Parallel algorithms

Overhead, time, speedup

Paolo Bientinesi

Umeå Universitet
pauldj@cs.umu.se

AQTIVATE workshop
28–29 November 2023

Sequential (serial) algorithms – 1/2

Sequential (serial) algorithms – 1/2

Resources

Sequential (serial) algorithms – 1/2

Resources

Task

Sequential (serial) algorithms – 1/2

Resources **Task**

Cost function: Money (or time)

Sequential (serial) algorithms – 2/2

- **Resources**: $p = 1$ processor (1 worker)

Sequential (serial) algorithms – 2/2

- **Resources**: $p = 1$ processor (1 worker)
- **Task**: Compute/solve/simulate XYZ

Sequential (serial) algorithms – 2/2

- **Resources**: $p = 1$ processor (1 worker)
- **Task**: Compute/solve/simulate XYZ
- **Objective**: Minimize the cost

Sequential (serial) algorithms – 2/2

- **Resources**: $p = 1$ processor (1 worker)
- **Task**: Compute/solve/simulate XYZ
- **Objective**: Minimize the cost
- **Cost**: # FLOPS (floating point operations), # of swaps, # bytes (amount of data) moved, . . . , execution time
- **Cost** as a **function** of the input
  - "size of the problem", "size of the input"
    length of array (n), number of vertices and edges (V, E), . . .
  - Matrix-matrix multiplication: $O(n^3)$, . . . , $O(n^{2.37286})$
    Shortest path: $O(V^2)$, $O((E + V)\log V)$, $O(E + V\log V)$

Parallel algorithms – 1/2

Resources

Task

Cost function Money (or time)

Parallel algorithms – 1/2

Resources

Task

Problems

1. how to split the work?
2. how to coordinate the workers?

Parallel algorithms – 2/2

- **Resources**: $p > 1$ processors
  p *workers; naturally, we want to exploit all of them*
- **Task**: Compute/solve/simulate XYZ
- **Objective**: Minimize the cost

Parallel algorithms – 2/2

- **Resources**: $p > 1$ processors
  p *workers; naturally, we want to exploit all of them*
- **Task**: Compute/solve/simulate XYZ
- **Objective**: Minimize the cost
- **Computational model**: PRAM, EREW
  how the workers "operate" and "communicate"

Parallel algorithms – 2/2

- **Resources**: $p > 1$ processors
  p workers; naturally, we want to exploit all of them
- **Task**: Compute/solve/simulate XYZ
- **Objective**: Minimize the cost
- **Computational model**: PRAM, EREW
  how the workers "operate" and "communicate"
- **Cost**: # FLOPS (floating point operations), # of swaps, # bytes (amount of data) moved, . . . , "execution time"
  definition for "parallel execution time" needed; more later
- **Cost** as a **function** of the input
  - Shortest path: $O(V)$ processors, $O(E+V)$
    $O(\log(V))$ processors, $O(E\bar{k}(N))$

General pattern for parallel algorithms

1. Decomposition of the problem into sub-problems

2. Parallel solution of the sub-problems

3. Composition of the sub-solutions

Example: sequential program

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: 1 grader (= sequential program)

Example: sequential program

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: 1 grader (= sequential program)

- The grader does everything, in whatever order the program dictates.

Example: data vs. task parallelism*

$\star$: definitions might differ

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: p graders

Example: data vs. task parallelism*

$\star$: definitions might differ

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: p graders

- Each grader works on n/p tests, all exercises $\Rightarrow$ **Data parallelism**

Implicit assumption: All exercises can be graded independently

Example: data vs. task parallelism*

$\star$: definitions might differ

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: p graders

- Each grader works on n/p tests, all exercises $\Rightarrow$ **Data parallelism**
  - Are all graders equally fast? Load balancing

Implicit assumption: All exercises can be graded independently

Example: data vs. task parallelism⋆

⋆: definitions might differ

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: p graders

- Each grader works on n/p tests, all exercises $\Rightarrow$ **Data parallelism**
  - Are all graders equally fast? Load balancing
  - Are all graders competent in all exercises? Specialization

Implicit assumption: All exercises can be graded independently

Example: data vs. task parallelism*

$\star$: definitions might differ

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: p graders

- Each grader works on n/p tests, all exercises $\Rightarrow$ **Data parallelism**
  - Are all graders equally fast? Load balancing
  - Are all graders competent in all exercises? Specialization
- Each grader works on t/p exercises, all tests $\Rightarrow$ **Task parallelism**

Implicit assumption: All exercises can be graded independently

Example: data vs. task parallelism*

*: definitions might differ

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: p graders

- Each grader works on n/p tests, all exercises $\Rightarrow$ **Data parallelism**
  - Are all graders equally fast? — Load balancing
  - Are all graders competent in all exercises? — Specialization
- Each grader works on t/p exercises, all tests $\Rightarrow$ **Task parallelism**
  - Are all graders equally fast? — Load balancing

Implicit assumption: All exercises can be graded independently

Example: data vs. task parallelism⋆

⋆: definitions might differ

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: p graders

- Each grader works on n/p tests, all exercises $\Rightarrow$ **Data parallelism**
  - Are all graders equally fast? Load balancing
  - Are all graders competent in all exercises? Specialization
- Each grader works on t/p exercises, all tests $\Rightarrow$ **Task parallelism**
  - Are all graders equally fast? Load balancing
  - Are all exercises equally time consuming? " "

