The Design of an API for Strict Multithreading in C++

Wolfgang Blochinger and Wolfgang Küchlin

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen, Sand 14, D-72076 Tübingen, Germany
{blochinger,kuechlin}@informatik.uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de/

Abstract. This paper deals with the design of an API for building distributed parallel applications in C++ which embody strict multithreaded computations. The API is enhanced with mechanisms to deal with highly irregular non-deterministic computations often occurring in the field of parallel symbolic computation. The API is part of the Distributed Object-Oriented Threads System DOTS. The DOTS environment provides support for strict multithreaded computations on highly heterogeneous networks of workstations.

1 Introduction

High level programming models typically exhibit a high degree of abstraction, thus shielding the programmer from many low-level details of the parallel execution process. In this paper we deal with the *multithreading* parallel programming model [8] (not to be confused with the shared memory model) which is located on a higher level of abstraction than other commonly employed parallel models (e.g. message passing). Within the multithreading parallel programming model, low-level synchronization, communication, and task mapping are carried out completely transparently. Many parallel algorithm models like master-slave, divide-and-conquer, branch-and-bound, and search with dynamic problem decomposition can be realized by multithreaded computations in a straight forward manner. This property makes the multithreading model also particularly attractive for the development of the now popular Internet-computing based parallel applications, provided it is available for highly heterogeneous distributed systems.

Parallel programs that employ the multithreading programming model are called *multithreaded programs*. A *multithreaded computation* results from the execution of a multithreaded program with a given input. A considerable amount of research has been carried out on (static) scheduling techniques for multithreaded computations in order to minimize time and/or space requirements [9, 10]. In this paper we study the design of an API (application programmers interface) that supports the creation of multithreaded programs and multithreaded computations with dynamic thread creation. It is part of our parallel middleware DOTS (Distributed Object-Oriented Threads System). The overall design goal of the presented API was on the one hand to support as large a class of multithreaded computations, while on the other hand keeping the API as lean as possible in order to enable the rapid and easy creation as well as maintenance of parallel programs.

The rest of the paper is organized as follows. In Section 2 a graph-theoretic definition and a classification of multithreaded computations are given. Section 3 provides details of the multithreading API of the DOTS parallel system environment. Section 4 gives a summary and comparison with related work.

2 Definition and Classification of Multithreaded Computations

In this section we summarize the graph-theoretic model for multithreaded computations introduced by *Blumofe and Leiserson* [8] which also provides a means to classify computations of this kind according to their expressiveness.

A multithreaded computation can be described by an *execution graph*. The main structural entities of multithreaded computations are *threads*. Each thread is composed of one or more unit-time *tasks* which represent the nodes of the execution graph. *Continue edges* between tasks indicate their sequential ordering within a thread and also define the extent of the thread.

Tasks of different threads can be executed concurrently, provided that two kinds of dependencies between the threads of a computation are considered:

- A thread can create new threads by executing special tasks, called *spawn tasks*. The resulting parent/child relationship among two threads is indicated by a *spawn edge* which goes from the spawn task of the parent to the first task of the child thread. The parent thread may pass an argument to its child thread along with its creation. The tree composed by all threads of a computation and the corresponding spawn edges is called *spawn tree*. It can be viewed as the parallel variant of a call tree.
- The second type of dependencies stem from producer/consumer relationships between two threads of a computation. A thread can produce one or more results which are later consumed by other threads. This relationship is indicated by *data-dependency edges* which go from the producing to the consuming task.

A multithreaded computation can thus be represented by a directed acyclic graph of unit-tasks connected by continue, spawn and data-dependency edges. Such computations are also called *general* multithreaded computations. In Figure 1 the execution graph of a general multithreaded computation is given. Threads are shaded gray, spawn edges are printed as solid arrows and data-dependency edges are printed as dotted arrows.

