Software Architectures for Flexible Task-oriented Program Execution on Multicore Systems

Thomas Rauber[1] Gudula Rünger[2]

University of Bayreuth, Germany
Chemnitz University of Technology, Germany

Complex Systems Design & Management (CSDM) 2010
Oct 2010, Paris, France

Outline

Introduction

- Future cluster and multicore systems will soon offer **ubiquitous parallelism** for all application areas.
- However: the mainstream programming model is still **sequential**. **sophisticated programming techniques** are required to access the parallel resources available
- A **change in programming and software development** is imperative for making the capabilities of the new architectures available to programmers and users of all kinds of software systems.
- Many effective parallel programming models are available often from **HPC**; **examples:** MPI, OpenMP, and Pthreads;
- **problem:** the **level of abstraction** provided may be **too low-level**.
- Trend towards large multicore systems with many (heterogeneous) cores requires a **higher level of abstraction** for software development to reach **productivity and scalability**.

Contribution of the article

- We propose a **parallel programming methodology** exploiting **task-based views** of software products.
- The main goal is to deliver a **hybrid, flexible and abstract parallel programming model** at a **high level of abstraction**.
- The **main feature** of the approach is a **decoupling of the specification of a parallel algorithm from the actual execution** of the parallel work units identified by the specification on a given parallel system.
- We propose to extend the standard model of a task-based execution to **multi-threaded tasks** that can be executed by multiple cores.
- An appropriate **specification mechanism** allows the expression of algorithms from different application areas on an abstract level.
  This **facilitates the development of parallel programs** significantly and makes parallel computing resources available to a **large community of software developers**.

Outline

Task decomposition

- The execution environment is based on a **decomposition of the computations** of the software system into **tasks**.
  task creation can be **static** or **dynamic**;
- Specification of **task creation** is **separated from task execution**.
- Input and output data may cause **dependencies between tasks**.
  illustration by a **task dependence graph (TDG)**.
- **Independent tasks** can be executed **in parallel**.
- different **execution modes** of tasks can be considered:
  - **Sequential execution of a task** on a **single core**;
  - **Parallel execution** of a task on multiple cores with a **shared address space** using a multi-threaded execution with **synchronization**;
  - **Parallel execution** of a task on multiple cores employing a **distributed address space** performing **intra-task communication** by message-passing;
  - **Parallel execution** of a task on multiple cores with a **mixed shared and distributed address space** (synchronization and communication).

Task decomposition

- An application program is represented by a **set of tasks** $\mathcal{T} = \{T_1, T_2, \ldots\}$ and a **coordination structure** $\mathcal{C}$ defining the **interactions between tasks**.
- A task captures a logical unit of the application and can be defined with **different granularities**.
- The **coordination structure** describes **possible cooperations** between the tasks.
- The cooperation between the tasks can be specified in different ways:
  - Tasks cooperate by specifying **input-output relations**, i.e., one task outputs data that is used by another task as input; in this case, the tasks must be executed **one after another**;
  - Tasks are **independent** of each other, allowing a **concurrent execution** without interactions;
  - Tasks can cooperate during their execution by exchanging information or data, thus **requiring a concurrent execution of cooperating tasks**.

