Chapter 11
Mixed Programming Models Using Parallel Tasks

Jörg Dümmler
Chemnitz University of Technology, Germany

Thomas Rauber
University Bayreuth, Germany

Gudula Rünger
Chemnitz University of Technology, Germany

ABSTRACT
Parallel programming models using parallel tasks have shown to be successful for increasing scalability on medium-size homogeneous parallel systems. Several investigations have shown that these programming models can be extended to hierarchical and heterogeneous systems which will dominate in the future. In this chapter, the authors discuss parallel programming models with parallel tasks and describe these programming models in the context of other approaches for mixed task and data parallelism. They discuss compiler-based as well as library-based approaches for task programming and present extensions to the model which allow a flexible combination of parallel tasks and an optimization of the resulting communication structure.

INTRODUCTION
Large modular parallel applications can be decomposed into a set of cooperating parallel tasks. This set of parallel tasks and their cooperation or coordination structure are a flexible representation of a parallel program for the specific application. The flexibility in scheduling and mapping the parallel tasks can be exploited to achieve efficiency and scalability on a specific distributed memory platform by choosing a suitable mapping and scheduling of the tasks. Each parallel task is responsible for the computation of a specific part or module of the application, and can be executed on an arbitrary number of processors. The terms multiprocessor tasks, malleable tasks and moldable tasks have been used to denote such parallel tasks. In the following, we use the term multiprocessor task (M-task). An M-task can be implemented
using an SPMD programming model (basic M-task) or can be hierarchically composed of other M-tasks and thereby support nested parallelism (composed M-task). The advantage of the M-task programming model is to exploit coarse-grained parallelism between M-tasks and fine-grained parallelism within basic M-tasks in the same program and thus the potential parallelism and scalability can be increased.

Each M-task provides an interface consisting of a set of input and output parameters. These parameters are parallel data structures that are distributed among the processors executing the M-task according to a predefined distribution scheme, e.g. a block-wise distribution of an array. A data dependence between M-tasks $M_1$ and $M_2$ arises if $M_1$ produces output data required as an input for $M_2$. Such data dependencies might lead to data re-distribution operations if $M_1$ and $M_2$ are executed on different sets of processors or if $M_1$ produces its output in a different data distribution than expected by $M_2$. Control dependencies are introduced by coordination operators, e.g. loop constructs for the repeated execution of an M-task or constructs for the conditional execution of an M-task. The data and control dependencies between M-tasks can be captured by a graph representation. Examples are macro dataflow graphs (Ramaswamy, Sapatnekar, & Banerjee, 1997) or series-parallel (SP) graphs (Rauber & Rünger, 2000).

The actual execution of an M-task program is based on a schedule of the M-tasks that has to take the data and control dependencies into account. M-tasks that are connected by a data or control dependence have to be executed subsequently. For independent M-tasks both, a concurrent execution on disjoint processor groups or an execution one after another are possible. The optimal schedule depends on the structure of the application and on the communication and computing performance of the parallel target platform. For the same application a pure data parallel schedule that executes all M-tasks consecutively on all available processors might lead to the best results on one platform but a mixed task and data parallel schedule may result in lower execution times on another platform. Thus, the parallel programming with M-tasks offers a very flexible programming style exploiting different levels of granularity and making parallel programs easily adoptable to a specific parallel platform.

Examples for M-task applications come from multiple areas. Large multi-disciplinary simulation programs consist of a collection of algorithms from different fields, e.g. aircraft design (Chapman, Haines, Mehrota, Zima, & van Rosendale, 1997; Bal & Haines, 1998) that uses models from aerodynamics, propulsion, and structural analysis or environmental simulations (Chapman et al., 1997) that combine atmospheric, surface water, and ground water models. Examples from numerical analysis include solution methods for ordinary differential equations (ODEs) like extrapolation methods (Rauber & Rünger, 2000), iterated Runge-Kutta methods (Rauber & Rünger, 1999a), implicitly iterated Runge-Kutta methods (Rauber & Rünger, 2000), or Parallel Adams methods (Rauber & Rünger, 2007). These time-stepping methods compute a fixed number of independent stage vectors within each time step and combine these vectors into the new approximation vector for the next time step. Partial differential equations (PDEs) can be defined over geometrically complex domains that are decomposed into sets of partially overlapping discretization meshes. Solution methods for PDEs can exploit coarse-grained parallelism between these meshes and fine-grained parallelism within the meshes (Merlin, Baden, Fink, & Chapman, 1999; Diaz, Rubio, Soler, & Troya, 2003). Hierarchical algorithms and divide-and-conquer algorithms compute partial solutions for independent subsets of the input and derive the final solution from these partial results. Examples are multi-level matrix multiplication algorithms (Hunold, Rauber, & Rünger, 2008). Stream-based applications process input streams by several pipeline stages and can exploit task and data parallelism by replicating non-scaling stages and executing the replicas concurrently. Examples come from image processing (Subhlok & Yang, 1997) and sensor-based programs that periodically process data produced by sensors (Subhlok & Yang, 1997; Bal & Haines, 1998; Orlando, Palmerini, & Perego, 2000).
Related Content

Architecturing Resource Aware Sensor Grid Middleware: Avoiding Common Errors in Design, Simulation, Test and Measurement
[www.igi-global.com/chapter/architecturing-resource-aware-sensor-grid/58745?camid=4v1a](www.igi-global.com/chapter/architecturing-resource-aware-sensor-grid/58745?camid=4v1a)

Design Methodologies and Mapping Algorithms for Reconfigurable NoC-Based Systems
[www.igi-global.com/chapter/design-methodologies-mapping-algorithms-reconfigurable/44223?camid=4v1a](www.igi-global.com/chapter/design-methodologies-mapping-algorithms-reconfigurable/44223?camid=4v1a)

Replica Placement Strategy for Data Grid Environment
[www.igi-global.com/article/replica-placement-strategy-for-data-grid-environment/78736?camid=4v1a](www.igi-global.com/article/replica-placement-strategy-for-data-grid-environment/78736?camid=4v1a)

Cloud Computing Security: Opportunities and Pitfalls
[www.igi-global.com/article/cloud-computing-security/62997?camid=4v1a](www.igi-global.com/article/cloud-computing-security/62997?camid=4v1a)