BASICS OF PARALLELIZATION

Dr Noor Mahamamd Sk

Why Parallelize?

- A single core may be too slow to perform the required task(s) in a “tolerable” amount of time.
- The definition of “tolerable” certainly varies, but “overnight” is often a reasonable estimate.

- The memory requirements cannot be met by the amount of main memory which is available on a single system, because larger problems (with higher resolution, more physics, more particles, etc.) need to be solved.

Parallelism

- Writing a parallel program must always start by identifying the parallelism inherent in the algorithm at hand.
- Different variants of parallelism induce different methods of parallelization.
- We will restrict ourselves to methods for exploiting parallelism using multiple cores or compute nodes.
- The fine-grained concurrency implemented with superscalar processors and SIMD capabilities.

Data parallelism

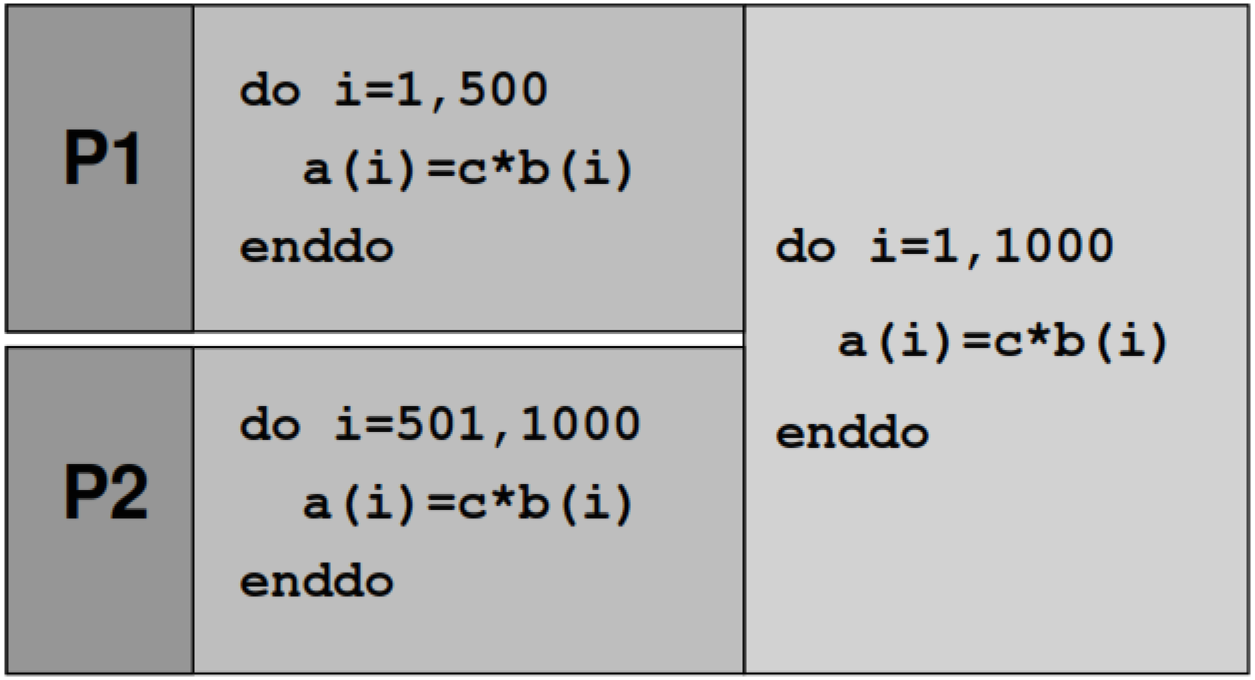
- Many problems in scientific computing involve processing of large quantities of data stored on a computer.
- If this manipulation can be performed in parallel i.e., by multiple processors working on different parts of the data, is called *data parallelism.*
- As a matter of fact, this is the dominant parallelization concept in scientific computing on MIMD-type computers.
- It also goes under the name of *SPMD (Single* Program Multiple Data), as usually the same code is executed on all processors, with independent instruction pointers.
- It is thus not to be confused with SIMD parallelism.