Subclasses of general multithreaded computations can be identified by restricting the kinds of data-dependencies that can occur between two threads of a computation. The simplest subclass is the class of *asynchronous procedure calls*. In such computations every thread produces exactly one result which is consumed by its parent. More formally, there is only one outgoing data-dependency edge per thread and it goes to its parent thread (see Figure 2). In *fully strict computations* a thread can produce several results, but the corresponding data-dependency edges also go to its parent thread without exceptions (see Figure 3). The more comprehensive class of *strict computations* is characterized by the property that all data-dependency edges of a thread go to an ancestor of the thread in the spawn tree (see Figure 4).

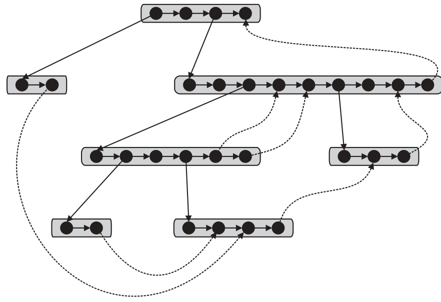

Fig. 1. Execution Graph of a General Multithreaded Computation

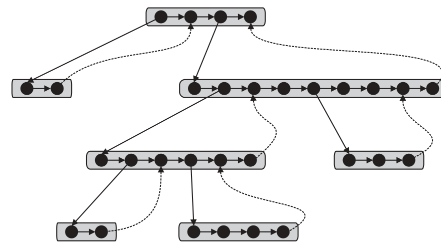

Fig. 2. Execution Graph of a Computation with Asynchronous Procedure Calls

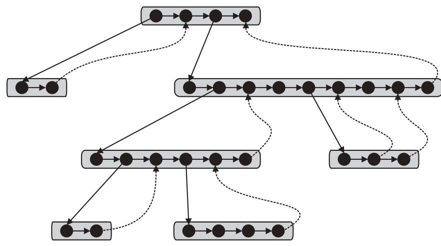

Fig. 3. Execution Graph of a Fully Strict Multithreaded Computation

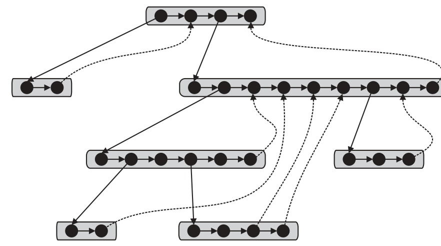

Fig. 4. Execution Graph of a Strict Multithreaded Computation

The class of strict computations plays an important role, since it is the most comprehensive class of multithreaded computations for which certain properties hold [9, 10]. In particular, it is considered to be the largest class of multithreaded computations which can be regarded as "well-structured". We will take advantage of this property in the design of the API for multithreading.

3 Multithreaded Computations in DOTS

3.1 DOTS Overview

To provide an appropriate context for the subsequent discussion on multithreading in DOTS, we first give a brief overview of the system. DOTS is a system environment for building and executing parallel C++ programs that integrates a wide range of different computing platforms into a homogeneous parallel environment [3]. Up to now, it has been deployed on heterogeneous clusters composed of the following platforms: Microsoft Windows 95/98/NT/2000/XP, Sun Solaris, SGI IRIX, IBM AIX, FreeBSD, Linux, QNX Realtime Platform and IBM Parallel Sysplex Cluster (clusters of IBM S/390 mainframes running under OS /390) [4].

Although DOTS was originally designed for the parallelization of algorithms from the realm of symbolic computation [5, 7, 6, 21], it has also been used in other application

domains like parallel computer graphics [17]. The primary design goal of DOTS was to provide a flexible and handy tool for the rapid prototyping of algorithms, especially supporting highly irregular symbolic computations, e.g. in data-dependent divide-and-conquer algorithms.

The DOTS system environment is organized in several layers. The lowest layer is the OS adaptation layer which is implemented by the ACE (ADAPTIVE Communication Environment) toolkit [1]. It homogenizes operating system APIs for a wide range of different platforms. On top of the OS adaptation layer, the DOTS run-time system is built. It provides several low-level services like object-serialization, message exchange, TCP connection caching and support for the internal component-architecture. Additionally, higher level services like task migration, node directory, logging, and load distribution are provided by the DOTS run-time.

