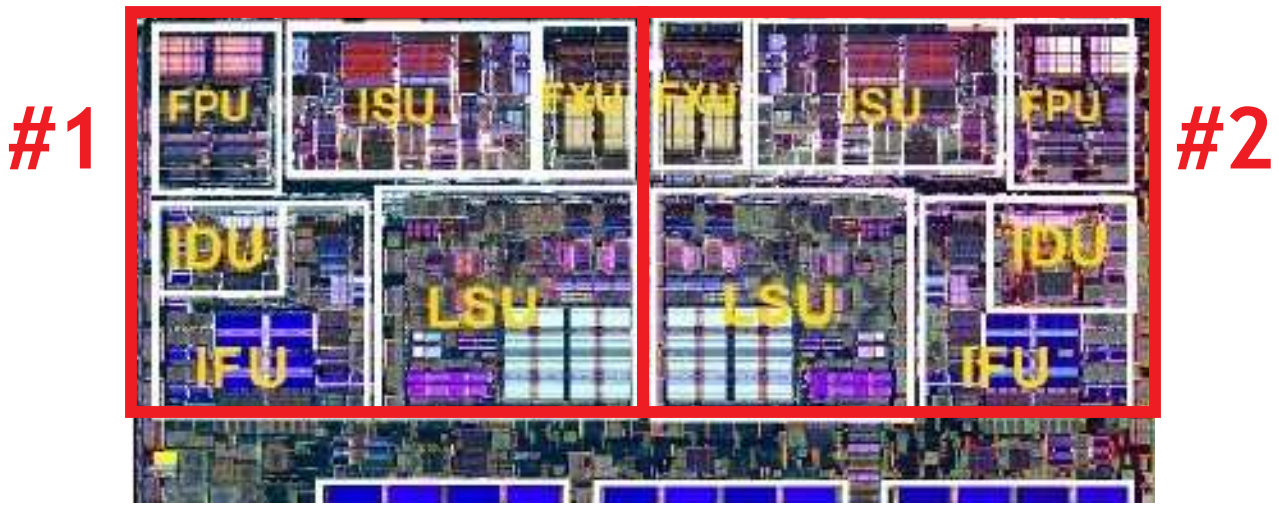
Exploiting a multi-core processor

```
unsigned counter = 0;

void *do_stuff(void * arg) {
   parallel_for (int i = 0 ; i < 200000000 ; ++ i) {
      counter ++;
   }
   return arg;
}
```

Split for-loop across multiple threads running on separate cores



Problem 1: Data Races

- Since `counter` is *shared* between threads, we can get a data race

```
parallel_for (int i = 0 ; i < 200000000 ; ++ i) {
    counter ++;
}
```

```
lw   $t0, counter
addi $t0, $t0, 1
sw   $t0, counter
```

Fix problem by making this operation atomic

Sequence 1

Processor 1	Processor 2
`lw $t0, counter`	
`addi $t0, $t0, 1`	
`sw $t0, counter`	
	`lw $t0, counter`
	`addi $t0, $t0, 1`
	`sw $t0, counter`

