Parallel programming model

We now have a series of von Neumann type machines that can communicate among each other.

There are variants:

- Not universal communication paths
- Shared memory area (communication might go through memory).
- Simultaneous operation on different data (vector operations).
- Independent operation (even on heterogeneous architectures)

Conceptual model

Task/Channel

We have a set of tasks (or processes) that have to be executed.

Tasks can communicate through channels between tasks.

Message Passing

Interprocess communication is done by sending *messages* from one process to another. Sent messages will be in a queue that can be read step-by-step.

The underlying implementation has to ensure tings such as preservation of message order and write/read synchronization.

Data Parallelism

It is often convenient to have different processes perform the same tasks on different parts of the data. Parallelization might be done (semi)automatically by a compiler.

Master/Slave Paradigm It can be convenient to have one process designated as a “master” process that distributes tasks to “slave” processes and collects their results. the task of this master process is often mainly organizational (and it might run in parallel with a slave process on one processor.

This paradigm is in particular useful if the communication infrastructure is rather low-key.

Desiderata for a parallel program

Concurrency It should be possibel to perform (many) operations in parallel.

Scalability The parallelization should not depend on a (fixed) number of processors given. Availability of more processors should improve the speed (at best linearly).

Data Locality It should not be necessary to transfer large amounts of data (beyond an initial initialization).

General Design Desiderata Modularily, Maintainability, etc.

Methods for designing parallel programs

Partitioning Split the initial task into smaller (independent) units that can be performed in parallel.

Such a decomposition might be based on a decomposition of the data (say: parts of a matrix) or on functionality (temperature/pressure forecast).

The 'divide-and-conquer' design paradigm naturally exposes such splits.

Partitioning checklist

- Is the partioning at leasta magnitude finer than the number of processors allocated (otherwise one can end up with much idling)?
- Does the algorithm avoid redundant computations/storage (otherwise scalability will suffer – Amdahl's law)
- Are tasks of comparable size? (Otherwise its hard to keep processors occupied).
- Does the number of tasks scale with the problem size?
- Investigate both domain and functional decomposition.

Communication

Specify what information has to be exchanged between processors and in what kind of messages this information should be sent.

- Synchronous vs. Asynchronous communication.
- Is communication cost equally distributed?
- Do tasks communicate only with a small number of neighbors?
- Can communications proceed simultaneously?

  (Example: Results of n calculations sent to a master process.)

Architectural adaption

We now translate the algorithm to the architecture that is to be used.

This might imply small changes in the communication structure or protocol.

Depending on the number of processors available and communication cost it can be useful to increase granularity and toi agglomerate tasks.

If communication is expensive in comparison to runtime, also (re)computation of results (or parallel building of large data structures) can be useful.

Load-balancing

Finally we have to decide (unless the operating system provides automatic tools) which tasks are to be executed on which processor.

If there are some tasks with high intermediate communication, they should run on the same or “close" processors.

