Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

Parallel Programming with Message Passing

Summer Term 2020 | 20.05.2020

Barry Linnert

Freie Universität Berlin

Objectives of Today's Lecture

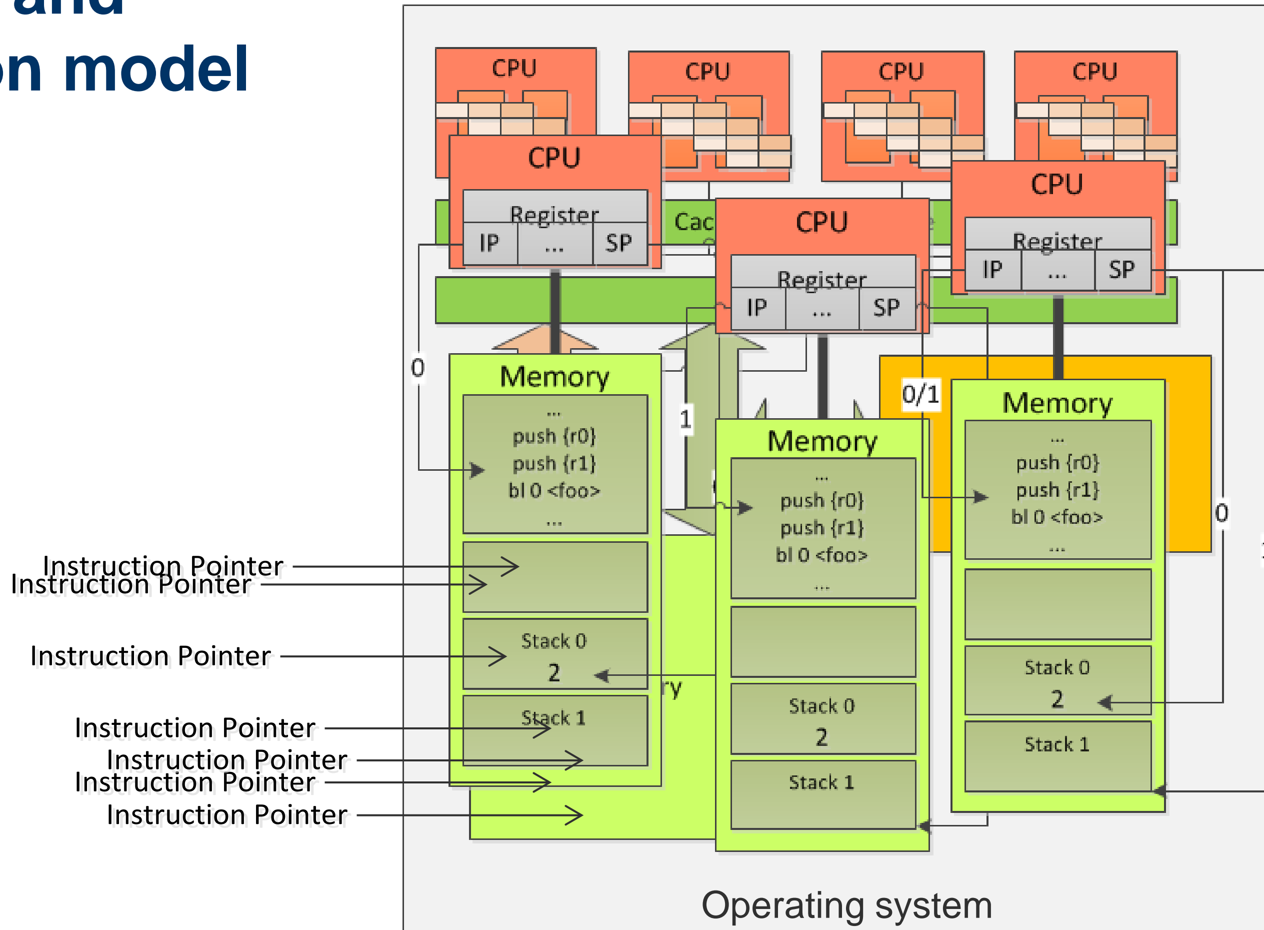
- Parallel programming with message passing
- Foster's Design Methodology