Implicit assumption: All exercises can be graded independently

Example: data vs. task parallelism*

$\star$: definitions might differ

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: p graders

- Each grader works on n/p tests, all exercises $\Rightarrow$ **Data parallelism**
  - Are all graders equally fast? Load balancing
  - Are all graders competent in all exercises? Specialization
- Each grader works on t/p exercises, all tests $\Rightarrow$ **Task parallelism**
  - Are all graders equally fast? Load balancing
  - Are all exercises equally time consuming? " "

Implicit assumption: All exercises can be graded independently

Example: data vs. task parallelism*

$\star$: definitions might differ

Data: n tests, each consisting of t exercises

Task: Grade all exercises of all tests

Resources: p graders

- Each grader works on n/p tests, all exercises $\Rightarrow$ **Data parallelism**
  - Are all graders equally fast? Load balancing
  - Are all graders competent in all exercises? Specialization
- Each grader works on t/p exercises, all tests $\Rightarrow$ **Task parallelism**
  - Are all graders equally fast? Load balancing
  - Are all exercises equally time consuming? " "

Implicit assumption: All exercises can be graded independently

What if this is not the case? **Dependencies** — more later

Demo: Deck sorting

Demo: Deck sorting

- 1 Paolo vs. 3 students
- 1 Paolo vs. 4 students

- Shared memory vs. distributed memory

General pattern for parallel algorithms

1. Decomposition of the problem into sub-problems

   Examples:
   - Data distribution
   - Task decomposition
   - Domain decomposition

2. Parallel solution of the sub-problems

3. Composition of the sub-solutions

All these steps might require

$$\left.\begin{array}{l}\blacktriangleright \text{ Extra computation} \\ \blacktriangleright \text{ Synchronization} \\ \blacktriangleright \text{ Data transfer}\end{array}\right\} \quad \text{OVERHEAD}$$

Time

Time

- **Wall time** or “wall-clock time” :
  real time between the beginning and the end of a computation

Time

- **Wall time** or “wall-clock time” :
  real time between the beginning and the end of a computation

  $T_p(n)$:= Wall time to solve a problem of size n using p procs.

  $T_p(n) = t_1 - t_0$

  t_0 : earliest time when one of the procs. starts its execution,
  t_1 : latest time when one of the procs. completes its execution

Time

- **Wall time** or “wall-clock time” :
  real time between the beginning and the end of a computation

  $T_p(n)$:= Wall time to solve a problem of size n using p procs.

  $T_p(n) = t_1 - t_0$

  t_0 : earliest time when one of the procs. starts its execution,
  t_1 : latest time when one of the procs. completes its execution

- **CPU-time** or “core time” :
  cumulative time spent by all processors in a computation

Time

- **Wall time** or “wall-clock time” :
  real time between the beginning and the end of a computation

  $T_p(n)$:= Wall time to solve a problem of size n using p procs.

  $T_p(n) = t_1 - t_0$

  t_0 : earliest time when one of the procs. starts its execution,
  t_1 : latest time when one of the procs. completes its execution

- **CPU-time** or “core time” :
  cumulative time spent by all processors in a computation

  Cumulative cost for all workers

Questions

- In the analogy of the house, what does wall time measure?

Questions

- In the analogy of the house, what does wall time measure?
- In the analogy of the house, what does CPU-time measure?

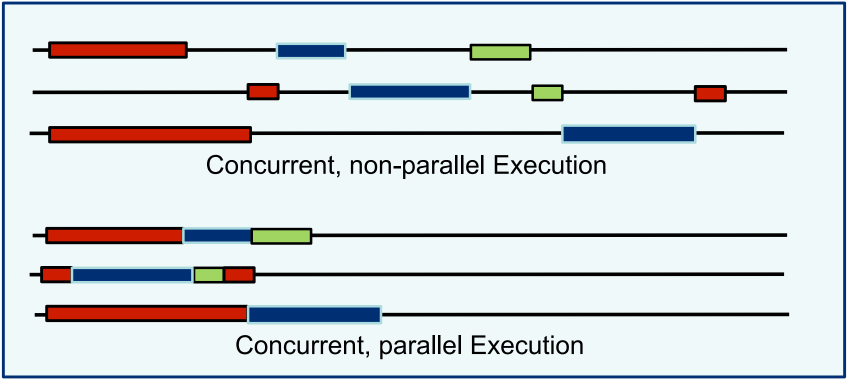
Questions

- In the analogy of the house, what does wall time measure?
- In the analogy of the house, what does CPU-time measure?
- Two programs with same wall time and different CPU-time?

Questions

- In the analogy of the house, what does wall time measure?
- In the analogy of the house, what does CPU-time measure?
- Two programs with same wall time and different CPU-time?
- Two programs with same CPU-time and different wall time?

