Theme: Towards Reconfigurable HPC
Lecture **9**

Advanced and Emerging Parallel Programming Paradigms

Manfred Mücke

Research Lab Computational Technologies and Applications
University of Vienna

Inverted CERN School of Computing, 3-5 March 2008

Objectives

Objectives

- Stressing importance of parallel problem specification
- Outlining common steps in parallel programming
- Presenting different || programming paradigms
- Presenting some (parallel) HPC languages

Contents

- Why parallel programming?
- Concurrency and parallelism
- From concurrency to parallelism
- Parallel programming paradigms
- New Languages in the making (HPCS)

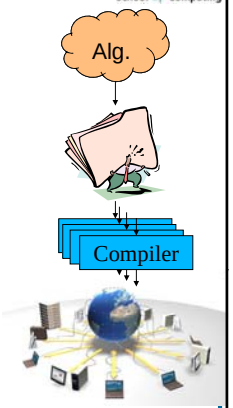
Don't we have enough problems?

Why Parallel Programming?

Why Parallel Programming?

- Contemporary computing platforms become increasingly parallel **at different levels**:
  - Grid, BOINC, Clusters
  - Manycores, multicores, GPUs
  - MMX, SSE, AltiVec
  - FPGAs
- Compilers can only exploit parallelism within the limits of the language used.

⇨ **Parallel languages allow much simpler and more reliable extraction of parallelism than sequential languages**.



(Grid, Cluster, Multicore, GPU, FPGA, Hybrid)

Why Parallel Programming II

⇨ **Because it appears to be the the only solution to exploiting the diverse range of computing platforms we keep inventing.**

"A major challenge for modern HPC systems is their lack of programmability."

M.Weiland, EPCC Edinburgh [1]

"Parallel programming languages are, I think, the most important issue in computing today"

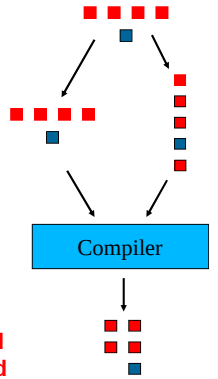
Burton Smith, Microsoft

But it might take a while....

A word on parallelizing compilers

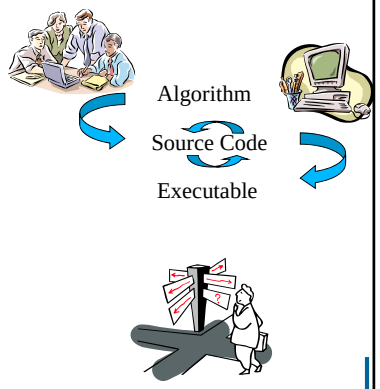
- **Most real-world problems feature some concurrency.**
- **Sequential programming arbitrarily serializes concurrent tasks.**
- **Parallelizing compilers need to reverse-engineer code to separate real from programmer-induced sequential code.**
- **This job gets harder, the larger the scope (ILP possible, applications).** ☹

⇨ **Parallelizing compilers have a hard and increasingly impossible job when faced with purely sequential code!**



Men vs. Machine

- We tend to optimize existing code.
- Only helpful if original code structure matches hardware.
- Few people nowadays know the platform, their software will run on (say in 5 years).
- Good performance across platforms is only possible, if concurrency (not explicit parallelism) is expressed in source code ... and if we have suitable compilers.



Two friends

Concurrency and Parallelism

Concurrency and Parallelism

Concurrency and parallelism are often used synonymously.

- **Concurrency:** The independence of parts of an algorithm (= independent of each other).
- **Parallelism** (also parallel execution): Two or more parts of a program are executed at the same moment in time.

Concurrency is a necessary **prerequisite** for parallel execution

but

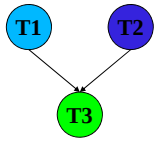
Parallel execution is only one **possible consequence** of concurrency.

Expressing Concurrency

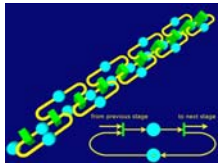
- **Mathematical notation (e.g. associativity)**

$$x = ab + cd + ef$$

- **Taskgraphs**

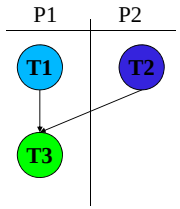


- **Petri Nets**
  (places, transitions, directed arcs, tokens, firing)
- **...**

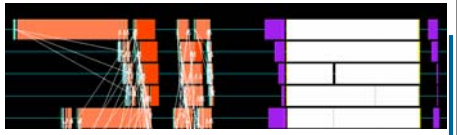


Expressing Parallelism

- **Annotated (sequential) source code**
- **Mapped task graphs**



- **Gantt Charts**



- **...**

The Ideal World

Our world is not sequential!

