UNIVERSITAS STUDIORUM MUTINENSIS ET REGIENSIS 1175

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

HiPeRT Lab
High-Performance Real-Time Lab

Code profiling and analysis

Roberto Cavicchioli
roberto.cavicchioli@unimore.it

Keys to parallel performance

- **Coverage** or extent of parallelism in algorithm
  - *Remember Amdahl's Law?*

- **Granularity** of partitioning among processors
  - Communication cost and load balancing

- **Locality** of computation and communication
  - Communication between processors or between processors and their memories

Rollback: Amdahl's Law

te_{old} : total exec time without the enhancement
te_{new} : total exec time with the enhancement
$ts = (te_{old} / te_{new})$: total speed-up
pe_{old} : (partial) exec time of the original component =1
pe_{new} : (partial) exec time of the enhanced component
$ps = (pe_{old} / pe_{new})$: (partial) speed-up of the enhanced component
f : fraction of time in which the component is used

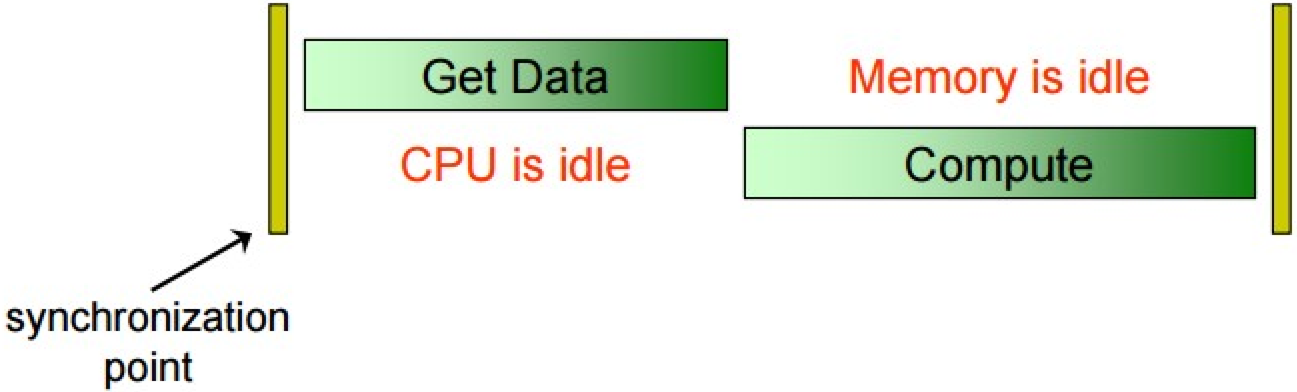
$$ts = te_{old} / te_{new} = 1 / [(1 – f) + (f / ps)]$$

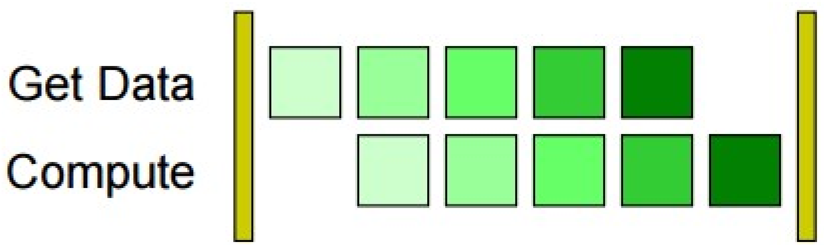
Communication Cost Model

$$C = f * \left(o + l + \frac{n/m}{B} + t - overlap\right)$$

- f: frequency of messages
- o: overhead per message (at both ends)
- l: network delay per message
- n: total data sent
- m: number of messages
- B: bandwidth along path (determined by network)
- t: cost induced by contention per message
- $overlap$: amount of latency hidden by concurrency with computation

