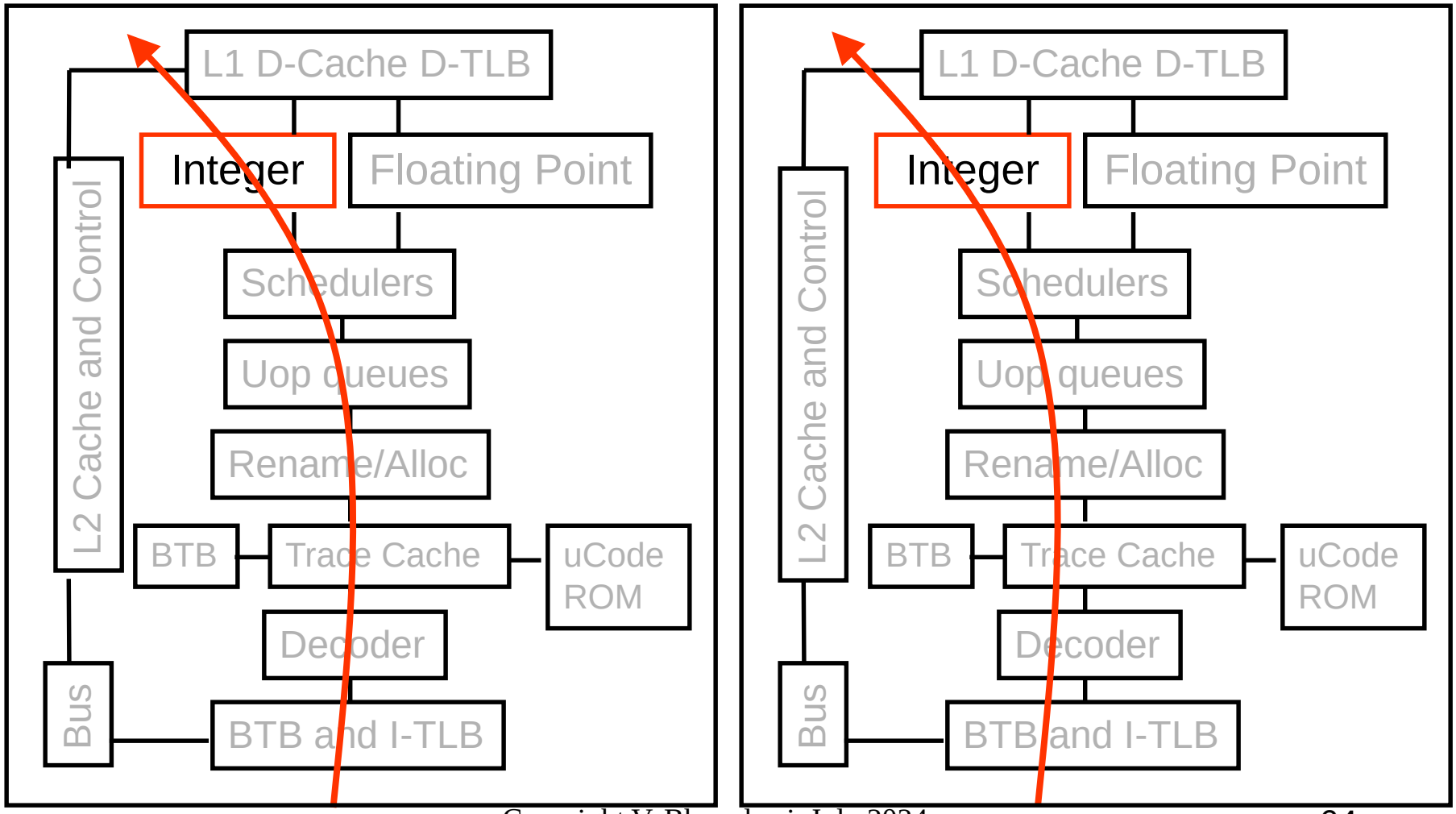


This scenario is impossible with SMT on a single core (assuming a single integer unit)

