TU Wien

Institute of Information Systems Engineering
Research Division of Compilers and Languages



# Übersetzer für Parallele Systeme
# Compilers for Parallel Systems

## LVA 185.A64
## SS 2018

Hans Moritsch
hans.moritsch@tuwien.ac.at

# Literature

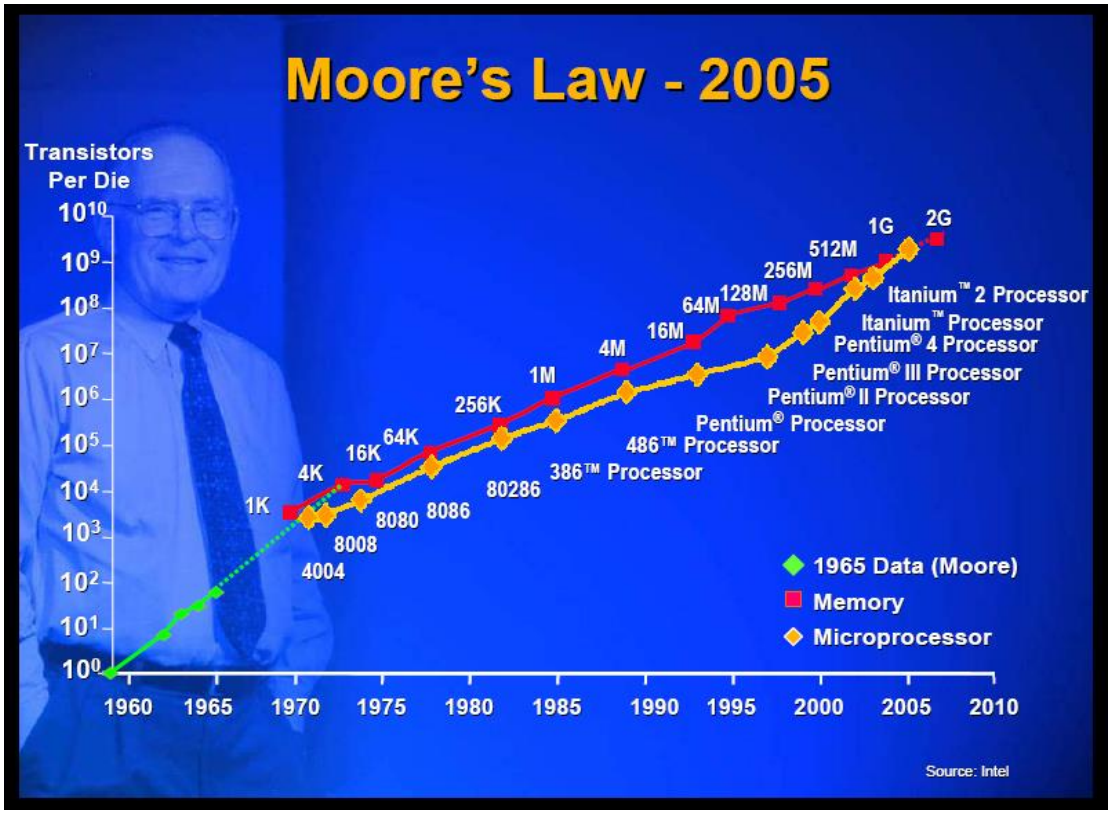

- Randy Allen, Ken Kennedy. *Optimizing compilers for modern architectures.* Kaufmann, 2002.
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques, & Tools (2nd Edition).* Pearson Addison Wesley, 2007.
- Michael J. Wolfe. High-*Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.
- Hans Zima, Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

# Parallel Computing Applications

- Parallel Computing is a fundamental tool of science
  - computational science and engineering is the "third pillar" of scientific discovery
  - in addition to theory and experiments
- Simulation instead of experiments for construction of models for natural phenomena
  - collision of galaxies cannot be studied using experiments
- Models are described by differential equations
  - discretized and solved numerically
  - finer discretization increases accuracy, e.g. spatial resolution of weather forecast models
  - ever-increasing demand for computing power
  - HPC High Performance Computing
- Application areas
  - biology, energy research, high energy physics, astrophysics, aerospace, automotive technologies, seismology, image processing, pattern matching, semiconductor technologies, VLSI design, tomography, genetic engineering, molecular simulation, drug design, …