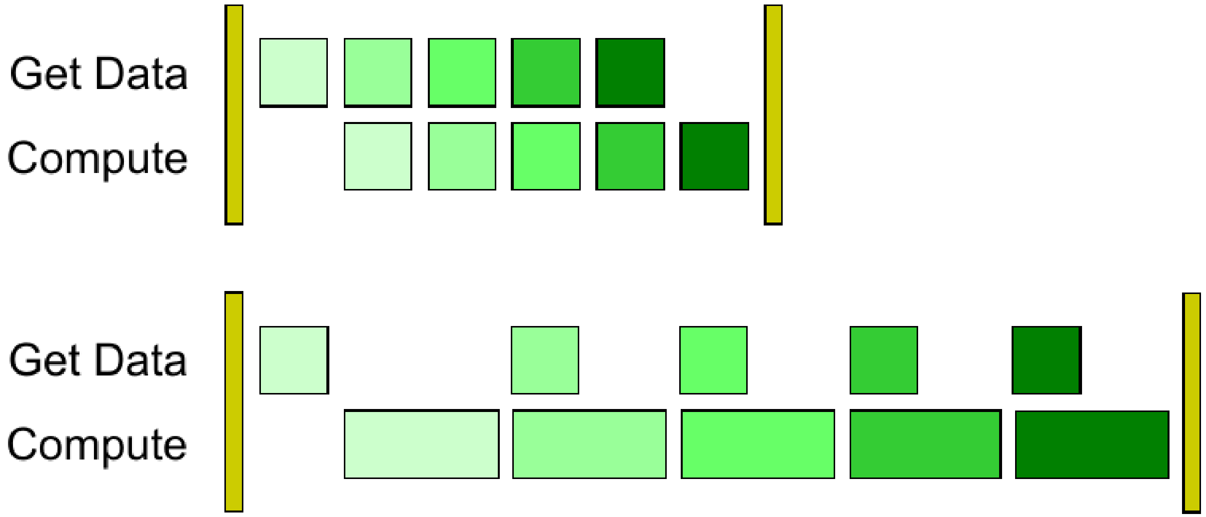
Overlapping Communication with Computation





Limits in Pipelining Communication

- Computation to communication ratio limits performance gains from pipelining



- Where else to look for performance?

Artifactual Communication

- Determined by program implementation and interactions with the architecture

- Examples:
  - Poor distribution of data across distributed memories
  - Unnecessarily fetching data that is not used
  - Redundant data fetches

Lessons From Uniprocessors

In uniprocessors, CPU communicates with memory

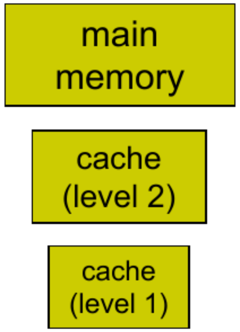
- Loads and stores are to uniprocessors equivalent to **get** and **put** in distributed memory multiprocessors

- How is communication overlap enhanced in uniprocessors?
  - Spatial locality
  - Temporal locality

Spatial Locality

- CPU asks for data at address 1000
- Memory sends data at address 1000 … 1064
  - Amount of data sent depends on architecture parameters such as the cache block size
- Works well if CPU actually ends up using data from 1001, 1002, …, 1064
- Otherwise wasted bandwidth and cache capacity

Temporal Locality



- Main memory access is expensive
- Memory hierarchy adds small but fast memories (caches) near the CPU
  - Memories get bigger as distance from CPU increases
- CPU asks for data at address 1000
- Memory hierarchy anticipates more accesses to same address and stores a local copy
- Works well if CPU actually ends up using data from 1000 over and over and over ...
- Otherwise wasted cache capacity