Example: **Medium-grained loop parallelism**

- Processing of array data by loops or loop nests is a central component in most scientific codes.
- A typical example are linear algebra operations on vectors or matrices.
- Often the computations performed on individual array elements are independent of each other and are hence typical candidates for parallel execution by several processors in shared memory.
- The distribution of work across processors is flexible and easily changeable down to the single data element

Example: **Medium-grained loop parallelism**

- The iterations of a loop are distributed to two processors P1 and P2 (in shared memory) for concurrent execution.



```
do i=1,500
   a(i)=c*b(i)
enddo
```

```
do i=501,1000
   a(i)=c*b(i)
enddo
```

```
do i=1,1000
   a(i)=c*b(i)
enddo
```

Example: Coarse-grained parallelism by domain decomposition

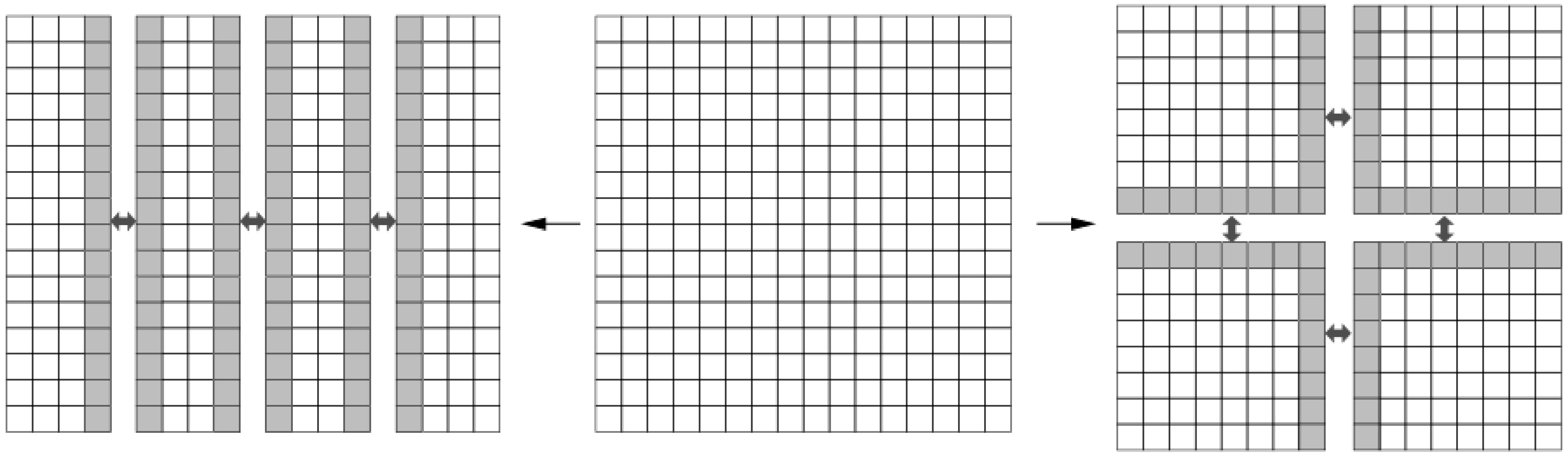
- Simulations of physical processes (like, e.g., fluid flow, mechanical stress, quantum fields) often work with a simplified picture of reality in which a *computational domain,*
- *E.g., some volume of a fluid, is represented as a grid that defines discrete* positions for the physical quantities under consideration
- The goal of the simulation is usually the computation of observables on this grid.
- A straightforward way to distribute the work involved across workers, i.e., processors, is to assign a part of the grid to each worker.
- This is called *domain decomposition.*

Coarse-grained parallelism by domain decomposition

- *load balancing:* The computational effort should be equal for all domains to prevent some workers from idling while others still update their own domains.
- After load imbalance has been eliminated one should care about reducing the communication overhead.
- The data volume to be communicated is proportional to the overall area of the domain cuts.
- Note that the calculation of communication overhead depends crucially on the *locality of data dependencies*
- *In the sense that communication cost grows linearly* with the distance that has to be bridged in order to calculate observables at a certain site of the grid.

- Domain decomposition has the attractive property that domain boundary area grows more slowly than volume if the problem size increases with *N constant.*
- One can sometimes alleviate communication bottlenecks just by choosing a larger problem size.

Domain decomposition



- Domain decomposition of a two-dimensional Jacobi solver, which requires next neighbor interactions.
- Cutting into stripes (left) is simple but incurs more communication than optimal decomposition (right).
- Shaded cells participate in network communication.

