THREAD LEVEL PARALLELISM

Mahdi Nazm Bojnordi

Assistant Professor

School of Computing

University of Utah

THE UNIVERSITY OF UTAH

CS/ECE 6810: Computer Architecture

Overview

- Announcement
  - Final exam: in-class, 10:30AM-12:30PM, Dec. 13th
- This lecture
  - Thread level parallelism (TLP)
    - Hardware multithreading
    - Multiprocessing
  - TLP Challenges
    - Communication

Hardware Multithreading

Recall: Hardware Multithreading

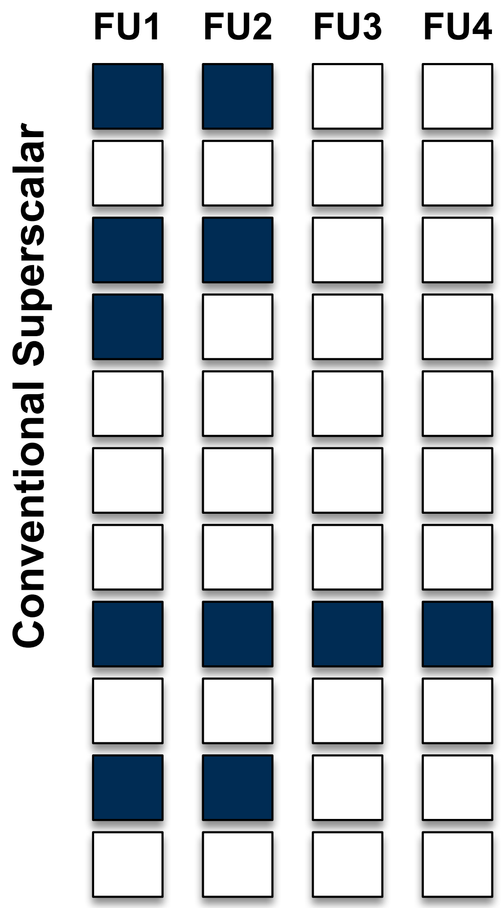
- Observation: CPU become idle due to latency of memory operations, dependent instructions, and branch resolution
- Key idea: utilize idle resources to improve performance
  - Support multiple thread contexts in a single processor
  - Exploit thread level parallelism
- Challenge: the energy and performance costs of context switching

Coarse Grained Multithreading

- Single thread runs until a costly stall—e.g. last level cache miss
- Another thread starts during stall for first
  - Pipeline fill time requires several cycles!
- At any time, only one thread is in the pipeline
- Does not cover short stalls
- Needs hardware support
  - PC and register file for each thread