Task definition

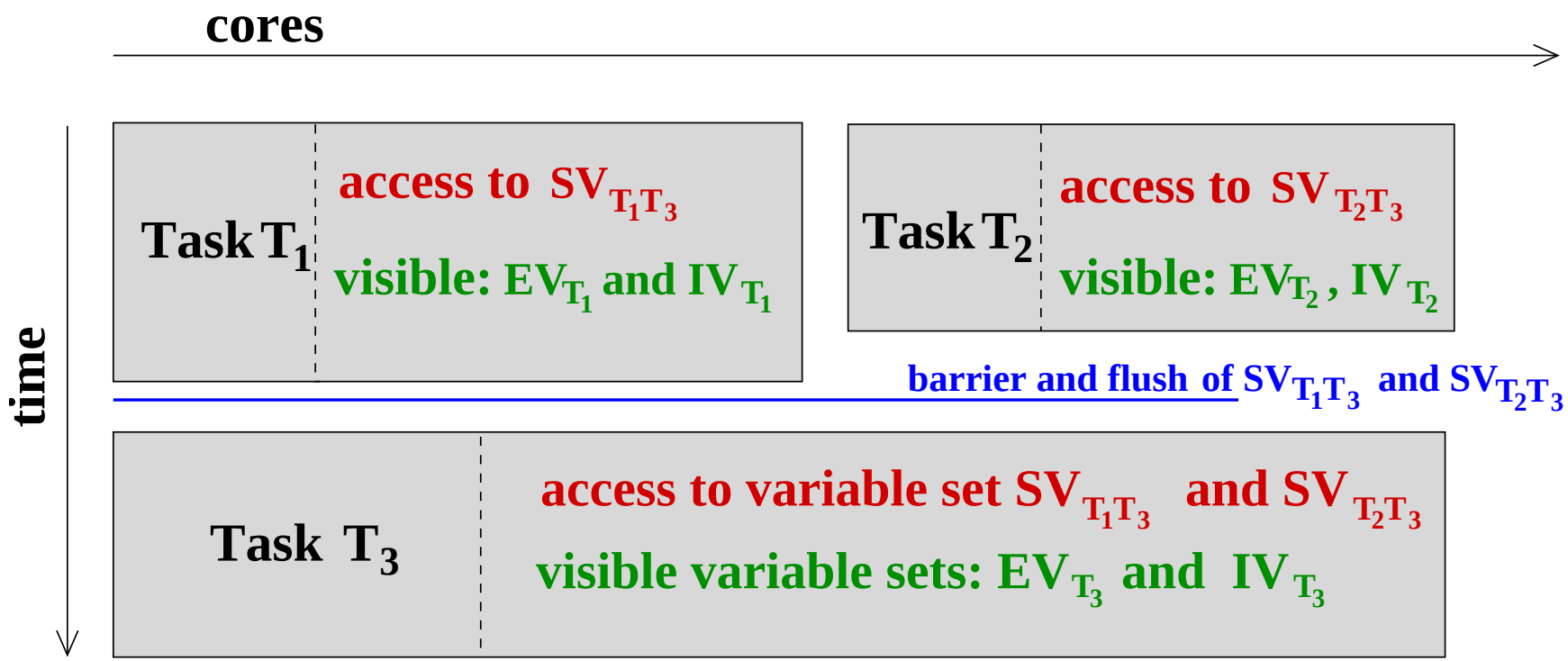
- Tasks and their interaction can be defined **statically** or **dynamically**
- **static definition:** all tasks and their interactions are known at **compile time** before the actual program execution;
  $\longrightarrow$ appropriate **scheduling and mapping techniques** can be used
- **dynamic definition:** tasks may also be created dynamically at runtime
  $\longrightarrow$ task deployment must be planned at runtime
- here: parallel tasks using a **shared address space**;
  coordination structure based on **input-output dependencies**.
- Tasks that are **not** connected by an **input-output relation** can be executed in **any execution order**.
- Each task is implemented as a multi-threaded program using **shared variables** for information exchange.
- For each task, **internal variables and external variables** can be distinguished.

Task interaction – internal and external variables

- **Internal variables** are only **visible** within the task and can be accessed by all threads executing that task.
  $\longrightarrow$ **synchronization mechanism** for a coordinated access to the internal variables required
- **External variables** of a task are visible to all other tasks of the program.
- visibility restricted to an **input-output relation** between tasks $T \longrightarrow T'$;
  $\longrightarrow$ the execution of T has to be **finished** before the execution of T' starts;
- T and T' can be executed on the **same set** or **different sets of cores**;
  $\longrightarrow$ output variables of T must be **made available** to the cores executing T'
- The **input variables** of a task T are provided **before** the actual execution starts;
- The **output variables** of T are exported **after** the execution of T has been finished.

