A Lightweight Task Graph Scheduler for Distributed High-Performance Scientific Computing

Josef Weinbub[1], Karl Rupp[1,2], and Siegfried Selberherr[1]

[1] Institute for Microelectronics, TU Wien
[2] Institute for Analysis and Scientific Computing, TU Wien, Vienna, Austria

Abstract. The continually growing demand for increased simulation complexity introduces the need for scientific software frameworks to parallelize simulation tasks. We present our approach for a task graph scheduler based on modern programming techniques. The scheduler utilizes the Message Passing Interface to distribute the tasks among distributed computing nodes. We show that our approach does not only offer a concise user-level code but also provides a high degree of scalability.

1 Introduction

The ever-growing demand of increased simulation complexity to better model physical phenomena requires, among other things, the combination of different simulation components [11]. This combination can be, for example, realized, by using the output of one tool as an input for another one. From a software point of view, this problem can be modelled as a task graph [10], which is governed by a software framework [5]. The individual simulation tools can be seen as vertices of the task graph, which are therefore executed based on the individual task dependencies. To improve the efficiency and therefore reduce the overall run-time of the framework, a parallelized approach for the task execution is required. A high-degree of flexibility is provided by a distributed approach based on the Message Passing Interface (MPI), as the execution can be spread among the nodes of a large-scale cluster environment as well as on the cores of a single workstation. In general, the distribution of parallelizable tasks among distributed [7] and shared computing [4] resources is a typical way to improve the overall run-time performance of a task graph. In this work we investigate a lightweight approach to implement a scheduler based on modern programming techniques, in particular, generic [12] and functional [8] programming in C++. By utilizing these techniques and external libraries we are able to achieve a highly concise user-level code, by simultaneously obtaining excellent scalability with regard to the execution performance.

This work is organized as follows: Section 2 introduces our approach and Section 3 validates the work by depicting performance results.

P. Manninen and P. Öster (Eds.): PARA 2012, LNCS 7782, pp. 563–566, 2013.

2 Our Approach

Our approach for distributing tasks on distributed computing nodes can be split into three parts. The first part is the mapping of the tasks and the corresponding dependencies on a graph datastructure; the second, the priorization based on the task dependences by utilizing a graph algorithm; and the third, the parallel and distributed execution on the computing nodes by using the Boost MPI Library [2]. Figure 1 depicts the principle of generating a task-graph and the parallel execution.

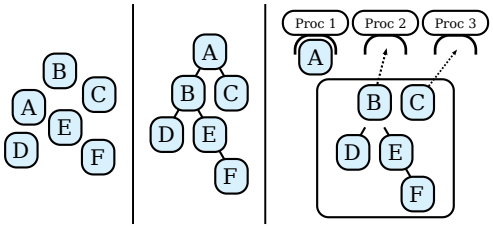


Fig. 1. Tasks are associated with vertices (**left**) and dependencies are related to edges (**middle**) in the graph. The tasks are executed on distributed processes according to their dependencies (**right**). For example, task `B` and task `C` are only executed, when task `A` is finished.

We utilize the Boost Graph Library (BGL) for the graph datastructure and the graph algorithms [1]. Each task is associated with a vertex in the graph, whereas the dependencies are mapped to edges connecting the respective vertices. Our implementation is based on the list scheduling technique, which requires a sequential list of prioritized tasks [9]. This priorization is computed by the BGL implementation of the topological sort graph algorithm [6]. According to the list scheduling approach, this prioritized list is traversed and every task is checked, whether it can be executed. This traversal is repeated until all tasks have been processed. In general, we utilize the generic and functional programming techniques. The generic programming paradigm is used to achieve a highly versatile and extendable implementation. The functional style allows to provide an intuitive user-level code and is applied by utilizing the Boost Phoenix Library (BPL) [3]. The utilization ot these programming paradigms enables to implement the following concise user-level code, which depicts the scheduling traversal of the prioritized tasks.

```
1 std::for_each(prioritized.begin(), prioritized.end(),
2    if_(is_executable)[execute(arg1,ref(process_manager))]);
```

The set of prioritized tasks (`prioritized`) is traversed. `if_(is_executable)[..]` checks if a task is ready for execution, which is done by testing the state of the immediate predecessors. If so, `execute(..)` tries to assign the task to a process