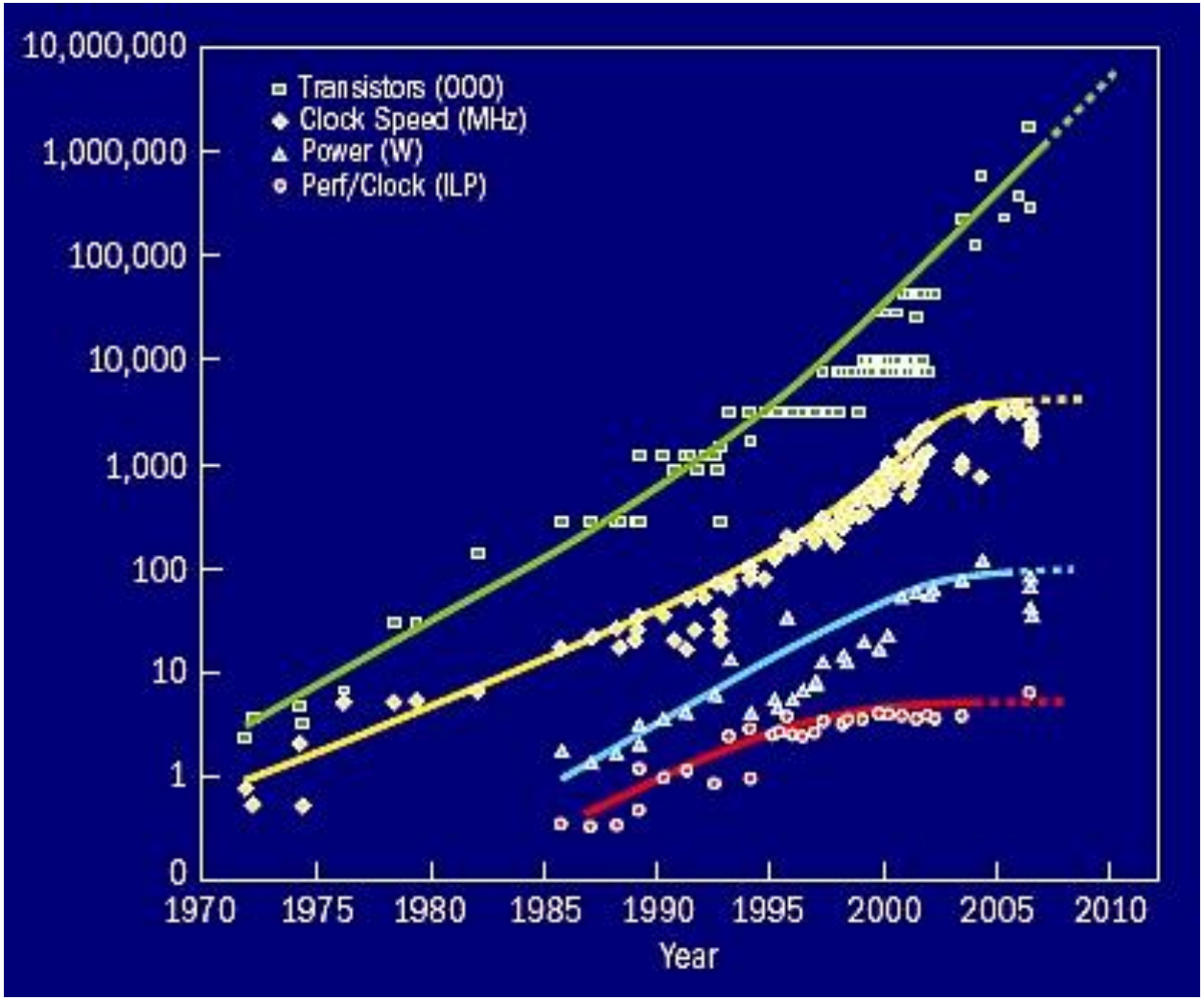
# Moore's Law

*Gordon Moore (co-founder of Intel), 1965: The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years.*




*Source: www.intel.com*

# Sources of Performance Improvements




- Frequency scaling
  - ⇨ heat dissipation
- Multiple functional units/superscalar: Instruction Level Parallelism ILP
  - ⇨ dependences between instructions
- Multicore
  - ⇨ parallelismus must be exploited by software

*Source: D. Patterson, UC-Berkeley. Illustration A. Tovey, www.scidacreview.org*

# Simulation of the Human Brain

- 1.73 billion virtual nerve cells connected to 10.4 trillion virtual synapses
  - synapses were connected at random
  - simulated network represents 1% of the neuronal network in the brain
- Virtual synapse contains 24 bytes of memory
  - 1 petabyte of main memory
  - memory of 250.000 desktop PCs.
- Simulation on K computer
  - simulates 1 sec of brain activity in 40 min computing time
  - 88.128 octacore nodes (705.024 cores)
  - 10.5 petaflops ($10.5*10^{15}$ floating point operations per second)
  - full-scale simulation of the brain will require a computer 100.000 times faster

*Source http://io9.com/this-computer-took-40-minutes-to-simulate-one-second-of-1043288954*

# Architectures and Programming

- Vector/Array computers
  - entire arrays handled with a single instruction
  - programming paradigm close to sequential programming
- Multiprocessing systems
  - introduce additional dimension of programming complexity
  - systems of processes
    - individual sequential threads of execution
    - running concurrently at unspecified speeds
  - synchronization required to prevent deadlocks, race conditions, unintentional data sharing, …

# Programming Support

- Languages: extensions to standard programming languages
  - C/C++, Fortran, Java, …
  - Fortran 90 vector extensions for array manipulation
  - High Performance Fortran HPF: sequential program + specification of data distribution
  - explicit control of parallelism, e.g. Occam, Ada, …
- Interfaces and Libraries
  - MPI, pthreads, Intel TBB, …
- Algorithms
- Tools