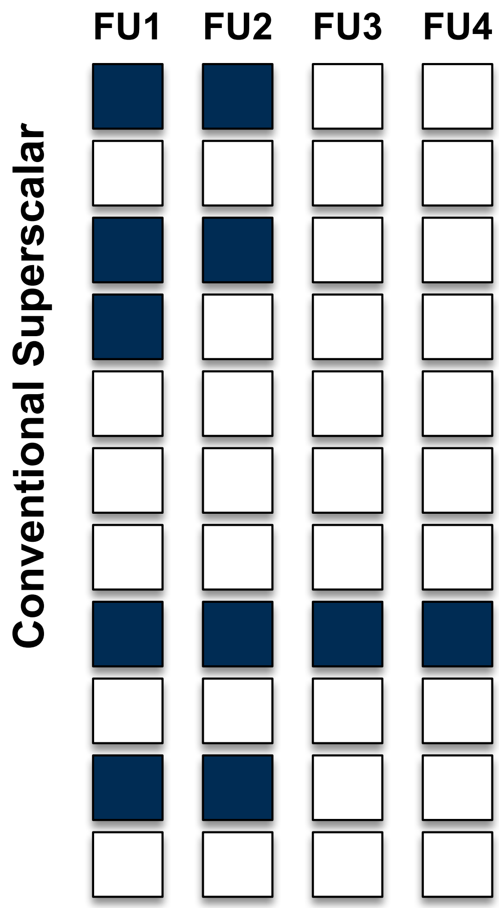
Coarse Grained Multithreading

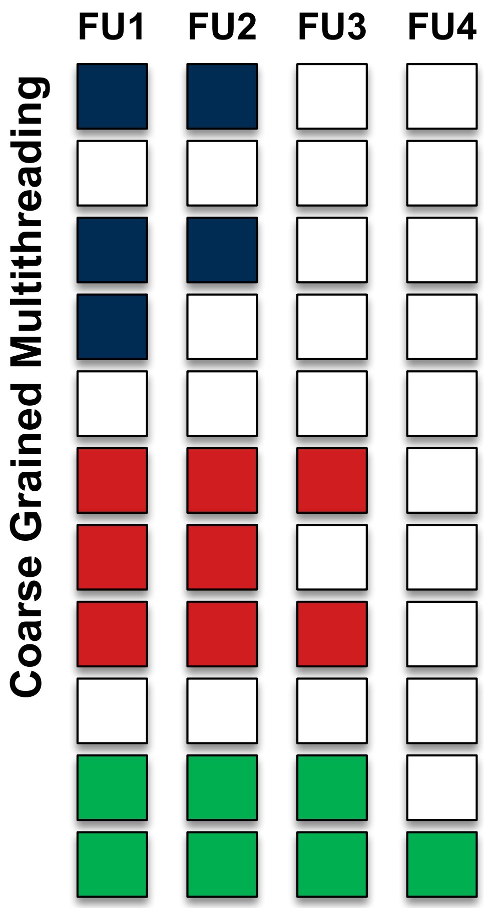
- Superscalar vs. CGMT



Coarse Grained Multithreading

- Superscalar vs. CGMT



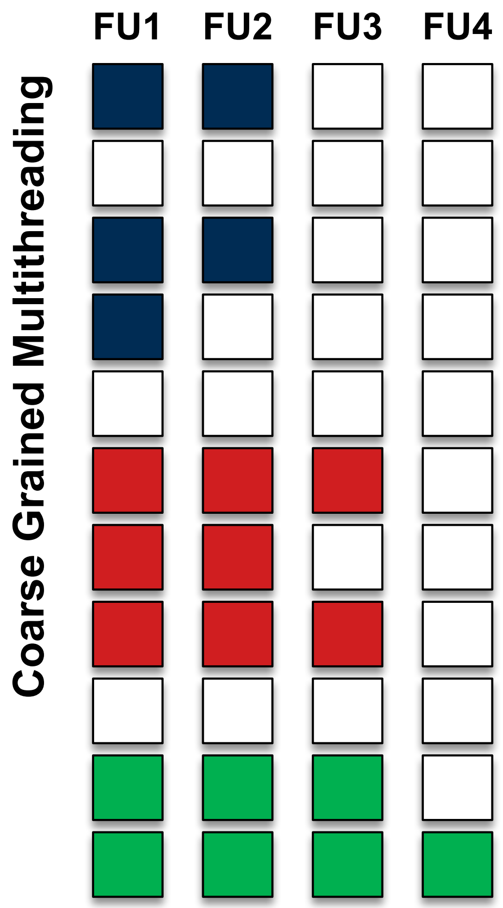


Fine Grain Multithreading

- Two or more threads interleave instructions
  - Round-robin fashion
  - Skip stalled threads
- Needs hardware support
  - Separate PC and register file for each thread
  - Hardware to control alternating pattern
- Naturally hides delays
  - Data hazards, Cache misses
  - Pipeline runs with rare stalls
- Does not make full use of multi-issue architecture

