Library Support for Hierarchical Multi-Processor Tasks*

THOMAS RAUBER
Institut für Informatik
Universität Halle-Wittenberg
06099 Halle (Saale), Germany
rauber@informatik.uni–halle.de

GUDULA RÜNGER
Fakultät für Informatik
Technische Universität Chemnitz
09107 Chemnitz, Germany
ruenger@informatik.tu–chemnitz.de

Abstract

The paper considers the modular programming with hierarchically structured multi-processor tasks on top of SPMD tasks for distributed memory machines. The parallel execution requires a corresponding decomposition of the set of processors into a hierarchical group structure onto which the tasks are mapped. This results in a multi-level group SPMD computation model with varying processor group structures. The advantage of this kind of mixed task and data parallelism is a potential to reduce the communication overhead and to increase scalability. We present a runtime library to support the coordination of hierarchically structured multi-processor tasks. The library exploits an extended parallel group SPMD programming model and manages the entire task execution including the dynamic hierarchy of processor groups. The library is built on top of MPI, has an easy-to-use interface, and leads to only a marginal overhead while allowing static planning and dynamic restructuring.

Keywords: mixed task and data parallelism, multi-processor tasks, multilevel group SPMD, hierarchical decomposition of processor sets, library support, distributed memory

1 Introduction

The paper considers parallel programs built up from multi-processor tasks (M-tasks) in a modular and hierarchical way resulting in a hierarchy of M-tasks on top of SPMD tasks. Many applications from scientific computing and other areas have an inherent modular structure of cooperating subtasks calling each other. Examples include environmental models combining atmospheric, surface water, and ground water models, or aircraft simulations combining models for fluid dynamics, structural mechanics, and surface heating, see [2] for an overview. Hierarchically structured M-tasks are a natural way of coding those problems. The parallel execution of hierarchically structured M-tasks on distributed memory machines (DMMs) requires a corresponding decomposition of the set of processors into a hierarchical group structure of disjoint groups. Each M-task is mapped onto one processor group and is executed concurrently to other independent tasks in the hierarchy. Due to the hierarchical structure different decompositions on different levels may be active at different points of program execution. This results in a multi-level group SPMD computation model with varying processor group structures.

The advantage of the described form of mixed task and data parallelism is a potential for reducing the communication overhead and for improving scalability, especially if collective communication operations are used. Collective communication operations performed on smaller processor groups lead to smaller execution times due to the logarithmic or linear dependence of the communication times on the number of processors [22, 14]. As a consequence, the concurrent execution of independent tasks on disjoint processor subsets of appropriate size can result in smaller parallel execution times than the consecutive execution of the same task one after another on the entire set of processors.

For a hierarchy of M-tasks there are several ways to map M-tasks onto processor subgroups and the interaction between M-tasks influences whether a pure data parallel or a mixed task and data parallel execution is advantageous for a specific set of independent M-tasks. The communication overhead also limits the scalability of pure data parallel programs. Additional M-task parallelism can improve scalability since the concurrent execution of tasks efficiently exploits larger numbers of processors.

In this paper, we present the runtime library Tlib to support the programming with hierarchically structured M-tasks. The library's API provides separate functions for the hierarchical structuring of processor groups and for the coordination of concurrent and nested M-tasks. The task structure can be nested arbitrarily which means that a coordination function can assign other coordination functions to subgroups for execution, which can then again split the corresponding subgroup and assign other coordination func-

* 0-7695-1524-X/02 $17.00 (c) 2002 IEEE

tions. It is also possible to specify recursive splittings of groups into subgroups and to assign coordination functions to these subgroups, so recursive algorithms like divide-and-conquer algorithms or tree-based algorithms can be formulated quite naturally. Tlib library functions are designed to be called in an SPMD manner which results in multi-level group SPMD programs. The entire management of groups and M-tasks at execution time is done by the library. This includes the creation and administration of a dynamic hierarchy of processor groups, the mapping of tasks to groups and their coordination, the handling and termination of recursive calls and group splittings, and the organization of communication between groups in the hierarchy. Internally, the library uses distributed information stored in distributed descriptors, which are hidden from the user. The current version of the library is based on C and is built on top of MPI.

The contribution of this paper is to provide a library support for the coordination of hierarchically structured M-tasks. A library approach has several advantages over compiler-based approaches or hand-coding. The Tlib library provides an easy-to-use API while hiding as much of the complexity of the group management and the multi-level group SPMD organization as possible. The application programmer is relieved from realizing the technical details of hierarchical M-tasks and the corresponding group management and can concentrate on how to exploit the potential M-task structure of the given application. Compared to manual program versions, which have to explicitly incorporate the specific M-task and subgroup management for a specific parallel variant of an application, many different program versions can be coded easily when using the library Tlib. In contrast to compiler approaches with coordination languages, the runtime library allows dynamic rearrangements of processor groups and reduces additional overhead. The Tlib library is implemented efficiently, so there is usually only a marginal overhead compared to hand-coded program versions. In particular, the library is implemented such that no additional communication is needed to exchange information within processor groups or between different processor groups. The MPI operations to establish the (nested) splitting of processor groups and to assign work to these groups are executed only by those processors that belong to these groups, thus minimizing the management overhead. The corresponding information about the processor partitioning is not stored globally, but is captured in distributed data structures existing only for those processors belonging to the resulting subgroups. A small overhead, which is neglectable in most cases, is caused by some local bookkeeping.