Functional Parallelism

- Sometimes the solution of a “big” numerical problem can be split into more or less disparate subtasks, which work together by data exchange and synchronization.
- In this case, the subtasks execute completely different code on different data items, which is why functional parallelism is also called *MPMD (Multiple Program Multiple* Data).
- This does not rule out, however, that each subtask could be executed in parallel by several processors in an SPMD fashion.

Pros and Cons of Functional Parallelism

- When different parts of the problem have different performance properties and hardware requirements, bottlenecks and load imbalance can easily arise.
- On the other hand, overlapping tasks that would otherwise be executed sequentially could accelerate execution considerably.

Master-worker scheme

- Reserving one compute element for administrative tasks while all others solve the actual problem is called the *master-worker scheme.*
- *The master distributes work* and collects results.

- A typical example is a parallel ray tracing program:
- A ray tracer computes a photorealistic image from a mathematical representation of a scene.
- For each pixel to be rendered, a “ray” is sent from the imaginary observer’s eye into the scene, hits surfaces, gets reflected, etc., picking up color components.
- If all compute elements have a copy of the scene, all pixels are independent and can be computed in parallel.
- Due to efficiency concerns, the picture is usually divided into “work packages” (rows or tiles).
- Whenever a worker has finished a package, it requests a new one from the master, who keeps lists of finished and yet to be completed tiles.
- In case of a distributed-memory system, the finished tile must also be communicated over the network.

Drawback of the master-worker scheme

- The potential communication and performance bottleneck that may appear with a single master when the number of workers is large.

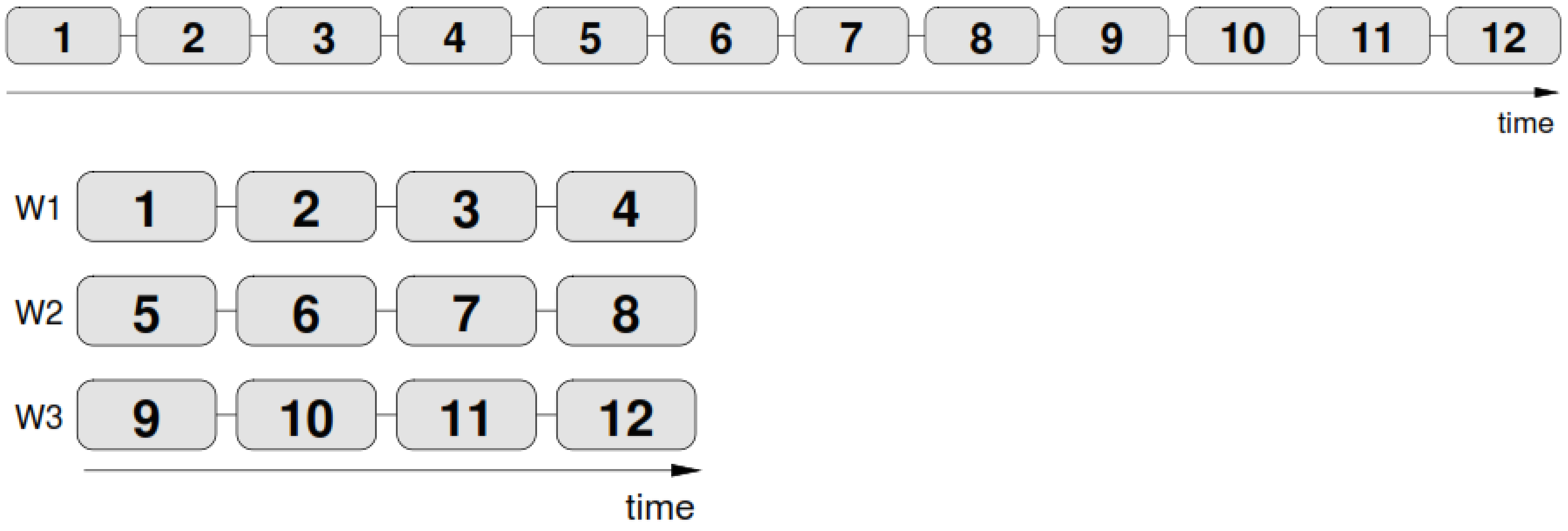
Functional decomposition

- Multiphysics simulations are prominent applications for parallelization by functional decomposition.
- For instance, the airflow around a racing car could be simulated using a parallel CFD (Computational Fluid Dynamics) code.
- On the other hand, a parallel finite element simulation could describe the reaction of the flexible structures of the car body to the flow, according to their geometry and material properties.
- Both codes have to be coupled using an appropriate communication layer.
- **Load balancing problem** because it is hard in practice to dynamically shift resources between the different functional domains.