Concurrency and parallelism*

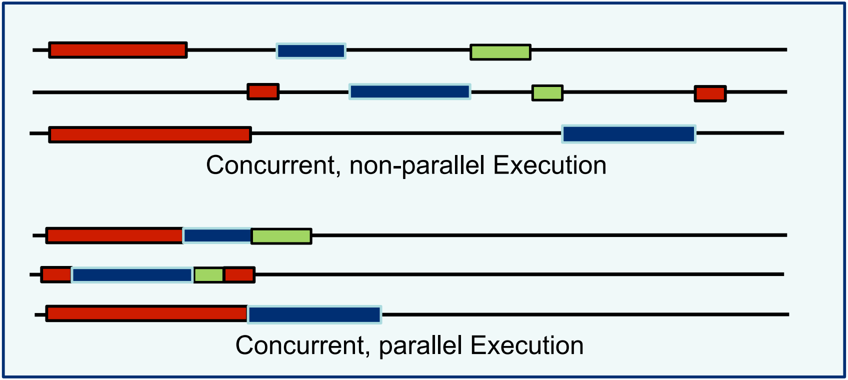


Tim Mattson

Example of programs with same CPU-time but different wall time

*: definitions differ

Concurrency and parallelism⋆



Tim Mattson

Example of programs with same CPU-time but different wall time

Did you spot the mistake?

⋆: definitions differ

$T_p(n)$

- $T_p(n)$:= Wall time to solve a problem of size n using p processors

$T_p(n)$

- $T_p(n)$:= Wall time to solve a problem of size n using p processors
- $n > m \quad \Rightarrow \quad T_p(n) \geq T_p(m)$ Typical evolution with size

$T_p(n)$

- $T_p(n)$:= Wall time to solve a problem of size n using p processors
- $n > m \quad \Rightarrow \quad T_p(n) \geq T_p(m)$ Typical evolution with size

  Example: Eigensolver

  Time complexity: $8n^3$ $T_p(\bar{n}) = t_0$ $T_p(1.5\bar{n}) =?$

$T_p(n)$

- $T_p(n)$:= Wall time to solve a problem of size n using p processors

- $n > m \quad \Rightarrow \quad T_p(n) \geq T_p(m)$ Typical evolution with size

  Example: Eigensolver

  Time complexity: $8n^3$ $T_p(\bar{n}) = t_0$ $T_p(1.5\bar{n}) =?$

  Cost$(1.5\bar{n}) \approx 8\,(1.5\bar{n})^3 = 8 \times 3.375\bar{n}^3 = 3.375 t_0$

$T_p(n)$

- $T_p(n)$:= Wall time to solve a problem of size n using p processors
- $n > m \quad \Rightarrow \quad T_p(n) \geq T_p(m)$ Typical evolution with size

  Example: Eigensolver

  Time complexity: $8n^3$ $T_p(\bar{n}) = t_0$ $T_p(1.5\bar{n}) = ?$

  Cost$(1.5\bar{n}) \approx 8\,(1.5\bar{n})^3 = 8 \times 3.375\bar{n}^3 = 3.375t_0$
- $p > q \quad \Rightarrow \quad T_p(n) < T_q(n)$ Expected evolution with #procs

$T_p(n)$

- $T_p(n)$:= Wall time to solve a problem of size n using p processors

- $n > m \quad \Rightarrow \quad T_p(n) \geq T_p(m)$ Typical evolution with size

  Example: Eigensolver

  Time complexity: $8n^3$ $\quad T_p(\bar{n}) = t_0 \quad T_p(1.5\bar{n}) = ?$

  Cost$(1.5\bar{n}) \approx 8\,(1.5\bar{n})^3 = 8 \times 3.375\bar{n}^3 = 3.375 t_0$

- $p > q \quad \Rightarrow \quad T_p(n) < T_q(n)$ Expected evolution with #procs

  Ideally: $T_{2p}(n) \approx \frac{1}{2}T_p(n), \quad T_{4p}(n) \approx \frac{1}{4}T_p(n), \quad \ldots$

$T_p(n)$

- $T_p(n)$:= Wall time to solve a problem of size n using p processors

- $n > m \quad \Rightarrow \quad T_p(n) \geq T_p(m)$ Typical evolution with size

  Example: Eigensolver

  Time complexity: $8n^3$ $\qquad T_p(\bar{n}) = t_0 \qquad T_p(1.5\bar{n}) =?$

  Cost$(1.5\bar{n}) \approx 8\,(1.5\bar{n})^3 = 8 \times 3.375\bar{n}^3 = 3.375t_0$

- $p > q \quad \Rightarrow \quad T_p(n) < T_q(n)$ Expected evolution with #procs

  Ideally: $T_{2p}(n) \approx \frac{1}{2}T_p(n), \quad T_{4p}(n) \approx \frac{1}{4}T_p(n), \quad \ldots$

  In practice, . . . OVERHEAD!

Parallel Speedup

- **Speedup**: $S_p(n) := \dfrac{T_1(n)}{T_p(n)}$

Parallel Speedup

- **Speedup**: $S_p(n) := \dfrac{T_1(n)}{T_p(n)}$
  - typically: $0 \leq S_p(n) \leq p$
  - if $S_p(n) > p$: "superlinear speedup" $\leftarrow$ rare, but possible

Parallel Speedup

- **Speedup**: $S_p(n) := \dfrac{T_1(n)}{T_p(n)}$
  - typically: $0 \leq S_p(n) \leq p$
  - if $S_p(n) > p$: "superlinear speedup" $\leftarrow$ rare, but possible

- **What is $T_1(n)$?**

Parallel Speedup

- **Speedup**: $S_p(n) := \dfrac{T_1(n)}{T_p(n)}$
  - typically: $0 \leq S_p(n) \leq p$
  - if $S_p(n) > p$: "superlinear speedup" $\leftarrow$ rare, but possible

- **What is $T_1(n)$?**
  - Time of the best sequential code.
  - NOT the time for the parallel code run with $p = 1$!!