Freie Universität Berlin

Concepts of Non-sequential and Distributed Programming

PARALLEL PROGRAMMING WITH MESSAGE PASSING

Machine and Execution model



Programming with Shared Memory

- Using the execution and machine model we can identify different levels of parallel processing:
  - Processor level
    - Pipelining
    - multiple processing units
  - Compiler level
    - Out-of-Order-Execution
    - Parallel loops
  - Thread level
    - OpenMP
      - Loop and functional parallelization
    - Pthreads
      - Functional parallelization

Implicit parallel processing (Processor level, Compiler level)

Explicit parallel processing (Thread level)

Implicit Parallel Processing (with Shared Memory)

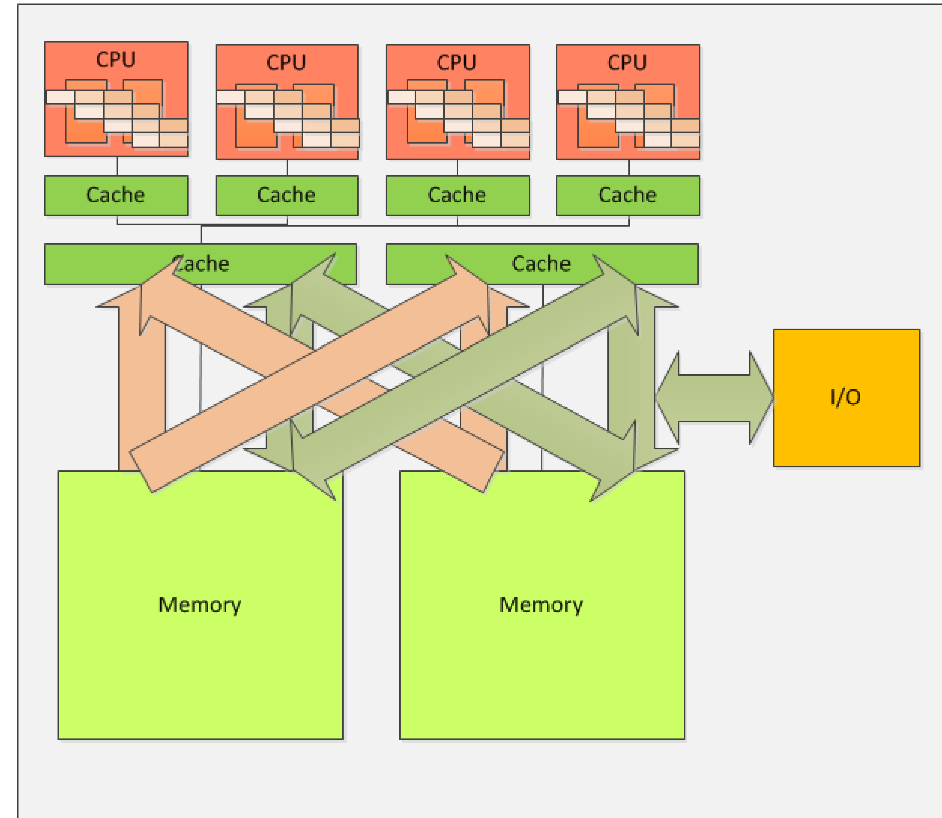
- The goal of implicit parallel processing is to maximize the usage of the existing computing units.
- The parallel execution of (parts) of the program is transparent to the program and therefore to the programmer and user.
- The unit responsible for the parallelization (compiler, processor) detects data independent parts of the program.
- Usually the parts are with respect to the whole program fine granular, such as blocks of few operations or even on level of micro-instruction.
- As the thread or process is executed without constrains the parallel executions works within the same address space. Thus, implicit parallel processing is performed on systems with shared memory.

