ORT – A Communication Library for Orthogonal Processor Groups

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

ROBERT REILEIN GUDULA RÜNGER
Fakultät für Informatik
Technische Universität Chemnitz
09107 Chemnitz, Germany
{reilein,ruenger}@informatik.tu–chemnitz.de

Abstract

Many implementations on message-passing machines can benefit from an exploitation of mixed task and data parallelism. A suitable parallel programming model is a group-SPMD model, which requires a structuring of the processors into subsets and a partition of the program into multi-processor tasks. In this paper, we introduce a library support for the specification of message-passing programs in a group-SPMD style allowing different partitions in a single program. We describe the functionality and the implementation of the library functions and illustrate the library programming style with example programs. The examples show that the runtime on distributed memory machines can be considerably reduced by using the library.

Keywords: communication library, mixed task and data parallelism, group-SPMD.

1 Introduction

The performance of message-passing programs depends on both the parallel target machine and the parallel programming model to be used. To get efficient programs the parallel programming model should be selected such that the characteristics of the parallel algorithm can be expressed appropriately. For target machines with a large number of processors, scalability properties of the parallel program also become an important issue. Pure data parallel implementations for example may lead to scalability problems when collective communication operations are used extensively in the program. Those programs can benefit from an implementation in a group-SPMD parallel programming model if a potential for mixed task and data parallelism is available.

SC2001 November 2001, Denver © 2001 ACM 1-58113-293-X/01/0011 $5.00

Many programs from scientific computing have a large potential of parallelism that is exploited best in such a programming model for mixed task and data parallelism where the parallelism can be structured in the form of concurrent multi-processor tasks[17]. A multi-processor task can be implemented on a subset of processors in a data parallel or SPMD style resulting in a group-SPMD programming model when performing several multi-processor tasks at the same time on disjoint subsets of processors. One of the advantages to consider subgroups of processors is based on the fact that for many message-passing machines communication costs are affected by the number of participating processors. A reason is that collective communication operations in a distributed memory environment have a smaller runtime when realized on a smaller processor group due to the logarithmic or linear dependence of the communication times on the number of participating processors. Consequently, for the sake of a low parallel runtime collective communication operations should be realized concurrently on disjoint processor subgroups whenever this is possible according to the potential parallelism provided by the algorithm. In scientific computing there are many applications with this property including, e.g., solvers for systems of ordinary differential equations or multidisciplinary applications.

Usually, the execution of a mixed task and data parallel program requires only one partition of the processor set. But applications can also benefit from the definition of multiple partitions that are defined orthogonal to each other which means the subsets of processors of different group partitions have some kind of orthogonal relationship to each other. Writing group-SPMD programs using one of the standard libraries like MPI or PVM requires the explicit administration of processor groups which can be an intricate task, especially when multiple processor groups are used within a single program. The use of multiple processor groups means that at different points of the program execution the implementation can be based on different decompositions of the entire processor set into a partition of processor subsets so that the assignment of processors to

groups can vary during the execution. To achieve efficiency the organization into groups should be predefined for the entire program in order to have a low overhead for building groups and switching between groups during runtime.

In this paper, we present a library support for the parallel programming in the group-SPMD model with orthogonal processor groups. This library is called ORT library. The advantage of a library support is to have a comfortable specification mechanism for group-SPMD programs and an executable implementation at the same time. We consider parallel programs consisting of a number of consecutive group-SPMD phases where each phase uses one of the predefined partitions. To indicate different program parts we provide library functions that structure the program. The execution of the library functions uses predefined processor group structures which are initialized by some starting routines. This programming style allows the application programmer to specify the program organization in a clear and readable program code. We have implemented the library for the use of orthogonal processor structures on distributed memory machines on top of the communication library MPI so that the specification program is executable on message-passing machines. The application programmer initializes the underlying group structures by calling the starting routine but does not have to cope with the processor organization so that he or she can concentrate on the most efficient realization of the program. Our target machines are a Cray T3E and a cluster of workstations. As example programs we use the well-known LU-decomposition and a specific solver for systems of ordinary differential equations with mixed task and data parallelism. Both examples are realized with the library. Measurements show that the runtime of the application program decreases when exploiting the potential of mixed task and data parallelism with the specific organization of orthogonal processor groups.

The rest of the paper is organized as follows. Section 2 describes the library support. Section 3 illustrates the programming style with example programs for which we present runtime experiments in Section 4. Related work is discussed in Section 5 and Section 6 concludes the paper.