Parallel Speedup

- **Speedup**: $S_p(n) := \dfrac{T_1(n)}{T_p(n)}$
  - typically: $0 \leq S_p(n) \leq p$
  - if $S_p(n) > p$: "superlinear speedup" $\leftarrow$ rare, but possible

- **What is $T_1(n)$?**
  - Time of the best sequential code.
  - NOT the time for the parallel code run with $p = 1$!!

Note: The sequential code could possibly implement a different algorithm than the parallel one.

Example: Eigenvalues of symmetric tridiagonal matrix

- Alg.1: `dqds` cost: $O(n^2)$, BUT inherently sequential
- Alg.2: `BX` cost: $O(n^3)$, BUT perfectly parallelizable

Speedup, Efficiency

- **What if p is large?** Then probably $T_1(n)$ is not obtainable, either because n is too large a problem to be solved sequentially, or because it would take too long to complete.

  In this case, $S_p(n) := \dfrac{T_{p_0}(n)}{T_p(n)}$, and $0 \le S_p(n) \le \frac{p}{p_0}$.

  $T_{p_0}(n)$ is used as a reference, possibly from a different code.

  Note: Excellent speedup (by itself) does not imply excellent algorithm.

Speedup, Efficiency

- **What if p is large?** Then probably $T_1(n)$ is not obtainable, either because n is too large a problem to be solved sequentially, or because it would take too long to complete.

  In this case, $S_p(n) := \dfrac{T_{p_0}(n)}{T_p(n)}$, and $0 \le S_p(n) \le \frac{p}{p_0}$.

  $T_{p_0}(n)$ is used as a reference, possibly from a different code.

  Note: Excellent speedup (by itself) does not imply excellent algorithm.

- **Parallel efficiency**:

  $E_p(n) := \dfrac{S_p(n)}{p}$ $\quad \Big($ or $E_p(n) := \frac{S_p(n)}{p/p_0}\Big)$ $\quad 0 \le E_p(n) \le 1$

  Note: Excellent efficiency (by itself) does not imply excellent algorithm.

Performance

- **Performance**: Number of floating point operations per second performed while solving a given problem.

Performance

- **Performance**: Number of floating point operations per second performed while solving a given problem.

- **Theoretical Peak Performance** (TPP): In ideal conditions, the highest number of floating point operations that a processor can perform in one second.

Performance

- **Performance**: Number of floating point operations per second performed while solving a given problem.

- **Theoretical Peak Performance** (TPP): In ideal conditions, the highest number of floating point operations that a processor can perform in one second.

- **Peak Performance** (“Practical peak performance”): The performance attained by highly tuned matrix-matrix multiplication kernels (DGEMM). For instance, MKL and OpenBLAS.

Performance

- **Performance**: Number of floating point operations per second performed while solving a given problem.
- **Theoretical Peak Performance** (TPP): In ideal conditions, the highest number of floating point operations that a processor can perform in one second.
- **Peak Performance** (“Practical peak performance”): The performance attained by highly tuned matrix-matrix multiplication kernels (DGEMM). For instance, MKL and OpenBLAS.
- **Efficiency**: The ratio between the performance attained while solving a given problem and the TPP (or the PPP).

Speedup & Efficiency in real life

- `time0.c`
  Cholesky factorization.
  No timings. Only correctness.
- `time1.c`
  Timings through `clock()`.
  Multithreading (via LAPACK/BLAS). CPU-time.
- `time2a.c`
  Cycle accurate timer.
  Cycles, frequency. Wall time vs. CPU-time.
- `time2b.c`
  Performance (#ops/sec), efficiency.
- Ex. 6 from Report #1

Scalability – 1/2

1) Scalability with respect to resources (# of processors).

   Question: *How does the execution time change as the amount of resources increases?*

Scalability – 1/2

1) Scalability with respect to resources (# of processors).

   Question: *How does the execution time change as the amount of resources increases?*

   Strong Scalability: Behaviour of $T_p(n)$, as p increases.
   Fixed problem size, increasing number of processes.

Scalability – 1/2

1) Scalability with respect to resources (# of processors).

Question: *How does the execution time change as the amount of resources increases?*

Strong Scalability: Behaviour of $T_p(n)$, as p increases.
Fixed problem size, increasing number of processes.

Example: $n = \bar{n}$; $p = 2^i$, with $i \in [10, \ldots, 14]$

$\Rightarrow \quad T_{1024}(n), T_{2048}(n), T_{4096}(n), T_{8192}(n), T_{16384}(n)$

Scalability – 1/2

1) Scalability with respect to resources (# of processors).

   Question: *How does the execution time change as the amount of resources increases?*

   Strong Scalability: Behaviour of $T_p(n)$, as p increases.
   Fixed problem size, increasing number of processes.

   Example: $\quad n = \bar{n}$; $\quad p = 2^i$, with $i \in [10, \ldots, 14]$

   $\Rightarrow \quad T_{1024}(n), T_{2048}(n), T_{4096}(n), T_{8192}(n), T_{16384}(n)$

2) Scalability with respect to data (problem size).

   Question: *How does the execution time change as the problem size increases?*

Scalability – 1/2

1) Scalability with respect to resources (# of processors).

Question: *How does the execution time change as the amount of resources increases?*

Strong Scalability: Behaviour of $T_p(n)$, as p increases.
Fixed problem size, increasing number of processes.