Explicit Parallel Processing with Shared Memory

- The goal of explicit parallel processing is to reduce the response time (execution and waiting time) of a specific program by using as much resources as possible.
- The parts of the program to be processed in parallel are to be identified by the programmer and has to encapsulate to be recognized by the execution system as unit to be executed separately.
- The encapsulation of the units to be processed in parallel usually have an expression at the level of the programming language.
- As it is easier (with respect to other approaches) to exchange intermediate results using shared data the corresponding variables are stored in shared memory address space.
- The identification and protection of critical sections has to be performed by the programmer her-/himself.

Extension of the Machine and Execution Model

- With explicit parallel processing the amount of parallelization – amount of computing units used by the program – can be controlled more effective.
- Larger systems with more CPUs may be used by these programs.
- If the architectures provide more than one unit of main memory and the access (times) to the address depends on the CPU performing the access this architecture is called NUMA – non-uniform memory access.



PRAM Model

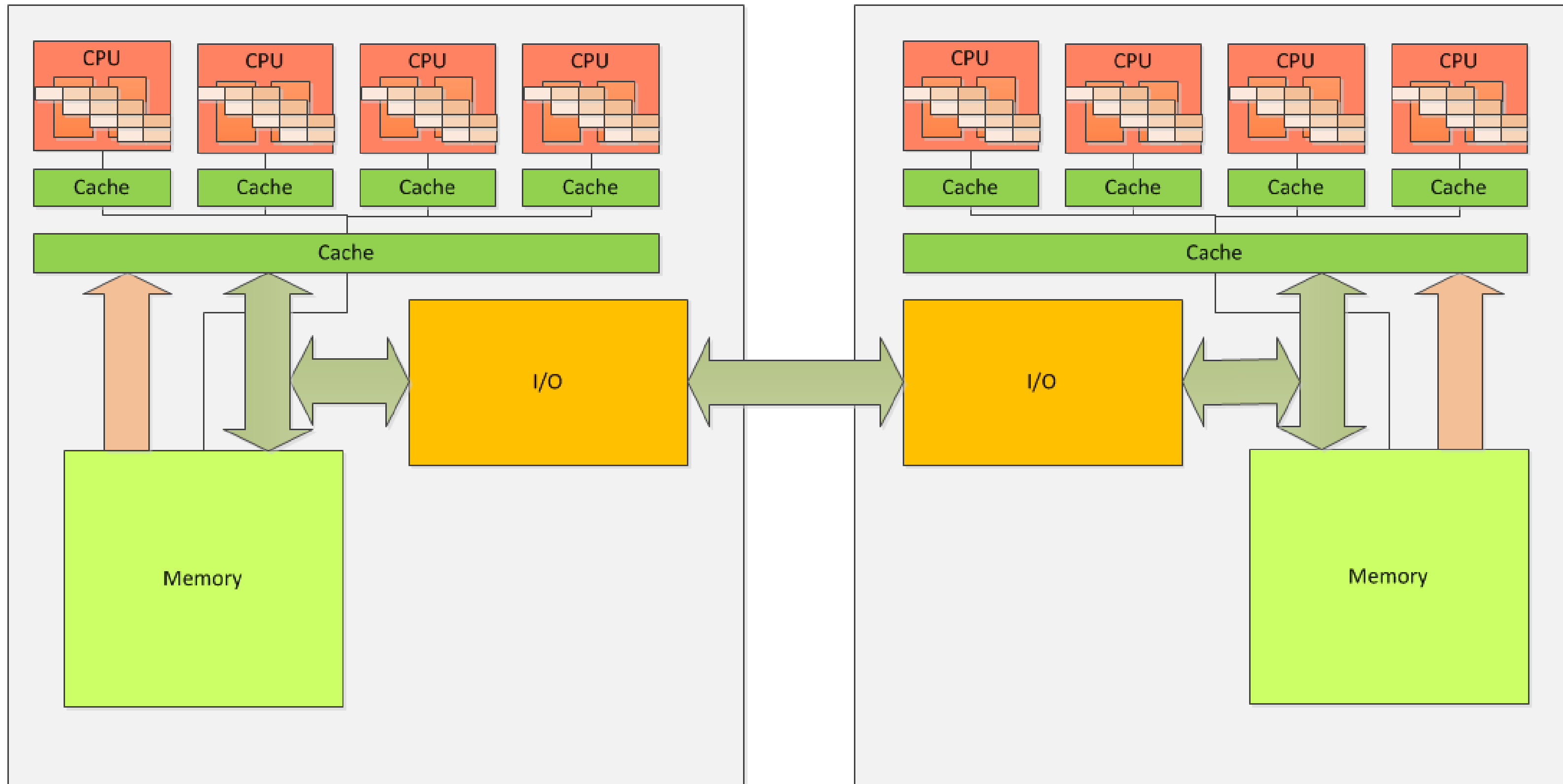
- The PRAM Model is a programming model for programs using shared memory based on UMA or NUMA architectures.
- PRAM stands for Parallel Random Access Machine.
- The PRAM model is an extension to the RAM model, which is known as the von Neumann architecture.
- The model consists of a set of independent random access machines each operate on a single processor and coordinate themselves via shared memory.