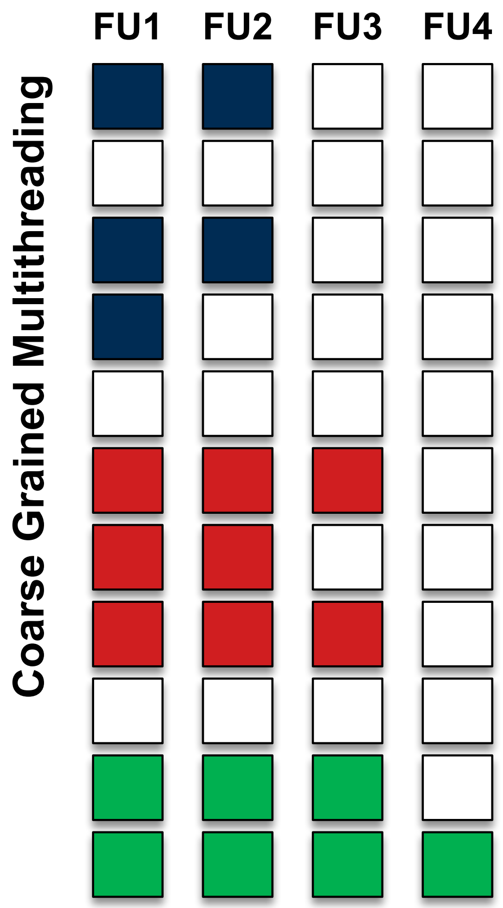
Fine Grained Multithreading

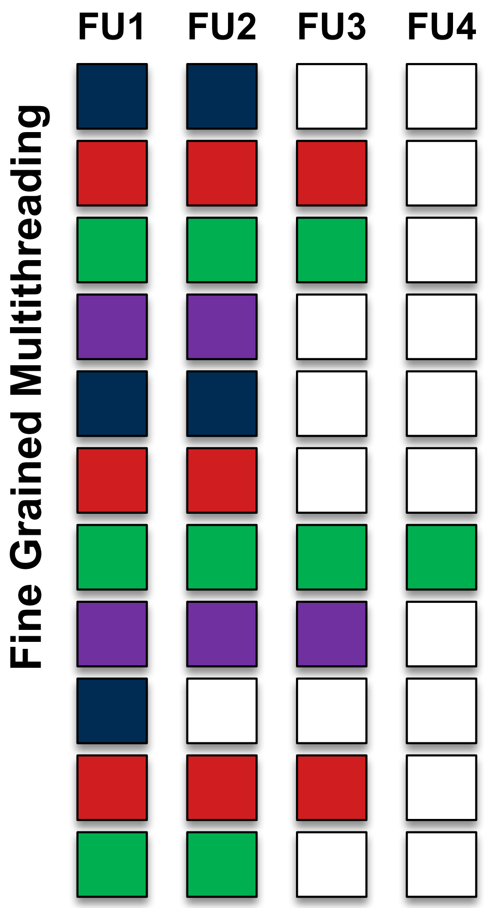
- CGMT vs. FGMT



Fine Grained Multithreading

- CGMT vs. FGMT



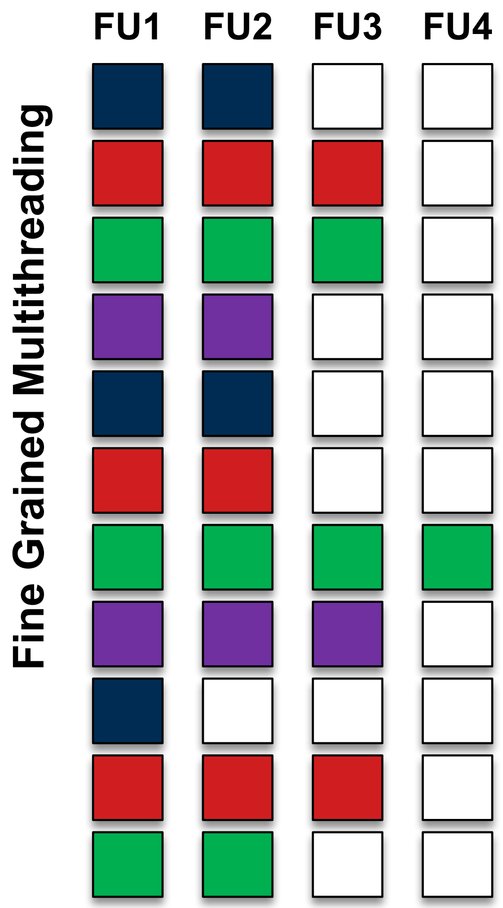


Simultaneous Multithreading

- Instructions from multiple threads issued on same cycle
  - Uses register renaming and dynamic scheduling facility of multi-issue architecture
- Needs more hardware support
  - Register files, PC's for each thread
  - Temporary result registers before commit
  - Support to sort out which threads get results from which instructions
- Maximizes utilization of execution units