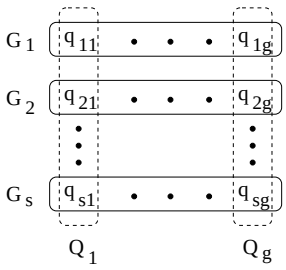

Figure 1. Illustration of an orthogonal group structure for the two-dimensional case.

2 The ORT Library

One of the main goals of the ORT programming library is the user-friendly management of orthogonal processor groups. The interface is intended to give the user the possibility to select processor groups for distinct program tasks and to organize group communication operations in an effective and comfortable way.

2.1 General approach and programming support

The ORT library supports the development of message-passing programs for parallel machines with a distributed address space. The processors are logically arranged in a grid structure and the programmer can assign computations to orthogonal slices of the processor grid.

The ORT library provides functions to support a group-SPMD parallel programming model that allows a mixed task and data parallel implementation of parallel algorithms. The ORT functions are used in a parallel program to structure the program into phases which are to be performed in a group-SPMD style with changing group partitions of the processors. Program parts outside orthogonal sections are executed in the usual SPMD style.

The concept is best illustrated for the two-dimensional case. The set of processors P is grouped in a two-dimensional grid. The rows of this grid form a partition $G_1, \ldots, G_s$ of the processors into subgroups. The columns of the grid form another partition $Q_1, \ldots, Q_g$ which is orthogonal to the first partition, see Figure 1 for an illustration. The idea of the ORT library is to allow the assignment of computations to the subgroups of the different partitions so that the subgroups of one partition work in a task parallel way. The flexibility of this programming approach comes from the fact that the different partitions may be active at different times of the program execution, but in one specific phase only one partition is active. The processors within one group of a partition perform the same computations and can communicate with each other, i.e., perform an SPMD computation. However, the processors of different groups of the same partition are not intended to communicate in this phase (although this is possible). If communication is required for the specific application, this should be done in a subsequent program phase with another group partition in which the communicating processors are members of the same group.

To support the programming with orthogonal processor structures we provide several library functions which are implemented on top of the MPI communication library. We use C as host language. The interface has been designed similarly to the Pthreads-interface, i.e., the function to be executed on the selected processor group is given as an argument to a specific library function. The arguments of that

function are provided as an additional parameter of type `void *`. 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. An advantage of the similarity to the Pthreads style is that the handling of the library is done in a familiar style although the programming model is completely different.

Besides the coded argument vector, the functions that are executed in a task parallel way using the ORT library must have an additional parameter that specifies an MPI communicator. For the execution of the function, this communicator is provided by the runtime system of the library according to the specification of the programmer. Group-internal communication during the execution of such a function is obtained by using this communicator for the execution of the communication operation. Global communication operations can still be executed by using other communicators like `MPI_COMM_WORLD`.

2.2 Interface of the ORT library

The user interface of the ORT library is defined by several library functions for the definition of processor partitions and the structuring of group-SPMD sections of the program. The ORT library functions are called in the parallel program and serve two purposes: First, they can be used to specify the parallel structures of the group-SPMD programs in a clear way. Second, the resulting program is executable on all platforms on which MPI is available.

The prototypes of the library functions are defined in `ortho.h` which must be included in the program.

To initialize the orthogonal processor structure, the function `ORT_Init()` is provided.

```
int ORT_Init( int dimension,
              int *proc_dim,
              MPI_Comm comm,
              ORT_Structure *ort)
```

This function has to be executed before any other function of the library is called. The function establishes a data structure which contains the MPI communicators for all orthogonal processor groups in which the calling processor is involved. The parameters have the following meaning:

- `dimension` specifies the number of dimensions of the processor structure;
- `proc_dim` is an array of length `dimension` that specifies the number of processors in each of the dimensions. This number must be at least 1. Moreover, the product of the entries of the array must be the same or smaller than the overall number of processors provided;
- `comm` is the global MPI communicator provided for the processors; this is usually `MPI_COMM_WORLD`, but can also be another communicator;
- `ort` is a pointer to an object of type `ORT_Structure` which is filled by the call to `ORT_Init()` and is used by later calls of ORT functions.

A call of the function

```
ORT_Finalize(ORT_Structure ort);
```

frees the `ORT_Structure`. This function should be called before the MPI environment is finalized. No other ORT function should be called after `ORT_Finalize()`.

