1. **Implementation: Target Architectures**

- different target architectures for numerical simulations:
  - monoprocessors
  - supercomputers

- modern microprocessors:
  - obvious trends:
    * increasing clock rates (> 2GHz almost standard)
    * more MIPS, more FLOPS
    * very-, ultra-, and ???-large scale integration; hence, more transistors and more functionality on the chip
    * longer words: 64 Bit architectures are standard (workstations) or coming (PCs)

  - important features:
    * RISC (Reduced Instruction Set Computer) technology
    * well-developed pipelining
    * superscalar processor organization
    * caching and multi-level memory hierarchy
    * VLIW, Multi Thread Architecture, On-chip multiprocessors, ...
2. **RISC Technology**

- counter-trend to CISC: more and more complex instructions entailing **microprogramming**
- now instead:
  - relatively small number of instructions (tens)
  - simple machine instructions, fixed format, few address modes
  - *load-and-store* principle: only explicit LOAD/WRITE instructions have memory access
  - no more need for microprogramming
3. Pipelining

• decompose instructions into simple steps involving different parts of the CPU:
  – load,
  – decode,
  – reserve registers,
  – execute,
  – write results

• further improvement: reorder steps of an instruction (LOAD as early as possible, WRITE as late as possible: avoids risk of idle waiting time)

• best case: identical instructions to be pipelined/overlapped, as in \textit{vector processors}

• pipelining needs different functional units in the CPU that can deal with the different steps in parallel; therefore:
4. Superscalar Processors

• several parts of the CPU are available in more than 1 copy
• example: MIPS R10000 has 5 execution pipelines
  – one for FP-multiplication, one for FP-addition
  – two integer ALU (arithmetic-logical units)
  – one address pipeline
5. Cache Memory

- CPU performance increased faster than memory access speed
- thus: reduce memory access time / latency
- cache memory: small and fast on-chip memory, keeps part of the main memory
- optimum: needed data is always available in cache memory
- look for strategies to ensure hit-probability $p$ close to 1:
  - choice of section: what to be kept in cache?
  - ensure locality of data (instructions in cache need data in cache)
  - strategies for fetching, replacement, and updating
  - association: how to check whether data are available in cache?
  - consistency: no different versions in cache and main memory
6. Memory Hierarchy

- today: several cache levels → memory hierarchy:
  - register,
  - (level-1/2/3) cache,
  - main memory,
  - hard disk,
  - remote memory

the faster, the smaller

- notion of the target computer’s memory hierarchy is important for numerical algorithms’ efficiency:
  - example: matrix-vector product $Ax$ with $A$ too large for cache
  - standard algorithm:
    - outer loop over rows of $A$,
    - inner loop for scalar product of one row of $A$ with $x$
  - if current contents of cache are some rows of $A$, it’s OK
  - if current contents of cache are some columns of $A$: slow!
  - tuning crucial: peak performance up to 4 orders of magnitude higher than performance observed in practice (without tuning)
7. Parallel Computers – Topologies

- parallel computers – distributed systems: frontier?
- different possibilities of arrangement:
  - static network topologies:
    * bus, ring, grid, or torus
    * binary tree or fat tree
    * hypercube
  - dynamic network topologies:
    * crossbar switch
    * shuffle exchange network
- crucial quantities:
  - diameter (longest path between two processors)
  - number of network connections (ports) per processor
  - parallel communications possible?
  - existence of bottlenecks?
8. Flynn’s Classification (1972)

- **SISD**: *Single Instruction Single Data*
  - classical von-Neumann monoprocessor

- **SIMD**: *Single Instruction Multiple Data*
  - *vector computers*: extreme pipelining, one instruction applied to a sequence (vector) of data (CRAY 1,2,X,Y,J/C/T90,…)
  - *array computers*: array of processors, concurrency (Thinking Machines CM-2, MasPar MP-1, MP-2)

- **MIMD**: *Multiple Instruction Multiple Data*
  - *multiprocessors:*
    * distributed memory* (loose coupling, explicit communication; Intel Paragon, IBM SP-2) or
    * shared memory* (tight coupling, global address space, implicit communication; most workstation servers) or
    * nets/clusters

- **MISD**: *Multiple Instruction Single Data*: rare
9. Memory Access Classification

- other criteria for classification:
  - scalability ($S$), programming model ($PM$), portability ($P$), and load distribution ($L$)

- **UMA**: Uniform Memory Access
  - shared memory systems: SMP (symmetric multiprocessors, parallel vector processors); PC- and WS-servers, CRAY YMP
  - advantage: $P$, $PM$, $L$; drawback: $S$

- **NORMA**: No Remote Memory Access
  - distributed memory systems; clusters, IBM SP-2, iPSC/860
  - advantage: $S$; drawback: $P$, $PM$, $L$

- **NUMA**: Non-Uniform Memory Access
  - systems with virtually shared memory; KSR-1, CRAY T3D/T3E, CONVEX SPP
  - Advantage: $PM$, $S$, $P$; drawback: cache-coherence, commun.
10. Parallelization

- classical programming paradigms are, in principle, all well-suited for explicit or implicit parallelization:
  - **imperative**: FORTRAN, C (dominant male, recently with some OO-touch like in C++)
  - **logical/reational**: PROLOG
  - **object-oriented**: SMALLTALK
  - **functional/applicative**: LISP