Parallel scalability: Factors that limit parallel execution

- Finding parallelism is not only a common problem in computing but also in many other areas like manufacturing, traffic flow and even business processes.
- In a very simplistic view, all execution units (workers, assembly lines, waiting queues, CPUs,. . .) execute their assigned work in exactly the same amount of time.
- Under such conditions, using *N workers, a problem that takes a time T to be solved sequentially* will now ideally take only *T/N.*
- *We call this a speedup of N.*

Speedup



- Parallelizing a sequence of tasks using three workers (W1. . . W3) with perfect speedup

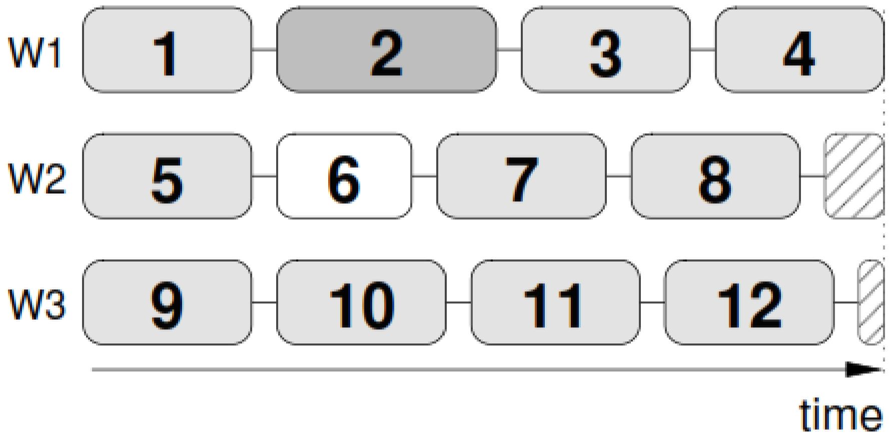
Factors that limit parallel execution

- Not all workers might execute their tasks in the same amount of time because the problem was not (or could not) be partitioned into pieces with equal complexity.
- Hence, there are times when all but a few have nothing to do but wait for the latecomers to arrive.
- This *load imbalance hampers performance because* some resources are underutilized.
- Moreover there might be shared resources like, e.g., tools that only exist once but are needed by all workers.
- This will effectively *serialize part of the concurrent execution.*
- *And finally, the parallel* workflow may require some communication between workers, adding overhead that would not be present in the serial case.
- All these effects can impose limits on speedup.

Use of Scalability Metric

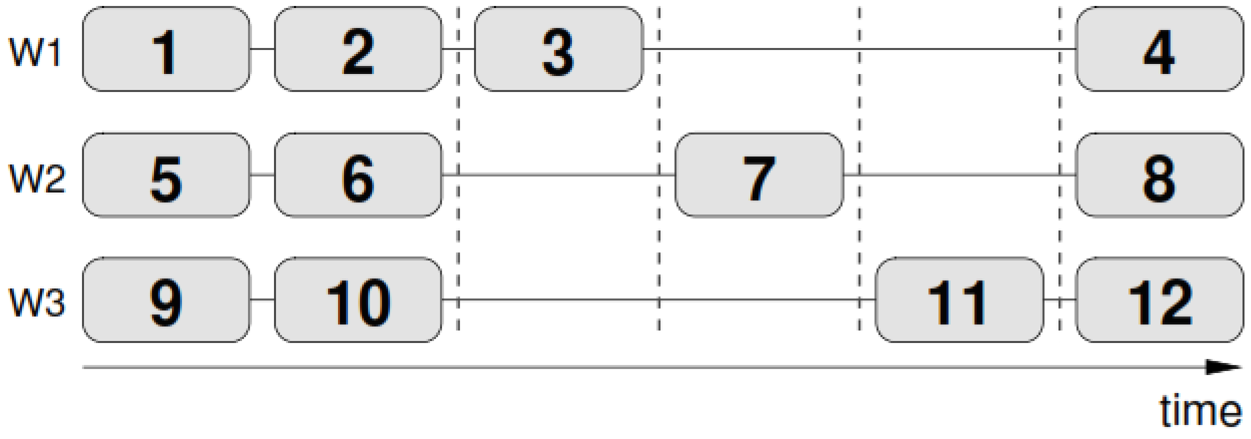
- How well a task can be parallelized is usually quantified by some *scalability metric.*
- Using such metrics, one can answer questions like:
- How much faster can a given problem be solved with *N workers instead of* one?
- How much more *work can be done with N workers instead of one?*
- What impact do the communication requirements of the parallel application have on performance and scalability?
- What fraction of the resources is actually used productively for solving the problem?