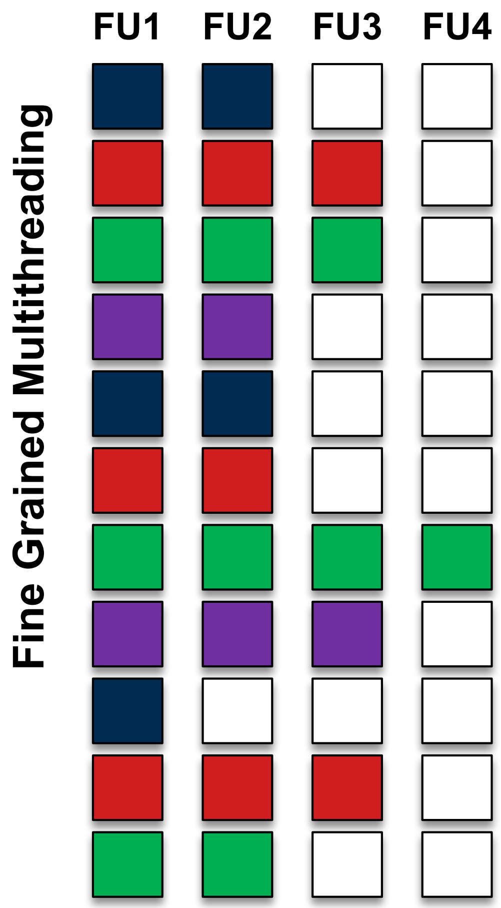
Simultaneous Multithreading

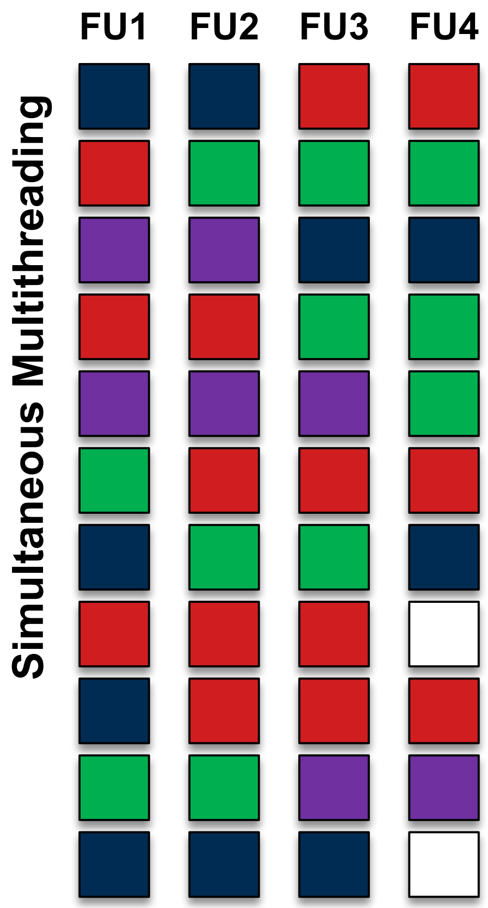
- FGMT vs. SMT



Simultaneous Multithreading

- FGMT vs. SMT





Recall: TLP Architectures

- Architectures for exploiting thread-level parallelism

Hardware Multithreading

- Multiple threads run on the same processor pipeline
- Multithreading levels
  - Coarse grained multithreading (CGMT)
  - Fine grained multithreading (FGMT)
  - Simultaneous multithreading (SMT)

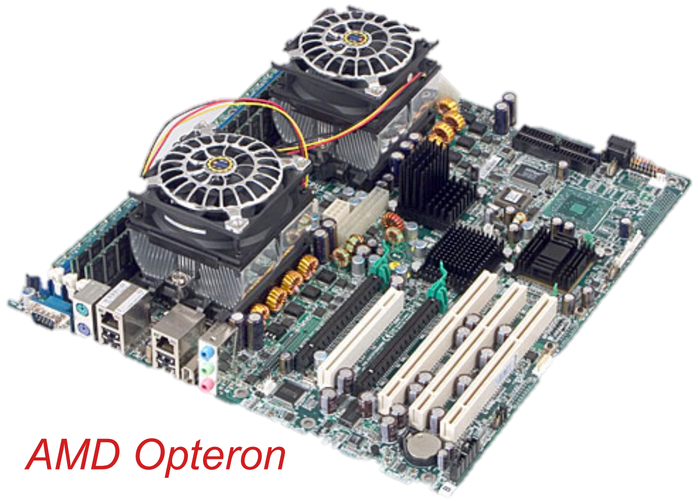
Multiprocessing

- Different threads run on different processors
- Two general types
  - Symmetric multiprocessors (SMP)
    - Single CPU per chip
  - Chip Multiprocessors (CMP)
    - Multiple CPUs per chip

Multiprocessing

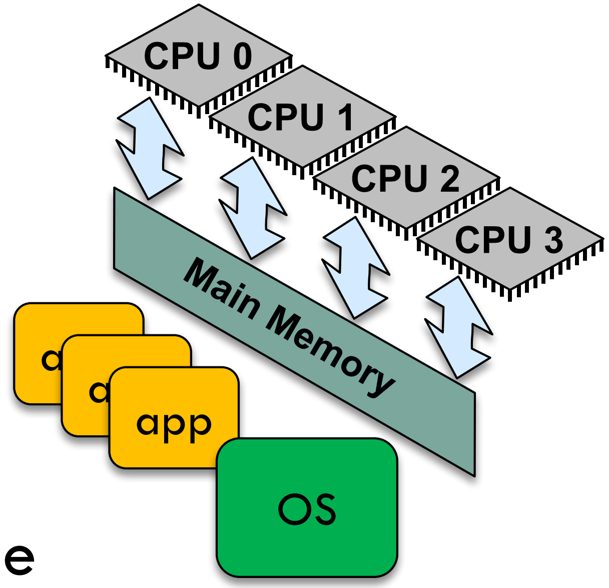
Symmetric Multiprocessors

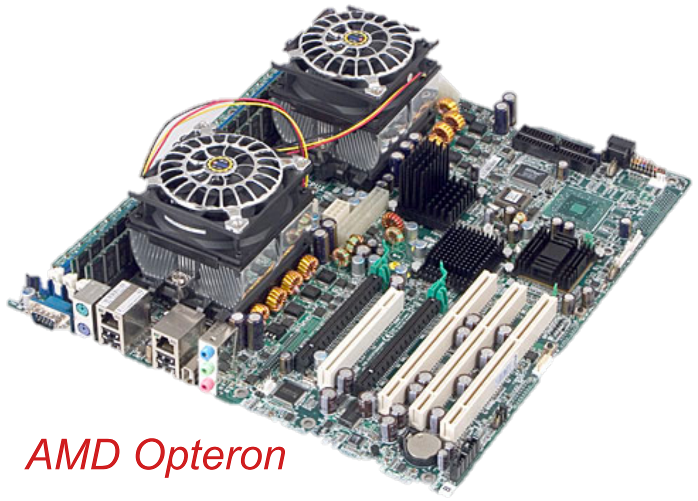
- Multiple CPU chips share the same memory
- From the OS's point of view
  - All of the CPUs have equal compute capabilities
  - The main memory is equally accessible by the CPU chips
- OS runs every thread on a CPU
- Every CPU has its own power distribution and cooling system