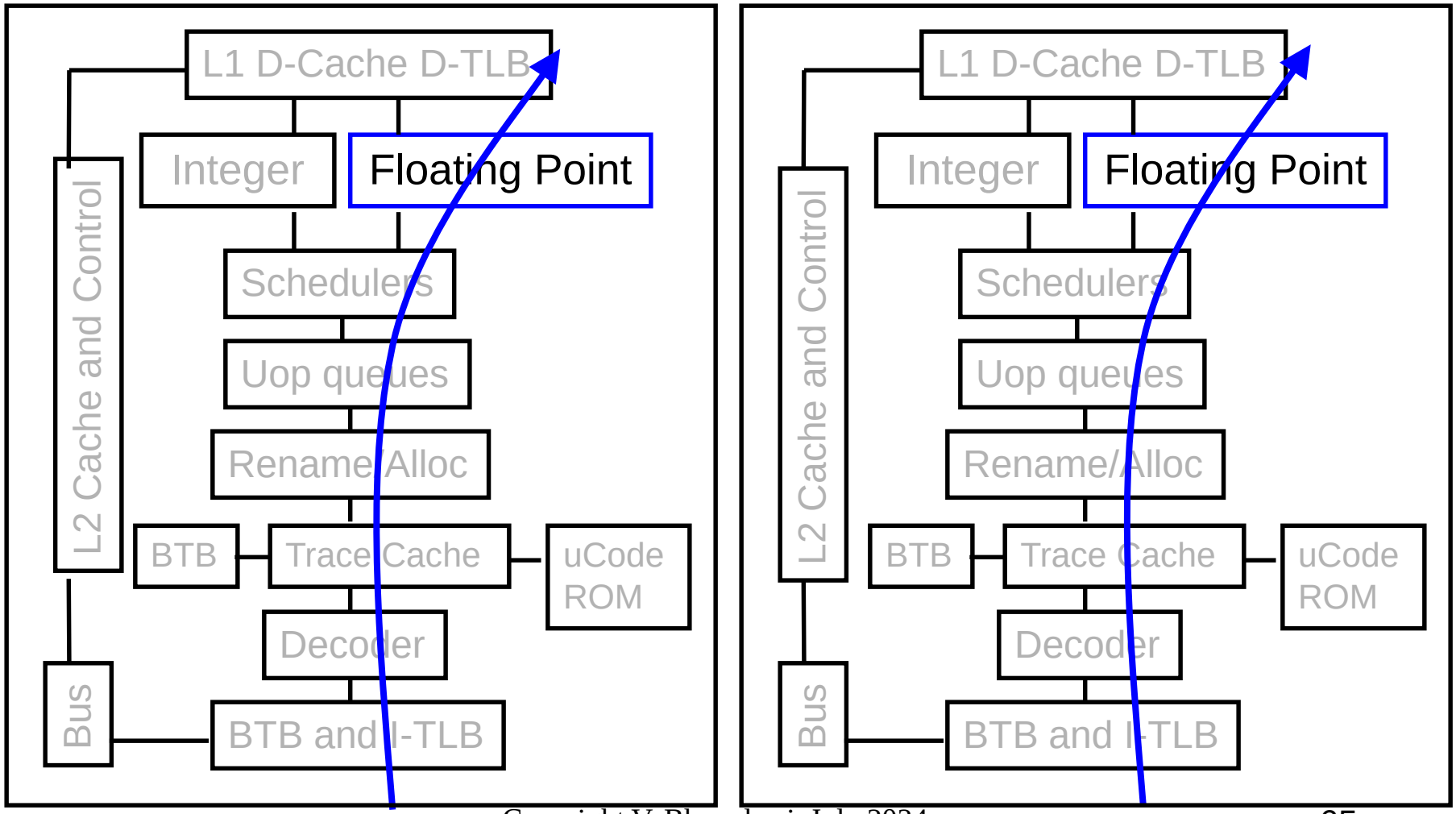
SMT not a “true” parallel processor

- Enables better threading
- OS and applications perceive each simultaneous thread as a separate “virtual processor”
- The chip has only a single copy of each resource
- Compared to multi-core each core has its own copy of resources