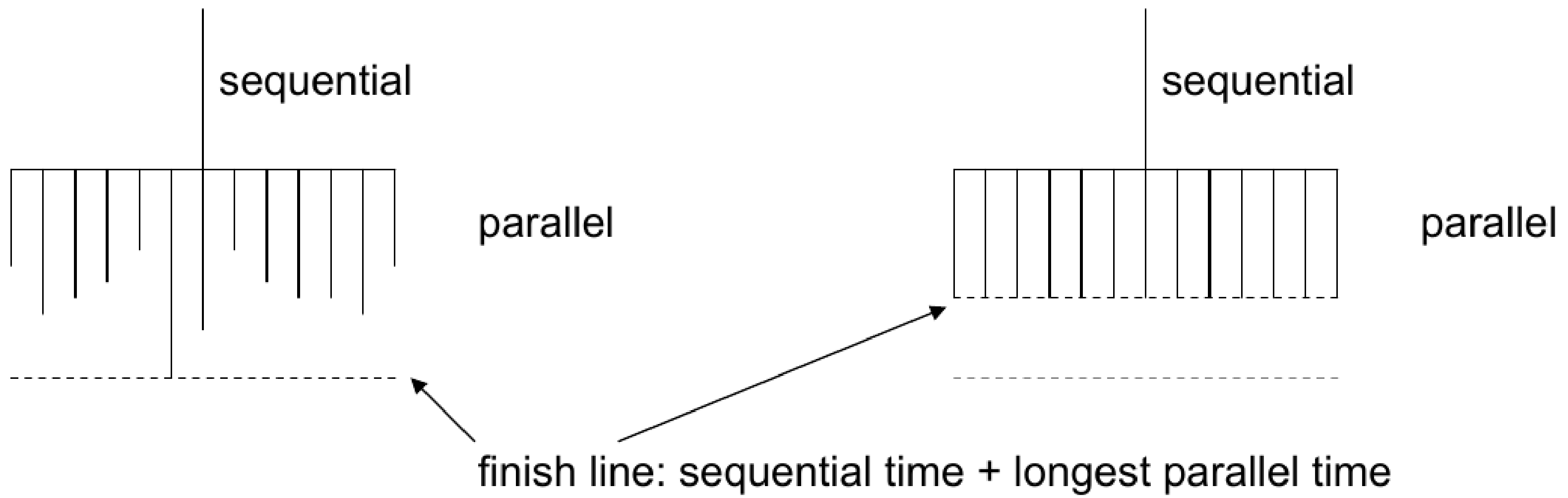
Reducing Artifactual Costs in Distributed Memory Architectures

- Data is transferred in chunks to amortize communication cost
- Spatial locality
  - Computation should exhibit good spatial locality characteristics
- Temporal locality
  - Reorder computation to maximize use of data fetched

Single Thread Performance

Single Thread Performance

- Tasks mapped to execution units (threads)
- Threads run on individual processors (cores)



- Two keys for faster execution
  - Load balance the work among the processors
  - Make execution on each processor faster

Understanding performance

- Need some way of measuring performance
  - Coarse grained measurements

```
% gcc sample.c
% time a.out
2.312u 0.062s 0:02.50 94.8%
% gcc sample.c -O3
% time a.out
1.921u 0.093s 0:02.03 99.0%
```

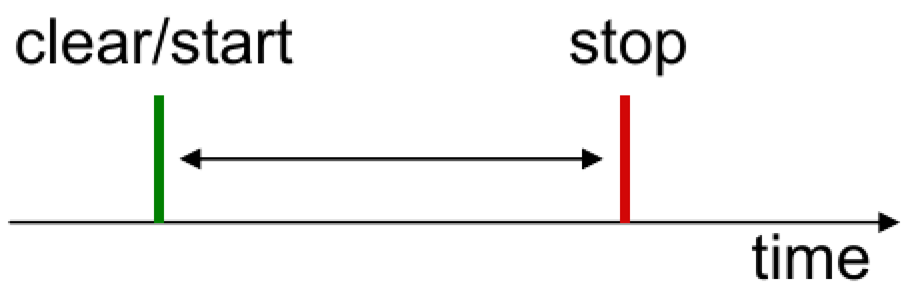
- *Did we learn something of what is going on?*

```
#define N (1 << 23)
#define T (10)
#include <string.h>
double a[N],b[N];
    void cleara(double a[N]) {
    int i;
    for (i = 0; i < N; i++) {
    a[i] = 0;
    }
}
int main() {
    double s=0,s2=0; int i,j;
    for (j = 0; j < T; j++) {
        for (i = 0; i < N; i++) {
            b[i] = 0;
    }
    cleara(a);
    memset(a,0,sizeof(a));
//start record
    for (i = 0; i < N; i++) {
        s += a[i] * b[i];
        s2 += a[i]*a[i] + b[i]*b[i];
        }
    }
//stop record
    printf("s %f s2 %f\n",s,s2);
}
```