AMD Opteron

Symmetric Multiprocessors

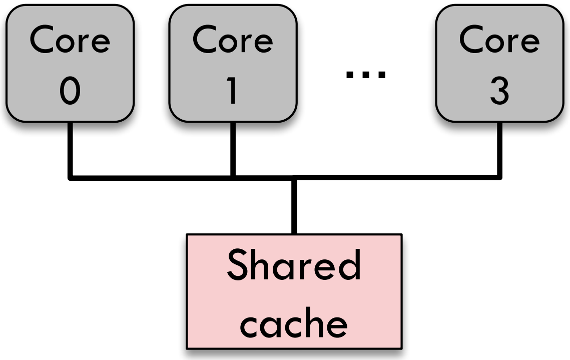
- Multiple CPU chips share the same memory
- From the OS's point of view
  - All of the CPUs have equal compute capabilities
  - The main memory is equally accessible by the CPU chips
- OS runs every thread on a CPU
- Every CPU has its own power distribution and cooling system

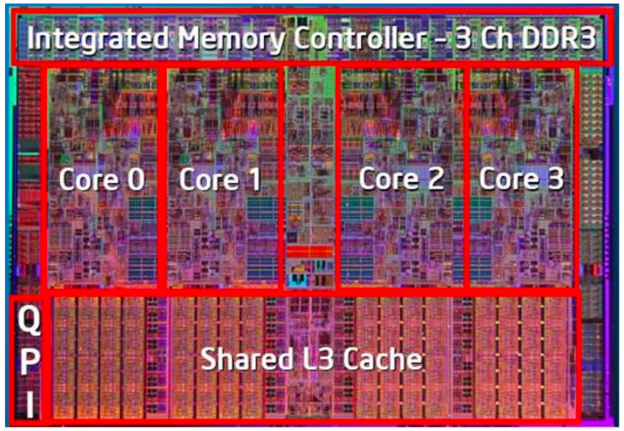



AMD Opteron

Chip Multiprocessors

- Can be viewed as a simple SMP on single chip
- CPUs are now called cores
  - One thread per core
- Shared higher level caches
  - Typically the last level
  - Lower latency
  - Improved bandwidth
- Not necessarily homogenous cores!





Intel Nehalem (Core i7)

Why Chip Multiprocessing?

- CMP exploits parallelism at lower costs than SMP
  - A single interface to the main memory
  - Only one CPU socket is required on the motherboard
- CMP requires less off-chip communication
  - Lower power and energy consumption
  - Better performance due to improved AMAT
- CMP better employs the additional transistors that are made available based on the Moore's law
  - More cores rather than more complicated pipelines

Efficiency of Chip Multiprocessing

- Ideally, n cores provide nx performance
- Example: design an ideal dual-processor
  - Goal: provide the same performance as uniprocessor

	Uniprocessor	Dual-processor
Frequency	1	?
Voltage	1	?
Execution Time	1	1
Dynamic Power	1	?
Dynamic Energy	1	?
Energy Efficiency	1	?

Efficiency of Chip Multiprocessing

- Ideally, n cores provide nx performance
- Example: design an ideal dual-processor
  - Goal: provide the same performance as uniprocessor

$f \propto V$ & $P \propto V^3 \rightarrow V_{dual} = 0.5V_{uni} \rightarrow P_{dual} = 2\times 0.125P_{uni}$

	Uniprocessor	Dual-processor
Frequency	1	0.5
Voltage	1	0.5
Execution Time	1	1
Dynamic Power	1	2x0.125
Dynamic Energy	1	2x0.125
Energy Efficiency	1	4

TLP Challenges

Example Code I

- A sequential application runs as a single thread

Kernel Function:

```
void kern (int start, int end) {
    int i;
    for(i=start; i<=end; ++i) {
        A[i] = A[i] * A[i] + 5;
    }
}
```

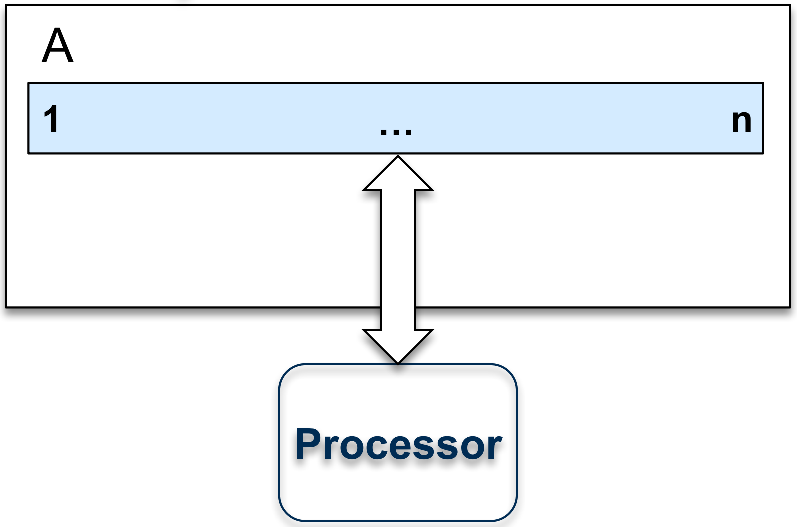
Example Code I

- A sequential application runs as a single thread

Kernel Function:

```
void kern (int start, int end) {
    int i;
    for(i=start; i<=end; ++i) {
        A[i] = A[i] * A[i] + 5;
    }
}
```