- implicit parallelization typically via special compilers
- explicit parallelization typically via linked communication libraries
- traditional way 😞 in Scientific Computing: FORTRAN code, vectorizing compiler, CRAY, wait for results
- explicit parallelization often difficult (cf. Gauß-Seidel), this makes non-conventional approaches attractive
11. The Programming Model MPI

• How to write parallel programs?
  – UMA systems: simple answer – just as sequential ones
  – distributed memory systems: MPI model or standard
    * Message Passing Interface
    * originally for clusters, today used even on massively parallel computers, too
    * MPI-1 developed 1992-1994
    * explicit exchange of messages: higher amount of programming work, but increasing possibilities of tuning and optimizing

• MPI Features:
  – parallel program: \( n \) processes, separate address spaces, no remote access
  – message exchange via system calls \textit{send} and \textit{receive}
  – MPI-kernel: library of communication routines, allowing to integrate MPI commands into standard languages
12. MPI Messages

- messages consist of a
  - header (recipient, buffer, type, context of communication) and of their
  - body (contents)
- messages are buffered (send buffer, receive buffer)
- sending a message can be
  - blocking (finished only after message has left node) or
  - non-blocking (finished immediately, message may be sent later)
- the same holds for receiving a message:
  - blocking: waiting;
  - non-blocking: looking for it from time to time

Cost of passing a message (length N, buffer cap. K):
\[ t(N) = \alpha \cdot \frac{N}{K} + \beta \cdot N \]

initializing cost/time \( \alpha \), transportation cost \( \beta \)
13. Programming with MPI

• a simple example:

P1: compute something
    store result in SBUF
    SendBlocking(P2,SBUF)
    RecBlocking(P2,RBUF)
    read data in RBUF
    compute again

P2: compute something
    store result in SBUF
    SendBlocking(P1,SBUF)
    RecBlocking(P1,RBUF)
    read data in RBUF
    compute again

• without buffering: deadlocks possible
  – nothing specified: buffering possible, but not imperative
  – never: no buffering (efficient, but risky)
  – always: secure, but sometimes costly

• collective communication features available:
  – broadcast, gather, gather-to-all, scatter, all-to-all,…
14. Load Distribution

• load: amount of work on processors
  – optimum: minimize idle times; needs estimates and monitoring
  – strategy: load balancing or load distribution or scheduling
  – important: avoid overhead

• one distinguishes
  – scheduling:
    * global: where do which processes run?
    * local: when does which processor which process
  – load balancing:
    * static: a priori
    * dynamic: during runtime

• in Scientific Computing applications load is often not predictable:
  – adaptive refinement of a finite element mesh,
  – convergence behaviour of iterations may differ
  – thus: static load balancing not sufficient
15. Designing Load Distribution

- Which are the primary objectives?
  - optimization of system load or application runtime?
  - placement of new processes or migration of running processes?

- Which is the level of integration?
  - Who initiates actions (measure load, chose strategy)?
    - application program
    - runtime system
    - OS?

- Any special features of the application to be considered?
  - restrictions in allocation process-to-processor frequent in S.C.

- Which units shall be distributed or displaced?
  - whole processes (coarse grain)
  - threads (fine grain)
  - objects or data (typical for simulation applications)
16. **Classification of Strategies**

- **origin of the idea:**
  from physics (diffusion model), from combinatorics (graph theory), economics (bidding, brokerage)

- **for networks, for bus topologies**

- **data represented as grids, trees, sets, or . . .**

- **distribution mechanisms:**
  - load handed over to neighbouring nodes only?
  - just distribution of new units or migration of running ones (how?)?

- **flow of information:**
  to whom is load communicated, from where comes information?

- **coordination:**
  who makes decisions? autonomous/cooperative/competitive?

- **algorithms:**
  who initiates measures? adaptivity? costs relevant? evaluation?
17. Examples of LD-Strategies

• diffusion model:
  permanent balancing process between neighbours

• bidding model:
  supply and demand, establishment of some market

• broker model:
  – esp. for heterogeneous hierarchical topologies, scalable
  – broker with partial knowledge, budget-based decision whether
    local processing or looking for better offers
  – prices for use of resources and brokerage

• matching model:
  construct matching in topology graph, balance along edges

• balanced allocation, space-filling curves, . . .
18. Performance Evaluation

- performance evaluation of algorithms and computers
- **average parallelism** (for \( p \) processors):
  \[
  A(p) = \frac{\text{sum of processor runtimes}}{\text{parallel runtime}}
  \]
- **speedup** \( S \):
  \[
  S = \frac{\text{sequential runtime}}{\text{parallel runtime}}
  \]
- **efficiency** \( E \):
  \[
  E = \frac{S}{p}
  \]
- **Amdahl’s Law**:
  assumption: each program has some part \( 0 < seq < 1 \) that can only be treated in a sequential way
  \[
  S \leq \frac{1}{seq + \frac{1-seq}{p}} \leq \frac{1}{seq}
  \]
- another important quantity: CCR (Communication-to-Computation Ratio)
  - CCR often increases with increasing \( p \) and constant problem size (example: iterative methods for \( Ax = b \))
  - therefore: do not compare speedups for different \( p \), but same problem size