- It is also used to evaluate the performance of a parallel program.
- To evaluate the performance of the program the performance values for the computing units (CPUs) and the overhead for memory access is incorporated.

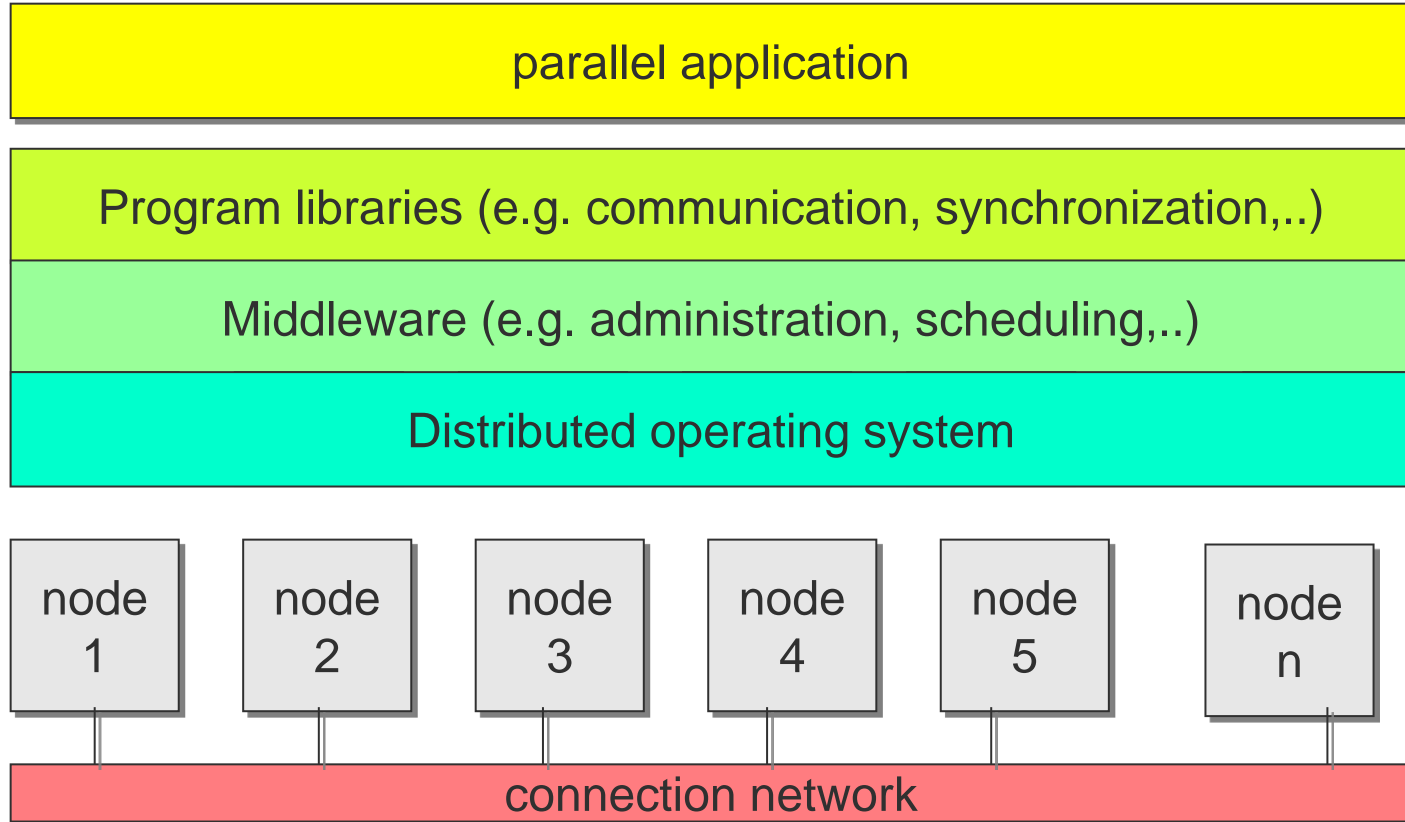
Parallel Computing

- If the programs are complex and the amount of resources needed to obtain reasonable response times more than one machine may have to work for the program.
- Systems coupled to provide resources for the execution of a parallel program are called parallel computers.
- The single machines forming the parallel computer are called nodes (of the parallel computer). Usually the nodes are connected by network interfaces and links.
- Thus, the ability to access addresses in main memory depends on the node the thread is running on. Usually, there is no comprehensive memory access implemented.
- Data needed to be exchanged between threads running on different nodes has to be transmitted explicitly – using message passing.

Architecture of a Parallel Computer



Examples


parallel application
Program libraries (e.g. communication, synchronization,..)
Middleware (e.g. administration, scheduling,..)
Distributed operating system
node 1
node 2
node 3
node 4
node 5
node n
connection network

Programming with Shared Memory vs. Message Passing

- The division of the program or problem space (data to be processed) is to be modeled in order to build a parallel program.
- The threads working the parallel program have to be synchronized when accessing or changing shared data.
- The performance of the program depends primarily on the extent of the parallel areas of the program.

Programming with shared memory:

- Exchange of date is done implicitly by using shared variables.
- Access to shared variables is usually as fast as to local/private variables.

Programming with message passing:

- Exchange of data takes place explicitly by sending and receiving the data (variable contents).
- Access to shared data requires explicit and slow message passing.

Example: Matrix Multiplication

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + A_{13} \cdot B_{31} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} + A_{13} \cdot B_{32} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + A_{23} \cdot B_{31} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} + A_{23} \cdot B_{32} \end{pmatrix}$$