Multi-core:
threads can run on separate cores



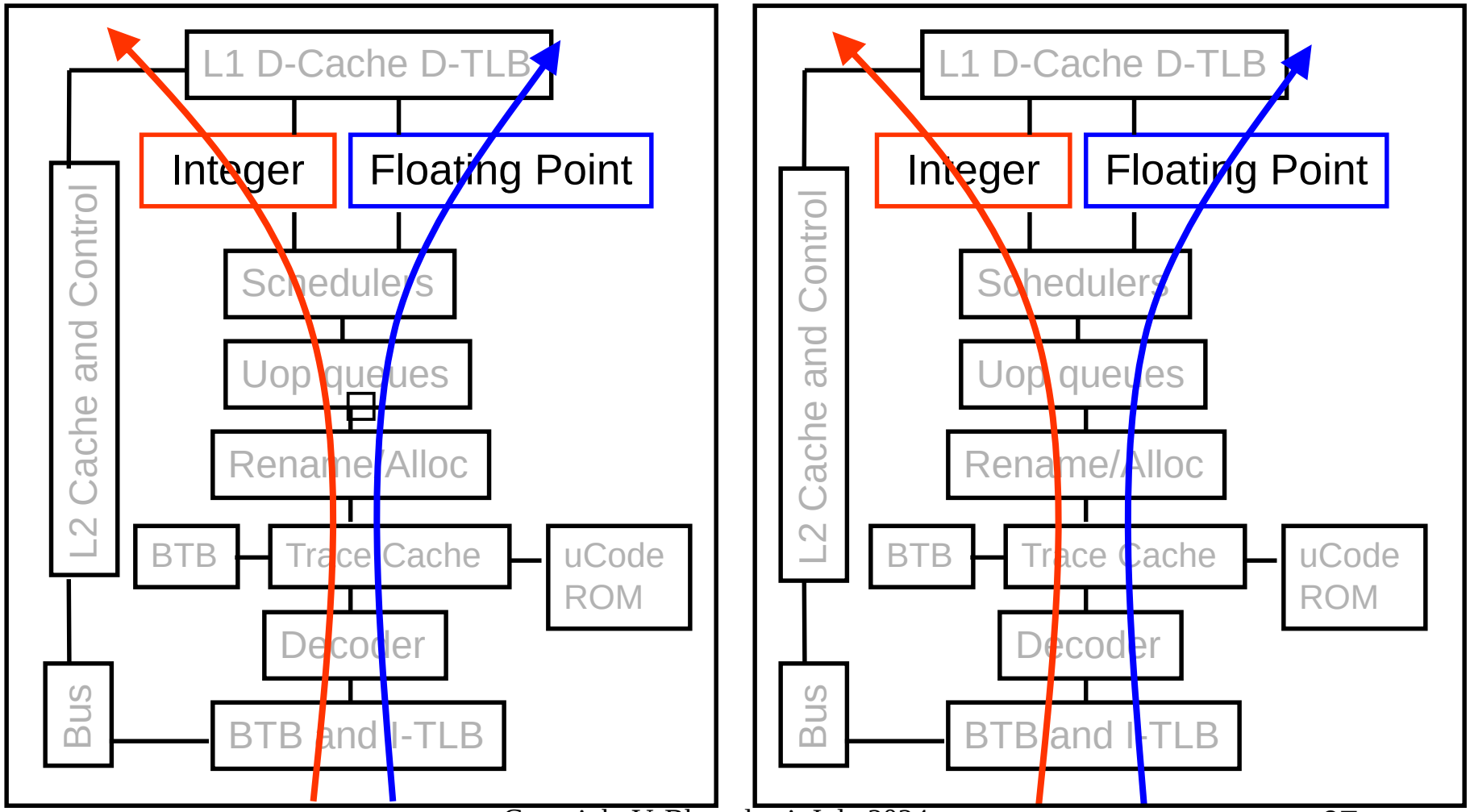
Multi-core:
threads can run on separate cores



Combining Multi-core and SMT

- Cores can be SMT-enabled (or not)
- The different combinations:
  - Single-core, non-SMT: standard uniprocessor
  - Single-core, with SMT
  - Multi-core, non-SMT
  - Multi-core, with SMT
- The number of SMT threads:
  2, 4, or sometimes 8 simultaneous threads
- Intel calls them “hyper-threads”

