IN3200/IN4200: More about parallelization

Chapter 5 in textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Plus examples from A. Grama, A. Gupta, G. Karypis, and V. Kumar: "Introduction to Parallel Computing", Addison Wesley, 2003

Content

- Simple theoretical insights into the factors that can hamper parallel performance
- More examples of identifying parallelism
- Simple design of parallel algorithms

Parallel scalability

The *ideal* goal: If a problem takes time T to be solved by one worker, we expect the solution time by using N identical workers to be T/N—a perfect **speedup** of N.

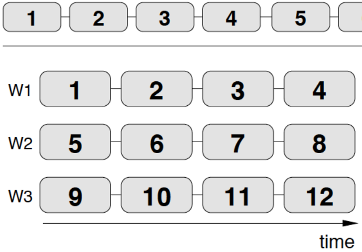


Figure 5.4: Parallelizing a sequence of tasks (top) using three workers (W1...W3) with perfect speedup (left).

However, perfect speedup is often not achievable in reality, why?

Factors that limit parallel execution

Reasons for non-perfect speedup:

- Not all workers might execute their tasks equally fast, because the problem was not (or could not be) partitioned into equal pieces—load imbalance;
- There might be shared resources which can only used by one worker at a time—serialization;
- New tasks may arise due to parallelization, such as communication between workers—overhead.

Example of load imbalance

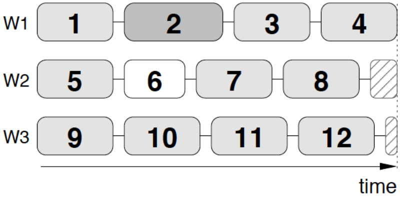


Figure 5.5: Some tasks executed by different workers at different speeds lead to *load imbalance*. Hatched regions indicate unused resources.

Example of serialization

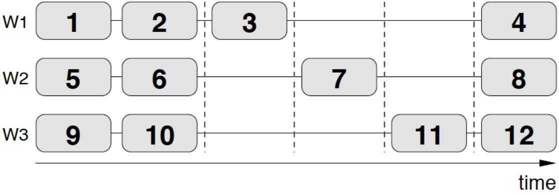


Figure 5.6: Parallelization with a bottleneck. Tasks 3, 7 and 11 cannot overlap with anything else across the dashed "barriers."

Example of communication overhead

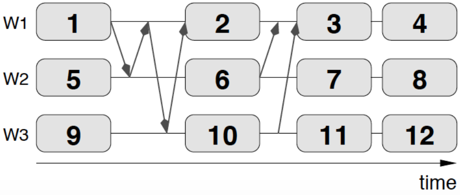


Figure 5.7: Communication processes (arrows represent messages) limit scalability if they cannot be overlapped with each other or with calculation.

Scalability metrics

How well can a computational problem be parallelized?

Scalability metrics help to answer the following questions:

- How much faster can a given problem be solved with N workers instead of one?
- How much more work can be done with N workers instead of one?
- What impact do the communication requirements have on performance and scalability?
- What fraction of the resources is actually used productively?

Strong and weak scaling

Starting point: The overall problem size ("amount of work") is *normalized* as

$$s + p = 1$$

where s is the serial, non-parallelizable fraction, p is the perfectly parallelizable fraction.

We can now define *strong scaling* and *weak scaling*, and study the relationship between single-worker serial runtime and multi-worker parallel runtime.

Strong scaling

Single-worker (serial) normalized runtime for a fixed-size problem:

$$T_\mathrm{f}^\mathrm{s} = s + p$$

Solving the same problem using N workers will require a runtime of

$$T_\mathrm{f}^\mathrm{p} = s + \frac{p}{N}$$

This is called **strong scaling**, because the total amount of work stays constant no matter how many workers are used.

Here, the goal of parallelization is minimization of time-to-solution for a given problem.

Weak scaling

For **weak scaling**, the goal is to solve an increasingly larger problem with more workers N.

More specifically, the total amount of work is scaled with some power of N

$$s + pN^{\alpha} \qquad (\alpha \text{ is a positive parameter})$$

which means that single-worker runtime for the variable-sized problem **would have been** $T_{\mathrm{v}}^{\mathrm{s}} = s + pN^{\alpha}$.

Using N workers, the parallel runtime is

$$T_{\mathrm{v}}^{\mathrm{p}} = s + pN^{\alpha-1}$$

Here, we have also assumed that s doesn't grow with N.

The most typical choice is $\alpha = 1$, then $T_{\mathrm{v}}^{\mathrm{s}} = s + pN$ and $T_{\mathrm{v}}^{\mathrm{p}} = s + p$.

Simple scalability laws

How to calculate speedup?

$$\text{application speedup} = \frac{\text{serial runtime}}{\text{parallel runtime}}$$

or equivalently

$$\text{application speedup} = \frac{\text{parallel performance}}{\text{serial performance}}$$

where "performance" is defined as "work over time".