Memory



Example Code I

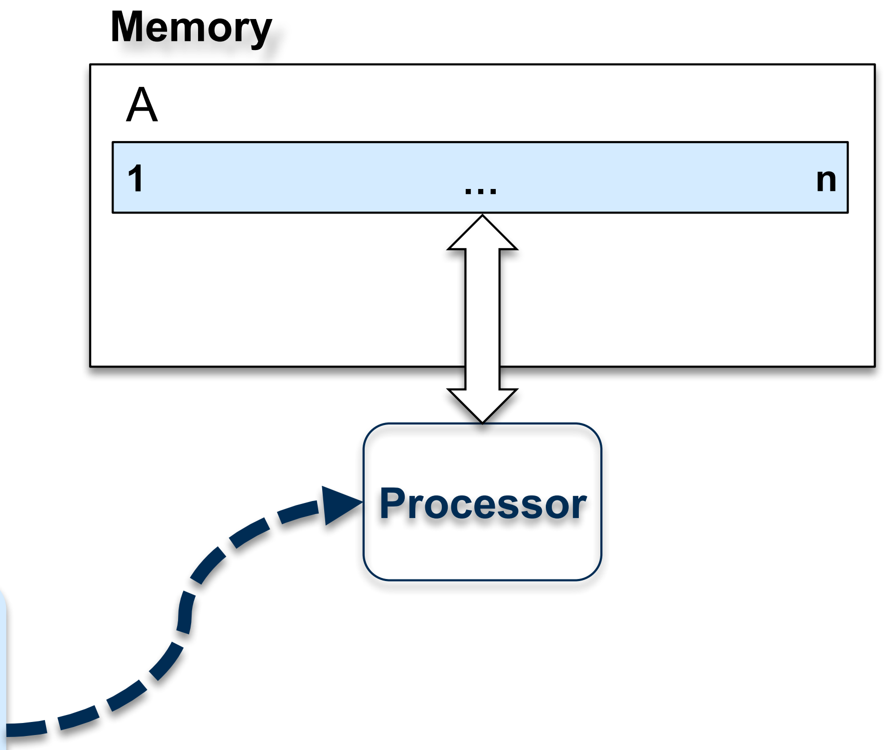
- A sequential application runs as a single thread

Kernel Function:

```
void kern (int start, int end) {
   int i;
   for(i=start; i<=end; ++i) {
      A[i] = A[i] * A[i] + 5;
   }
}
```

Single Thread

```
main() {
   …
   kern (1, n);
   …
}
```



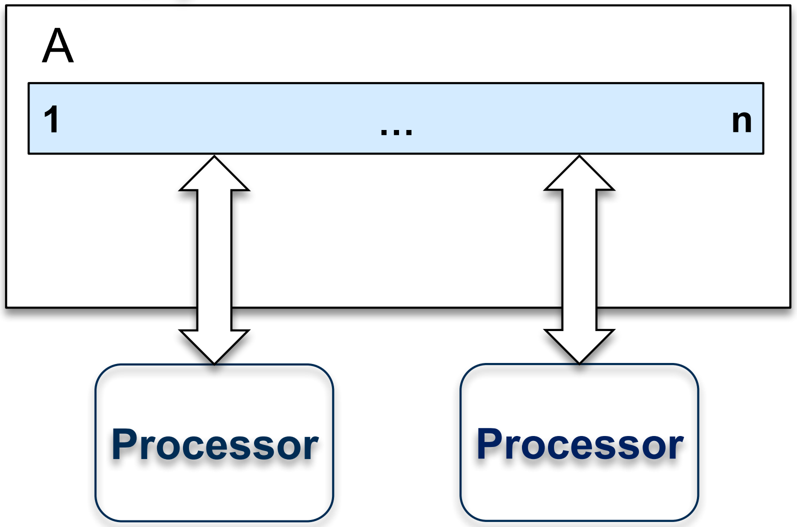
Example Code I

- Two threads operating on separate partitions

Kernel Function:

```
void kern (int start, int end) {
   int i;
   for(i=start; i<=end; ++i) {
      A[i] = A[i] * A[i] + 5;
   }
}
```

Memory



How to run the kernel on two processors?