Measurements Using Counters

- Increasingly possible to get accurate measurements using performance counters
  - Special registers in the hardware to measure events

- Insert code to start, read, and stop counter
  - Measure exactly what you want, anywhere you want
  - Can measure communication and computation duration
  - But requires manual changes
  - Monitoring nested scopes is an issue
  - Heisenberg effect: counters can perturb execution time



Dynamic Profiling

- Event-based profiling
- Interrupt execution when an event counter reaches a threshold

- Time-based profiling
- Interrupt execution every t seconds

- Works without modifying your code
  - Does not require that you know where problem might be
  - Supports multiple languages and programming models
  - Quite efficient for appropriate sampling frequencies

Counter Examples

- Cycles (clock ticks)
- Pipeline stalls
- Cache hits
- Cache misses
- Number of instructions
- Number of loads
- Number of stores
- Number of floating point operations
- …

Useful Derived Measurements

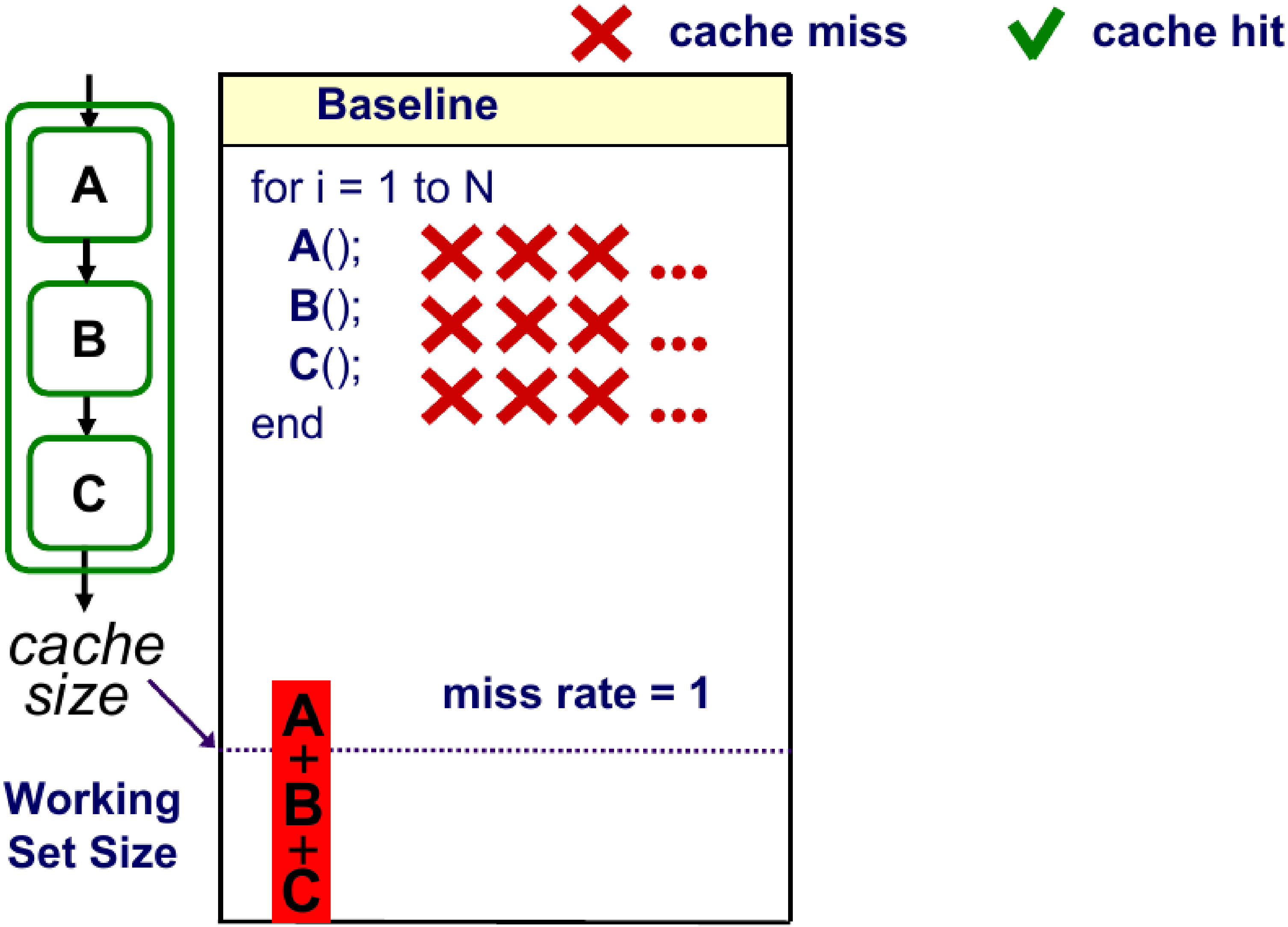
- Processor utilization
  - Cycles / Wall Clock Time
- Instructions per cycle
  - Instructions / Cycles
- Instructions per memory operation
  - Instructions / Loads + Stores
- Average number of instructions per load miss
  - Instructions / L1 Load Misses
- Memory traffic
  - Loads + Stores * Lk Cache Line Size
- Bandwidth consumed
  - Loads + Stores * Lk Cache Line Size / Wall Clock Time
- Many others
  - Cache miss rate
  - Branch misprediction rate
  - …

