11th International Workshop on Computational Electronics
TU Wien, 25-27 May 2006

A Novel Framework for Distributing Computations

T. Fühner, S. Popp*, and T. Jung
Fraunhofer Institute of Integrated Systems and Device Technology,
Schottkystrasse 10, 91058 Erlangen, Germany, fuehner@iisb.fraunhofer.de
* University of Applied Science Regensburg, Department of Computer Science and Mathematics,
Universitaetsstrasse 31, 93053 Regensburg, Germany, st.popp@gmx.de

I. Introduction

Decreasing costs and increasing performance characteristics of desktop-style computers lead to a significant displacement of traditional supercomputers. Nowadays, many of the high-performing computer systems are based on networks of standard PC computers (clusters) [1]. In order to efficiently make use of those distributed systems, sophisticated parallel programming techniques are required.

There is large number of programming frameworks that deal with this. Among them are:

– *Message Passing Interface (MPI):* A communication framework which provides a high-level interface for parallelization tasks on distributed memory systems [2]. Including fault-tolerance and load-balancing mechanisms has to be specially tailored per application. In addition, there is a number of different MPI implementations, many of which are not compatible to each other.

– *Parallel Virtual Machine (PVM):* This approach uses a virtual machine in which a network of heterogeneous hosts is represented as one parallel computer to the application [3]. In contrast to MPI, PVM provides a basic mechanism for fault tolerance: The failure of one node will not lead to a crash of the entire multi-node/-process application. As a drawback, compared to MPI PVM lacks a feature-rich communication interface (an often stated shortcoming is the lack of non-blocking operations).

II. DisPyTE

In this work we propose a new concept: A Dispatchable Python Tasks Environment (DisPyTE), implemented in the free, efficient, and increasingly popular interpreter language Python [4]. The main design and implementation goals of this framework are fault-tolerance and an easy-to-use load-balancing mechanism. Another important criterion was to achieve a maximum facility in both application and implementation.

The result is a requester/worker type implementation (see Fig. 1). The main components of DisPyTE are:

– *Worker*: *Workers* are realized as processes residing on any computers within the network. They are connected to by the requesters' administrator. Once a *Worker* receives a task, it it will be computed, and the result is sent back to the administrator.

– *Proxy Factory*: The factory checks whether new *Workers* are available, generating a proxy for each of them. Proxies are made available to the *Admin*. Additionally, failures of *Workers* are communicated to the *Admin*.

– *Admin*: This module manages the actual distribution process. It keeps track of the state of the *Workers* and distributes the tasks accordingly.

– *Producer*: The *Producer* generates *Tasks*. A task is conferred on the *Admin*, which assigns it to the next idle *Worker* (using the corresponding *Proxy*). Once a *Worker* has completed a task, the result is transferred to the *Requester* using a call-back routine.

The *Proxy Factory*, the *Admin*, and the *Producer* are within the scope of the same process. No extra programs such as schedulers are required. An *implicit* load-balancing mechanism is inherent, since the *Admin* is assigning tasks only to idle *Workers*. Thus, for example, a *Worker* that is twice as fast as another one, will in general be assigned twice as many tasks. The proposed approach is also fault-tolerant: If a *Worker* fails, the pending task is reassigned to a different *Worker*. Even in case no *Workers* are available (any longer), the *Admin* will wait for new computing entities to attach.

ISBN 3-901578-16-1

DisPyTE does not depend on a specific inter-process communication (IPC) mechanism. As a first implementation the Python package Twisted proved an ideal framework [5]. It provides a wide range of protocols and is freely available for all platforms that support Python.

III. Application Example: Genetic Algorithm

A genetic algorithm (GA) is a heuristic global optimization routine, which is well suited for optimization tasks that exhibit little information on the search space [6]. Each GA iteration consists of the following two steps: (1) evaluation of the (current) set of solutions and (2) recombination of these solutions. The implicitly parallel behavior of GAs can be exploited in a straightforward manner using a distributed system (see Fig. 2).

In order to use DisPyTE as the underlying distribution layer, following steps are required:

– *Producer* implementation: The GA "produces" a set of solutions per iteration. Instead of directly evaluating these solutions, they act as tasks of a DisPyTE *Producer* and are hence distributed to available *Workers*.

– *Worker* implementation: A *Worker's* job in the context of a GA is to compute the merit of a specific parameter set. As DisPyTE is using a callback mechanism, the only adjustment that has to be made is to have the DisPyTE *Worker* call the object function.

IV. Conclusions

The proposed DisPyTE framework proved very reliable and well suited for the described task. Especially in the regime of heavily loaded desktop computers, the fault-tolerance and the implicit load-balancing mechanisms payed off well. A very moderate implementation effort had to be made in order to use DisPyTE for the parallelization of the genetic algorithm. The concept has already been successfully applied to the optimization of crystal growth and lithography simulation processes (for a discussion on these applications see [7] and [8], resp.). Future work on DisPyTE will be focused on the integration of explicit load-balancing mechanisms. Moreover, as this design seems well suited for other parallelization tasks, it is also planned to adapt additional applications such that they can make use of DisPyTE.

References

[1] TOP500 Supercomputer Sites. `www.top500.org`.

[2] W. Gropp, E. Lusk, and A. Skjellum. *Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, 1999.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT press, 1994.

[4] Programming Language Python Homepage. `www.python.org`.

[5] A. Fetting. *Twisted Network Programming Essentials*. O'Reilly, Sebastopol, CA, 2005.

[6] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[7] T. Fühner and T. Jung. Use of genetic algorithms for the development and optimization of crystal growth processes. *Journal of Crystal Growth*, 266(1-3):229–238, 2004.

[8] T. Fühner, A. Erdmann, R. Farkas, B. Tollkühn, and G. Kókai. Genetic algorithms to improve mask and illumination geometries in lithographic imaging systems. In Raidl et al., editor, *EvoWorkshops 2004*, pages 208–217, 2004.

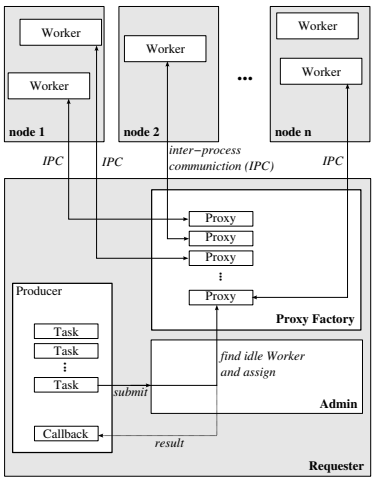

Fig. 1. Main components and call scheme of DisPyTE.

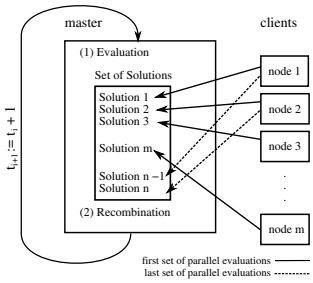

Fig. 2. Distributed computations in the GA regime.