- OpenMP Focus is on Optimization of Loops

But be careful about:

Shared vs. Private Variables

Loop Carried Dependencies

Why Do We Care?

Loops are the favorite control structures of High Performance Computing, because compilers know how to ***optimize*** their performance using instruction-level parallelism: superscalar, pipelining and vectorization can give excellent speedup.

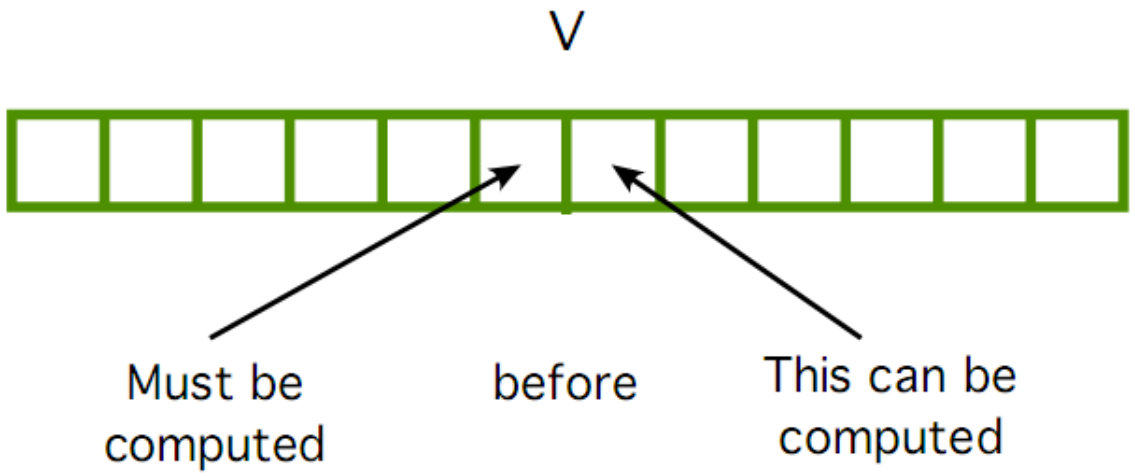
Loop carried dependencies affect whether a loop can be parallelized, and how much.

Example of Backward Loop Dependency

Loops often introduce real, or apparent, dependencies.

```
do i=1,n
    V[i]=V[i] – 2*V[i–1]
enddo
```

Backward dependency: cannot be parallelized because each value depends upon value from previous iteration.



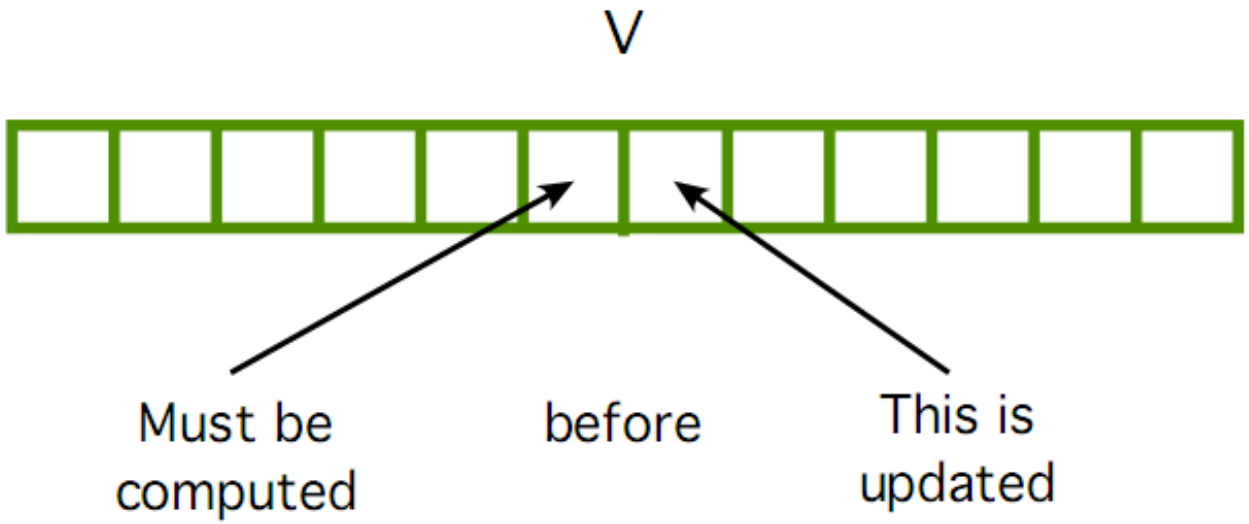
T1: V[1] = V[1] - 2V[0]
V[2] = V[2] - 2V[1]
V[3] = V[3] - 2V[2]
V[4] = V[4] - 2V[3]

T2: V[5] = V[5] - **2V[4]** RAW
V[6] = V[6] - 2V[5]
V[7] = V[7] - 2V[6]
V[8] = V[8] - 2V[7]

Example of Forward Loop Dependency

To parallelize

```
do i=1,n
    V[i]=V[i] – 2*V[i+1]
enddo
```



T1: V[1] = V[1] - 2V[2]
V[2] = V[2] - 2V[3]
V[3] = V[3] - 2V[4]
V[4] = V[4] - 2**V[5]**

T2: **V[5]** = V[5] - 2V[6]
V[6] = V[6] - 2V[7]
V[7] = V[7] - 2V[8]
V[8] = V[8] - 2V[9]

WAR

The bag of programming tricks

that has served us so well

for the last 50 years

is

the wrong way to think

going forward and

must be thrown out.