Popular Runtime Profiling Tools

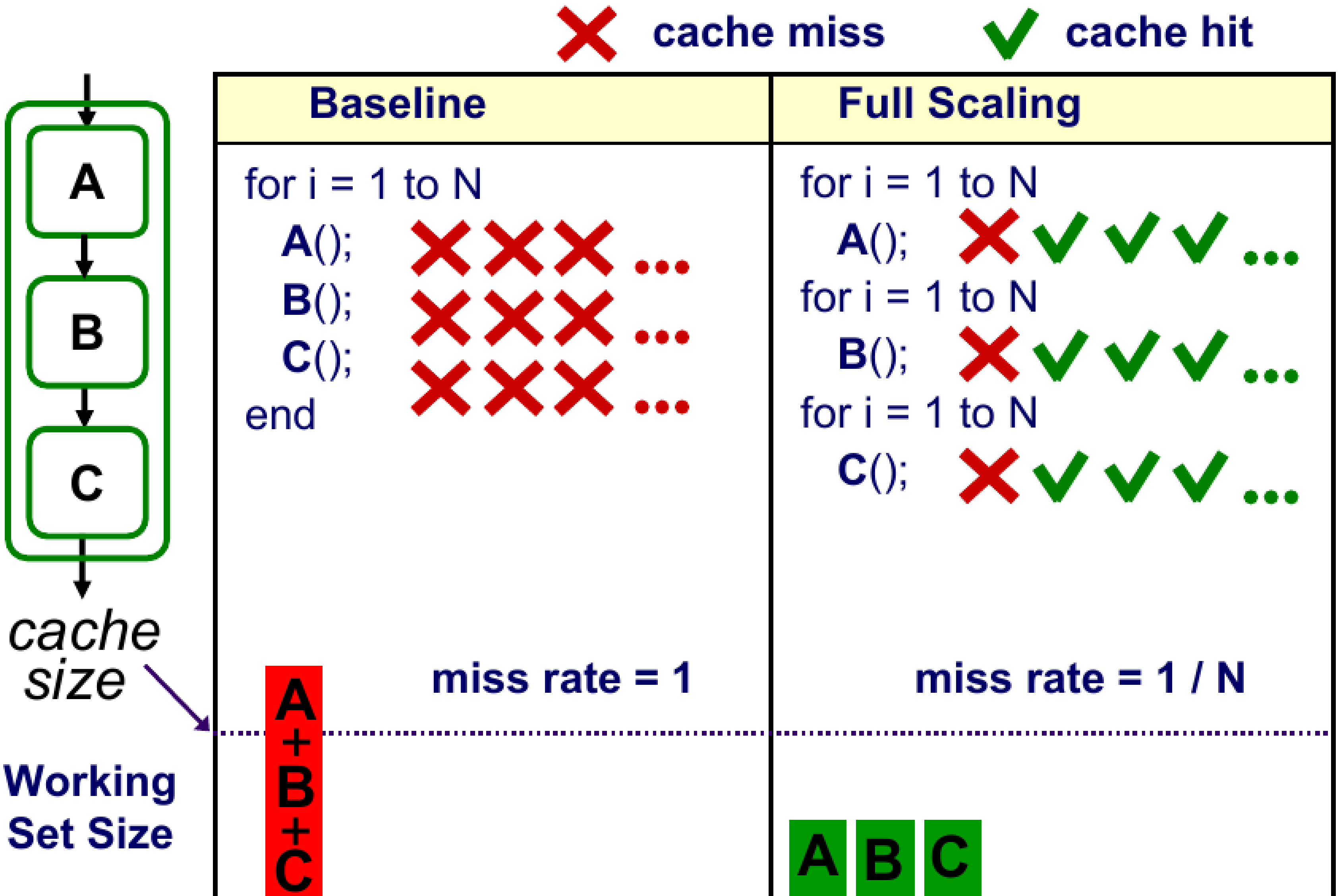
- GNU gprof
  - Widely available with UNIX/Linux distributions
    - `gcc -O2 -pg foo.c -o foo`
    - `./foo`
    - `gprof foo`
- perf tools https://perf.wiki.kernel.org/index.php/Tutorial
  - Needs prior machine configuration
    - `perf`
- Valgrind and its tools: http://valgrind.org/info/tools.html
  - `valgrind --tool=callgrind ./foo`
  - `kcachegrind` to visualize