Task interaction – example

- input-output relation $T_1 \longrightarrow T_3$ and $T_2 \longrightarrow T_3$;
  $\longrightarrow$ T_1 and T_2 can be executed in parallel;



information exchange $T_1 \longrightarrow T_3$ and $T_2 \longrightarrow T_3$;

Coordination language

- The coordination language provides operators for specifying **dependencies** or **independencies** between task.
  operators to express **dependence** and **independence** between tasks;
  generalization: **parallel loops**
- **Example 1**: **iterated RK method** for solving differential equations;
- **Example 2**: **extrapolation method** for solving differential equations;
- **compiler framework** for analysis and translation into executable parallel programs;

Specification program of the iterated RK method

External task declarations:

StageVector(IN f: scal $\times$ vec(n) $\rightarrow$ vec(n), x: scal, y: vec(n),
s: scal, A: mat(s $\times$ s), h: scal, V: list[s] of vec(n);
OUT vnew: vec(n))

ComputeApprox(IN f: scal $\times$ vec(n) $\rightarrow$ vec(n), x: scal, y: vec(n),
s: scal, b: vec(s), h: scal, V: list[s] of vec(n);
OUT ynew: vec(n))

StepsizeControl(IN y: vec(n), ynew: vec(n); OUT hnew: scal, xnew: scal)

Task definitions:

ItRKmethod(IN f: scal $\times$ vec(n) $\rightarrow$ vec(n), x: scal, xend: scal, y: vec(n),
s: scal, A: mat(s $\times$ s), b: vec(s), h: scal;
OUT X: list[] of scal, Y: list[] of vec(n))
= while(x $<$ xend) $\{$
ItComputeStagevectors (f, x, y, s, A, h ; V)
$\circ$ ComputeApprox (f, x, y, s, b, h, V ; ynew)
$\circ$ StepsizeControl (y, ynew; xnew, hnew) $\}$

ItComputeStagevectors(IN f: scal $\times$ vec(n) $\rightarrow$ vec(n), x: scal, y: vec(n),
s: scal, A: mat(s $\times$ s), h: scal;
OUT V: list[s] of vec(n))
= InitializeStage(y ; V)
$\circ$ for(j=1,...,m)
parfor (l=1,...,s) StageVector(f, x, y, V ; Vnew)

Specification program of the extrapolation method

External task declarations:

BuildExtrapTable(IN Y: list[r] of vec(n); y: OUT vec(n))

ComputeMicroStepsize(IN j: scal, H:scal, r:scal; OUT h_j:scal)

EulerStep(IN f: scal $\times$ vec(n) $\longrightarrow$ vec(n), x: scal, y: vec(n), h: scal;
OUT ynew: vec(n))

StepsizeControl(IN y: vec(n), ynew: vec(n); OUT hnew: scal, xnew: scal)

Task definitions:

ExtrapMethod(IN f: scal $\times$ vec(n) $\longrightarrow$ vec(n), x: scal, xend: scal, y: vec(n),
r: scal, H: scal;
OUT X: list[] of scal, Y: list[] of vec(n))
= while(x $<$ xend)
parfor(j=1,...,r) MicroSteps(j, f, x, y, H ; y_j)
$\circ$ BuildExtrapTable((y_1,...,y_r) ; ynew)
$\circ$ StepsizeControl(y, ynew ; Hnew, xnew)

MicroSteps(IN j:scal , f: scal $\times$ vec(n) $\longrightarrow$ vec(n), x: scal, y: vec(n), H: scal;
OUT ynew: vec(n))
= ComputeMicroStepsize(j, H, r ; h_j)
$\circ$ for(i=1,...,j) EulerStep(f, x, y, h_j ; ynew)