The rest of the paper illustrates the parallel programming model in Section 2. Section 3 describes the API of the Tlib library and Section 4 highlights some of the implementations details. Section 5 shows runtime experiments, Section 6 discusses related work, and Section 7 concludes.

2 Example Application

Many applications from scientific computing or methods from numerical analysis exhibit a nested or hierarchical structure which makes them amenable for a mixed task and data parallel realization using the Tlib library. Examples are diagonal-implicitly iterated Runge-Kutta (DIIRK) methods which are implicit solution methods with integrated stepsize and error control for ordinary differential equations (ODEs) arizing, e.g., when solving time-dependent partial differential equations with the method of lines [21].

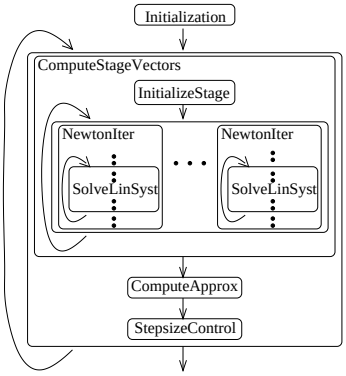

Figure 1. Modular structure of DIIRK methods.

The modular structure of DIIRK methods is given in Figure 1. Boxes denote M-tasks and back arrows denote loop structures. In each time step, the DIIRK method computes a fixed number of s stage vectors (ComputeStageVectors) which are then combined to the final approximation (ComputeApprox). DIIRK methods offer the potential of a task parallel execution, since the computation of the different stage vectors are independent of each other [18]. The computation of each stage vector requires the solution of a non-linear equation system whose size is determined by the size of the ODE system to be solved. The non-linear systems are solved with a modified Newton method (NewtonIter) requiring the solution of a linear equation system (SolveLinSyst) in each iteration step. Depending on the characteristics of the linear system and the solution method used, a further internal mixed task and data parallel execution can be used leading to another level of the task hierarchy.

A parallel implementation can exploit the modular structure in several different ways but can also execute the entire method in a pure SPMD fashion. The different implementations differ in the coordination of the M-tasks. How to write coordination functions with the Tlib library is addressed in the next section.

3 Interface of the Tlib library

The Tlib library provides an API consisting of several library functions that are implemented on top of MPI. A parallel program using the library consists of a collection of basic SPMD tasks and a set of *coordination functions* which embody M-tasks. In the coordination functions, concurrent execution is obtained by calling the corresponding *library functions* and providing a description of the (hierarchical) decomposition of the processor set. This description is stored in a *group descriptor* that is built up before the concurrent execution of tasks is initiated.

The hierarchical task structure induces a hierarchy of group descriptors with different stages of activity during program execution. The internal structure of the group descriptors and their relationship is hidden from the application programmer, see Section 4.

Interface Design The interface of the library has been designed similar to the Pthreads-interface, i.e., the functions to be executed concurrently are given as arguments to a specific library function together with an additional parameter of type `void *` containing the arguments of these functions. This requires the programmer to pack the function arguments in a single data structure which keeps the interface of the library function clean and provides flexibility for the specification of the argument function.

Because of the specific way to pass functions as arguments to a library function, all basic SPMD tasks and all coordination functions that may be passed as an argument to a library function are required to have type

```
void *F (void * arg,
         MPI_Comm comm,
         T_Descr * pdescr)
```

where the MPI communicator `comm` can be used for group-internal communication and `pdescr` is a pointer to a group descriptor that contains information about the processor group onto which the function is mapped. The body of `F` may contain further calls to library functions to generate subgroups and to initiate task parallel executions on these subgroups. `F` may also generate a recursive call of itself on a smaller subgroup, thus enabling the implementation of divide-and-conquer algorithms. The recursive splitting of processor groups can be programmed such that the recursion is stopped as soon as the different groups generated do not contain enough processors for a further splitting or a further splitting would lead to an inefficient program. As well as group-internal communication using the `comm` parameter, global communication is possible in the body of `F` by using a different communicator like `MPI_COMM_WORLD`.

The library provides functions for initialization, splitting of groups into two or more subgroups, assignment of tasks to processor groups, and getting information on the subgroup structure. The prototypes of the library functions are collected in `twol.h` which has to be included in the program. Each call of a library function returns 0 if the function call was successful and an appropriate error code otherwise.

Initialization To initialize a Tlib program, the library function `T_Init()` is provided. This function has to be executed before any other function of the library is called.

```
int T_Init( int argc,
            char * argv[],
            MPI_Comm comm,
            T_Descr * pdescr) .
```

The parameter `pdescr` is a pointer to an object of type `T_Descr` which is filled by the call to `T_Init()` and is used by later calls of library functions. The data structures of type `T_Descr` are opaque objects that represent a communicator, the corresponding processor group, and information about the relative position and organization within the decomposition hierarchy. The function `T_Init()` is used to establish the initial group and communication structures for the executing processors. All subsequent library functions are called in the context of this initial `T_Descr` object until a succeeding `T_Descr` object is created and becomes active.

Splitting operations Before initiating a concurrent execution, appropriate processor groups and corresponding communicators have to be established. By calling the following library function, two disjoint processor groups are generated:

```
int T_SplitGrp( T_Descr * pdescr,
                T_Descr * pdescr1,
                float per1,
                float per2) .
```

where `pdescr` is a pointer to an existing group descriptor; `pdescr1` is a pointers to a group descriptor that is filled by the library for each processor calling the `T_SplitGrp()` function; `per1` and `per2` specify fractional values with $per1 + per2 \leq 1$. The effect of the call of `T_SplitGrp()` is the creation of two disjoint processor groups and their corresponding communicators such that the processor groups contain a fraction of `per1` and `per2` of the processors of the original processor group described by `pdescr`, respectively. The group descriptor `pdescr1` contains the descriptions of the new processor groups and their communicators after the return to the calling function. Each processor specifies a separate descriptor object of name `pdescr1` filled by the call of the splitting

operation, so that each processor has access to processor-specific information as well as to global information about the overall group structure.