Now let's try them

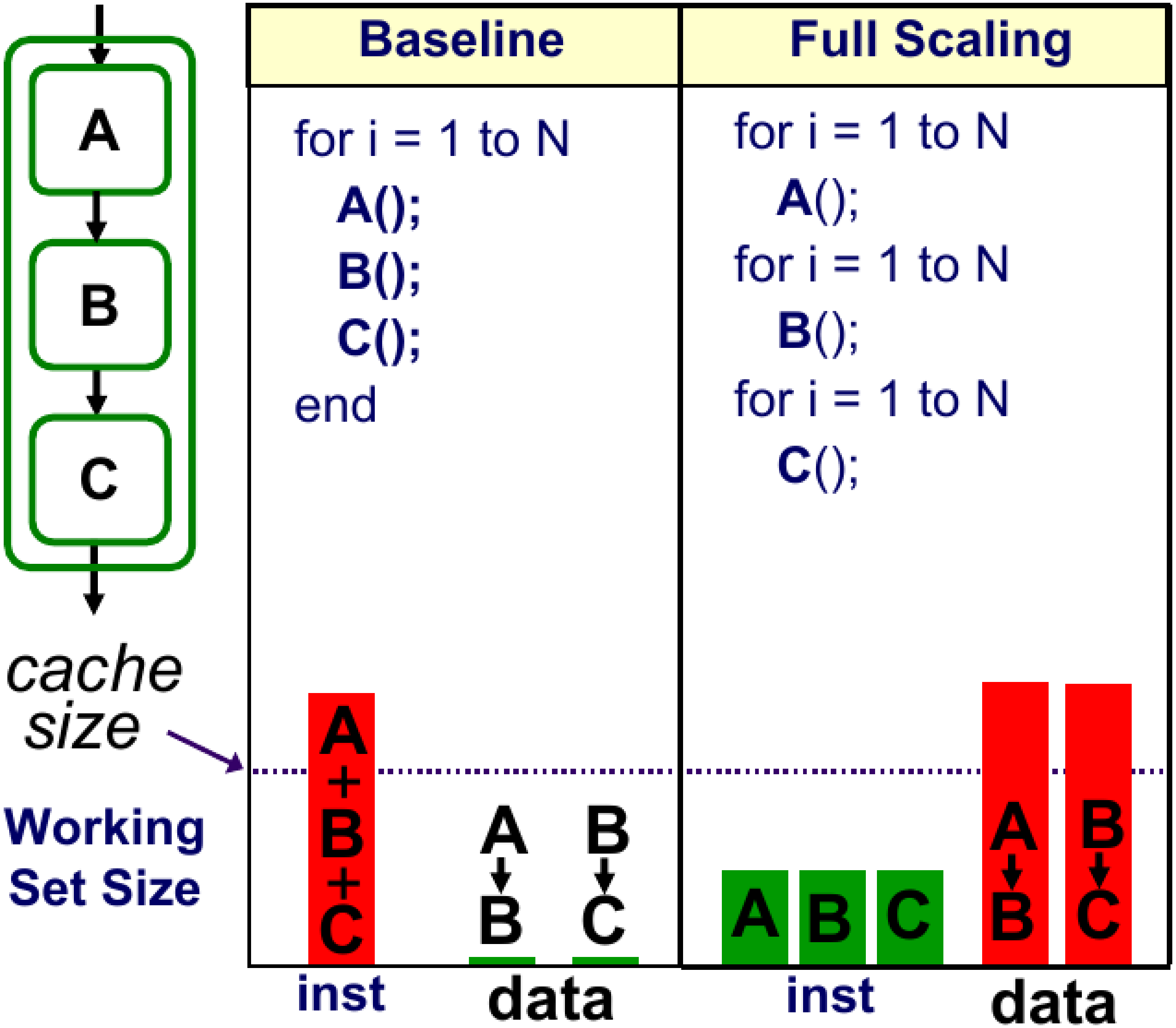
Instruction Locality



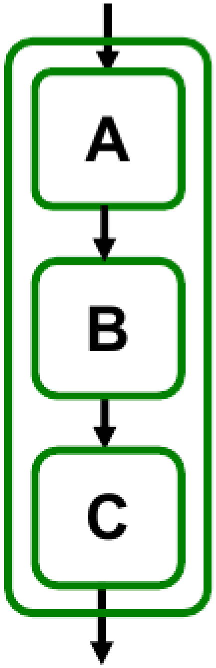
Instruction Locality



Cache Optimization

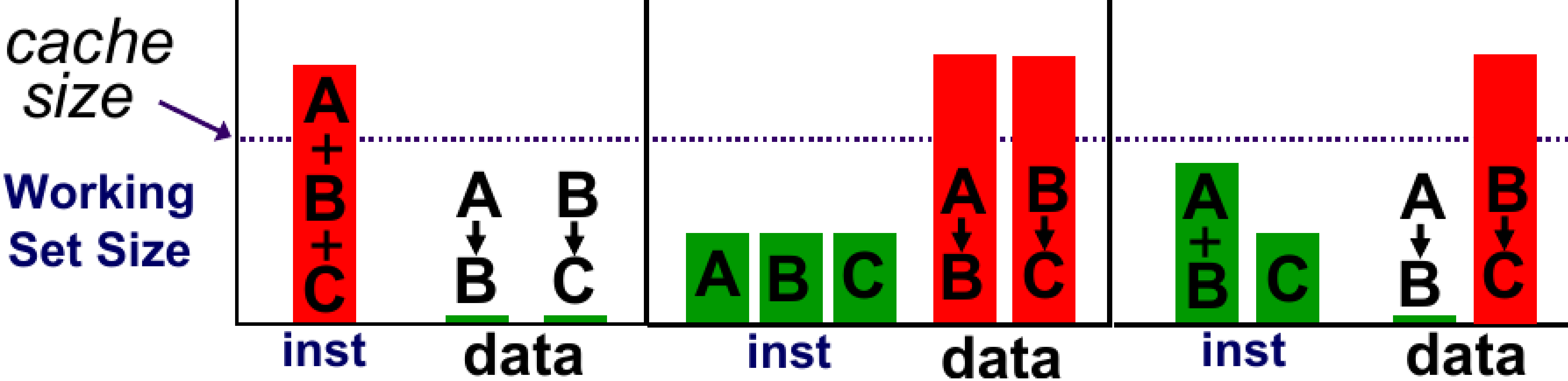


Cache Optimization

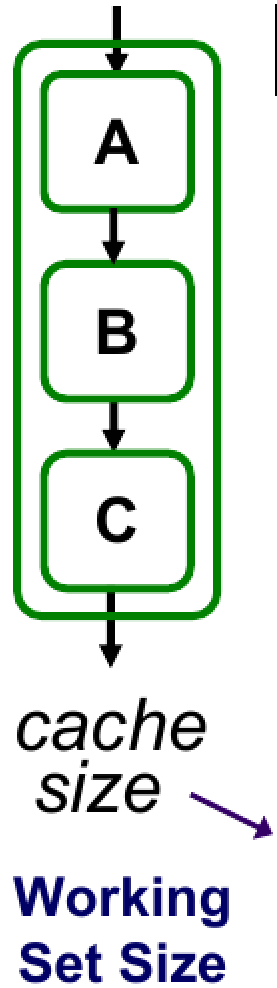


Baseline	Full Scaling	

for i = 1 to N
  A();
  B();
  C();
end
	for i = 1 to N
	  A();
	for i = 1 to N
	  B();
	for i = 1 to N
	  C();
		for i = 1 to N