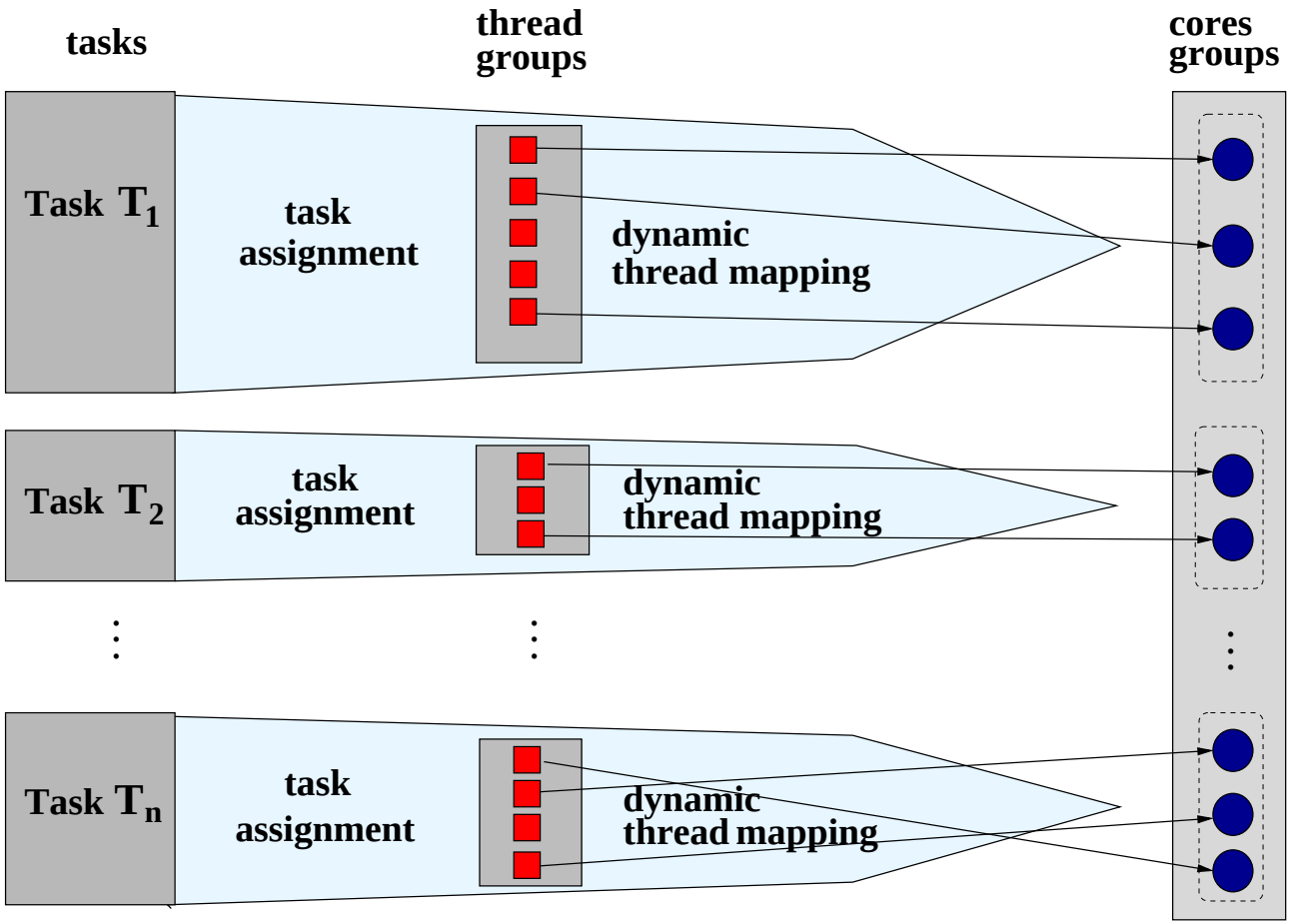
Outline

Software architectures for task-based programs

- For the **execution of a single task**, the task is **mapped** to a **set of cores** together with its internal variables.
- The coordination structure does **not** specify an **exact execution order** of the tasks, but leaves some **degree of freedom** when tasks are **independent** of each other.
- task **scheduling problem**: For a given coordination structure, how can the tasks be mapped to the cores such that a **minimum overall execution time** results?
- There are usually **several tasks** that can be executed next at each point of program execution.
- The scheduler has knowledge about idle cores of the multicore platform and selects tasks for execution from the **set of ready tasks**.
- The **set of tasks** to be executed next must be defined and the **number of cores** used has to be determined for each of the tasks selected.