A more general splitting operation is provided to split a processor group into more than two subgroups.

```
int T_SplitGrpParfor( int n,
                      T_Descr * pdescr,
                      T_Descr * pdescr1,
                      float p[]) .
```

The first parameter `n` specifies the number of different groups in the partition; `pdescr` is a pointer to an existing group descriptor that has previously been established; `pdescr1` is a pointer to a group descriptor filled by the call of this function; `p` is an array of length `n` specifying fractional values with $\sum_{i=0}^{n-1} p[i] \leq 1$. If `T_SplitGrp()` or `T_SplitGrpParfor()` are called with a parameter `pdescr` representing less than 2 or `n` processors, respectively, the split operation will not be performed and an appropriate error code will be returned. The library function `T_SplitGrpParfor` internally exploits MPI mechanisms to create subgroups and new communicators. Also each participating processor builds up the new group descriptor and there are several error detecting tests.

An additional splitting operation with a similar interface as the MPI function `MPI_Comm_Split()` takes an existing group descriptor as input and creates a decomposition of this group using `color` and `key` values that are specified by the different processors individually. The function has the following form:

```
int T_SplitGrpExpl( T_Descr * pdescr,
                    T_Descr * pdescr1,
                    int color,
                    int key) .
```

Again `pdescr` is a pointer to an input descriptor of an existing group; `pdescr1` is a pointer to an output descriptor that is filled by the call of this function; `color` is an integer value specified by each processor individually; processors that specify the same `color` value belong to the same processor group, so the number of disjoint processor groups corresponds to the number of different `color` values; `key` is also an integer value specified by each processor individually; this value defines the order of the processors in the resulting subgroups.

Concurrent task execution After a splitting operation, the concurrent execution of two independent tasks can be initiated by calling the library function

```
int T_Par( void * (*f1)(void *,
                        MPI_Comm, T_Descr *),
           void * parg1,
           void * pres1,
           void * (*f2)(void *,
                        MPI_Comm, T_Descr *),
           void * parg2,
           void * pres2,
           T_Descr *pdescr)
```

where `f1` and `f2` are pointers to functions to be executed concurrently by the different processor groups induced by the descriptor `pdescr`; `parg1` and `parg2` contain pointers to the parameters for the function calls of `f1` and `f2`, respectively; `pres1` and `pres2` are pointers to the results to be computed by `f1` and `f2`, respectively; `pdescr` is a pointer to an object of type `T_Descr` that has previously been generated by a `T_SplitGrp()` operation.

The library function `T_Parfor()` allows the parallel activation of an arbitrary number of different functions according to a previously performed creation of concurrent subgroups. The function has the following form:

```
int T_Parfor( void *(*f[])(void *,
                           MPI_Comm, T_Descr *),
              void * parg[],
              void * pres[],
              T_Descr * pdescr) .
```

The parameter `f` is an array of length `n` with pointers to functions where `n` is the number of concurrent subgroups previously generated by a splitting operation like `T_SplitGrpParfor()` (the entry `f[i]` specifies the function to be executed by group `i` of the group partition); `parg` is an array of length `n` containing pointers to the arguments for the different function calls in `f`; `pres` is an array of length `n` containing pointers to results to be produced by the functions `f[i]`; `pdescr` is a pointer to a descriptor for a group decomposition into `n` disjoint subgroups, each of which is executing one of the functions given in `f`. The number `n` of functions to be executed in parallel should correspond to the number of groups created in `pdescr` as well as to the number of arguments and results.

If `T_Par()` or `T_Parfor()` are called with a parameter `pdescr` representing less than 2 or `n` subgroups, respectively, no task-parallel execution is initiated. Instead, the specified tasks are executed consecutively one after another in a data-parallel way by all processors of the group represented by `pdescr`. This situation may occur, if `T_SplitGrp()` or `T_SplitGrpParfor()` are previously called with a processor group that is too small for a further splitting.

The library function `T_Parfor()` can also be used for the realization of parallel loops by specifying the body of the loop as a specific function and by using this function for all entries of the parameter `f`. Since each iteration of the parallel loop usually works on different data, the entries of `parg` are in this case occupied with different data.

Architecture of the Tlib library The parallel behavior of a Tlib program is induced by a collection of cooperating user-defined coordination functions. A user-defined coordination function in a hierarchically structured M-task program is an SPMD C-function calling basic SPMD functions, Tlib functions for incrementally organizing the group hierarchy like `T_SplitGrp`, Tlib functions to initiate the concurrent execution of independent tasks like `T_Par`, and other user coordination functions including recursive calls. The different functions interact in the way illustrated in Figure 2. The underlying multi-level group structure is implicitly given by a hierarchy of linked group descriptors. The execution of an SPMD function or a coordination function takes place with respect to a specific group by referring to a group descriptor.

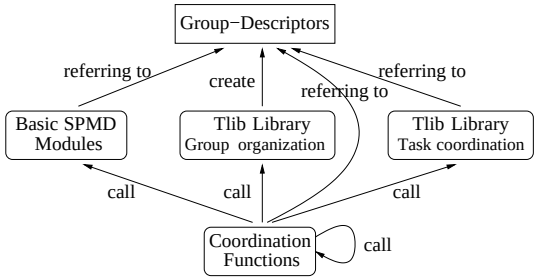

Figure 2. Architecture of the Tlib library.