Amdahl's law

For a fixed problem size $s + p = 1$, the application speedup ("scalability") is

$$S_{\mathrm{f}} = \frac{T_{\mathrm{f}}^{\mathrm{s}}}{T_{\mathrm{f}}^{\mathrm{p}}} = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{1-s}{N}}$$

This is "Amdahl's law"—maximum speedup is $1/s$ when $N \to \infty$.

Gustafson's law

The problem size is scaled with the number of workers N.

Recall that for $\alpha = 1$ we have $T_{\mathrm{v}}^{\mathrm{s}} = s + pN$ and $T_{\mathrm{v}}^{\mathrm{p}} = s + p$. Therefore the application speedup is

$$S_{\mathrm{v}} = \frac{T_{\mathrm{v}}^{\mathrm{s}}}{T_{\mathrm{v}}^{\mathrm{p}}} = \frac{s + pN}{s + p} = \frac{s + (1 - s)N}{1} = s + (1 - s)N$$

This is "Gustafson's law"—speedup can be arbitrarily large when $N \to \infty$.

Parallel efficiency

How effectively is the resource used by parallel program?

Parallel efficiency is defined as

$$\varepsilon = \frac{\text{speedup}}{N}$$

This will be a value between 0 and 100%.

Negative impact of load imbalance

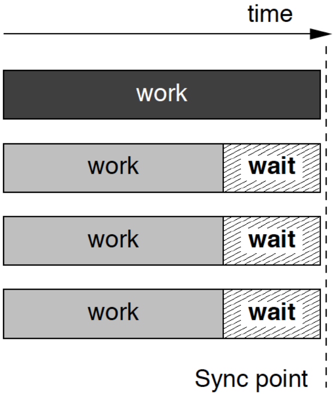


Figure 5.13: Load imbalance with few (one in this case) “laggers”: A lot of resources are underutilized (hatched areas).

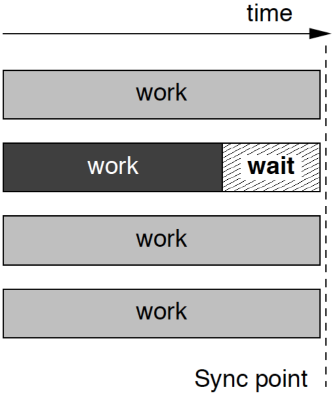


Figure 5.14: Load imbalance with few (one in this case) “speeders”: Underutilization may be acceptable.

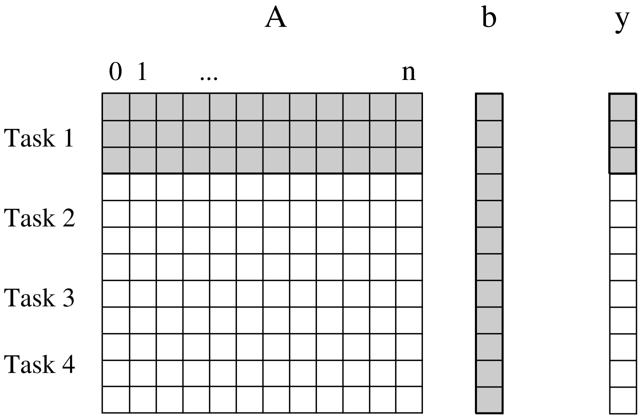
Example: dense matrix-vector multiply

Dense matrix-vector multiply

$$\mathbf{y} = \mathbf{Ab}$$

```
for (i=0; i<N; i++) {
  double tmp = 0.;
  for (j=0; j<N; j++)
    tmp += A[i][j]*b[j];
  y[i] = tmp;
}
```

Parallelization



Decomposition of the outer loop (index `i`) into P chunks, each as the computational task for a processor core. All the tasks are *completely independent*.

Work decomposition

Let N denote the number of entries in vector $\mathbf{y}$ (same as the number of rows in matrix $\mathbf{A}$). If N is divisible by the number of processor cores P, then work decomposition will be perfectly even.

For example: processor core number k ($0 \leq k < P$) can be responsible for computing the following entries of vector $\mathbf{y}$:

```
y[k*chunk_size],
y[k*chunk_size+1],
...
y[(k+1)*chunk_size-1]
```

where `chunk_size=N/P`

Danger for severe load imbalance

What if N is not divisible by P?

Integer division `chunk_size=N/P` will result in

$$\texttt{chunk\_size} = \lfloor \frac{N}{P} \rfloor = \frac{N - \text{modulo}(N, P)}{P}$$

That can easily lead to that $P-1$ processor cores compute each `chunk_size` entries of vector $\mathbf{y}$, whereas one processor core computes modulo(N, P) entries **extra**.