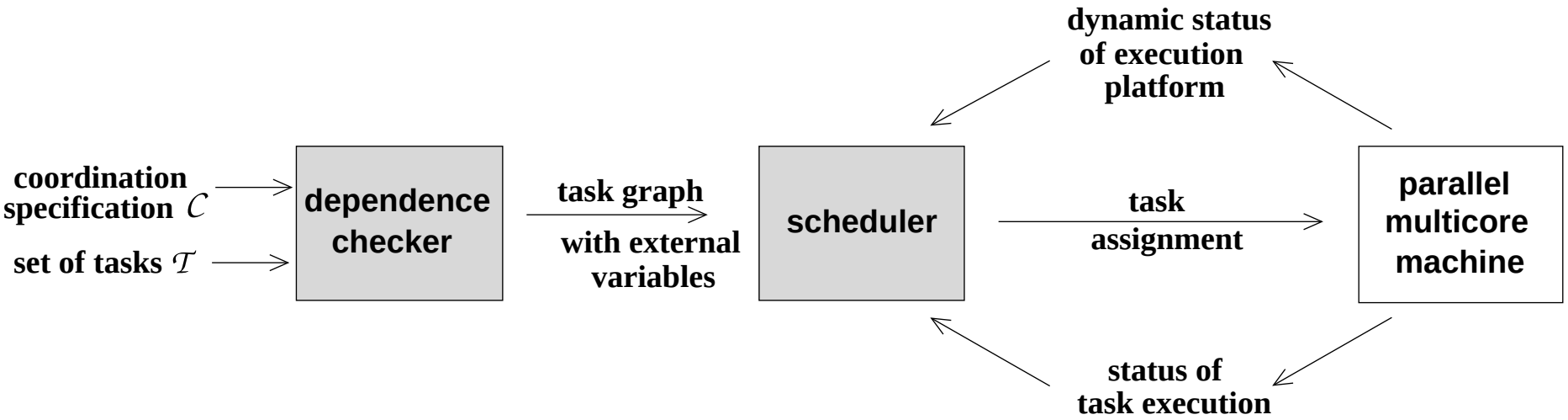
Task scheduling

- The assignment of the threads to cores is done by a **thread scheduler**.



Software environment for task-based programs

- **correctness checker** checks correctness of the input task graph
- The scheduler requires **feedback** about the status of the execution platform