Example: $n = \bar{n}$; $p = 2^i$, with $i \in [10, \ldots, 14]$

$\Rightarrow \quad T_{1024}(n), T_{2048}(n), T_{4096}(n), T_{8192}(n), T_{16384}(n)$

2) Scalability with respect to data (problem size).

Question: *How does the execution time change as the problem size increases?*

Example: $p = \bar{p}$; $n = 2^i \bar{n}$, with $i \in [0, \ldots, 4]$

$\Rightarrow \quad T_p(n),\ T_p(2n),\ T_p(4n),\ T_p(8n),\ T_p(16n)$

Scalability – 2/2

- **Weak Scalability**: Behaviour of $T_k(n)$, as n and k increase to keep the memory usage per process constant.

  Memory load per processor is fixed, problem size and number of processes increase.

Scalability – 2/2

- **Weak Scalability**: Behaviour of $T_k(n)$, as n and k increase to keep the memory usage per process constant.

  Memory load per processor is fixed, problem size and number of processes increase.

  Example: Algorithm $\mathcal{A}$; input: $n \in \mathbf{N}$
  Time complexity: $O(n^3)$ (info not needed)
  Space complexity: $O(n^2)$
  Reference: $\bar{n} = 100$, $\bar{p} = 16$

Scalability – 2/2

- **Weak Scalability**: Behaviour of $T_k(n)$, as n and k increase to keep the memory usage per process constant.

  Memory load per processor is fixed, problem size and number of processes increase.

  Example: Algorithm $\mathcal{A}$; input: $n \in \mathbf{N}$

  Time complexity: $O(n^3)$ (info not needed)

  Space complexity: $O(n^2)$

  Reference: $\bar{n} = 100$, $\bar{p} = 16$

  $\Rightarrow \quad T_{16}(100), T_{64}(200), T_{256}(400), T_{1024}(800)$

Scalability – 2/2

- **Weak Scalability**: Behaviour of $T_k(n)$, as n and k increase to keep the memory usage per process constant.

  Memory load per processor is fixed, problem size and number of processes increase.

  Example: Algorithm $\mathcal{A}$; input: $n \in \mathbf{N}$

  Time complexity: $O(n^3)$ (info not needed)

  Space complexity: $O(n^2)$

  Reference: $\bar{n} = 100$, $\bar{p} = 16$

  $\Rightarrow \quad T_{16}(100), T_{64}(200), T_{256}(400), T_{1024}(800)$

Question: Why two definitions of scalability (strong, weak)?
What are they useful for?

Weak Scalability – Example #1

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$) = $O(n^2)$ Space($\mathcal{B}(n)$) = $3n$ Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors

Weak Scalability – Example #1

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$) = $O(n^2)$ Space($\mathcal{B}(n)$) = $3n$ Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $3\bar{n}$ $\Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$

Weak Scalability – Example #1

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$) = $O(n^2)$ Space($\mathcal{B}(n)$) = $3n$ Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $3\bar{n}$ $\Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$
    Space($2\bar{n}$) = $6\bar{n}$ $\Rightarrow$ $6\bar{n}/?? = const$ $\Rightarrow$

Weak Scalability – Example #1

- Algorithm $\mathcal{B}(n)$;   input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$) = $O(n^2)$ Space($\mathcal{B}(n)$) = $3n$ Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $3\bar{n}$ $\Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$
    Space($2\bar{n}$) = $6\bar{n}$ $\Rightarrow$ $6\bar{n}/?? = const$ $\Rightarrow$ $?? = 2\bar{p}$

Weak Scalability – Example #1

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$) = $O(n^2)$ Space($\mathcal{B}(n)$) = $3n$ Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $3\bar{n}$ $\Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$
    Space($2\bar{n}$) = $6\bar{n}$ $\Rightarrow$ $6\bar{n}/?? = const$ $\Rightarrow$ $?? = 2\bar{p}$
  - $T_{2\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?

Weak Scalability – Example #1

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$) = $O(n^2)$ Space($\mathcal{B}(n)$) = $3n$ Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $3\bar{n}$ $\Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$
    Space($2\bar{n}$) = $6\bar{n}$ $\Rightarrow$ $6\bar{n}/?? = const$ $\Rightarrow$ $?? = 2\bar{p}$
  - $T_{2\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?
    Assumption: perfect scalability wrt size, strong scalability
    $t_1 = T_{2\bar{p}}(2\bar{n}) \approx 4T_{2\bar{p}}(\bar{n}) \approx 4T_{\bar{p}}(\bar{n})/2 = 2t_0$

Weak Scalability – Example #1

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$) = $O(n^2)$ Space($\mathcal{B}(n)$) = $3n$ Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $3\bar{n}$ $\Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$
    Space($2\bar{n}$) = $6\bar{n}$ $\Rightarrow$ $6\bar{n}/?? = const$ $\Rightarrow$ $?? = 2\bar{p}$
  - $T_{2\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?
    Assumption: perfect scalability wrt size, strong scalability
    $t_1 = T_{2\bar{p}}(2\bar{n}) \approx 4T_{2\bar{p}}(\bar{n}) \approx 4T_{\bar{p}}(\bar{n})/2 = 2t_0$
  - $T_{2\bar{p}}(??)$ – from number of processors to problem size