		  A();
		  B();
		end
		for i = 1 to N
		  C();

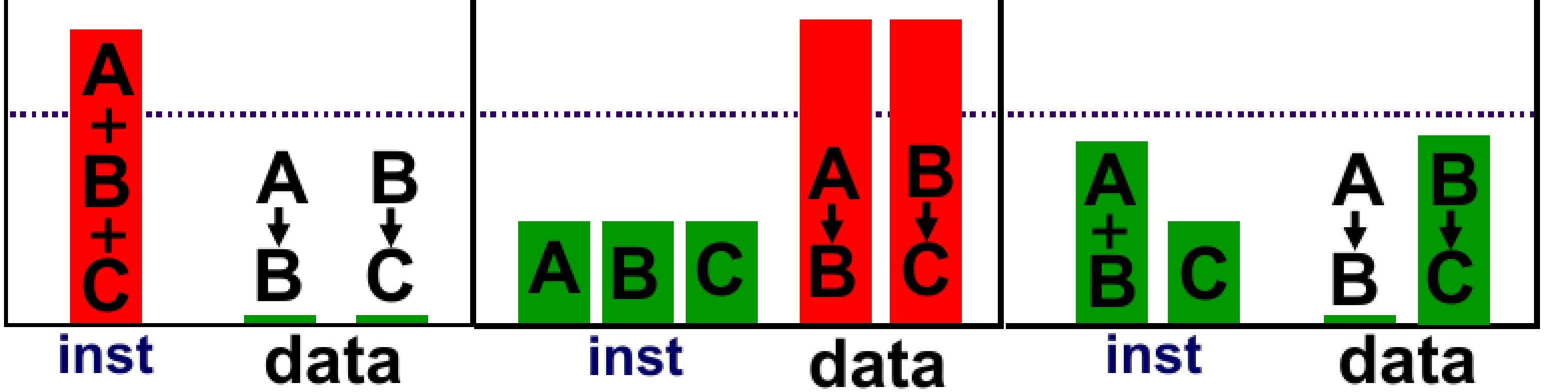


Cache Optimization



Baseline	Full Scaling	Cache Aware

for i = 1 to N
  A();
  B();
  C();
end
	for i = 1 to N
	  A();
	for i = 1 to N
	  B();
	for i = 1 to N
	  C();
		for i = 1 to 64
		  A();
		  B();
		end
		for i = 1 to 64
		  C();
		
		end



Programming for performance

- Tune the parallelism first!
- Then tune performance for each individual processor
  - Instruction level parallelism …
  - Profiling requires lots of probing
- Optimize Memory
  - It is much slower than processor
  - Data locality is essential for performance
  - Remember the model you are using:
    - Hierarchical memory
    - Communication
- May have to change **everything**!
  - Algorithm, data structure, program structure
- Focus on the biggest impediment
  - Amdahl...