DOTS applications can be based on several parallel programming models which are represented by separate APIs. It is possible to mix primitives from different APIs within an application.

- The *Task API* represents the basic API layer of DOTS on which all other APIs are based. It supports object-oriented task parallel programming at a basic level using directly the services of the DOTS run-time.
- The *Active Message API* provides support for object-oriented message passing.
- The *Autonomous Tasks API* can be employed to create task objects that operate as mobile agents in a heterogeneous distributed system.
- The *Thread API* provides support for strict multithreaded computations. For the rest of the paper we will focus on this API.

3.2 The DOTS API for Multithreading

The design of the multithreading API of DOTS was guided by three main goals:

- Providing high-level and easy-to-use primitives.
  The API should provide a small number of powerful and orthogonal primitives facilitating the creation of distributed parallel programs. Also the parallelization of object-oriented programs should be supported.
- Support for a comprehensive class of multithreaded computations.
  In order to support more complex multithreaded computations than asynchronous procedure calls and at the same time preserving the ”well-structured” property of computations the class of strict multithreaded computations should be supported.
- Support for typical requirements of the field of symbolic computation.
  Typical properties of parallel applications of this field are a high degree of irregularity and non-determinism. Also primitives for cancellation are required in order to efficiently support parallel search procedures.

The key concept of the DOTS API for multithreaded computations is the use of *thread group* objects as links between different primitives of the API. A thread group represents one or more active threads of a computation. Within a program, any number of thread group objects can be used. A thread can be placed *explicitly* or *implicitly* into a thread group. In the former case, the thread group object has to be supplied as

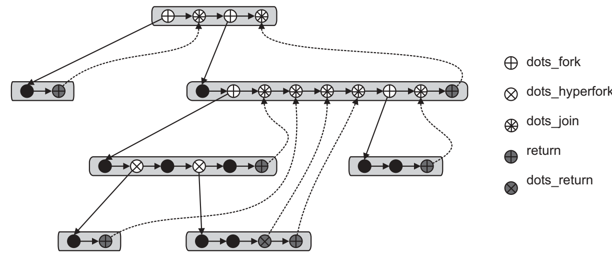

Fig. 5. A Strict Computation with DOTS Primitives

argument to a primitive. In the latter case, the corresponding thread group is determined by the dynamic context in which a primitive is executed. For all primitives that are subsequently applied to a thread group no distinctions are made between threads that have been placed explicitly and threads that have been placed implicitly into the group.

The basic primitives provided by the Thread API are dots_fork and dots_hyperfork for thread creation, dots_join for synchronizing with the results computed by other threads (which are returned using dots_return), and dots_cancel for thread cancellation. To facilitate the parallelization of existing C++ programs within the multithreading programming model, the DOTS Thread API is enhanced with object-oriented features, like argument and result objects for threads.

If a thread is created using the dots_fork primitive, it is placed explicitly into a specified thread group. Whereas, if a thread is created using dots_hyperfork, it is placed implicitly in the same thread group as its closest ancestor in the spawn tree which has been created using dots_fork. In both cases a procedure to be executed by the child thread and an argument-object for the child thread must be supplied.

Threads return result objects using the dots_return primitive. The last result of a thread is delivered by the final return statement of the procedure.

When dots_join is called on a thread group *join-any* semantics is applied: The first result which becomes available from any thread in the given group is delivered, regardless of whether the thread has been placed explicitly or implicitly into the group. If no result is available, the calling thread is blocked until one thread of the group delivers a result. If a thread has finished its execution and all result objects of the thread have been joined, it is removed from the thread group. In case dots_join is applied to an empty group (i.e. a thread group which doesn't hold explicit or implicit threads any more), 0 is returned by dots_join. By checking this return value, the end of a computation can be determined. (Note, that in general the joining thread cannot know how many results are available in a thread group.)

Figure 5 depicts the realization of the strict multithreaded computation previously shown in Figure 4 using DOTS primitives.