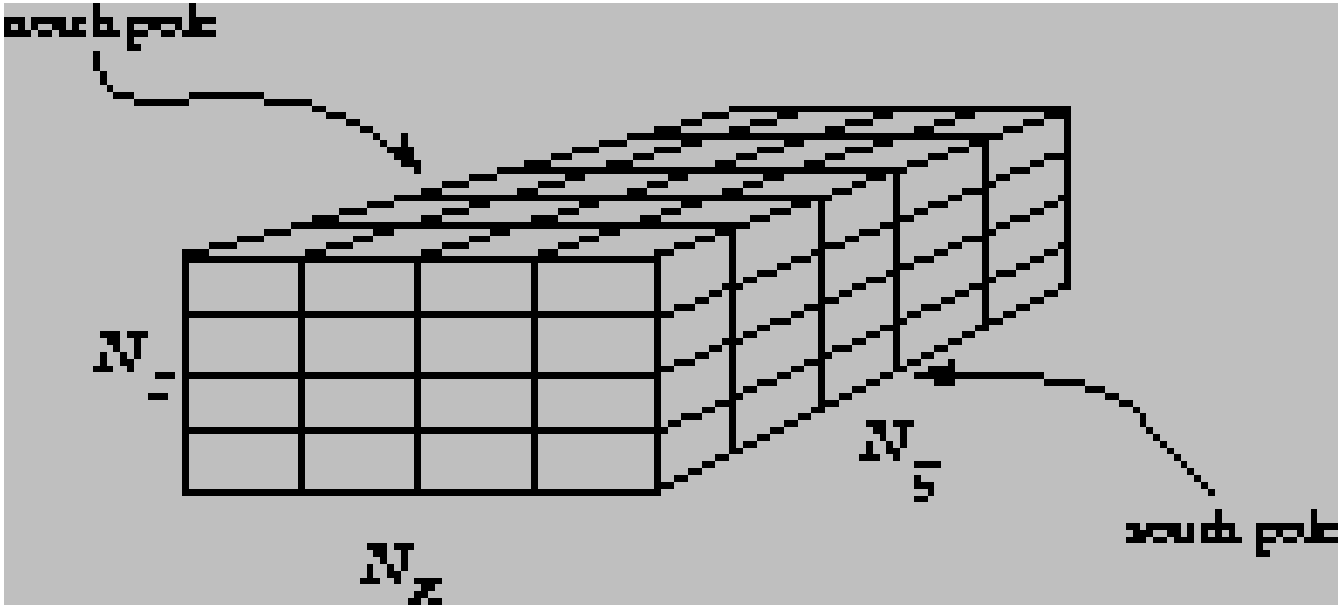
We also have to make sure that we do not keep processors idling.

If there is a “master" process, it has to be made sure this process does not become a bottle-neck.

Case Study: Atmosphere Model

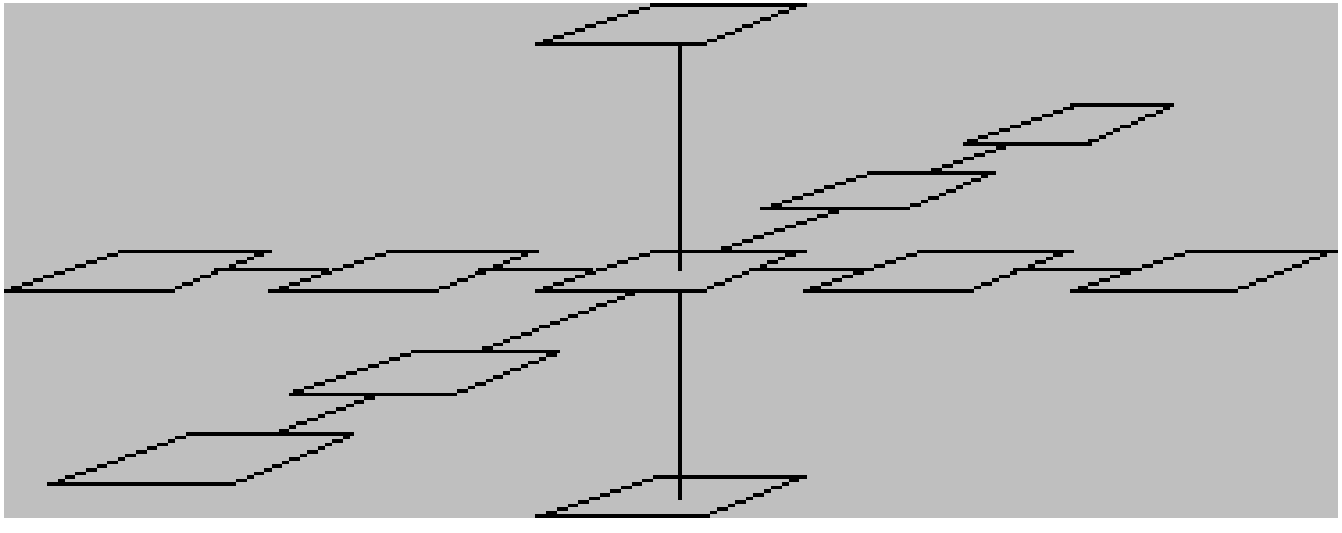
Suppose we want to model the behaviour of the atmosphere (such as temperature, pressure, air flow, humidity, ...), based on basic physical principles (such as the heat equation, mass tranfer etc.)