Example: Raytracing

Rendering (interior architecture)

Rendering (Movie: Titanic)

Photo realistic presentation using sophisticated illumination modules

Foster's Design Methodology

- Ian Foster
  - He is director at the Argonne National Laboratory and professor in the department of computer science at the University of Chicago.
  - He is – together with Carl Kesselman and Steve Tuecke – inventor of the term Grid Computing to connect different supercomputers to build a transparent resource for high performance computing (HPC) applications.

datasysys.cs.iit.edu

- Foster described an approach to model and design parallel applications to be run on supercomputers. It’s called Foster's Design Methodology.

Task/Channel (Programming) Model

- The Task/Channel (Programming) Model serves as foundation for Forster‘s design methodology.
- A Task of the task/channel model is a part of the application with its own address space (process).
- Tasks can exchange data via messages using channels.
- A channel is a message queue connecting two specific tasks.
- If a task wants to receive a message, the task waits until the message is received (is blocked).
- Messages are sent immediately by the sender. The sender does not wait until the message is received.
- Thus, the task/channel model implements synchronous receive and asynchronous sending.

Freie Universität Berlin

Foster's Design Methodology

- The design methodology by Foster is a four-step process consisting of:
    1. Partitioning
    2. Communication
    3. Agglomeration
    4. Mapping (Assignment)

Partitioning

- The first step deals with the division of the problem space.
- The goal is to identify as many independent pieces of the program or data as possible.
- Thus, we can use domain and data decomposition or functional decomposition.
- Domain and data decomposition is up to identify data that can be processed independently.
- Functional decomposition focuses on identification of computational functions that can be processed independently.
- Partitioning is usually performed in a recursive top-down process.
- Often a complete decomposition is not possible, so there has to be made a trade-off between the amount of tasks to be created and the communication cost.