SMT Dual-core: all four threads can run concurrently

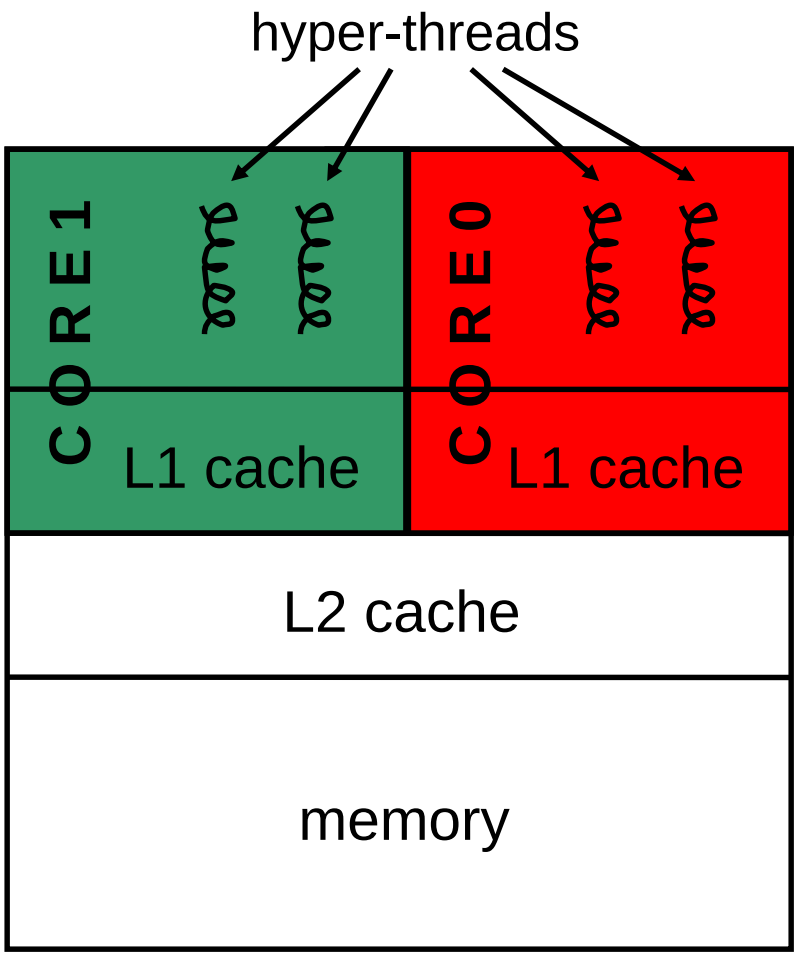


Memory hierarchy in multi-core processors

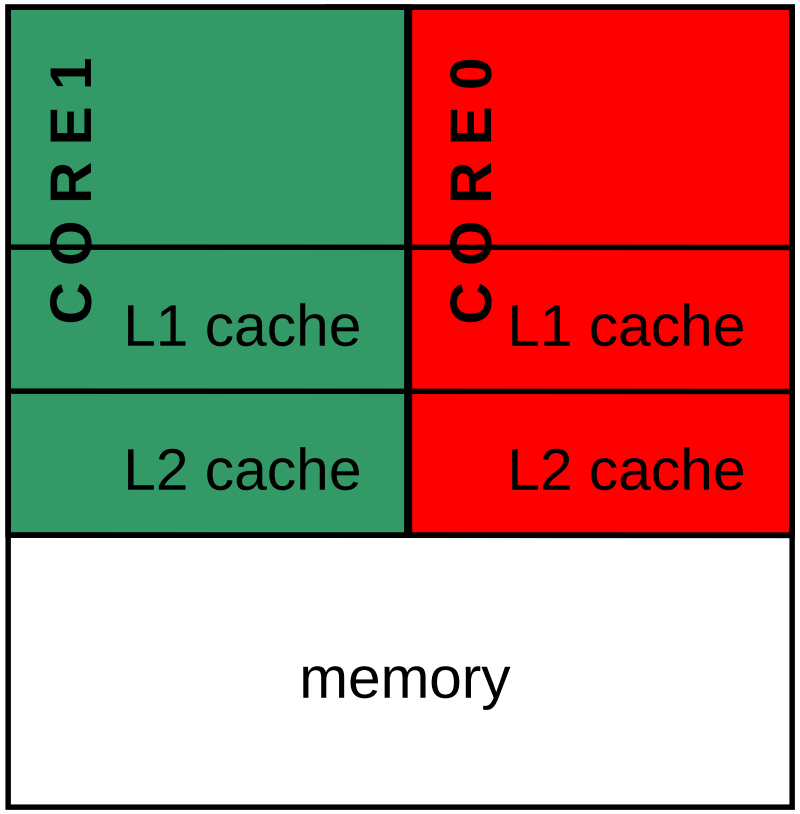
- If simultaneous multithreading only:
  - all caches are shared
- Multi-core chips:
  - L1 caches private
  - L2 caches private in some architectures and shared in others
- Memory is always shared

Typical Dual Core Features

- Dual-core
  Intel Xeon processors
- Each core is
  hyper-threaded
- Private L1 caches
- Shared L2 caches