Example For an illustration of the programming with the library Tlib we consider a simple hierarchical M-task program using SPMD tasks $F1, \ldots, F7$. Figure 3 illustrates a) a hierarchical M-task structure of a modular program together with b) the corresponding multi-level group structure of processors, c) the implicitly constructed group hierarchy, and d) local descriptors of selected processors. After the execution of F_1 on all processors, two disjoint subgroups G^a and G^b are built. $F2$ is executed on G^a and $F3$ is executed on G^b. After the execution of $F2$ on G^a, G^a is again decomposed into two subsets G^c and G^d such that $F4$ is executed on G^c and $F5$ is executed on G^d. Then $F6$ is executed on the larger group G^a. After the completion of $F6$ and $F3$, $F7$ is executed on all processors. Figure 4 sketches the corresponding coordination program. $F1, \ldots, F7$ are assumed to be provided as SPMD functions and are not shown explicitly. Group-internal communication within $F1, \ldots, F7$ is obtained by using the respective communicator `comm` that is provided as argument to these functions. The function `FH` capturing the coordination function to be executed by group G^a is an example for a user-defined coordination function containing calls to library functions and SPMD functions.

```
main( int argc, char * argv[] ) {
  T_Descr descr, descr1;
  void *arg_f1, *arg_fh, *arg_f3, *arg_f7;
  void *res_fh, *res_f3;
  T_Init(argc,argv,MPI_COMM_WORLD,&descr);
  /* pack argument arg_f1 */
  F1(arg_f1, MPI_COMM_WORLD, &descr);
  T_SplitGrp( &descr, &descr1, 0.7, 0.3);
  /* pack argument arg_fh in G^a */
  /* pack argument arg_f3 in G^b */
  T_Par(FH, &arg_fh, &res_fh,
        F3, &arg_f3, &res_f3, &descr1);
  /* possible redistribution of data from G^a and G^b */
  /* to G = G^a ∪ G^b; */
  F7(arg_f7, MPI_COMM_WORLD, &descr)
}

void * FH(void * arg, MPI_Comm comm,
                      T_Descr * pdescr){
   T_Descr descr2;
   void *arg_f2, *arg_f4, *arg_f5, *arg_f6;
   void *res_f2, *res_f4, *res_f5, *res_f6;
   res_f2 = F2( arg_f2, comm, pdescr);
   T_SplitGrp ( pdescr, &descr2, 0.5, 0.5);
   /* pack argument arg_f4 in G^c */
   /* pack argument arg_f5 in G^d */
   T_Par ( F4, &arg_f4, &res_f4,
           F5, &arg_f5, &res_f5, &descr2);
   /* possible redistribution of data from G^c and G^d */
   /* to G^a = G^c ∪ G^d; */
   res_f6 = F6( arg_f6, comm, pdescr);
   /* build result of FH in res_fh; */
   return res_fh;
}
```

Figure 4. Realization of a descriptor hierarchy in the Tlib library for the nested task structure from Figure 3.

4 Internal realization of the Tlib library

The library Tlib exploits distributed information about the group structure stored in group descriptors. A *group descriptor* is a data structure which holds information about a current subdivision of processors into subsets. All participating processors hold a version of the descriptor with global information about the subdivision and local information, which might be different for each processor.

Creating a group descriptor A group descriptor is built up by calling `T_SplitGrp()` or related functions from the Tlib library. A group descriptor is always constructed from an existing group descriptor. At the beginning of the program this is the group descriptor corresponding to the entire set of processors assigned to the program, which is

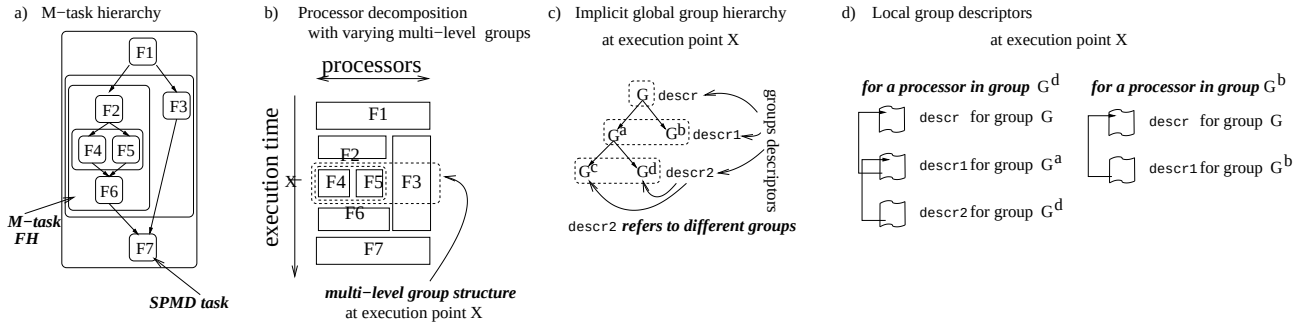

Figure 3. Illustration of a hierarchical task structure.

constructed when calling `T_init`. The creation of a new group descriptor is performed by a group of processors in an SPMD fashion. Each processor has different information in different parts of the descriptors: (i) global information about the most recent subdivision, like the number of subgroups generated and the members of those subgroups, is stored consistently for all processors participating in the splitting operation; (ii) group local information stores information about the subgroup to which the processors belong after creating the subgroups (like the communicator used for group-local communication); (iii) local information stores information relevant only to the specific processor, like the processor Id with respect to the new subgroup. All this information can either be obtained by the MPI functions for the subdivision or can be computed locally from the specification of the library function. So no additional communication is required compared to a hand-coded version. After their creation, the group descriptors can be used by functions like `T_Par()` to assign tasks to the different subgroups. The given group descriptor in a library call guarantees that each processor participating in the SPMD call automatically has knowledge about its specific situation in the group hierarchy and within the currently active groups. The separation of the creation and the use of the descriptors is useful in iterative methods where the same descriptor can be efficiently reused several times after its creation.