When applying the dots_cancel primitive to a thread group object, the execution of all threads within this group is aborted and results which have not yet been joined are deleted. All threads are removed from the thread group. A thread can abort its own

execution process with dots_abort. Analogously it is removed from its thread group and unjoined results of the thread are deleted.

The described interaction of dots_fork, dots_hyperfork and thread groups enables the programmer to succinctly formulate strict multithreaded computations. Additionally, the join-any property of dots_join together with the dots_cancel primitive extend this strict multithreading model to a flexible tool for dealing with high degrees of non-determinism, for example occurring in highly irregular search problems. Moreover, since all primitives are completely orthogonal the API can be applied easily.

3.3 Example Program

In this Section we present an example program illustrating the DOTS multithreading primitives. Figure 6 shows the (simplified) code of a parallel search process with dynamic problem decomposition.

```
bool search(SearchDesc search_space) {

  while(!search_space.empty()) {

    // begin search step
    ...
    if (solution_found)
      return true;
    ...
    // end search step

    if (dots_task_queue_length() < SPLIT_THRESHOLD) {
      // split search space and spawn new thread
      SearchDesc new_search_space = split(search_space);
      dots_hyperfork(search, new_search_space);
    }
  }

  return false;
}
```

```
int main(int argc, char* argv[]) {
  dots_reg(search); dots_init(argc, argv);

  // read in description of search space
  SearchDesc search_space; read(search_space);

  DOTS_Thread_Group thread_group;
  dots_fork(thread_group, search, search_space);

  // dots_join returns 0 if no more threads
  // are available
  bool solution;
  while (dots_join(thread_group, solution) > 0)
    if (solution==true)
        break;

  if (solution) {
    printf("solution found");
    dots_cancel(thread_group);
  } else
    printf("no solution found");

   return 0;
}
```

Fig. 6. Parallel Search with Dynamic Problem Decomposition Employing Strict Multithreading

The parallel search starts with one search thread which is forked by the main thread. Initially, this search thread has the whole search space assigned. For load balancing, new search threads are (dynamically) spawned. Each additional search thread receives a part of the search space of its parent. New threads are created during the initial phase of the parallel search in order to assign work to all available processors and every time a processor runs out of work when it has completed the execution of a thread. Typically, newly created threads will be assigned to a processer by a receiver initiated strategy which is transparently provided by a built-in DOTS load distribution component. In order to provide enough work that can be transferred to other nodes, new search threads are created when the length of the local task queue falls below a predefined limit.

Since all additional search threads are spawned using dots_hyperfork their results need not to be joined by their parents but are directly passed to the main thread resulting in a strict computation. In many search problems, the size of the individual search threads cannot be predicted. The resulting non-determinism is handled by the join-any

semantics of the dots_join primitive. The strict multithreading execution model decouples parent and child threads, leading to a smaller number of active threads. A search thread can complete its execution without prior synchronization with results of its descendants. Moreover, dynamic search space splitting can be carried out in a distributed manner. This enhances the scalability of the search process, compared to an alternative approach, where the master thread is also responsible for performing search space splits and creating corresponding search threads.

4 Related Work

In the last decade, many parallel system environments which are able to support (distributed) multithreaded computations in some way have been developed. In this section we carry out a classification of the more relevant of these approaches into three categories focusing on the programming models which they provide.

4.1 Shared Memory Multithreading based on Distributed Shared Memory (DSM)

The central theme of approaches to multithreading that fall into this category is to support the shared memory multithreading programming model, e.g. provided by many modern operating systems, on parallel architectures with distributed memory as transparently as possible. Examples of DSM multithreading systems are DSM-Threads [18], Millipede [16] or DSM-PM2 [2].

Since there are no restrictions for carrying out communication between threads in the shared memory model, even general multithreaded computations can be realized with the system environments belonging into this category. But the APIs of these platforms have always to be located on a comparably low-level of abstraction, so the formulation of structured multithreaded computations, like the strict ones, in principle cannot lead to equally structured and compact multithreaded programs.