Load Imbalance.



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

Effectively *serialize part of the concurrent execution*



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

Scalability metrics

- The overall problem size ("amount of work") shall be $s + p = 1$,
- *Where s is the serial* (nonparallelizable) part and *p is the perfectly parallelizable fraction.*
- *There can be* many reasons for a nonvanishing serial part:
- *Algorithmic limitations.*
- *Bottlenecks*
- *Startup overhead*
- *Communication*

Algorithmic limitations

- Operations that cannot be done in parallel
- Example:
- Mutual dependencies,
- Can only be performed one after another, or even in a certain order.

Bottlenecks

- *Shared resources are common in computer systems:*
- *Execution* units in the core, shared paths to memory in multicore chips, I/O devices.
- Access to a shared resource *serializes execution.*
- *Even if the algorithm itself could* be performed completely in parallel, concurrency may be limited by bottlenecks.

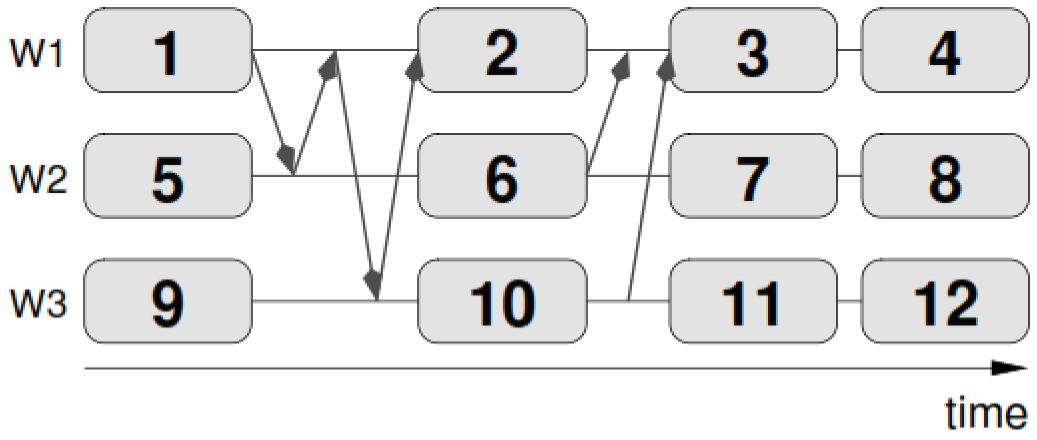
Startup overhead

- *Starting a parallel program, regardless of the technical details,* takes time.
- Of course, system designs try to minimize startup time, especially in massively parallel systems, but there is always a nonvanishing serial part.
- If a parallel application’s overall runtime is too short, startup will have a strong impact.

Communication

- *Fully concurrent communication between different parts of* a parallel system cannot be taken for granted.
- If solving a problem in parallel requires communication, some serialization is usually unavoidable.

Communication



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

Strong Scaling

- First we assume a fixed problem, which is to be solved by *N workers.*
- *We normalize* the single-worker (serial) runtime
- $T_f^s = s + p$
- Solving the same problem on *N workers will require a runtime of*
- $T_f^p = s + (P/N)$
- This is called *strong scaling because the 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

- If time to solution is not the primary objective because larger problem sizes (for which available memory is the limiting factor) are of interest
- It is appropriate to scale the problem size with some power of *N so that the total amount of work is* $s + pN^{\alpha}$
- Where α is a positive but otherwise free parameter.
- Here we use the implicit assumption that the serial fraction s *is a constant.*
- *We define the serial runtime for the* scaled (variably-sized) problem as
- $T_v^s = s + pN^{\alpha}$
- Consequently, the parallel runtime is
- $T_v^p = s + pN^{\alpha-1}$

Parallel Efficiency

- Parallel efficiency is then defined as

$$\varepsilon = \frac{\text{performance on } N \text{ CPUs}}{N \times \text{ performance on one CPU}} = \frac{\text{speedup}}{N}.$$