The more concurrency a program contains,

- the more parallel instructions/threads/tasks can be extracted *reliably* by (suitable) compilers and
- the better the automatic mapping on distributed hardware architectures.

⇨ In the ideal world, compilers map efficiently your parallel high-level description on a set of different parallel architectures

Our World

- Classical CPUs are sequential.
- There is an enormous **sequential programming knowledge** out there.
- Parallel Programming is requiring **new skills and new tools**.
- **"Saving existing investments":** Everyone is reluctant to make the big move. Small transitional steps are preferred.

Let's face the challenges ahead!

Capitalizing on Concurrency

From Concurrency to Parallelism

From Concurrency to Parallel Execution

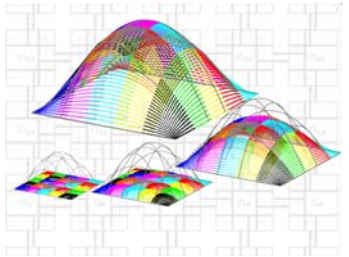
This is not a one-stop show!

- **Inherent Concurrency**
  - Decomposition
    - Mapping
      - Add Communication
        - Add Synchronization
          - Enjoy Parallel Execution

now in more detail...

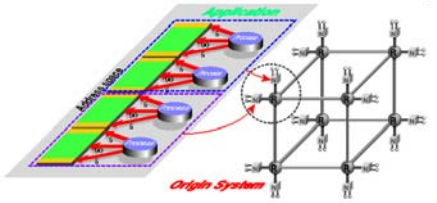
Decomposition

- Divide problem (program/data) into separate tasks /threads/operations to be executed/evaluated in parallel on distinct devices.



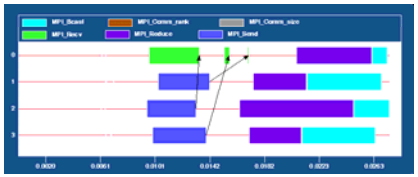
Mapping

- Assign identified tasks to specific hardware units.
- Observe
  - Communication patterns
  - Network topology
  - Special functionality (FPU, ..)



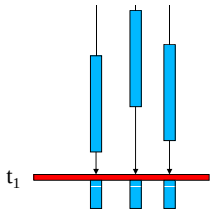
Communication

- Implement communication action if nonlocal data is required.
- Hardware-dependent
- **Deadlock**
  (waiting for data which never arrives)



Synchronization

- Make sure each task within a set has reached a certain point.
- Waiting time accumulates fast!
- **Deadlock**
  (waiting for absent task)



Explicit vs. Implicit

- **Explicit specifications** allow the programmer to guide the implementation, but pollute the code with (low-level) details.

```
float a, b, c;
x = (a + b) + c
```

- **Implicit specifications** give more freedom to the compiler, but make hand-tuning difficult

```
smallfloat a, b;
biggerfloat c;
x = a + b + c;
```

- Explicit specifications define well the **level of abstraction** provided.

Decomposition	✗✓
Mapping	✗✓
Communication	✗✓
Synchronization	✗✓

Don't be pragmatic..

Parallel Programming Paradigms

|| Programming Paradigms

Paradigm = Framework, Mindset,
Fundamental style of programming

Evaluation criteria:

- Level of abstraction
- Achievable quality of results
- Ease of use
- Match with hardware architectures (Compiler complexity)

|| Programming Paradigms

- Message Passing
  - master/slave
  - subgroups
  - individual
- Data Parallelism
- Functional Parallelism
  - all parallel
  - parallel sequential processes
  - sequential parallel processes
- Hybrid
  - parallel objects

Every paradigm can be implemented on any hardware!

Message Passing

- Focus is on **independent, communicating units**.
  Each unit has its own memory and processor.
- Unknown concept to sequential programming languages
  ⇨ easily implemented as additional library
- **MPI** (Message Passing Interface) is the best-known incarnation of message passing.
  - MPI is a standardized set of routines and respective bindings for C, C++ and Fortran.
  - MPI separates communication from implementation
    ⇨ a new library release can improve performance
  - MPI programs are portable

Message Passing Variations

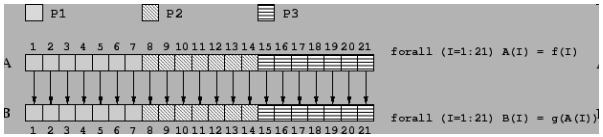
- **Uniform Master/Slave**
  Master distributes data and synchronizes all slaves.
  Direct Implementation: MPI collective communication primitives (Broadcast, Reduce, ...)
  - \+ Code scales easily (just change n)
  - \+ Just one code for all machines
  - \- No load balancing possible
- **Nonuniform Master/Slave**
  - \+ Load balancing possible
  - \- Explicit communication with each slave