We approximate the (continuous) space by a finite set of points (regularly spaced though in some cases one would space denser in critical areas):



Source: I.Foster, Designing and building parallel programs

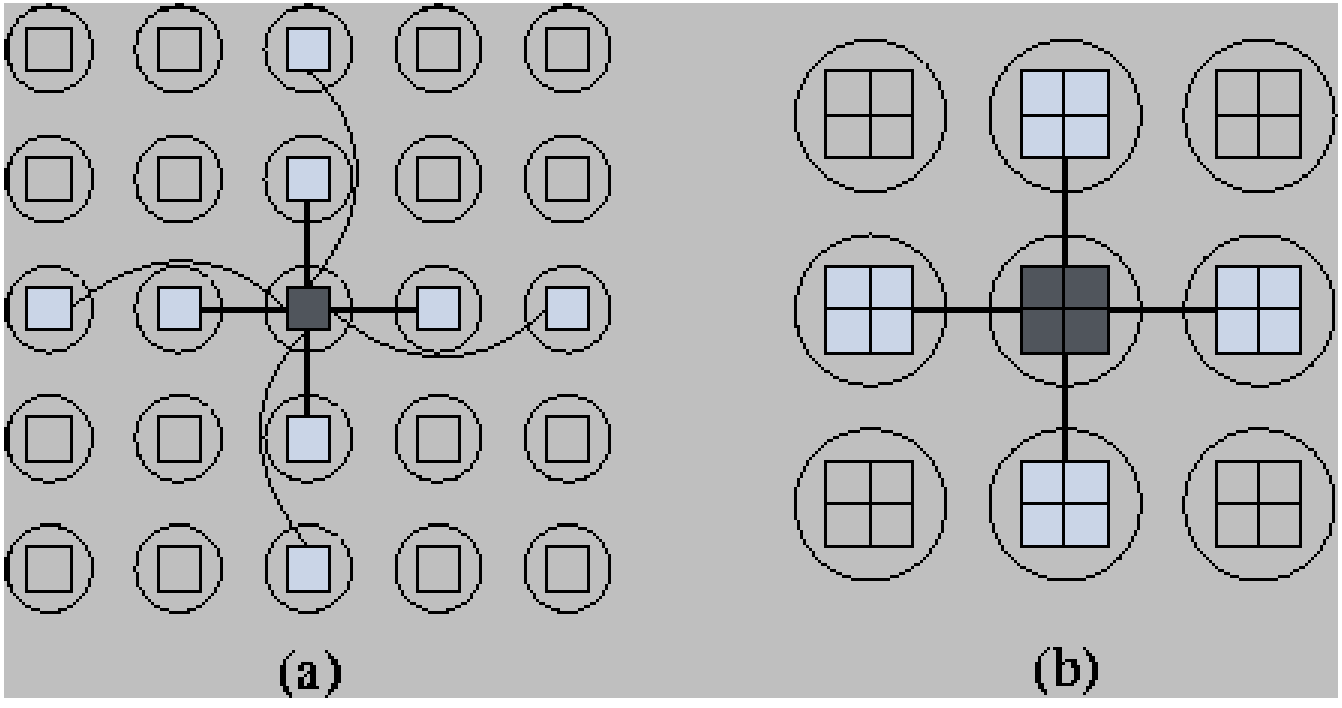
Starting with an initial state, the model calculates new values (after a given time interval Δt) at each grid point, based on the values of "neighboring" points, neighborhood being defined by a "stencil" pattern:



Source: I.Foster, Designing and building parallel programs

Partition The most extreme partitioning possible is to consider one task for each grid point.

Communication Each task has to communicate with the tasks defined by the neighborhood stencil. In one "layer" this gives 8 neighbors to communicate with.