Additionally, due to the consistency problems caused by the replication of shared data objects, all approaches to DSM multithreading have to cope with the problem of combining efficiency and programmability (transparency) on large scale distributed (heterogeneous) parallel machines. Up to now, no sufficient solutions for this problem domain exist [22].

For applications that do not further profit from DSM, using DOTS can ensure the efficient applicability in larger scale heterogeneous distributed systems and at the same time enabling rapid application development using its high level API for multithreading.

4.2 HPC Middleware for Integrating Communication and Multithreading

The system platforms discussed in this section pursue the tight integration of communication and multithreading. However, the main intention of their parallel programming models is not to carry out communication transparently, resulting in a lower level of abstraction. Typical representatives are Nexus [15], Panda [20] or Athapascan-0 [11].

These system platforms are primarily designed to be used as compiler targets or as middleware for building higher-level parallel system platforms. Due to their general nature, general multithreading can be realized with all of these platforms but communication is not transparent on the application programming level. Using such low level programming models can lead to artificially complex programs of well structured computations like strict computations.

4.3 Platforms Supporting the Fork/Join Multithreading Programming Model

Systems in this category are the most similar to DOTS; they support distributed multithreading by providing a fork/join like API. This approach to distributed multithreading carries out communication completely transparently by using argument-result semantics. However, communication between the threads of a computation is restricted to specific points during the execution of a thread.

DTS [12] (which is the predecessor of DOTS) realizes asynchronous remote procedure calls in C and Fortran. No support for object-oriented programming is provided in DTS and its deployment is limited to distributed systems composed of UNIX nodes.

Cilk [19] is a language for multithreaded parallel programming that represents a superset of ANSI C. It uses pre-compilation techniques for static code instrumentation in order to support the Cilk runtime system. There exists a prototype implementation of a distributed version of Cilk, called distributed Cilk [14] that spans clusters of SMPs.

The current version of the Cilk language features a very compact and easy to use set of parallel keywords, but no support for strict multithreaded computations is included. DOTS is library based and therefore avoids typical problems of systems that extend standard languages like the lack of standard development tools (e.g. debuggers). Moreover, DOTS is based on C++ and supports object-oriented programming. Since distributed Cilk is currently available only on a few platforms, its usability in highly heterogeneous distributed environments is limited.

Virtual Data Space (VDS) [13] is a load balancing system for irregular applications also supporting strict multithreading. In contrast to DOTS the VDS API for strict computations is very complex. For example, the VDS Fibonacci program presented in [13] needs about double the number of lines of code than an equivalent DOTS program of the same functionality.

VDS is implemented in C and therefore provides no direct support for object-oriented programming and it is not available for a wider range of common platforms, resulting in limited support for heterogeneous high performance computing.

5 Conclusion

In this paper we presented an API for building parallel programs based on the multithreading parallel programming model. Our approach is distinguished from related work by supporting the comprehensive class of strict multithreaded computations while at the same time keeping the API compact and and easy to use by completely orthogonal primitives . Additionally, the API includes primitives for efficiently dealing with non-determinism occurring in highly irregular computations.

References

[1] The ADAPTIVE Communication Environment (ACE). *http://www.cs.wustl.edu/˜schmidt/ACE.html*.

[2] ANTONIU, G., AND BOUGÉ, L. DSM-PM2: A portable implementation platform for multithreaded dsm consistency protocols. In *In Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)* (San Francisco, April 2001), vol. 2026 of *Lect. Notes in Comp. Science*, Springer-Verlag, pp. 55–70.

[3] BLOCHINGER, W. Distributed high performance computing in heterogeneous environments with DOTS. In *Proc. of Intl. Parallel and Distributed Processing Symp. (IPDPS 2001)* (San Francisco, CA, U.S.A., April 2001), IEEE Computer Society Press, p. 90.