Weak Scalability – Example #1

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$) = $O(n^2)$ Space($\mathcal{B}(n)$) = $3n$ Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $3\bar{n}$ $\Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$
    Space($2\bar{n}$) = $6\bar{n}$ $\Rightarrow$ $6\bar{n}/?? = const$ $\Rightarrow$ $?? = 2\bar{p}$
  - $T_{2\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?
    Assumption: perfect scalability wrt size, strong scalability
    $t_1 = T_{2\bar{p}}(2\bar{n}) \approx 4T_{2\bar{p}}(\bar{n}) \approx 4T_{\bar{p}}(\bar{n})/2 = 2t_0$
  - $T_{2\bar{p}}(??)$ – from number of processors to problem size
    Space($\bar{n}$) = $3\bar{n}$ $\Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$

Weak Scalability – Example #1

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$) = $O(n^2)$ Space($\mathcal{B}(n)$) = $3n$ Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $3\bar{n}$ $\Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$
    Space($2\bar{n}$) = $6\bar{n}$ $\Rightarrow$ $6\bar{n}/?? = const$ $\Rightarrow$ $?? = 2\bar{p}$
  - $T_{2\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?
    Assumption: perfect scalability wrt size, strong scalability
    $t_1 = T_{2\bar{p}}(2\bar{n}) \approx 4T_{2\bar{p}}(\bar{n}) \approx 4T_{\bar{p}}(\bar{n})/2 = 2t_0$
  - $T_{2\bar{p}}(??)$ – from number of processors to problem size
    Space($\bar{n}$) = $3\bar{n}$ $\Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$
    Space($??$) = $3\ ??$ $\Rightarrow$ Mem/proc: $3\ ??/2\bar{p} = const$ $\Rightarrow$

Weak Scalability – Example #1

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$) = $O(n^2)$ Space($\mathcal{B}(n)$) = $3n$ Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $3\bar{n} \Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$
    Space($2\bar{n}$) = $6\bar{n} \Rightarrow 6\bar{n}/?? = const \Rightarrow ?? = 2\bar{p}$
  - $T_{2\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?
    Assumption: perfect scalability wrt size, strong scalability
    $t_1 = T_{2\bar{p}}(2\bar{n}) \approx 4T_{2\bar{p}}(\bar{n}) \approx 4T_{\bar{p}}(\bar{n})/2 = 2t_0$
  - $T_{2\bar{p}}(??)$ – from number of processors to problem size
    Space($\bar{n}$) = $3\bar{n} \Rightarrow$ Mem/proc: $3\bar{n}/\bar{p} =: const$
    Space($??$) = $3\,?? \Rightarrow$ Mem/proc: $3\,??/2\bar{p} = const \Rightarrow ?? = 2\bar{n}$

Weak Scalability – Example #2

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$): $O(n^2)$ Space($\mathcal{B}(n)$): n^2 Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors

Weak Scalability – Example #2

- Algorithm $\mathcal{B}(n)$;   input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$): $O(n^2)$ Space($\mathcal{B}(n)$): n^2 Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $\bar{n}^2$ $\Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$

Weak Scalability – Example #2

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$): $O(n^2)$ Space($\mathcal{B}(n)$): n^2 Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $\bar{n}^2$ $\Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$
    Space($2\bar{n}$) = $4\bar{n}^2$ $\Rightarrow$ $4\bar{n}^2/?? = const$ $\Rightarrow$

Weak Scalability – Example #2

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$): $O(n^2)$ Space($\mathcal{B}(n)$): n^2 Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $\bar{n}^2$ $\Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$
    Space($2\bar{n}$) = $4\bar{n}^2$ $\Rightarrow$ $4\bar{n}^2/?? = const$ $\Rightarrow$ $?? = 4\bar{p}$

Weak Scalability – Example #2

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$): $O(n^2)$ Space($\mathcal{B}(n)$): n^2 Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $\bar{n}^2$ $\Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$
    Space($2\bar{n}$) = $4\bar{n}^2$ $\Rightarrow$ $4\bar{n}^2/?? = const$ $\Rightarrow$ $?? = 4\bar{p}$
  - $T_{4\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?

Weak Scalability – Example #2

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$): $O(n^2)$ Space($\mathcal{B}(n)$): n^2 Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $\bar{n}^2$ $\Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$
    Space($2\bar{n}$) = $4\bar{n}^2$ $\Rightarrow$ $4\bar{n}^2/?? = const$ $\Rightarrow$ $?? = 4\bar{p}$
  - $T_{4\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?
    Assumption: perfect scalability
    $t_1 = T_{4\bar{p}}(2\bar{n}) \approx 4T_{4\bar{p}}(\bar{n}) \approx 4T_{\bar{p}}(\bar{n})/4 = t_0$

Weak Scalability – Example #2

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$): $O(n^2)$ Space($\mathcal{B}(n)$): n^2 Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $\bar{n}^2$ $\Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$
    Space($2\bar{n}$) = $4\bar{n}^2$ $\Rightarrow$ $4\bar{n}^2/?? = const$ $\Rightarrow$ $?? = 4\bar{p}$
  - $T_{4\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?
    Assumption: perfect scalability
    $t_1 = T_{4\bar{p}}(2\bar{n}) \approx 4T_{4\bar{p}}(\bar{n}) \approx 4T_{\bar{p}}(\bar{n})/4 = t_0$
  - $T_{2\bar{p}}(??)$ – from number of processors to problem size