Why?

- Good sequential code minimizes total number of operations.
  - > Clever tricks to reuse previously computed results.
  - > Good parallel code often performs redundant operations to reduce communication.
- Good sequential algorithms minimize space usage.
  - > Clever tricks to reuse storage.
  - > Good parallel code often requires extra space to permit temporal decoupling.
- Sequential idioms stress linear problem decomposition.
  - > Process one thing at a time and accumulate results.
  - > Good parallel code usually requires multiway problem decomposition and multiway aggregation of results.

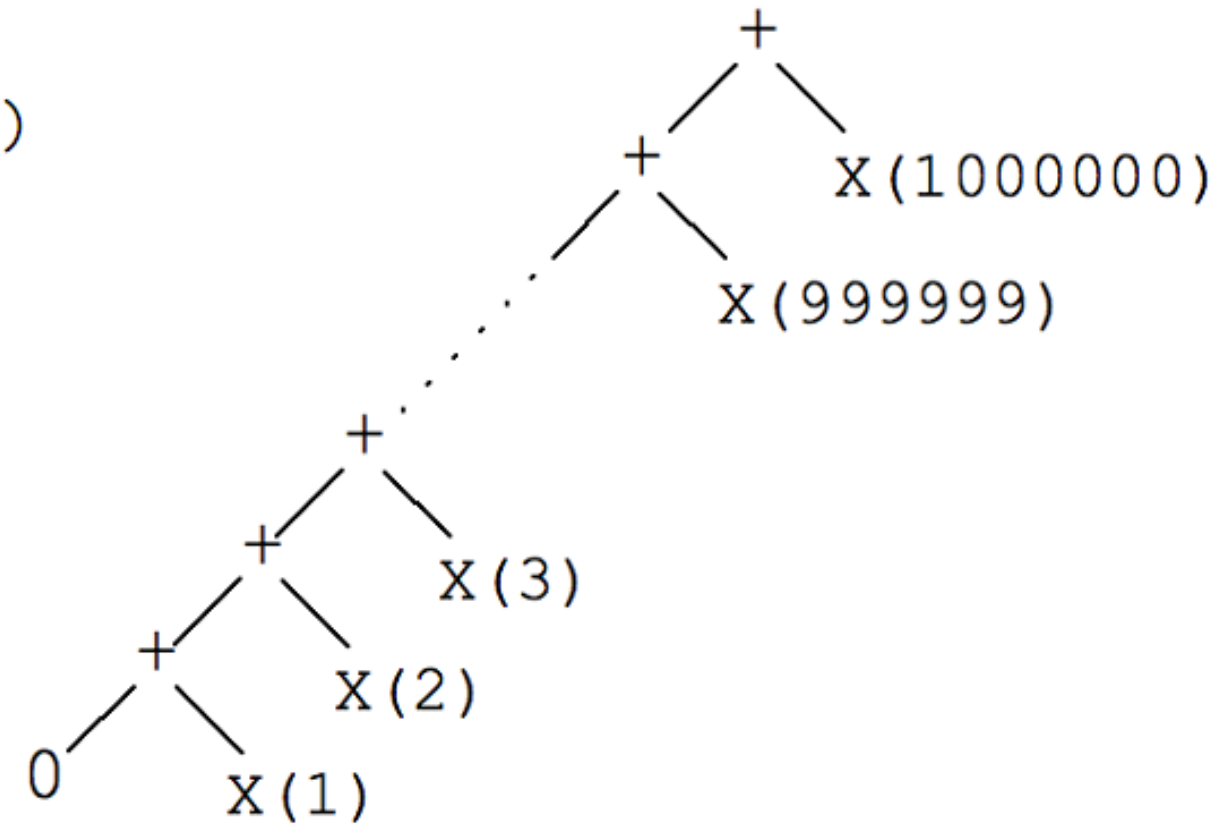
Let's Add a Bunch of Numbers

```
DO I = 1, 1000000
   SUM = SUM + X(I)
END DO
```

Can it be parallelized?