Information within group descriptors For the coding of M-task programs it might be useful to exploit information stored in a group descriptor `descr`. This is supported by several access functions in the library. A processor can obtain its rank by calling

```
int T_GetMyRank (T_Descr * descr).
```

The leaders of the different groups in a partition can be obtained by calling

```
int *T_GetGroupLeaders (T_Descr * descr).
```

This call yields an array with the ranks of the group leaders. The length of the array is the number of groups in the partition which is provided by a call of

```
int *T_GetNoOfGroups (T_Descr * descr).
```

The parent group descriptor which allows the access to information on the parent partition is provided by

```
T_Descr * T_GetParent (T_Descr * descr).
```

Hierarchy of group descriptors The construction mechanism for the partitioning into subgroups and for the corresponding group descriptor results in a hierarchy of subgroups that can be represented by a subdivision tree. The root of this hierarchy is the initial set of processors. Deeper levels of the hierarchy correspond to divisions into subsets. Depending on the specific level of the hierarchy, a processor belongs to different subgroups and different group organizations. The hierarchy of subgroups is accompanied by a hierarchy of group descriptors which are built up when creating subgroups and which mimics the subgroup hierarchy. The global group descriptor hierarchy is given implicitly and each processor stores only the relevant group descriptors of the groups it belongs to. Each computation in the task parallel program can be associated with a specific subgroup situation which is uniquely specified by the group descriptor. The computation is automatically adapted to this subgroup situation when the corresponding group descriptor is used in the `T_Par()` or `T_Parfor()` specification.

Visibility within the group hierarchy After the creation of a group hierarchy, the programmer can address different subgroups by using different group descriptors. For a specific subdivision, the group descriptor of each processor not only stores the information about the subgroup that the processor belongs to, but also stores information about the sibling groups and the parent group. The Tlib library provides functions to access this information. If necessary for the application program, communication between neighboring

subgroups of the same partition can be performed via the communicator stored in the parent descriptor. On the other hand, in order to keep the visibility structure clear and concise, no functions are provided to access information about a later subdivision from a parent descriptor, i.e., to access this information, a processor would have to use a descriptor at the appropriate level of the descriptor hierarchy.

5 Runtime Experiments

Diagonal-Implicitly iterated RK methods For the DIIRK methods, see Section 2, we have realized different parallel versions exploiting the task structure. All versions use a direct method to solve the linear equation systems in each iteration step of the internal Newton method. We have used these versions to solve dense and sparse ODE systems arizing when solving reaction-diffusion equations describing the behavior of binary mixtures with different characteristics.

For the Intel Paragon and the IBM SP2 the task parallel execution leads to a better performance than the pure data parallel execution that computes the stage vectors by all available processors one after another. **Figure 5** shows the resulting runtimes in seconds on 32 processors of an IBM SP2 and Intel Paragon for an application of a DIIRK method with 5 stages to the solution of a sparse, stiff ODE system. For dense ODE systems (not shown here), the runtimes and the resulting speedup values are larger for both the task parallel and data parallel execution. The task parallel execution is still much faster than the data parallel execution, but the percentage difference is smaller. Similar results for sparse and dense ODE systems are obtained on a beowulf cluster, see **Figure 6**. On a Cray T3E, however, this is no longer the case. For 5 stages, the execution time of the two execution schemes are quite similar, see **Figure 7**. For 3 stages, the pure data parallel execution scheme is even faster than the task parallel scheme. The figure shows the results for sparse ODE systems. This situation is similar for dense ODE systems. For comparison, the runtimes of the corresponding implicit RK methods are also shown in the figures.

The communication behavior of the DIIRK methods is dominated by the broadcast operations that are used within the solution methods for the linear equation systems. A performance analysis shows that the execution times of broadcast operations exhibit a logarithmic dependence on the number of participating processors on all machines considered. Both the startup time and the byte-transfer time show this dependence, so the formula

$$t_{sb}(p,n) = \tau \cdot log_2(p) + t_c \cdot log_2(p) \cdot n$$

can be used where p denotes the number of processors and n is the message size [14]. The difference between the T3E