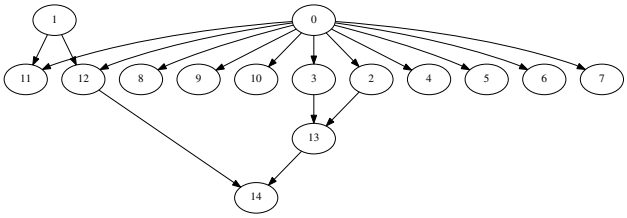
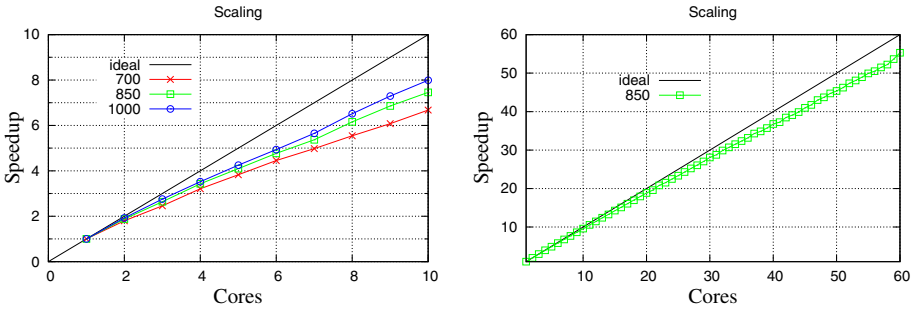


Fig. 2. An exemplary task graph is shown containing a maximum of 11 parallelizable tasks (task 2-12).



(a) Task-Graph with Dependencies **(b)** Task-Graph without Dependencies

Fig. 3. Left: The scaling for different task problem sizes is depicted based on a task graph with dependencies. The scaling efficiency for 10 cores is improved from 68% for a problem size of 700 to 80% for a problem size of 1000. **Right:** The scaling for a task problem size of 850 is shown based on a task graph without dependencies. A scaling efficiency of 91% is achieved for 60 cores.

by utilizing a process manager facility. Note that `arg1` and `ref(..)` are BPL expressions which enable access to the traversal object and the reference to an existing object, respectively.

3 Performance

In this section we present the scalability of our approach. Each task computes the dense matrix-matrix product for different problem sizes to model a computational load. Note that in this work we do not investigate the data transfer between the individual tasks, as we solely focus on the scheduling and the execution of the tasks. Our approach is evaluated based on two different test cases. First, we evaluate the speedup of a task graph with various dependencies. For this investigation we basically use the same graph layout as depicted in Figure 2. However, instead of a maximum number of 11 tasks on the second level of the graph, we use a problem offering a maximum number of 100 parallelizable tasks. Furthermore, we investigate different problem sizes with respect to the dense matrix-matrix product. The scalability is investigated for up to 10 cores. The

hardware environment for this investigation consists of three workstations, two AMD Phenom II X4 965 with 8 GB of memory, and one INTEL i7 960 with 12 GB of memory, connected by a gigabit Ethernet network. Figure 3a depicts the gained performance results. A scaling efficiency of 68% for a problem size of 700 is improved to 80% for a problem size of 1000. Second, we investigate the speedup for 600 tasks without task dependencies, to investigate the optimal parallelization capabilities. This hardware environment is based on our computing cluster, where the nodes offer four six-core AMD Opteron 8435, 128 GB of system memory, and an Infiniband DDR network connection each. Figure 3b shows the speedup for this test. A scaling efficiency of around 91% for 60 cores is achieved.

4 Conclusion

Our approach based on modern programming techniques provides not only concise user-level code but also offers excellent scalability for up to 60 cores. Furthermore, the scalability improves for larger problems, which underlines the suitability of our scheduling approach for large-scale simulations.

Acknowledgments. This work has been supported by the European Research Council through the grant #247056 MOSILSPIN. Karl Rupp gratefully acknowledges support by the Graduate School PDETech at the TU Wien.

References

1. The Boost Graph Library, `http://www.boost.org/libs/graph/`
2. The Boost MPI Library, `http://www.boost.org/libs/mpi/`
3. The Boost Phoenix Library, `http://www.boost.org/libs/phoenix/`
4. Agrawal, K., et al.: Executing Task Graphs Using Work-Stealing. In: Proc. IPDPS (2010)
5. Carey, J.O., Carlson, B.: Framework Process Patterns (2002)
6. Cormen, T.H., et al.: Introduction to Algorithms (2009)
7. Dutot, P.F., et al.: Scheduling Parallel Task Graphs on (Almost) Homogeneous Multicluster Platforms. IEEE Trans. Parallel Distrib. Syst. 20 (2009)
8. Hughes, J.: Why Functional Programming Matters. The Comput. J. 32(2) (1989)
9. Kwok, Y.K., et al.: Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Comput. Surv. 31(4) (1999)
10. Miller, A.: The Task Graph Pattern. In: Proc. ParaPLoP (2010)
11. Quintino, T.: A Component Environment for High-Performance Scientific Computing. Ph.D. thesis, Katholieke Universiteit Leuven (2008)
12. Reis, G.D.: et al.: What is Generic Programming? In: Proc. LCSD (2005)