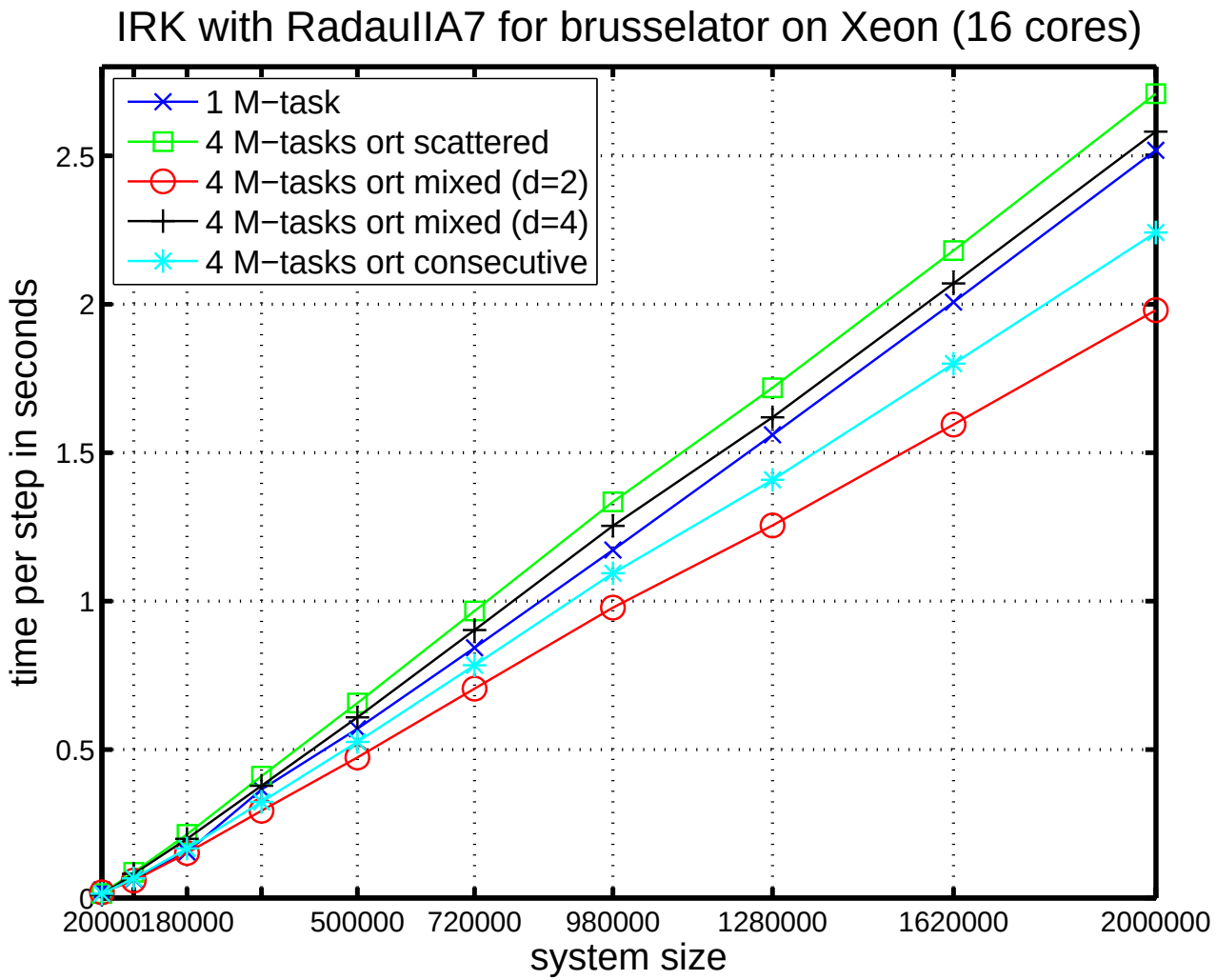


Outline

Runtime experiments

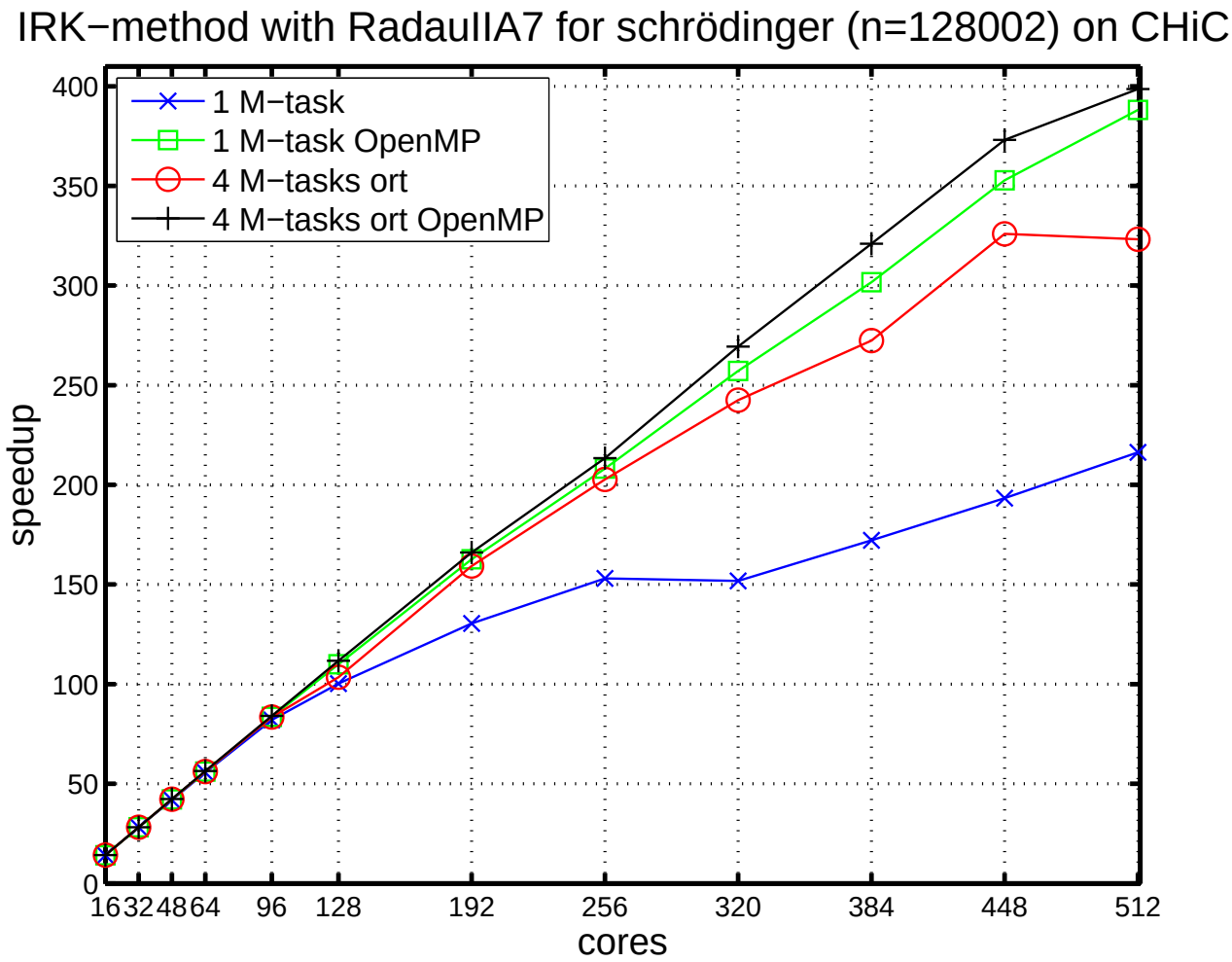
- Task-based executions can provide **competitive runtimes** compared to traditional parallel programming techniques.
- Illustration for **three platforms**:
  Xeon cluster with two nodes with two Intel Xeon E5345 “Clovertown“ quad core processors, infiniband network
  Chemnitz High Performance Linux Cluster (CHiC): 538 nodes, each consisting of two AMD Opteron 2218 dual core processors
  SGI Altix LRZ Munich: 2 Itanium 2 dual-core per node;
  128 nodes per partition;
- **application**: iterated RK method with four stage vectors;
  execution times of **one time step**;
  spatial discretization of the **2D Brusselator equation**;
  Galerkin approximation of a **Schrödinger-Poisson system**

Runtime per time step for iterated RK method (sparse ODE)



spatial discretization of the 2D Brusselator equation

Speedup for iterated RK method (dense ODE)