`counter` increases by 2

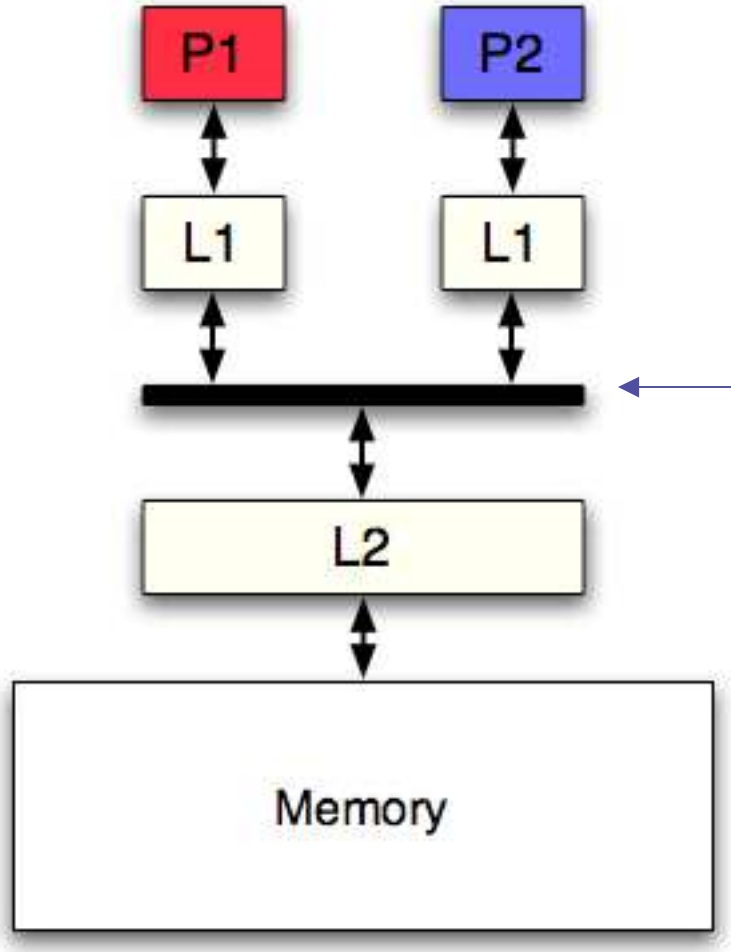
Sequence 2

Processor 1	Processor 2
`lw $t0, counter`	
	`lw $t0, counter`
`addi $t0, $t0, 1`	
	`addi $t0, $t0, 1`
`sw $t0, counter`	
	`sw $t0, counter`

`counter` increases by 1 !!

Problem 2: This is SLOW!

- Memory is shared, but the L1 caches are not!



Multiple bouncing slows performance

Fast, parallel code (software approach)

- Hardware guarantees correctness with atomic operations, but its slow

```
parallel_for (i = 0; i < N; ++i){
  counter++;
}
```

- What if each thread had its *own* copy of `counter`? (*private*, not shared)

```
parallel_for (i = 0; i < N; ++i) private(counter) {
  counter++; // increment local copy
}
// Now reduce the local copies of counter into a single variable
```

- This works because "+" is associative and commutative
  - fortunately, common operations have these properties

An example: Selection Sort

```
for(i = 0; i < LENGTH - 1; i++) {
  min = i;       // assume ith element is smallest

  for(j = i + 1; j < LENGTH; j++) {
    if(a[j] < a[min])
      min = j;  // oops, jth element is smaller
  }

  swap(a[min], a[i]);
}
```

Another example: Register Allocation

- Recall the register allocation problem
  - assign register names for every program variable and temporary
  - variables that are live at the same time must get different registers

- Graph model
  - Vertices = Variables
  - edge between *u* and *v* if both variables are live at the same time

- Register allocation == Graph coloring
  - assign colors to every vertex so that neighbors get different colors

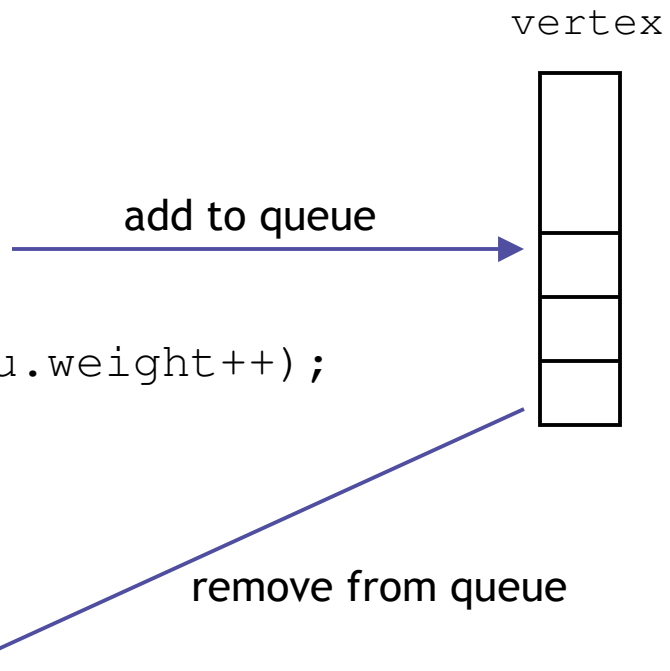
- For special graphs, there is a simple optimal coloring algorithm…