Weak Scalability – Example #2

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$): $O(n^2)$ Space($\mathcal{B}(n)$): n^2 Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $\bar{n}^2 \Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$
    Space($2\bar{n}$) = $4\bar{n}^2 \Rightarrow 4\bar{n}^2/?? = const \Rightarrow ?? = 4\bar{p}$
  - $T_{4\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?
    Assumption: perfect scalability
    $t_1 = T_{4\bar{p}}(2\bar{n}) \approx 4T_{4\bar{p}}(\bar{n}) \approx 4T_{\bar{p}}(\bar{n})/4 = t_0$
  - $T_{2\bar{p}}(??)$ – from number of processors to problem size
    Space($\bar{n}$) = $\bar{n}^2 \Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$

Weak Scalability – Example #2

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$): $O(n^2)$ Space($\mathcal{B}(n)$): n^2 Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $\bar{n}^2$ $\Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$
    Space($2\bar{n}$) = $4\bar{n}^2$ $\Rightarrow$ $4\bar{n}^2/?? = const$ $\Rightarrow$ $?? = 4\bar{p}$
  - $T_{4\bar{p}}(2\bar{n}) = t_1$ $t_0 > / = / < t_1$?
    Assumption: perfect scalability
    $t_1 = T_{4\bar{p}}(2\bar{n}) \approx 4T_{4\bar{p}}(\bar{n}) \approx 4T_{\bar{p}}(\bar{n})/4 = t_0$
  - $T_{2\bar{p}}(??)$ – from number of processors to problem size
    Space($\bar{n}$) = $\bar{n}^2$ $\Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$
    Space($??$) = $??^2$ $\Rightarrow$ Mem/proc: $??^2/2\bar{p} = const$ $\Rightarrow$ $??^2 = 2\bar{p}\bar{n}^2/\bar{p}$
    $\Rightarrow$

Weak Scalability – Example #2

- Algorithm $\mathcal{B}(n)$; input: $n \in \mathbf{N}$
  Time($\mathcal{B}(n)$): $O(n^2)$ Space($\mathcal{B}(n)$): n^2 Reference: $T_{\bar{p}}(\bar{n}) = t_0$
  - $T_{??}(2\bar{n})$ – from problem size to number of processors
    Space($\bar{n}$) = $\bar{n}^2 \;\Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$
    Space($2\bar{n}$) = $4\bar{n}^2 \;\Rightarrow\; 4\bar{n}^2/?? = const \;\Rightarrow\; ?? = 4\bar{p}$
  - $T_{4\bar{p}}(2\bar{n}) = t_1 \qquad t_0 > / = / < t_1$?
    Assumption: perfect scalability
    $t_1 = T_{4\bar{p}}(2\bar{n}) \approx 4T_{4\bar{p}}(\bar{n}) \approx 4T_{\bar{p}}(\bar{n})/4 = t_0$
  - $T_{2\bar{p}}(??)$ – from number of processors to problem size
    Space($\bar{n}$) = $\bar{n}^2 \;\Rightarrow$ Mem/proc: $\bar{n}^2/\bar{p} =: const$
    Space($??$) = $??^2 \;\Rightarrow$ Mem/proc: $??^2/2\bar{p} = const \;\Rightarrow\; ??^2 = 2\bar{p}\bar{n}^2/\bar{p}$
    $\Rightarrow\; ?? = \bar{n}\sqrt{2}$

Weak Scalability in real life

“A Parallel Eigensolver for Dense Symmetric Matrices Based on Multiple Relatively Robust Representations”

`https://hpac.cs.umu.se/~pauldj/pubs/PMR3.pdf`

Best possible parallel efficiency?

Best possible parallel efficiency?

1. #flops/p no dependencies

Best possible parallel efficiency?

1. #flops/p — no dependencies
2. Amdahl's law — seq. vs. parallel portions

Best possible parallel efficiency?

1. #flops/p — no dependencies
2. Amdahl's law — seq. vs. parallel portions
3. Length of the critical path — sequence of deps

Amdahl's law

Maximum possible speedup when only a portion of the code scales

- T_{seq} portion of the algorithm (in secs) which is strictly sequential

Amdahl's law

Maximum possible speedup when only a portion of the code scales

- T_{seq} portion of the algorithm (in secs) which is strictly sequential
- T_{par} portion of the algorithm (in secs) that can be parallelized

Amdahl's law

Maximum possible speedup when only a portion of the code scales

- T_{seq} portion of the algorithm (in secs) which is strictly sequential
- T_{par} portion of the algorithm (in secs) that can be parallelized
- $\beta := \dfrac{T_{\text{seq}}}{T_{\text{seq}} + T_{\text{par}}}$ fraction of the algorithm that is strictly sequential

Amdahl's law

Maximum possible speedup when only a portion of the code scales

- T_{seq} portion of the algorithm (in secs) which is strictly sequential
- T_{par} portion of the algorithm (in secs) that can be parallelized
- $\beta := \dfrac{T_{\text{seq}}}{T_{\text{seq}} + T_{\text{par}}}$ fraction of the algorithm that is strictly sequential
- $T_1(n) := T_{\text{seq}} + T_{\text{par}}$ by definition

Amdahl's law

Maximum possible speedup when only a portion of the code scales

- T_{seq} portion of the algorithm (in secs) which is strictly sequential
- T_{par} portion of the algorithm (in secs) that can be parallelized
- $\beta := \dfrac{T_{\text{seq}}}{T_{\text{seq}} + T_{\text{par}}}$ fraction of the algorithm that is strictly sequential
- $T_1(n) := T_{\text{seq}} + T_{\text{par}}$ by definition
- $T_p(n) := T_{\text{seq}} + T_{\text{par}}/p$ ideal parallelisation

