Steps in parallelization

Decomposition

How we decompose a problem has a lot to do with the speedup it can achieve.

Example: Consider a program with two phases.

- In the first phase, a single operation is performed on all points of a 2D *n*-by-*n* grid, as in
- In the second phase, the sum of the n^2 grid-point values is computed.

Assume: In a sequential program, the first phase and second phase take the same amount of time (say, 1 unit per point, or __ altogether).

If we have *p* processors, we can assign $\frac{n^2}{p}$ points to each processor, and complete the first phase in parallel in time

In the second phase, each processor adds each of its $\frac{n^2}{p}$ values into a global-sum variable.

What's wrong with this?

Thus, the second phase takes _____ time, regardless of *p*.

What is the best possible speedup in this decomposition? Well, ...

How much time does it take to execute the program sequentially?

How much time does it take to execute it in parallel, with the above decomposition?

The speedup is thus at most ________________ , or $\frac{2p}{p+1}$.

What is the highest speedup we can get, regardless of the number of processors?

An easy way to improve this is to have each processor sum up its $\frac{n^2}{p}$ values into its own local sum.

This gives us a 3-phase program:

1. Operation performed on all n^2 points.
2. Processes add their values independently into private sum.
3. Processes add their private sums into global sum.

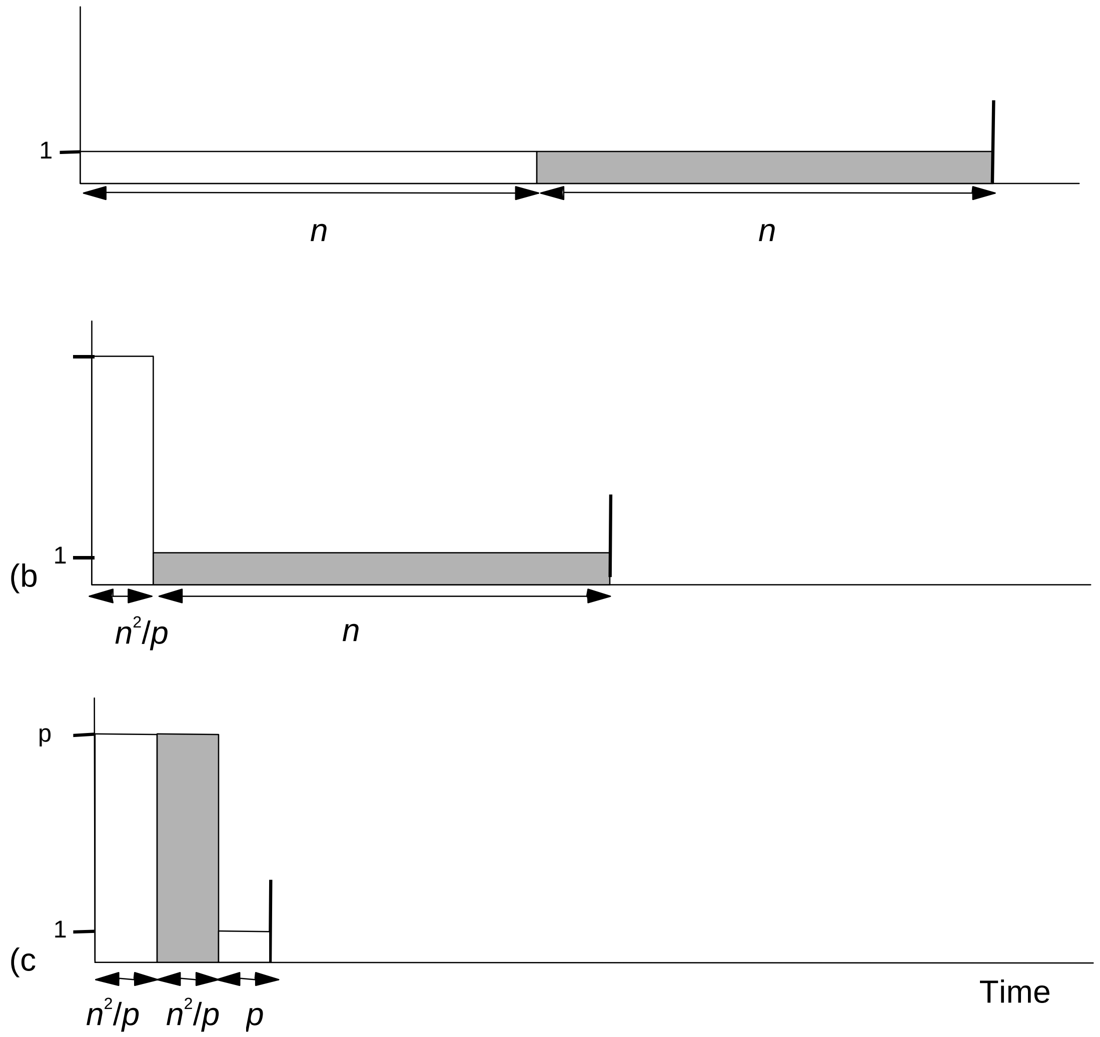
Phase 2 is now fully parallel (all *p* processors operate independently). Phase 3 is sequential, but it is shorter; there are only ____ operations in it, not