Greedy Coloring

```
for all vertices v {
  v.color = ∞;
  v.weight = 0;
  bag.insert(v);
}

for(i = 0; i < N; ++i) {
  v = bag.remove_heaviest();
  available_colors = {1, 2, ..., N};

  for all neighbors u of v
     if(u.color == ∞)
       bag.update(u, u.weight++);
     else
       available_colors.remove(u.color);

  v.color = available_colors.smallest();
}
```

Greedy Coloring - slightly modified

```
for all vertices v {
  v.color = ∞;
  v.weight = 0;
  bag.insert(v);
}

for(i = 0; i < N; ++i) {
  vertex[i] = bag.remove_heaviest();
  for all neighbors u of vertex[i]
     if(u.color == ∞) bag.update(u, u.weight++);
}

for(i = 0; i < N; ++i) {
  available_colors = {1, 2, ..., N};
  for all neighbors u of vertex[i]
     if(u.color != ∞) available_colors.remove(u.color);
  vertex[i].color = available_colors.smallest();
}
```

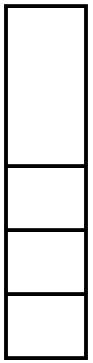


Greedy Coloring - slightly modified, rearranged

```
for all vertices v {
  v.color = ∞;
  v.weight = 0;
  bag.insert(v);
}
```

```
for(i = 0; i < N; ++i) {
  vertex[i] = bag.remove_heaviest();
  for all neighbors u of vertex[i]
    if(u.color == ∞)
      bag.update(u, u.weight++);
}
```

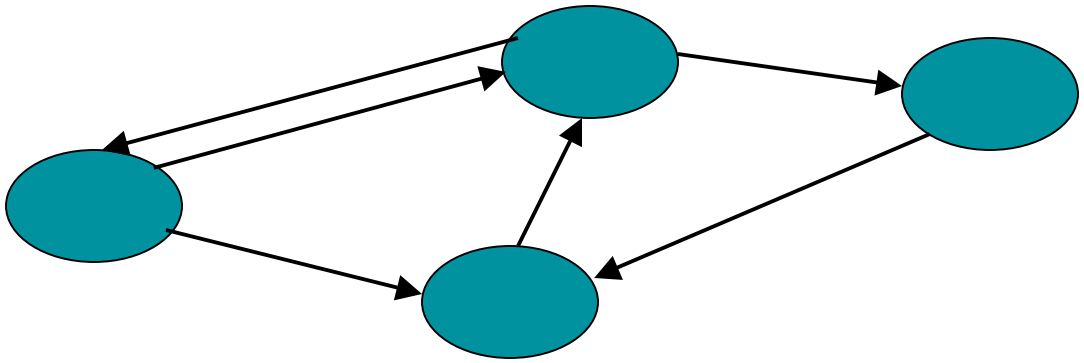
vertex



```
for(i = 0; i < N; ++i) {
  available_colors = {1, 2, ..., N};
  for all neighbors u of vertex[i]
    if(u.color != ∞)
      available_colors.remove(u.color);
  vertex[i].color =
    available_colors.smallest();
}
```

- What does this look like?

A generalization: Actors



- Each actor executes its own code
- Actors communicate via messages
  - *GEN*'s output becomes *COL*'s input
- "Everything is an actor", rather like "everything is an object"
  - actors act *concurrently*

Conclusions

- The hardware must implement some basic atomic operations
  - we can use these to safely share data across processors

- Can't rely too much on atomic operations – they are slow!
  - they force one processor to wait

- Software approaches:

1. If possible, use private (instead of shared) variables in each thread
   - After parallel region, reduce private copies into a single variable
   - Common operations are associative + commutative, so this is OK

2. Pipelining
   - More generally, actors