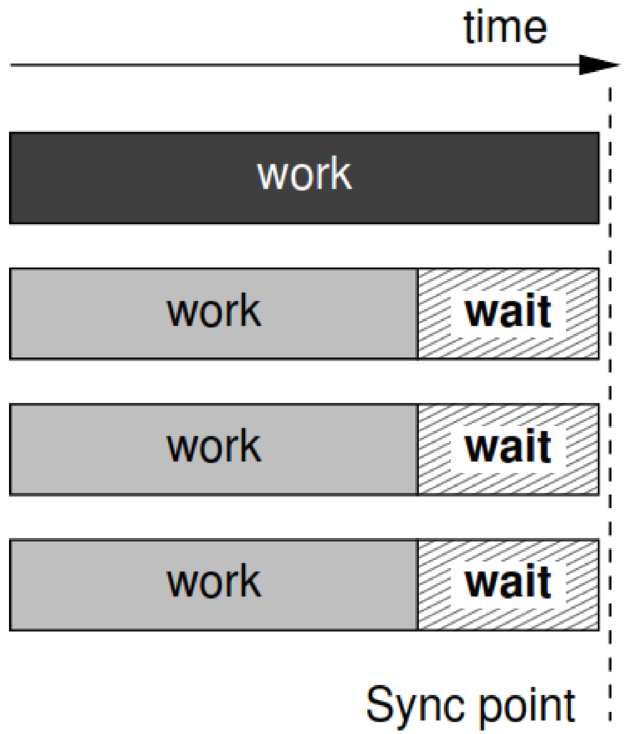
Load imbalance

- Inexperienced HPC users usually try to find the reasons for bad scalability of their parallel programs in the hardware details of the platform used and the specific drawbacks of the chosen parallelization method:
- Communication overhead, synchronization loss, false sharing, NUMA locality, bandwidth bottlenecks, etc.
- While all these are possible reasons for bad scalability (and are covered in due detail elsewhere in this book), load imbalance is often overlooked.
- Load imbalance occurs when synchronization points are reached by some workers earlier than by others, leading to at least one worker idling while others still do useful work.
- As a consequence, resources are underutilized.

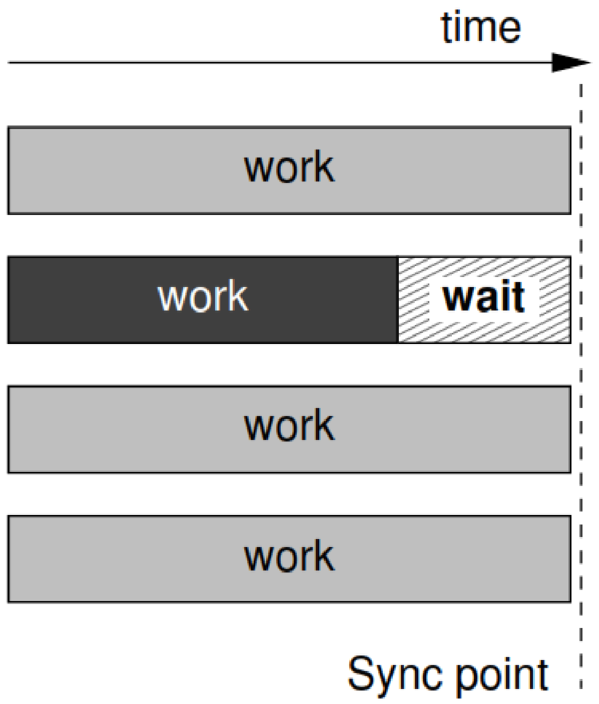
Load imbalance

- The consequences of load imbalance are hard to characterize in a simple model without further assumptions about the work distribution.
- Also, the actual impact on performance is not easily judged:
- Having a few workers that take longer to reach the synchronization point ("laggers") leaves the rest,
- i.e., the majority of workers, idling for some time, incurring significant loss.
- On the other hand, a few "speeders," i.e., workers that finish their tasks early, may be harmless because the accumulated waiting time is negligible.

Load imbalance



- Load imbalance with few (one in this case) "laggers": A lot of resources are underutilized (hatched areas).



- Load imbalance with few (one in this case) "speeders": Underutilization may be acceptable.

Reference

- **Georg Hager and Gerhard Wellein**, Introduction to High Performance Computing for Scientists and Engineers, CRC Press, 2011.