The total time for the parallel program is now

The speedup can be as great as

What happens to this speedup as the number of processors *p* increases?

Here is a diagram of what is happening in these decompositions.

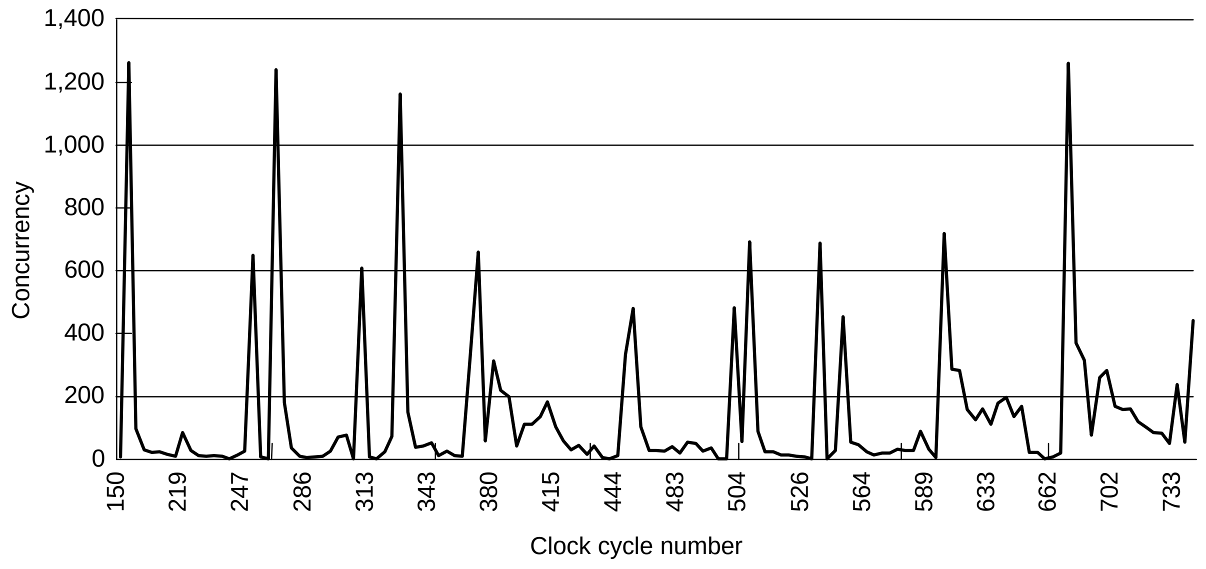


This diagram, depicting how many tasks can be performed concurrently at each point in time, is called a *concurrency profile*.

The concurrency profile depends on the problem and the decomposition. It is independent of the number of processors (it is given in terms of the number of processors *p*).

It is also independent of the assignment or orchestration.

Concurrency profiles may be regular, as above, or they may be irregular, like the profile for this discrete-event logic simulator.



The area under the curve is the amount of work done.

The horizontal extent is a lower bound on the time it would take to run the best parallel program given that decomposition, assuming—

- an infinite number of processors
- ____________________________________ are free.

The area divided by the horizontal extent therefore gives us a limit on the achievable speedup.

Amdahl's law can be expressed as—

$$\text{Speedup} \leq \frac{\text{Area under concurrency profile}}{\text{horizontal extent of concurrency profile}}\,.$$

It is easy to show (see p. 87 of CS&G) that the speedup with an infinite number of processors is limited by 1/*s*, and with *p* processors, it is limited by

$$\frac{1}{s + \frac{1 - s}{p}}\,.$$

Assignment

Assignment means specifying the mechanisms by which tasks will be divided among processes.