Partitioning II

- The best designs satisfy the following attributes:
  - There are many more independent tasks identified than there are processors in the system.
  - The number of redundant calculations and redundant data structures (variables) has been minimized.
  - The tasks (primitive tasks) should have approximately the same size or processing effort.
  - The number of tasks should grow at the same rate as the problem size.

Communication

- After splitting the problem space into pieces the relations between the tasks have to be considered.
- Thus, the data exchange between the tasks is modeled using the channels and the appropriate properties of the channels.
- The message passing to transmit data from one specific task to another specific task is called local communication.
- The sending of messages from one task to all of the other tasks or of a significant amount of tasks is called global communication.
- The total amount of local and global communication (for a task) determines the communication overhead (of the specific task).

Communication II

- The requirements for this step are:
  - The communication efforts of the respective tasks should be (almost) equally distributed.
  - Each task communicates only with a small number of other tasks (neighbours).
  - Tasks can perform their communication concurrently. This means, different tasks communicate independently of each other.
  - Tasks can perform their computations concurrently. Different tasks perform their calculations independently of each other.
    - There are no permanent data-level dependencies between tasks.

Agglomeration

- As the amount of tasks exceeds the number of processors tasks have be combined or grouped together to be assigned to a specific processor.
- The amount of combined tasks should correspond to the number of processors available.
- The combination of tasks should reduce the amount of communication between tasks as the tasks to be combined uses shared memory to provide shared data.
- Thus, the agglomeration should increase locality of data usage.
- With combination of the tasks the effort of task creation (process creation) is reduced, too.

Agglomeration II

- The requirements for the step of agglomeration are:
  - The combination of the tasks has led to a higher locality of the parallel program.
  - There is less communication effort across all tasks with respect to the communication before the agglomeration.
  - If there is additional work to do because a task was combined with two different sets of tasks, the replicated computation should need less time than the communication it replaces.
  - The amount of data combined into common variables is small enough that the program is able to scale as the problem space increases.
  - The combined tasks are (still) similar in terms of calculation and communication effort.
  - The number of tasks should (still) grow at the same rate as the problem size.
  - If changes to the parallel program has to be made the cost for this programming has to be reasonable.

Mapping

- The combined tasks has to be assigned to specific processors of the parallel computer.
- This is called mapping.
- The mapping should lead to a maximum of utilization of the single processor.
- If there are (still) more tasks available the mapping will lead to an assignment with more than on tasks to be mapped to a specific processor.
- The mapping of more than one task to a processor should lead to a significant increase in an unbalance of computational load.