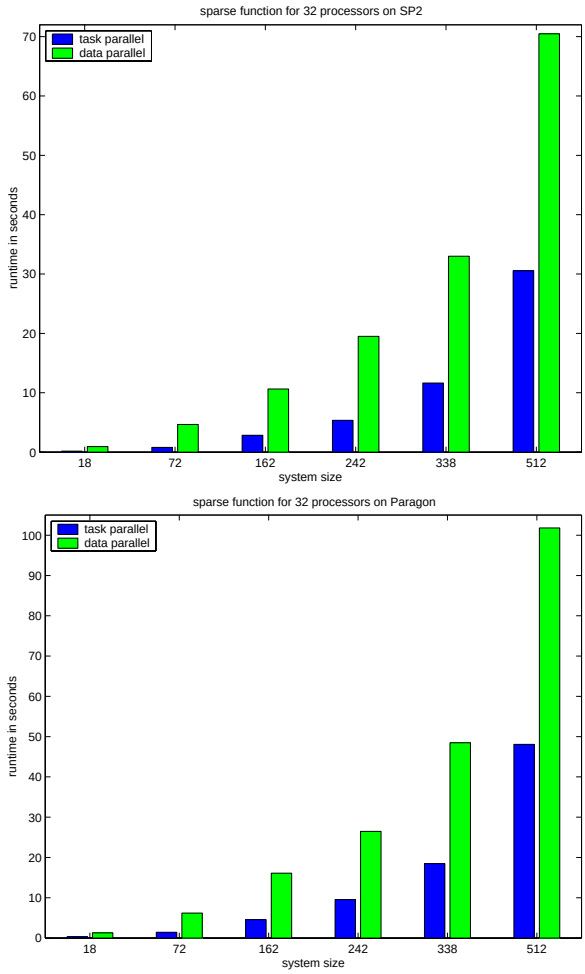

Figure 5. Runtimes of task-parallel and data-parallel execution schemes of the DIIRK method on the IBM SP2 (top) and the Intel Paragon (bottom).

and the other machines lies in the fact that the startup coefficient τ is quite small (7.7 μs) compared to the other machines (31.3 μs on the SP2 and 564 μs on the CLiC). Thus, a concurrent execution of the broadcast operations in a task-parallel scheme only saves a small fraction of the corresponding communication time, whereas the effect on the other machines is much larger and leads to a significant overall performance improvement of the execution time.

Extrapolation methods Extrapolation methods are explicit solution methods for ODEs with a possibly high convergence order [9, 17]. They are widely used and are especially suited if high precision is required. In each time step, they compute r different approximations for the same time step with different step sizes h_i, $i = 1, \ldots r$, which are combined to obtain an approximation solution of higher

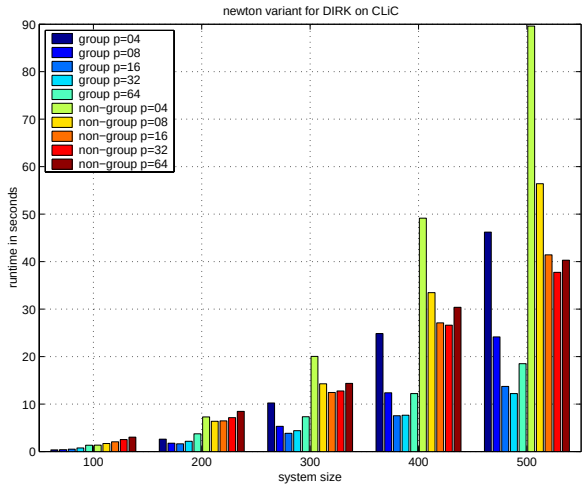

Figure 6. Runtimes of task-parallel and data-parallel execution schemes of the DIIRK method for dense ODE systems on the CLiC (Chemnitzer Linux Cluster).

order. Since these computations are independent of each other, extrapolation methods offer the potential for a mixed task and data parallel execution. In particular, the following task structures can be used:

1. a pure data parallel execution scheme which executes the generating method with the different stepsizes one after another with all processors available (*consecutive execution*);

2. a task parallel execution scheme using r disjoint processor groups such that the size of the groups is adapted to the computational work that they have to perform (*linear partitioning*);

3. a task parallel execution scheme with $\lceil r/2 \rceil$ disjoint groups where each group i contains the same number of processors and applies the generating method with two different step sizes h_i and h_{r-i} (r even) (*extended* partitioning).

Figure 8 shows the resulting runtimes in seconds on a Cray T3E for one time step of the extrapolation method with $r = 4$ different stepsizes. Similar results are obtained for $r = 8$. The top part of **Figure 8** shows the runtimes for $p = 4$ and $p = 8$ processors, the bottom part shows the runtimes for $p = 16$ and $p = 32$ processors. As example, we consider sparse ODE systems resulting from a discretization of reaction-diffusion equations with different grids [9]. For all numbers of processors, one of the task parallel execution schemes leads to the smallest execution time and

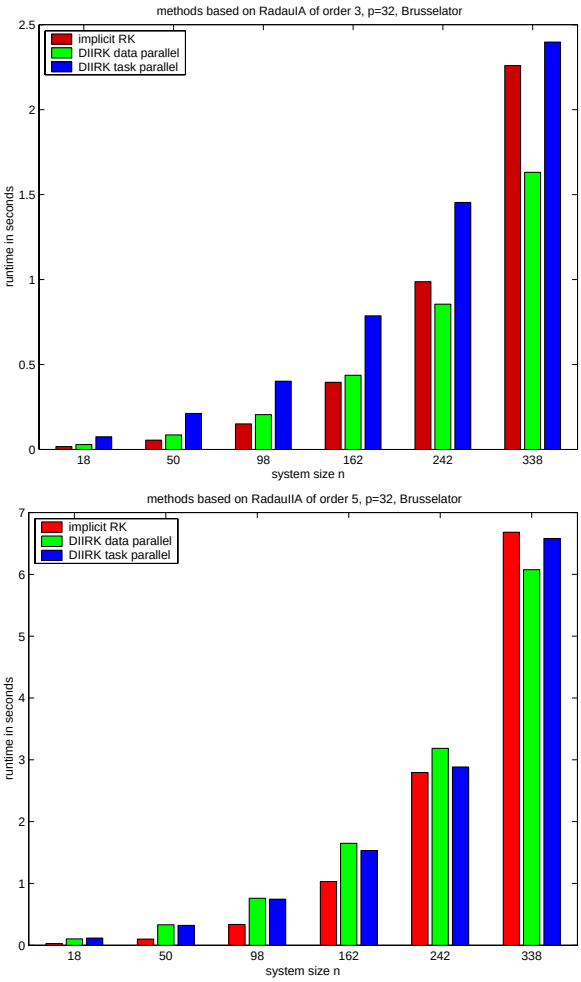

Figure 7. Runtimes of task-parallel and data-parallel execution schemes of the DIIRK method on 32 processors of an Cray T3E for a 3-stage RadauIA method (top) and a 5-stage RadauIIA method (bottom).

the percentage difference to the pure data-parallel execution time is increasing with the number of processors.