- E.g., which process computes forces on which stars, or which rays. What choices do we have in assignment in the ocean simulation?

- Assignment, together with decomposition, is also called *partitioning.*

- Goals of assignment: Balance workload, reduce communication and management cost. Can you explain these?

Structured approaches to assignment usually work well

- Programs are often structured in phases. Programmer can decide on an assignment for each phase.

  Example: A divide-and-conquer algorithm on a message-passing machine, e.g., parallel search, or VLSI design-rule checking.

  Phase 1: One processor starts dividing up the work.

  Depending on the amount of dividing to do, and the amount of data to be moved, other processors may be called in to help.

  Phase 2: Eventually the work is divided among all the processors, and each does the computation on its own portion.

  Phase 3: Processors are assigned to gather up the results from Phase 2. E.g., on a 256-processor system,

  – 16 processors gather the results from all processors in their (16-processor) cluster.

  – One processor gathers the results from all the clusters.

- Assignment can be determined from code inspection (e.g., parallel loops) or other understanding of the application.

Static versus dynamic assignment: If the assignment is predetermined at the beginning of the program, or after reading and analyzing the input, and does not change thereafter, it is called a *static*, or *predetermined*, assignment.

If it is determined as the program executes, it's called a *dynamic assignment*.

As programmers, we worry about partitioning first.

- It is *usually* independent of the architecture or programming model.
- But the cost and complexity of using primitives may affect our decisions.

As architects, we assume the program does a reasonable job of partitioning.

Orchestration

Questions in the orchestration phase include—

- Naming data: What names shall processes use to refer to other processes?
- Structuring communication: How and when shall processes communicate with each other? Small or large messages?
- Synchronization: How and when to synchronize.
- How to scheduling tasks temporally to exploit data locality.

Goals of orchestration

- Reduce cost of communication and synchronization.
- Preserve locality of data reference.
- Schedule tasks early if many other tasks depend on them.

- Reduce the overhead of parallelism management.

These issues are much more dependent on the multiprocessor architecture, programming model, and programming language than choices in earlier steps of parallelization.

- Choices depend a lot on communication abstraction, efficiency of primitives
- Architects should provide appropriate primitives that simplify orchestration.

Mapping

After orchestration, we already have a parallel program.

Mapping has two aspects.

- Which processes will run on the same processor, if necessary.
- Which process runs on which particular processor (e.g., mapping to a network topology).

One extreme: *space-sharing*

- The machine is divided into subsets, with only one application at a time in a particular subset.
- Processes can be *pinned* to processors, or the choice of where to run a process can be left to the OS.

  Why would you want to do this?

  Some form of space-sharing has been typical of large-scale multiprocessors.

Another extreme: Complete resource management control to OS.

  Why would you want to do this?

Scheduling constraints may include—

- Trying to run certain processes at the same time
- Trying to run a process on the same processor as much as possible

In the real world, the situation is usually between the two extremes.

User specifies desires in some aspects, system may ignore for effective resource management.

For simplicity, we will usually adopt the view that there is—

- one process per processor;
- processes are assigned randomly to processors;
- processes remain on the same processor throughout execution.

This lets us use the terms "process" and "processor" interchangeably.

Parallelizing computation vs. data

[§2.2.2] We have talked about parallelizing a computation.

In some cases, the decomposition of work and decomposition of data are strongly related. Which of our example parallel applications is like this?

In this case, the computation follows the data; this is called *owner computes.*

Another example is in data-mining; the database is parceled out among processes, with each being responsible for mining its own portion.

Language systems like High Performance Fortran (HPF) allow the programmer to specify the decomposition and assignment of data structures. Computation then follows the data.

But data and computation are not always so strongly linked.

Examples: Barnes-Hut (*n*-body), ray-tracing

We will consider partitioning to mean partitioning of computation, and consider data access and communication to be part of orchestration

High-level goals of parallelization

[§2.2.3] High performance (speedup over sequential program) ...

but low resource usage and development effort

Table 2.1 Steps in the Parallelization Process and Their Goals

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchronization cost as seen by the processor Reduce serialization at shared resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary Exploit locality in network topology

Implications for algorithm designers and architects:

- Algorithm designers should strive for high performance, low resource needs

- Architects should strive for high performance, low cost, reduced programming effort.

  For example, gradually improving performance with programming effort may be preferable to a sudden threshold after large programming effort.