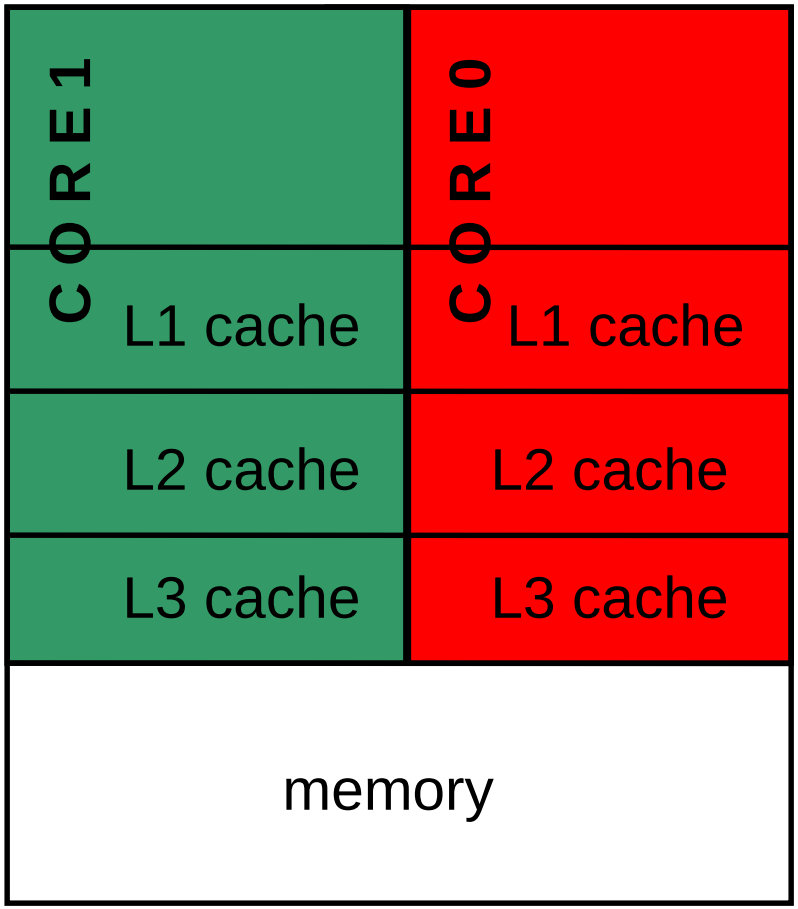


Designs with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron,
AMD Athlon, Intel Pentium D



A design with L3 caches

Example: Intel Itanium 2

Private vs shared caches

- Advantages of private:
  - They are closer to core, so faster access
  - Reduces contention
- Advantages of shared:
  - Threads on different cores can share the same cache data
  - More cache space available if a single (or a few) high-performance thread runs on the system

Example: Multi-threaded way of solving a computational problem

Typical application is heat wave propagation computation:

-- Averaging and testing for convergence

(a) Single Threaded Solution &
(b) Multi-threaded approach using Shared Memory Multiprocessor System
(c) Message Passing approach (suitable for HPC)

```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
     diff = 0;
     for i ← 1 to n do
       for j ← 1 to n do
          temp = A[i,j]; /* saving the original value */
          A[i,j] ← 0.2 * (A[i,j] + neighbors);
          diff += abs(A[i,j] – temp); /* difference computation */
       end for
     end for
     if (diff < TOL) then done = 1;   /* convergence testing */
  end while
end procedure
```

SINGLE THREDED EXECUTION

```
int  n, nprocs;
float  **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);


main()
begin
  read(n); read(nprocs);
  A ← G_MALLOC();
  initialize (A);
  CREATE (nprocs,Solve,A);
  WAIT_FOR_END (nprocs);
end main
```

MULTI-THREDED EXECUTION - SM

```
procedure Solve(A)
  int i, j, pid, done=0;
  float temp, mydiff=0;
  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;
    BARRIER(bar1,nprocs);
    for i ← mymin to mymax
      for j ← 1 to n do
        ...
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER (bar1, nprocs);
  endwhile
```

```
main()
  read(n); read(nprocs);
  CREATE (nprocs-1, Solve);
  Solve();
  WAIT_FOR_END (nprocs-1);

procedure Solve()
  int i, j, pid, nn = n/nprocs, done=0;
  float temp, tempdiff, mydiff = 0;
  myA ← malloc(...)
  initialize(myA);
  while (!done) do
    mydiff = 0;
    if (pid != 0)
      SEND(&myA[1,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      SEND(&myA[nn,0], n, pid+1, ROW);
    if (pid != 0)
      RECEIVE(&myA[0,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
    for i ← 1 to nn do
      for j ← 1 to n do
        ...
      endfor
    endfor
    if (pid != 0)
      SEND(mydiff, 1, 0, DIFF);
      RECEIVE(done, 1, 0, DONE);
    else
      for i ← 1 to nprocs-1 do
        RECEIVE(tempdiff, 1, *, DIFF);
        mydiff += tempdiff;
      endfor
      if (mydiff < TOL) done = 1;
      for i ← 1 to nprocs-1 do
        SEND(done, 1, I, DONE);
      endfor
    endif
  endwhile
```

MESSAGE PASSING APPROACH