An extreme case of load imbalance arises when $N = 2P - 1$. It will mean that the amount of work for the "heavy-load" processor core is P times of the other processor cores!

Remedy for load balance

The following work decomposition will guarantee that the maximum difference between "heavy-load" and "light-load" tasks is at most 1.

Processor core number k computes

```
y[start_k],
y[start_k+1],
...
y[stop_k-1]
```

where `start_k=k*N/P` and `stop_k=(k+1)*N/P` (integer divisions are used to compute both values).

Example: summing an array of values

```
sum=0.;
for (i=0; i<N; i++)
  sum += y[i];
```

Basic strategy of parallelization:

- Divide the entries of array `y` into as equal-sized chunks as possible

  `start_k=k*N/P` *and* `stop_k=(k+1)*N/P`

- Each processor core *independently* computes a partial sum as

  `y[start_k]` + `y[start_k+1]` +...+ `y[stop_k-1]`

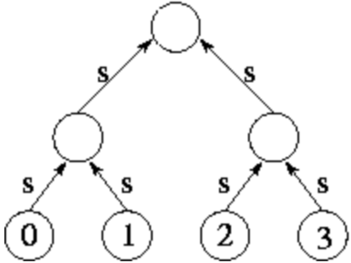
- When all the P partial sums are computed, they are added up to produce the correct value of `sum`

How to sum up P values from P processor cores?

Approach 1: Pick a "master" processor core, and let the master add the P values together.

Downside of this approach: The master core can become a bottleneck if P is large.

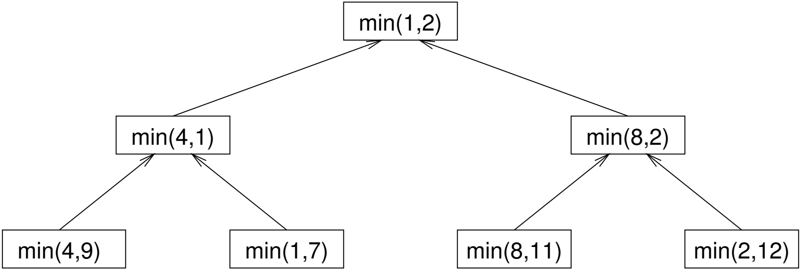
Approach 2: *reverse recursive decomposition*



The "bottom" tasks represent individual partial sums on the processor cores, the other tasks are pair-wise additions until `sum` is computed at the "top".

Another example of reverse recursive decomposition

Suppose we want to find the minimum value in the set $\{4, 9, 1, 7, 8, 11, 2, 12\}$.



Example: Database Query Processing

Consider the execution of the query:

```
MODEL = ''CIVIC'' AND YEAR = 2001 AND
(COLOR = ''GREEN'' OR COLOR = ''WHITE)
```

on the following database:

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Decomposition into tasks

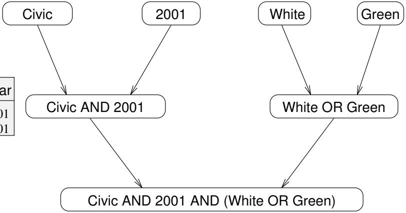
The execution of the query can be divided into tasks. Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause.

ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green



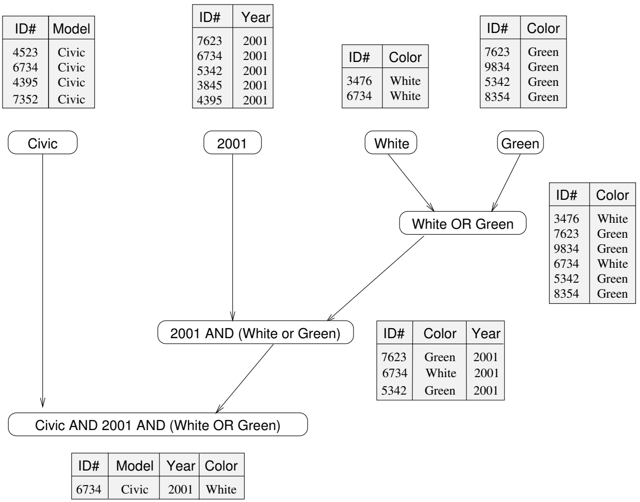
ID#	Model	Year
6734	Civic	2001
4395	Civic	2001

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Model	Year	Color
6734	Civic	2001	White

Decomposing the given query into several tasks. Edges denote that the output of one task is needed to accomplish the next.

Another decomposition



ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Color	Year
7623	Green	2001
6734	White	2001
5342	Green	2001

ID#	Model	Year	Color
6734	Civic	2001	White

Different task decompositions may lead to significant differences with respect to their eventual parallel performance.

Task dependency graph & critical path

Task dependency graph: A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.

The length of the longest path in a task dependency graph is called the critical path length. It also gives the minimum time needed by parallel execution.