Sequential Computation Tree

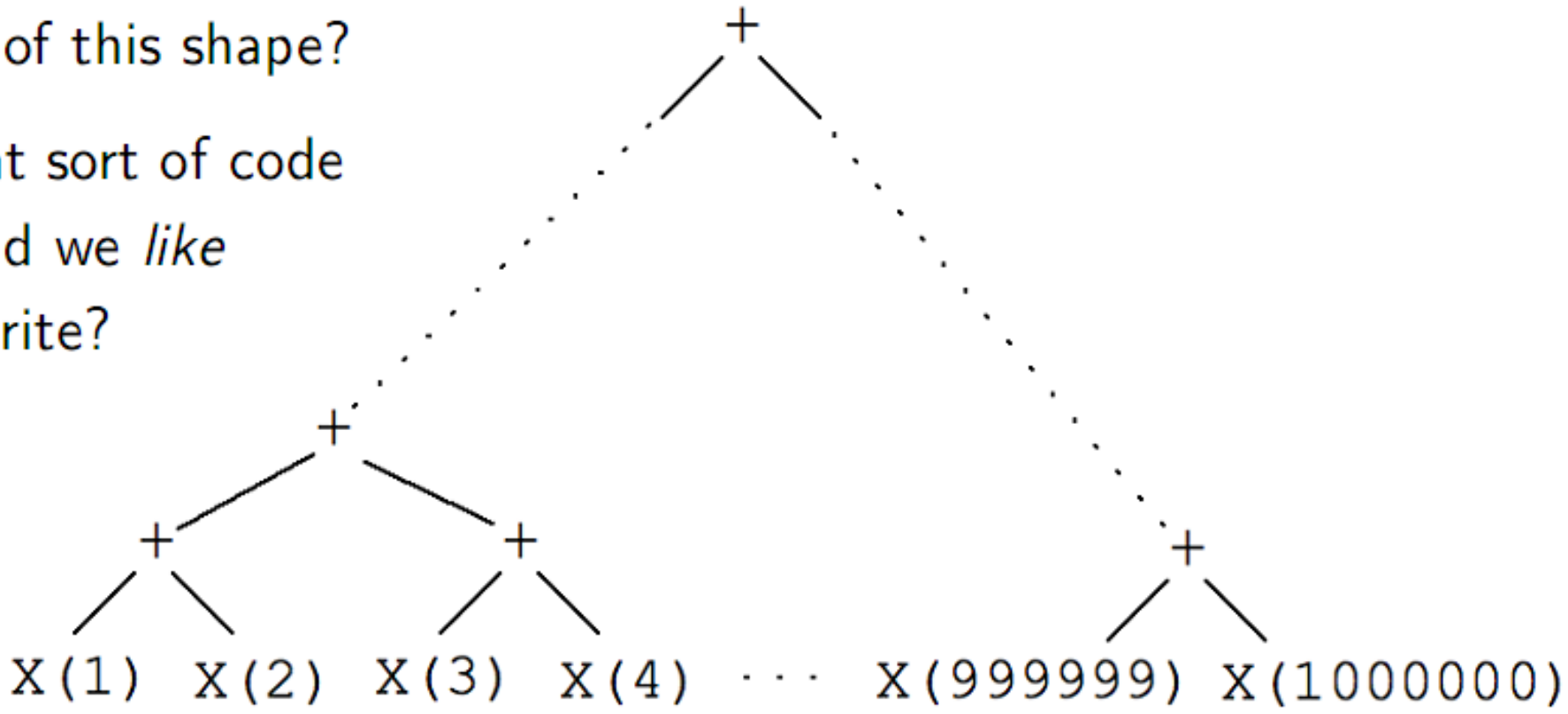
```
SUM = 0
DO I = 1, 1000000
   SUM = SUM + X(I)
END DO
```



Parallel Computation Tree

What sort of code
should we write
to get a computation
tree of this shape?

What sort of code
would we *like*
to write?



The Parallel Future

- We need new strategies for problem decomposition.
  - > Data structure design/object relationships
  - > Algorithmic organization
  - > Don't split a problem into "the first" and "the rest."
  - > Do split a problem into roughly equal pieces.
    Then figure out how to combine general subsolutions.
  - > Often this makes combining the results a bit harder.
- We need programming languages and runtime implementations that support parallel strategies *and* hybrid sequential/parallel strategies.
- We must learn to manage new space-time tradeoffs.

Conclusion

- A program organized according to linear problem decomposition principles can be really hard to parallelize.
- A program organized according to parallel problem decomposition principles is easily run either in parallel or sequentially, according to available resources.
- The new strategy has costs and overheads. They will be reduced over time but will not disappear.
- This is our only hope for program portability in the future.
- Better language design can encourage better parallel programming.

Lessons

- Actual performance of a simple program can be a complicated function of the architecture
  - Slight changes in the architecture or program change the performance significantly
  - To write fast programs, need to consider architecture
    - True on sequential or parallel processor
  - We would like simple models to help us design efficient algorithms

What does this mean to you?

- In theory, the compiler understands all of this
  - When compiling, it will rearrange instructions to get a good "schedule" that maximizes pipelining, uses FMAs and SIMD
  - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help
  - Choose a different compiler, optimization flags, etc.
  - Rearrange your code to make things more obvious
  - Using special functions ("intrinsics") or write in assembly ☹

Conclusions

- Era of programmers not caring about what is under the hood is over
- A lot of variations/choices in hardware
- Many will have performance implications
- Understanding the hardware will make it easier to make programs get high performance
- *A note of caution*: If program is too closely tied to the processor → cannot port or migrate
  - back to the era of assembly programming