Source: I.Foster, Designing and building parallel programs

Agglomeration If we agglomerate 2×2 points in one unit, communication reduces to 4 neighbors.

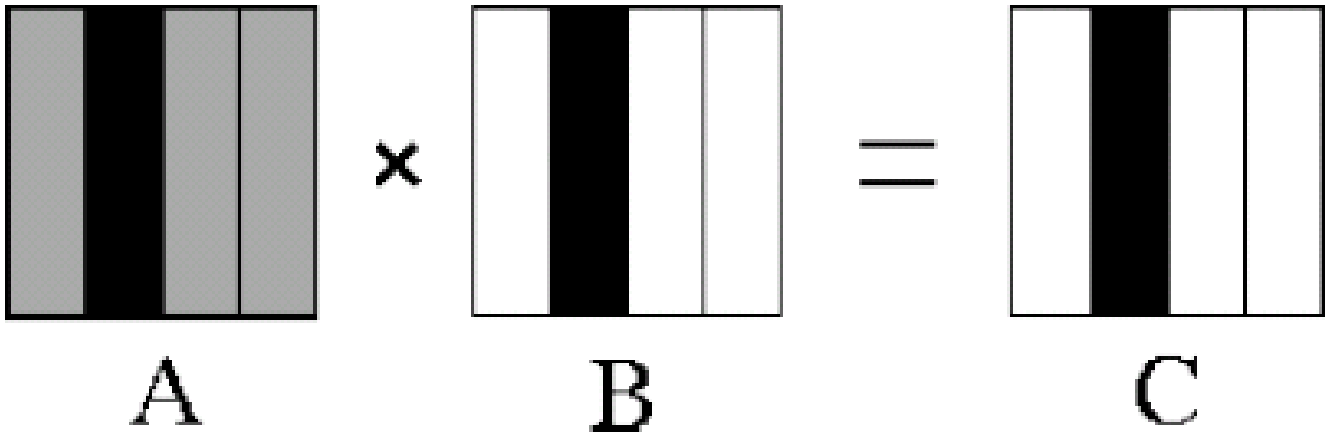
In some cases there might be more vertical interaction (say weight/pressure) which would indicate that it can be also useful to agglomerate “columns" vertically.

Mapping Since loads for every point are equal, we can simply distribute the same number of (neighboring) tasks on each processor.

Case Study: Matrix Multiplication

We now consider the problem of multiplying (dense) $n \times n$ matrices $A = (a_{i,j})$ and $B = (b_{i,j})$ to a product $C = (c_{i,j})$ where $c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$. Suppose we have p tasks

Consider possible ways to distribute the contents of the matrices A,B, C: We first consider a decomposition into columns:



Source: I.Foster, Designing and building parallel algorithms

In this situation, each task needs to know the whole of matrix A to compute the corresponding entries of C. From each of the remaining $p-1$ tasks we need to obtain n^2/p matrix entries.

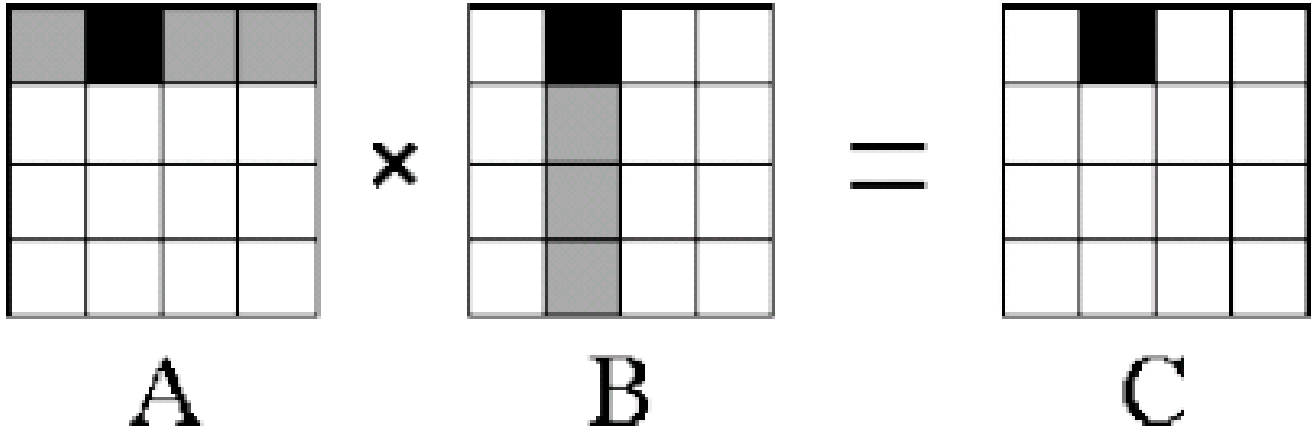
If we denote the time required to initiate interprocess communication by s and the actual communication time by t, we get a total communication cost of:

$$(p-1) * (s + t\frac{n^2}{p}) \approx sp + tn^2$$

Each task performs $\mathcal{O}(n^3/p)$ operations. Thus if $n \approx p$, the communication cost is similar to the cost of arithmetic.

The algorithm would be efficient only if $n \gg p$ or the arithmetic cost is substantial higher than the communication cost.

Now consider a decomposition by rows and columns (similar to blocked multiplication):



Source: I.Foster, Designing and building parallel algorithms

Here the task to calculate an entry of C needs access to a row of A and a column of B.

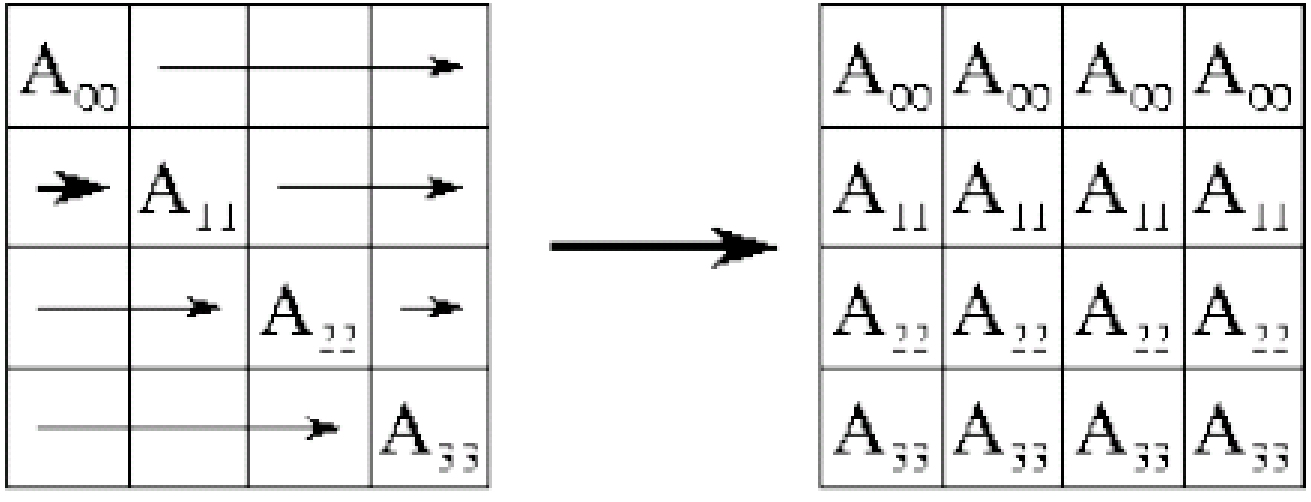
We thus have to get $2(\frac{n}{\sqrt{p}} - 1)$ blocks of size $\frac{n^2}{p}$ each, for a total communication cost of $\mathcal{O}\left(\frac{n^2}{p}\right)$, which is notably less (in particular considering that p is larger).