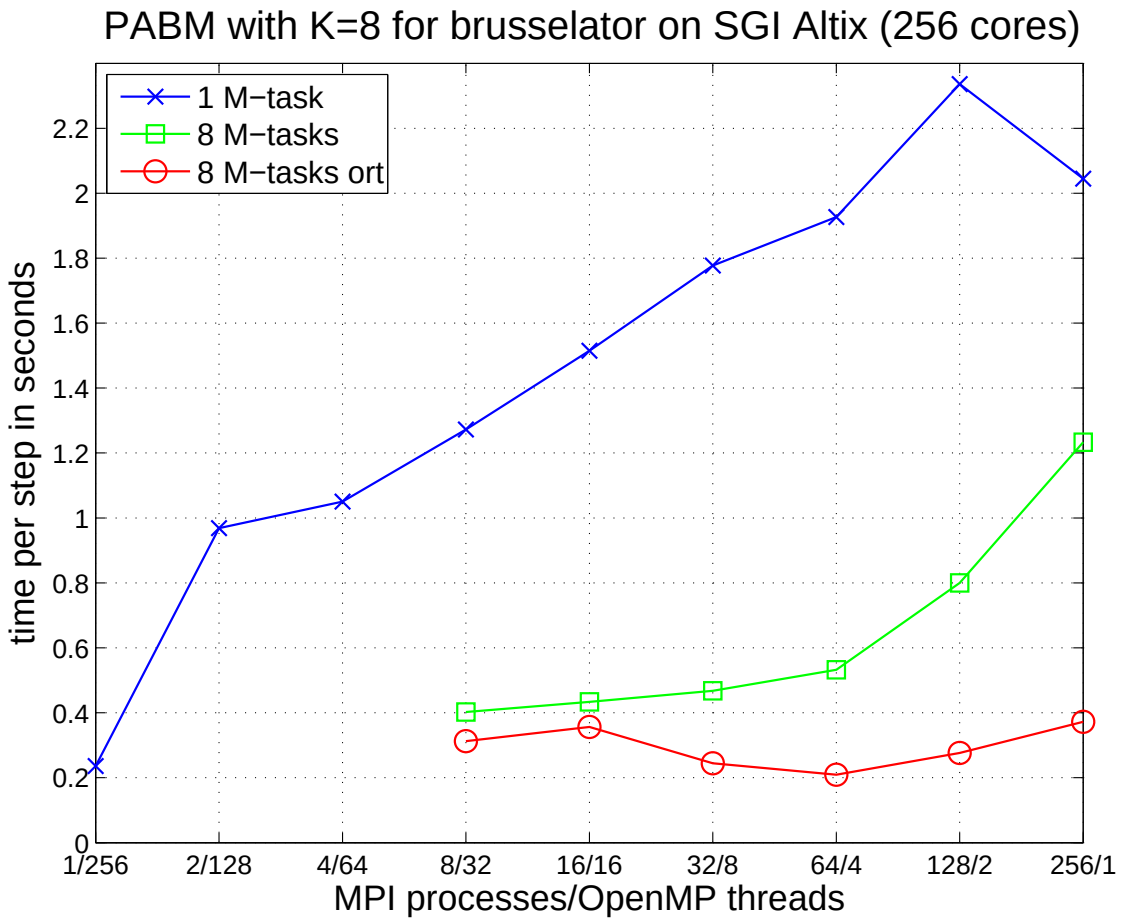


Galerkin approximation of a Schrödinger-Poisson system

MPI tasks vs. OpenMP threads

different combinations of MPI processes and OpenMP threads;

8-stage PABM for sparse ODEs on SGI Altix: time per step:



orthogonal: 64 processes and 4 threads minimizes execution time;

Outline

Conclusions

- The **portability** and **efficient execution on multicore architectures** will be an important property of all future software products;
- A **task-based parallel programming model** provides a useful abstraction;
  the software system is **decomposed into tasks**.
- The software system can exhibit a **dynamic behavior** such that new tasks can be activated during the execution of another task.
- The correct and efficient execution on a multicore platform is supported by a **software architecture** and a **task scheduler** at application program level.
- Both are integrated into a **separate runtime library** which supports the execution of arbitrary task-based software systems.