Amdahl's law

Maximum possible speedup when only a portion of the code scales

- T_{seq} portion of the algorithm (in secs) which is strictly sequential
- T_{par} portion of the algorithm (in secs) that can be parallelized
- $\beta := \dfrac{T_{\text{seq}}}{T_{\text{seq}} + T_{\text{par}}}$ fraction of the algorithm that is strictly sequential
- $T_1(n) := T_{\text{seq}} + T_{\text{par}}$ by definition
- $T_p(n) := T_{\text{seq}} + T_{\text{par}}/p$ ideal parallelisation
- Speedup: $S_p(n) := \dfrac{T_1(n)}{T_p(n)} = \dfrac{T_{\text{seq}} + T_{\text{par}}}{T_{\text{seq}} + T_{\text{par}}/p}$
- Ideal Speedup: $\lim\limits_{p\to\infty} S_p(n) = \dfrac{1}{\beta}$

Amdahl's law

Maximum possible speedup when only a portion of the code scales

- T_{seq} portion of the algorithm (in secs) which is strictly sequential
- T_{par} portion of the algorithm (in secs) that can be parallelized
- $\beta := \dfrac{T_{\text{seq}}}{T_{\text{seq}} + T_{\text{par}}}$ fraction of the algorithm that is strictly sequential
- $T_1(n) := T_{\text{seq}} + T_{\text{par}}$ by definition
- $T_p(n) := T_{\text{seq}} + T_{\text{par}}/p$ ideal parallelisation
- Speedup: $S_p(n) := \dfrac{T_1(n)}{T_p(n)} = \dfrac{T_{\text{seq}} + T_{\text{par}}}{T_{\text{seq}} + T_{\text{par}}/p}$
- Ideal Speedup: $\lim_{p \to \infty} S_p(n) = \dfrac{1}{\beta}$

Note: Counterpart of Amdahl's law for weak scalability: Gustafson's law.

Amdahl's law in real life

- `time3.c`
  Timings breakdown: Malloc, init, compute, test.
  Scalability. Serial vs. parallel code. Amdahl's law.

- "A Scalable, Linear-Time Dynamic Cutoff Algorithm for MD"

  `https://arxiv.org/pdf/1701.05242.pdf`

- Ex. 3 from Report #1

Critical path

Longest sequence of dependent tasks

$\Rightarrow$ Dependencies

On dependencies and parallel execution

- Two independent instructions can be executed "concurrently" = "in parallel" (by different resources)

On dependencies and parallel execution

- Two independent instructions can be executed “concurrently” = “in parallel” (by different resources)

- Parallel execution: **NO** time ordering!
  When two instructions are executed in parallel,
  NO assumptions about start, duration, end.

On dependencies and parallel execution

- Two independent instructions can be executed
  “concurrently” = “in parallel” (by different resources)

- Parallel execution: **NO** time ordering!
  When two instructions are executed in parallel,
  NO assumptions about start, duration, end.

- Dependencies $\Rightarrow$ ordering
  The “right” ordering is dictated by the semantics of the program

On dependencies and parallel execution

- Two independent instructions can be executed
  "concurrently" = "in parallel"   (by different resources)

- Parallel execution: **NO** time ordering!
  When two instructions are executed in parallel,
  NO assumptions about start, duration, end.

- Dependencies $\Rightarrow$ ordering
  The "right" ordering is dictated by the semantics of the program

- Some dependencies can be removed by duplicating data

True / Flow dependency

```
               {x = 1, y = 2, a = 3}


...
y := a * x + y
w := 3 * y
...

{y = 5, w = 15}
```

True / Flow dependency

```
               {x = 1, y = 2, a = 3}

...
y := a * x + y
w := 3 * y
...

{y = 5, w = 15}
```

- The value of `w` depends on the updated value of `y`

True / Flow dependency

```
            {x = 1, y = 2, a = 3}

...                        ...
y := a * x + y       |     w := 3 * y
w := 3 * y           |     y := a * x + y
...                        ...

{y = 5, w = 15}            {y = 5, w = 6}
```

- The value of `w` depends on the updated value of `y`
- The semantics of the program depends on the **order** of the statements

Anti dependency

```
                 {x = 1, y = 2, a = 3}


...
w := 3 * y
y := a * x + y
...

{y = 5, w = 6}
```

- The value of `w` depends on the initial value of `y`

Anti dependency

```
              {x = 1, y = 2, a = 3}

...                           ...
w := 3 * y             |     y := a * x + y
y := a * x + y         |     w := 3 * y
...                           ...

{y = 5, w = 6}               {y = 5, w = 15}
```

- The value of `w` depends on the initial value of `y`
- The semantics of the program depends on the **order** of the statements

Output dependency

```
              {x = 1, y = 2, a = 3}


...
w := 3 * y
w := a * x
...

{w = 3}
```

- The value of `w` depends on the order of the statements

Output dependency

```
                 {x = 1, y = 2, a = 3}

  ...                                ...
  w := 3 * y              |          w := a * x
  w := a * x              |          w := 3 * y
  ...                                ...

  {w = 3}                            {w = 6}
```

- The value of `w` depends on the order of the statements
- The semantics of the program depends on the **order** of the statements