We now need to design a communication strategy:

Consider for example $C_{1,2} = A_{1,0}B_{0,2} + A_{1,1}B_{1,2} + A_{1,2}B_{2,2} + A_{1,3}B_{3,2}$. We will calculate the summands one by one in each task and eventually add them up to obtain the result.

Also note that *every* $C_{1,x}$ will contain a summand that is a product of $A_{1,1}$ with $B_{1,x}$.

We thus first transmit each A-block to the whole row:



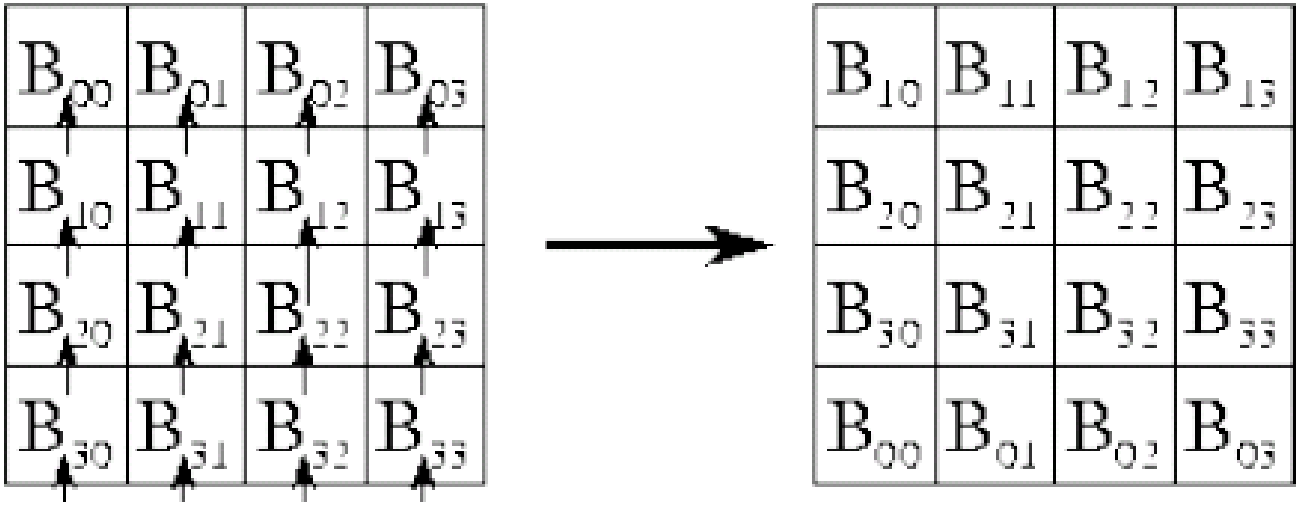
Source: I.Foster, Designing and building parallel algorithms

Next we perform a calculation in each block:

$$\begin{array}{cccc} A_{0,0} \cdot B_{0,0} & A_{0,0} \cdot B_{0,1} & A_{0,0} \cdot B_{0,2} & A_{0,0} \cdot B_{0,3} \\ A_{1,1} \cdot B_{0,0} & A_{1,1} \cdot B_{0,1} & A_{1,1} \cdot B_{0,2} & A_{1,1} \cdot B_{0,3} \\ A_{2,2} \cdot B_{2,2} & A_{2,2} \cdot B_{0,1} & A_{2,2} \cdot B_{0,2} & A_{0,0} \cdot B_{0,3} \\ A_{3,3} \cdot B_{3,3} & A_{3,3} \cdot B_{0,1} & A_{3,3} \cdot B_{0,2} & A_{3,3} \cdot B_{0,3} \end{array}$$

and store the results.

Next, consider how $A_{1,2}$ has to be multiplied: We have to multiply it with $B_{2,x}$. To keep the “row distribution" structure we thus transfer the B-blocks one step up in their column:



Source: I.Foster, Designing and building parallel algorithms

Then we distribute $A_{0,1}$, $A_{1,2}$, &c.to the whole row and again perform all multiplications.

The process is iterated through the number of blocks.

In this scheme communication has to take place only with the next neighbor in a (cyclic) grid-like setup.