For both the extrapolation methods and the DIIRK methods, there is no noticeable difference between a hand-coded version and the implementation with the Tlib library.

6 Related work

The SPMD model proposed in [5] was from it's inception more general than a data parallel model and did allow a hierarchical expression of parallelism; however most implementations exploit only a data parallel form. Many research groups have proposed models for mixed task and data parallel executions with the goal to obtain parallel programs with faster execution time and better scalability properties, see

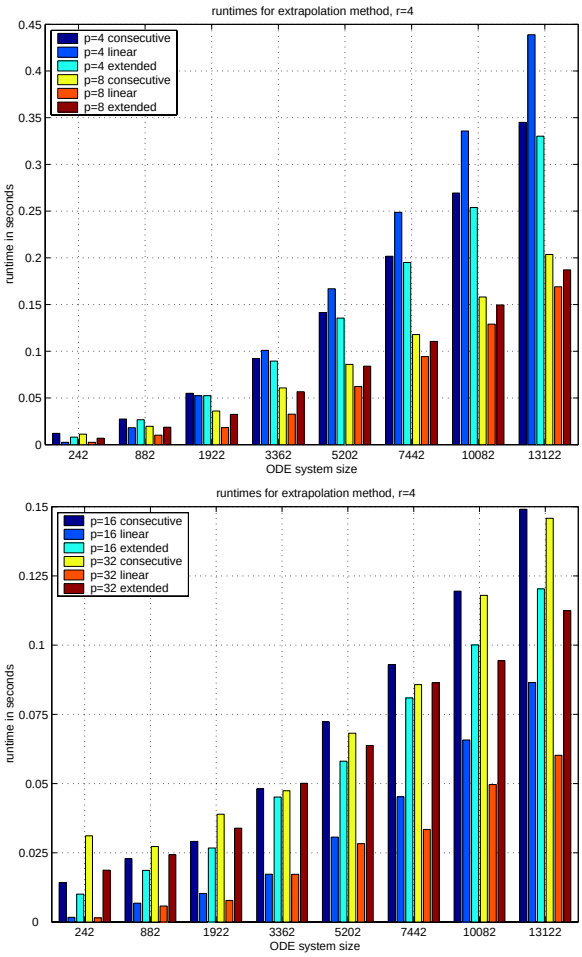

Figure 8. Runtimes of the different execution schemes of the extrapolation method with 4 different stepsizes for $p = 4$ and $p = 8$ processors (top) and for $p = 16$ and $p = 32$ processors (bottom) of a Cray T3E.

[2, 19] for an overview of systems and approaches and see [4] for a detailed investigation of the benefits of combining task and data parallel executions.

Most closely related to our work concerning the parallel programming model are approaches which combine multiprocessor task and data parallelism. Several models support the programmer in writing efficient programs without dealing too much with the underlying communication and coordination details of a specific parallel machine. Language approaches include Braid, Fortran M, Fx, Opus, and Orca, see [2] for an overview and comparison. Fortran M [8, 7] allows the creation of processes which can communicate with each other by predefined channels and which can be combined with HPF Fortran for a mixed task and data parallel execution. The organization of communication in Fortran M is different than within the Tlib library, since communication in the TLib library primarily takes part within a multiprocessor task, although communication between different multiprocessor tasks is possible for sibling groups by using an appropriate parent communicator. The Fx [20] approach enables task parallelism by the definition of *parallel sections*. The body of a parallel section may contain subroutine calls and loops with constant bounds. Subroutines can be executed in a data parallel way and their interface with cooperating subroutines is expressed by input and output directives. Subroutines without data dependencies can be executed by independent groups of processors. In contrast to the Tlib library, communication between tasks in the Fx model is prohibited except on entry and exit points whereas using the Tlib library, concurrent tasks can communicate using the communicator of the parent descriptor or any other descriptor that includes all processors executing the concurrent tasks. Moreover, Fortran M and Fx do not allow recursive specifications of task parallelism.

An exploitation of task and data parallelism in the context of a parallelizing compiler can be found in the Paradigm project [11, 15]. The Paradigm compiler provides a framework that expresses task parallelism by a macro data-flow graph which has been derived from the hierarchical task graphs used in the Parafrase compiler [3]. A major difference between the Paradigm and the Tlib approach is also that Paradigm expects a sequential program as input whereas Tlib starts with a specification of the available degree of task parallelism.

Many other environments for mixed parallelism in scientific computing are extensions to the HPF data parallel language, see [6] for an overview. An example is HPJava which adopts the data distribution concepts of HPF but uses a high level SPMD programming model with a fixed number of logical control threads and includes collective communication operations encapsulated in a communication library. A language description is given in [23]. The concept of processor groups is supported in the sense that global data to be distributed over one process group can be defined and that the program execution control can choose one of the process groups to be active. A library for the realization of orthogonal processor groups allowing the definition of different partitions that are orthogonal to each other is described in [16].