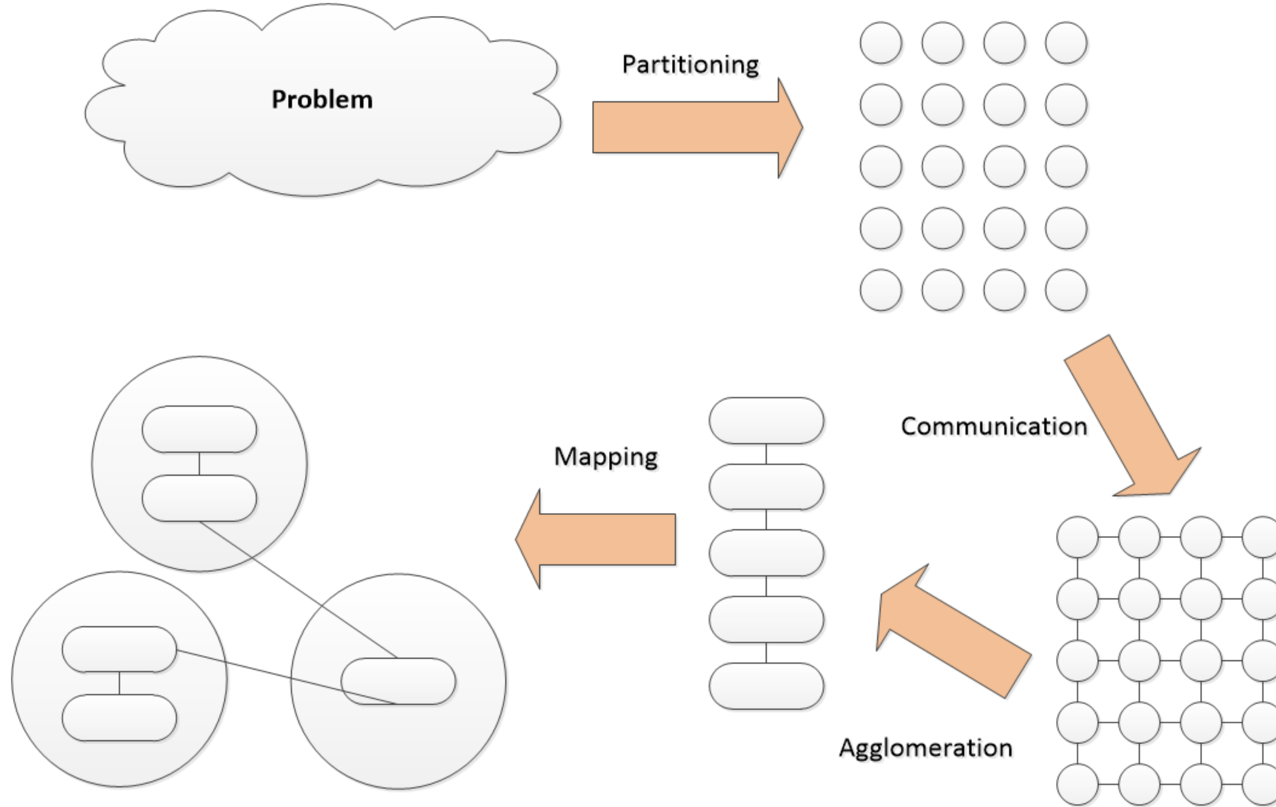
Freie Universität Berlin

Mapping II

- The requirements for the last step of the design methodology are :
  - The machine and execution model has to be considered sufficiently.
  - Thus, the amount of processors available for the parallel program is used.
  - The available allocation strategies – static or dynamic – have to be considered and evaluated.
  - If a dynamic allocation strategy has been chosen, the manager – within the program or of the management systems – should not become the bottleneck for performance.
  - If a static assignment has been chosen, about 10 tasks should be mapped to a processor.

Freie Universität Berlin

Foster's Design Methodology



Consideration of Time Sequences

- Foster's approach considers time sequences or dependencies only as a summary of communication efforts.
- Often intermediate results form the basis for the next processing steps.
- The exchange of intermediate results thus has a synchronization aspect in addition to the aspect of providing shared data.

- Modelling the time sequence (program runtime behavior) during programming can be a great challenge:
  - The processing of the tasks of the program also dependent on the input data.
  - The influence of the time sequences to the overall program runtime behavior depends on all (previous) steps of execution.

BSP Model

- To model the program runtime behavior at time of programming the BSP model was introduced.
- BSP stand for bulk synchronous parallel.
- It is a well used programming model for parallel applications and can be used to evaluate the performance of the parallel program as well.
- The computational work is split into super steps.
- A super step consists of calculation in parallel for all of the participating processes and the interchange of the intermediate results between all of these processes.
- For the synchronization of calculation and of the transmission a barrier synchronization is performed. This barrier ends the super step and afterwards the next super step starts.
- The synchronization ensures that every process always works on valid data.

BSP Model

- To model the performance of the parallel program the sequence of super steps and the execution time for the super steps is used.
- The communication costs are modeled via broadcast communication and barrier synchronization.

- The performance of the parallel program depends heavily on the balanced distribution of computational work between all of the processes.
- The synchronization part (with broadcast communication) may lead to overload situations at the network level temporary.

concepts of non-sequential and distributed programming

NEXT EVENT

Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

MPI

APL IV: Concepts of Non-sequential and Distributed Programming (Summer Term 2020)