Message Passing Variations

- **Subgroups**
  Several masters can coexist, directing slave subgroups.
  MPI: Communicators
  - \+ Different tasks can be accomplished in parallel
  - \- Coordination of Masters and respective subgroups
- **Individual**
  N coexisting, independent tasks
  - \+ Arbitrary communication complexity
  - \- Only send/receive
  - \- Deadlock

Data Parallelism

- The same operation is applied to multiple data sets.
  Data sets implicitly define task distribution
- Extends existing data structures
  Simple implementation as annotations, e.g.
  - ⇨ High-Performance Fortran
  - ⇨ OpenMP
-
  - \+ Very efficient for matching problems
  - \+ Easy to debug (communication is implicit)
  - \- Load balancing difficult



Functional Parallelism

- Parallelism is derived from **program flow**.
  Independent loops/blocks/functions can be evaluated in parallel.
- How to **identify independent code**?

Imperative languages	--
Single-assignment languages	+
Functional languages	+
Mathematical Notation	++

⇨ Intimate link with matching programmming language

- **All parallel**
  All actions are parallel, except if stated otherwise

Functional Parallelism

- **Parallel sequential processes**
  Program is composed of sequential building blocks, which are **executed in parallel.**
  + Encapsulates sequential programming model
  - Defining correct interfaces is crucial

  ⇨ fork/merge
  ⇨ pThreads

- **Sequential parallel processes**
  Program is composed of inherently parallel building blocks, which are **executed sequentially**
  + Exploits well fine/medium-grained parallelism
  + Merging of neighbouring operations possible

  ⇨ Parallel Skeletons

Parallel Objects

- OOP: **Object = Data + applicable functions**
- OOP maps all objects on one sequential machine.
- Parallel Objects can reside on different processors.
  ⇨ Messages become communication
  ⇨ Objects can migrate (load balancing)
  ⇨ Smart run-time system required
- Implementation example: Mentat (1993, University of Virginia)
  C++ extension + run-time system

Hardware Support

- All programming paradigms can be mapped (more or less efficient) on any hardware architecture.
- Usually a few hardware features can greatly simplify or complicate this mapping:
  - Synchronization ⇔ Hardware Semaphores
  - Message passing ⇔ Communication Co-Processor
  - Shared Memory ⇔ Transactional Memory
  - Debugging ⇔ Distributed Trace Memory

⇨ Parallel systems will change the typical CPU feature set

Anyone daring enough to invent (kind of) new languages?

HPCS

HPC Languages

High-Performance Fortran (HPF): Last big attempt (1993) to create a language for HPC, supporting parallel execution. HPF failed to attract enough interest.

Current situation: Exploiting more decent, but generic approaches (**MPI, OpenMP**)

2002: DARPA launches High Productivity Computing Systems (**HPCS**) programme, because "it becomes ever more difficult to exploit all the resources a [HPC] system has to offer."

The languages developed should:
- support general parallelism and
- separate algorithms from implementation
- easy to learn for anyone who has programming experience

HPC Languages in the Making

- **Fortress** by Sun Microsystems (funding 2002 - 2006)
  Syntax very close to mathematical notation
  Interpreted language on top of JVM
- **X10** by IBM (funding 2002 - 2010)
  Extension of Java
  Compiles into Java
- **Chapel** by Cray (funding 2002 - 2010)
  OOP-style, borrowing from C, Java, Fortran and Ada
  Compiles into C
- Common: global view model (= single, partitioned address space)
- Full implementations awaited.
- Will Sun continue work on Fortress?

Fortress, X10, Chapel & FPGAs?

An empty space so far...

Further Reading

[1] M. Weiland, **"Chapel, Fortress and X10: Novel languages for HPC"** The University of Edinburgh, Tech. Rep., October 2007. http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0706.pdf

[2] Skillicorn, D. B., Talia, D., June 1998. **Models and languages for parallel computation.** ACM Comput. Surv. 30 (2), 123-169. http://portal.acm.org/citation.cfm?id=280277.280278

[3] Dehon, A., Hutchings, B., Rudusky, D., Hwang, J., Nikhil, Raje, S., Stoica, A., 2004. **What is the right model for programming and using modern FPGAs?** In: FPGA '04. ACM, New York, NY, USA, pp. 119-119. http://portal.acm.org/citation.cfm?id=968281

Q & A

Combining two extremes (in one language)

Controlflow vs. Dataflow

Dataflow

Dataflow is a paradigm based on the idea that changing the value of a variable should **automatically force recalculation** of the values of depending variables

- Dataflow architecture = Reactive systems
- Dataflow programming = Reactive programming

⇨ Native paradigm to describe I/O behaviour of stateless hardware.

Controlflow

- Controlflow is a paradigm, based on the idea that a machine executes a stream of instructions, one at a time. Controlflow statements manipulate the stream of instructions.
- Basic controlflow operation: Conditional Branch

⇨ **Native paradigm to manage execution of programs on CPUs (von-Neumann architectures).**