LPARX is a parallel programming system for the development of dynamic, nonuniform scientific computations supporting block-irregular data distributions [13]. KeLP extends LPARX to support the development of efficient programs for hierarchical parallel computers such as clusters of SMPs [1, 6]. In comparison to our approach, LPARX and KeLP are more directed towards the realization of irregular grid computations whereas our approach considers the hierarchical decomposition of processor groups for the realization of quite regular applications.

NestStep extends the BSP model [10] by supporting

group-oriented parallelism based on the nesting of supersteps and a hierarchical processor group concept [12]. The group concept of NestStep is similar to the group concept of Tlib, but the BSP-based programming model of NestStep is quite different from the Tlib programming model where no concept of supersteps consists. In particular, each communication operations takes effect as soon as the data arrives at the receiving processors.

7 Conclusions

To allow an easy and flexible exploitation of mixed task and data parallelism with hierarchically structured M-tasks, we provide a runtime library that enables the programmer to express arbitrarily nested task parallel structures without dealing with the complications of the underlying group and communicator management.

The easy-to-use interface facilitates the comparison of different task-parallel organizations of the same parallel program considerably, since it only requires to exchange the library functions called in the coordination functions. No changes in the data-parallel SPMD functions are necessary. Moreover, the library allows dynamic reorganizations of task and groups structures, e.g. to adapt load balancing, and the separation of group and task management allows the efficient reuse of group structures.

Since the communication library is based on MPI, it is easily portable and can be arbitrarily mixed with MPI calls. An advantage of the similarity with the Pthreads library is the possibility to use Pthreads functions instead of MPI functions as building blocks, e.g. for clusters of SMPs, thus obtaining a mixed programming model with Pthreads tasks within SMP nodes and a task organization with MPI communication between SMP nodes.

Acknowledgement

We thank the NIC Jülich for providing access to the Cray T3E.

References

[1] S.B. Baden and S.J. Fink. A Programming Methodology for Dual-Tier Multicomputers. *IEEE Transactions on Software Engineering*, 26(3):212–226, 2000.

[2] H. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3):74–84, July-August 1998.

[3] C.J. Beckmann and C. Polychronopoulos. Microarchitecture Support for Dynamic Scheduling of Acyclic Task Graphs. Technical Report CSRD Report 1207, University of Illinois, 1992.

[4] S. Chakrabarti, J. Demmel, and Yelick K. Modeling the benefits of mixed data and task parallelism. In *Symposium on Parallel Algorithms and Architecture (SPAA)*, pages 74–83, 1995.

[5] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational mode for EPEX/FORTRAN. *Parallel Comput.*, 7(1):11–24, 1988.

[6] S.J. Fink. *A Programming Model for Block-Structured Scientific Calculations on SMP Clusters*. PhD thesis, University of California, San Diego, 1998.

[7] I. Foster and K.M. Chandy. Fortran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 25(1):24–35, April 1995.

[8] I. Foster, M. Xu, B. Avalani, and A. Choudhary. A Compilation System That Integrates High Performance Fortran and Fortran M. In *Proceedings 1994 Scalable High Performance Computing Conference*, pages 293–300. IEEE Computer Society Press, 1994.

[9] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer–Verlag, Berlin, 1993.

[10] M. Hill, W. McColl, and D. Skillicorn. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[11] P. Joisha and P. Banerjee. PARADIGM (version 2.0): A New HPF Compilation System. In *Proc. 1999 International Parallel Processing Symposium (IPPS'99)*, 1999.

[12] C.W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. *The Journal of Supercomputing*, 17:245–262, 2001.

[13] S.R. Kohn and S.B. Baden. Irregular Coarse-Grain Data Parallelism under LPARX. *Scientific Programming*, 5:185–201, 1995.

[14] M. Kühnemann, T. Rauber, and G. Rünger. Performance Modelling for Task-Parallel Programs. In *Proc. of the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002)*, pages 148–154, San Antonio, USA, 2002.

[15] S. Ramaswamy. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[16] T. Rauber, R. Reilein, and G. Rünger. ORT – A Communication Library for Orthogonal Processor Groups. In *Proc. of the Supercomputing 2001*, Denver, USA, 2001. IEEE Press.

[17] T. Rauber and G. Rünger. Load Balancing Schemes for Extrapolation Methods. *Concurrency: Practice and Experience*, 9(3):181–202, 1997.

[18] T. Rauber and G. Rünger. Diagonal-Implicitly Iterated Runge-Kutta Methods on Distributed Memory Machines. *Int. Journal of High Speed Computing*, 10(2):185–207, 1999.

[19] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[20] J. Subhlok and B. Yang. A New Model for Integrating Nested Task and Data Parallel Programming. In *8th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, pages 1–12, 1997.

[21] P.J. van der Houwen, B.P. Sommeijer, and W. Couzy. Embedded Diagonally Implicit Runge–Kutta Algorithms on Parallel Computers. *Mathematics of Computation*, 58(197):135–159, January 1992.

[22] Z. Xu and K. Hwang. Early Prediction of MPP Performance: SP2, T3D and Paragon Experiences. *Parallel Computing*, 22:917–942, 1996.

[23] G. Zhang, B. Carpenter, G.Fox, X. Li, and Y. Wen. A high level SPMD programming model: HPspmd and its Java language binding. Technical report, NPAC at Syracuse University, 1998.