[4] BLOCHINGER, W., BÜNDGEN, R., AND HEINEMANN, A. Dependable high performance computing on a Parallel Sysplex cluster. In *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)* (Las Vegas, NV, U.S.A., June 2000), H. R. Arabnia, Ed., vol. 3, CSREA Press, pp. 1627–1633.

[5] BLOCHINGER, W., KÜCHLIN, W., LUDWIG, C., AND WEBER, A. An object-oriented platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation 49* (1999), 161–178.

[6] BLOCHINGER, W., SINZ, C., AND KÜCHLIN, W. Parallel consistency checking of automotive product data. In *Proc. of the Intl. Conf. ParCo 2001: Parallel Computing – Advances and Current Issues* (Naples, Italy, 2002), G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, Eds., Imperial College Press, pp. 50–57.

[7] BLOCHINGER, W., SINZ, C., AND KÜCHLIN, W. Parallel propositional satisfiability checking with distributed dynamic learning. *J. Parallel Computing* (2003). To appear.

[8] BLUMOFE, R. D., AND LEISERSON, C. E. Space efficient scheduling of multithreaded computations. In *Proc. of the Twenty Fifth Annual ACM Symp. on Theory of Computing* (San Diego, CA, May 1993), pp. 362–371.

[9] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. In *35th Annual Symp. on Foundations of Computer Science (FOCS '94)* (Mexico, November 1994), pp. 356–368.

[10] BLUMOFE, R. D., AND LEISERSON, C. E. Space-efficient scheduling of multithreaded computations. *SIAM Journal Computing 27*, 1 (Feb 1998), 202–229.

[11] BRIAT, J., GINZBURG, I., PASIN, M., AND PLATEAU, B. Athapascan runtime : Efficiency for irregular problems. In *Proc. of the Europar'97 Conference* (Passau, Germany, August 1997), Springer Verlag, pp. 590–599.

[12] BUBECK, T., HILLER, M., KÜCHLIN, W., AND ROSENSTIEL, W. Distributed symbolic computation with DTS. In *Parallel Algorithms for Irregularly Structured Problems, 2nd Intl. Workshop, IRREGULAR'95* (Lyon, France, Sep 1995), A. Ferreira and J. Rolim, Eds., vol. 980 of *LNCS*, Springer-Verlag, pp. 231–248.

[13] DECKER, T. Virtual data space - load balancing for irregular applications. *Parallel Computing 26* (2000), 1825–1860.

[14] distributed Cilk. *http://supertech.lcs.mit.edu/ cilk/home/distcilk5.1.html*.

[15] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing 37* (1996), 70–82.

[16] FRIEDMAN, R., GOLDIN, M., ITZKOVITZ, A., AND SCHUSTER, A. Millipede: Easy parallel programming in available distributed environment. *Software: Practice and Experience 27*, 8 (August 1997), 929–965.

[17] MEISSNER, M., HÜTTNER, T., BLOCHINGER, W., AND WEBER, A. Parallel direct volume rendering on PC networks. In *Proc. of the Intl. Conf. on Parallel and Distributed*

Processing Techniques and Applications (PDPTA '98) (Las Vegas, NV, U.S.A., July 1998), H. R. Arabnia, Ed., CSREA Press.

[18] MUELLER, F. On the design and implementation of DSM-threads. In *Int. Conference on Parallel and Distributed Processing Techniques and Applications* (June 1997), pp. 315–324.

[19] RANDALL, K. H. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 1998.

[20] RÜHL, T., BAL, H. E., BENSON, G., BHOEDJANG, R. A. F., AND LANGENDOEN, K. Experience with a portability layer for implementing parallel programming systems. In *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)* (Sunnyvale, CA, 1996), pp. 1477–1488.

[21] SINZ, C., BLOCHINGER, W., AND KÜCHLIN, W. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)* (Boston, MA, June 2001), H. Kautz and B. Selman, Eds., vol. 9 of *Electronic Notes in Discrete Mathematics*, Elsevier Science Publishers.

[22] TANENBAUM, A. S., AND VAN STEEN, M. *Distributed Systems – Principles and Paradigms*. Prentice-Hall, 2002.