# Parallel Program Development

- Programmer formulates concurrent algorithm explicitly, using language extensions and/or libraries
  - machine specific vs. higher abstraction level
  - shield user from details of concurrency management
- Large body of existing sequential code
  - e.g. legacy code
  - manual transformation using language extensions is very time consuming

# Compiler Support

- "Supercompilers" perform automatic restructuring
  - sequential program specifies computation
  - transform same operations into parallel form
  - program analysis to detect hidden concurrency
  - vectorization - vector code
  - parallelization - code for parallel computer
- User supported transformation process
  - restructuring guided by program annotations and assertions
  - user supplies information and makes decisions
  - semi-automatic parallelization
  - interactive systems
- Advantages
  - development, verification, debugging of sequential programs is easier
  - existing IDE's for sequential programming
  - Supports portability between different parallel architectures

# Limitations

- Quality of parallelized program depends on structure of algorithm and target architecture
  - mainly for data parallelism
- Can only parallelize a given algorithm, not construct a new parallel algorithm, e.g. for sorting
  - compilers do not change essential features of the algorithm
  - scalable parallel algorithms can be qualitativly different from sequential algorithms
- Vectorization well established
- Parallelization more difficult
  - complexity, variety, heterogeneity of parallel systems
- Exploitation of machine specific features in source program impedes automatic parallelization

# Data Dependence

- Execution order in procedural languages is fixed
- Change of order can affect program semantics
  - hidden, implicit concurrency

    $S_1$ a=b+c

    $S_2$ d=e*f
  - reverse order ? ok
  - concurrent execution ? ok
- Bernstein's conditions, 1966
  - $S_1$ executed before $S_2$
  - $DEF_i$ … variables written in $S_i$, $USE_i$ … variables read in $S_i$,
  - $(DEF_1 \cap USE_2) \cup (USE_1 \cap DEF_2) \cup (DEF_1 \cap DEF_2) = \emptyset$

    *true* *anti* *output* dependence

# Control Dependence

* Conditional statement

```
S1 if (a<0) then
S2     x = -y
   else
S3     x = y+1
end if
```

* $S_2$ and $S_3$ are control dependent on $S_1$
* Condition in $S_3$ must be evaluated before execution of $S_2$ and $S_3$

# Parallelizing Compiler Components

- Front end
  - scanning
  - parsing
  - (static) semantic analysis
  - construction of intermediate representation (IR)
    - atttributed abstract tree
    - symbol table
- Based on IR
  - program analysis
    - flow analysis
    - dependence analysis
  - program normalization
    - e.g. loops increment always 1

# Program Analysis

- Flow analysis
  - information about flow of scalar data
  - control flow analysis
  - data flow analysis
  - arrays treated as a whole
- Dependence analysis
  - dependence relation between statements
  - constructs dependence graph
  - examines array indices
  - in general undecidable
  - effective tests available if indices are linear functions of loop variables

# Transformation Component

- Program transformations
- Loop transformations
  - loop interchange, - distribution, - unrolling, ...
- Application determined by
  - program properties
    - analysis information
    - target architecture
- More or less specific to target architecture
- Program analysis results are used for
  - verification of preconditions (transformation specific)
  - selection of „good" transformations
  - evaluation of effect of transformation after application

# Example

* Sequential code

```
for i=100 to 1 step -1
   a[i-2] = b[i+1] + c[i]
   e[i]   = f[i-3] / a[i-2]
end for
```

* Loop normalization

```
for $i=1 to 100
   a[100+($i-1)*(-1)-2)] = b[100+($i-1)*(-1)+1] +
                           c[100+($i-1)*(-1)]
   e[100+($i-1)*(-1)]    = f[100+($i-1)*(-1)- 3] /
                                  a[100+($i-1)*(-1)- 2]
end for
i=0  dead code, if not read before next assignment to i
```

# Example (2)

* Dead code elimination, standardization, simplification

```
for i=1 to 100
   a[99-$i]  = b[102-$i] + c[101-$i]  form: a0+a1*$i
   e[101-$i] = f[98-$i] / a[99-$i]
end for
```

* Dependence analysis

```
for i=1 to 100
   a[99-$i]  = b[102-$i] + c[101-$i]
   e[101-$i] = f[98-$i] / a[99-$i]    true dependence
end for
```

* Loop distribution: generate loops with single statement body

```
for i=1 to 100
   a[99-$i] = b[102-$i] + c[101-$i]
end for
for i=1 to 100
   e[101-$i] = f[98-$i] / a[99-$i]
end for
```

# Example (3)

- Vector code generation can be applied to distributed loops
  - `a[l:u]` denotes elements <a[l], a[l+1], … a[u]>
  - addition of all respective elements concurrently

```
a[-1:98] = b[2:101] + c[1:100]
e[1:100] = f[-2:97] / a[-1:98]
```