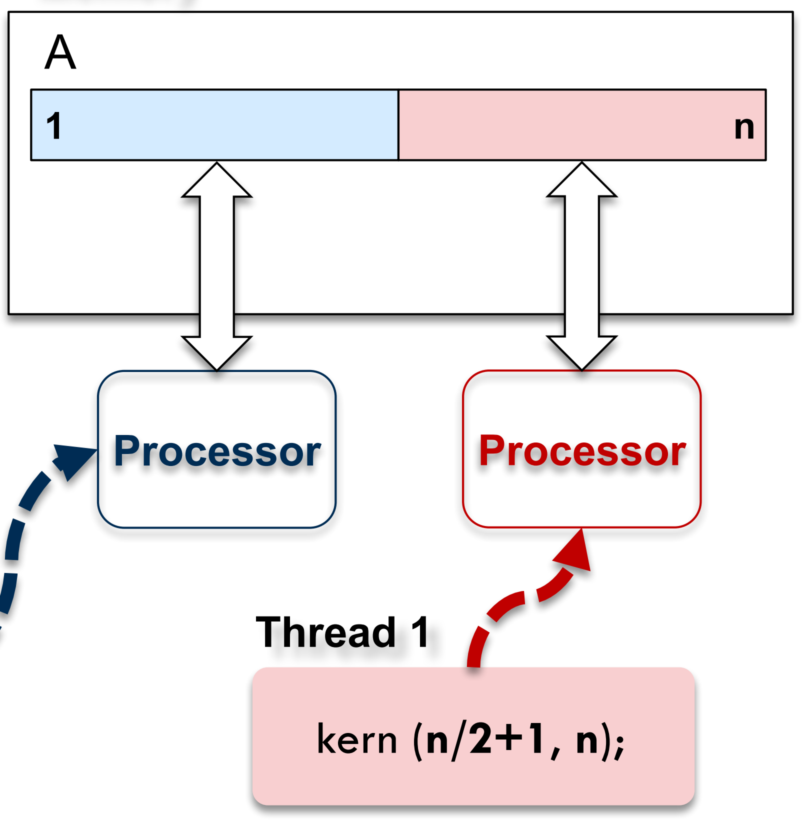
Example Code I

- Two threads operating on separate partitions

Kernel Function:

```
void kern (int start, int end) {
   int i;
   for(i=start; i<=end; ++i) {
      A[i] = A[i] * A[i] + 5;
   }
}
```

Memory



Thread 0

```
main() {
   …
   kern (1, n/2);
   …
}
```

Thread 1

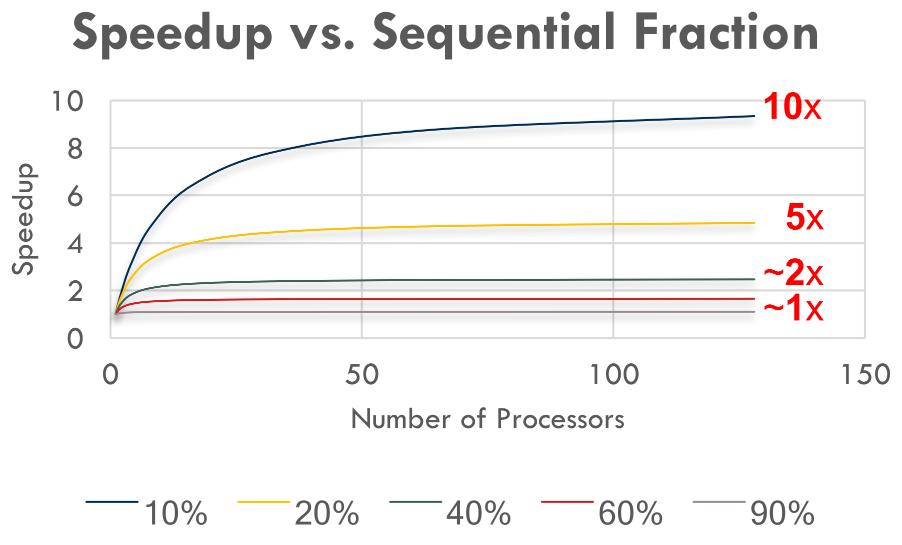
```
kern (n/2+1, n);
```

Performance of Parallel Processing

- Recall: Amdahl's law for theoretical speedup
  - Overall speedup is limited to the fraction of the program that can be executed in parallel

$$speedup = \frac{1}{f + \frac{1-f}{n}}$$

f: sequential fraction



Example Code II

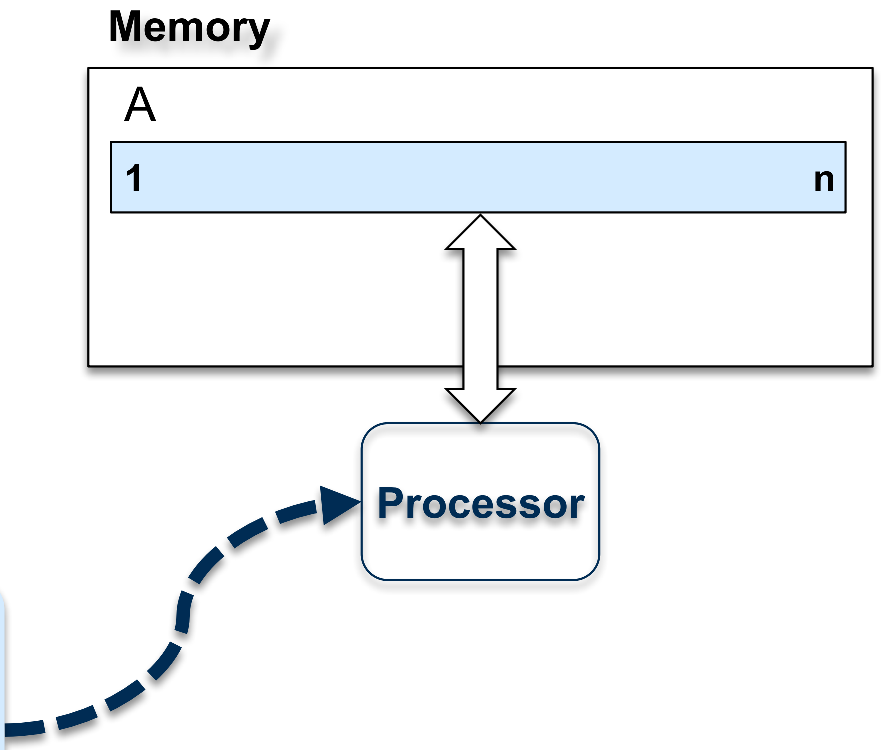
- A single location is updated every time

Kernel Function:

```
void kern (int start, int end) {
   int i;
   for(i=start; i<=end; ++i) {
      sum = sum + A[i];
   }
}
```

Thread 0

```
main() {
   …
   kern (1, n);
   …
}
```



Example Code II

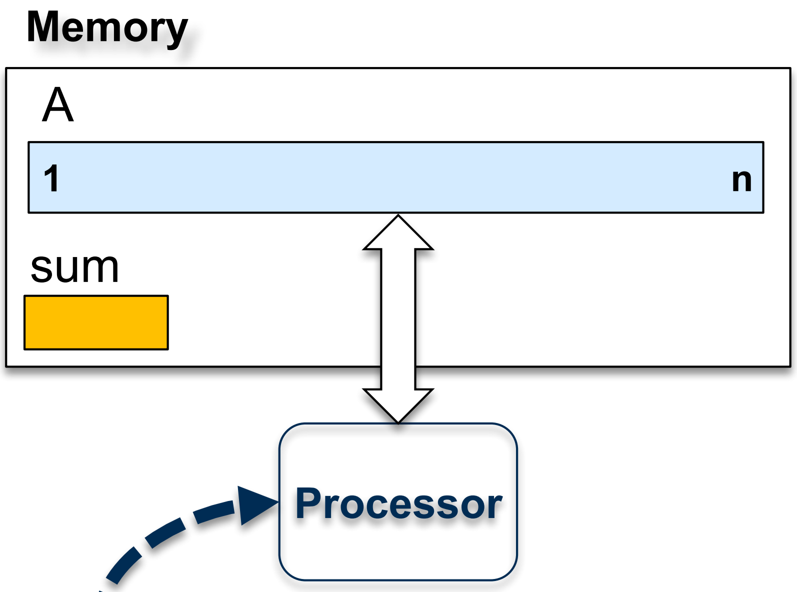
- A single location is updated every time

Kernel Function:

```
void kern (int start, int end) {
    int i;
    for(i=start; i<=end; ++i) {
        sum = sum + A[i];
    }
}
```

Thread 0

```
main() {
    …
    kern (1, n);
    …
}
```



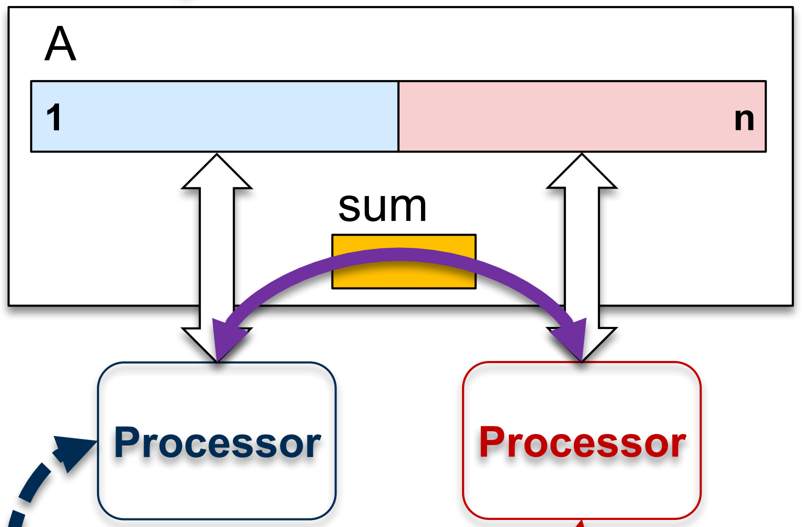
Example Code II

- Two threads operating on separate partitions

Kernel Function:

```
void kern (int start, int end) {
   int i;
   for(i=start; i<=end; ++i) {
      sum = sum + A[i];
   }
}
```

Memory



Thread 0

```
main() {
   …
   kern (1, n/2);
   …
}
```

Thread 1

```
kern (n/2+1, n);
```