To execute tasks on subgroups of the orthogonal processor structure, the ORT function `ORT_Section()` is provided which expects a pointer to the function to be executed as argument and has additional parameters for the selection of the substructure:

```
int ORT_Section(
        int dimension,
        int *dim_select,
        void * (*f)(void *, MPI_Comm comm),
        void *arg,
        ORT_Structure ort)
```

The parameters have the following meaning:

- `dimension` specifies the number of dimensions of the processor structure.
- `dim_select` is an array of length `dimension` that specifies the dimension of the selected sub-structure by specifying for each dimension how it is represented in the sub-structure; the following values are allowed:
  - `ORT_ALL`: the corresponding dimension is contained in the sub-structure;
  - `ORT_PARALLEL`: the corresponding dimension is not contained in the sub-structure, all processor groups in the corresponding dimension of the processor grid work in parallel;
  - `k` where k is a value between 0 and the size of the processor structure in this dimension minus 1: one specific position is selected in this dimension, the remaining entries are not used for the computation;
- `f` is the function to be executed by the processors of the selected sub-structure; the function should expect two arguments, one of type `void *` to which arbitrary arguments may be mapped, and one communicator for the execution of the internal communications within `f`. Collective communication and reduction operations that are performed during the execution of

f with this communicator perform group-based communication operations. At runtime, this communicator is automatically selected by the implementation of ORT_Section.

- arg contains the parameters for the function call.
- ort is an object of type ORT_Structure that has been previously returned by ORT_Section().

Since two-dimensional virtual processor grids are often used, several functions are provided for convenience, see Figure 2 for an illustration. The function

```
int ORT_Vertical_section(
        int k,
        void * (*f) (void *, MPI_Comm comm),
        void *arg,
        ORT_Structure ort);
```

is provided as an abbreviation for the activation of the general function ORT_Section(2,dim_select,f,arg,ort) with argument value dim_select={k,ORT_ALL}. Each processor in the vertical group k executes the specified function f in an SPMD-like way together with all other processors in the same group. Only the processors in this group are active during the execution of the function. The function

```
int ORT_Vertall_section(
        void * (*f) (void *, MPI_Comm comm),
        void *arg,
        ORT_Structure ort);
```

is an abbreviation for the activation of the general function ORT_Section(2,dim_select,f,arg,ort) with argument value dim_select={ORT_PARALLEL,ORT_ALL}. Each processor executes the specified function in an SPMD-like way together with the other processors in the same column group. A processor in column group k may perform computations as well as collective communication and reduction operations involving the processors in the same column. This is done concurrently to the other vertical processor groups. Thus, ORT_Vertall_section() corresponds to a parallel loop over all column groups k where each iteration executes ORT_Vertical_section(f,arg,k,ort). The function

```
int ORT_Horizontal_section(
        int k,
        void * (*f) (void *, MPI_Comm comm),
        void *arg,
        ORT_Structure ort);
```

is an abbreviation for the activation of the general function ORT_Section(2,dim_select,f,arg,ort) with argument value dim_select={ORT_ALL,k}, i.e., ORT_Horizontal_section() is similar to ORT_Vertical_section() but uses the horizontal group structure. The function

```
int ORT_Horall_section(
        void * (*f) (void *, MPI_Comm comm),
        void *arg,
        ORT_Structure ort);
```

is an abbreviation for the activation of the general function ORT_Section(2,dim_select,f,arg,ort) with argument value dim_select={ORT_ALL,ORT_PARALLEL}, i.e., ORT_Horall_section() is similar to ORT_Vertall_section() but uses the horizontal group structure.

Several functions are provided to allow the processors to obtain information about active or other substructures during the execution of a function within an orthogonal section. For example, there is a function to give each processor information on its position in a specific substructure and on its membership to a specific orthogonal group:

```
int ORT_Group_ranks(int ndims,
                    int *dim_select,
                    int *ranks,
                    ORT_Structure ort);
```

The array dim_select of size ndims is used to specify a subgroup structure that has been previously built by ORT_Init(). The selection is made as described for ORT_Section(). The function ORT_Group_ranks() can be used for processor grids of arbitrary dimension. It returns the array ranks of size ndims with the specific positions in the remaining dimensions for the orthogonal processor group.

2.3 Implementation issues

The data structure ORT_Structure is the core of the ORT library implementation. The implementation is hidden within the library implementation and consists of several components which cannot be accessed directly by the programmer. The main component is an array to store the MPI communicators for each orthogonal processor group. The communicators which are either constructed at beginning or at runtime are ordered using a binary number representation of the corresponding orthogonal processor group. To obtain the position in that array each dimension is represented as one bit. If a dimension is included in a group the corresponding bit will be set to 1. This allows a simple and quick translation of the dim_select array contents used by ORT_Section into the right array position. The communicators are constructed using the MPI library functions MPI_Cart_create() and MPI_Cart_sub(). An internal ORT library parameter defines the dimension limit up to which the communicators are constructed at the initialization. Currently this limit is set to 4. All array positions are initialized with MPI_COMM_NULL to test at runtime if a communicator is already constructed.

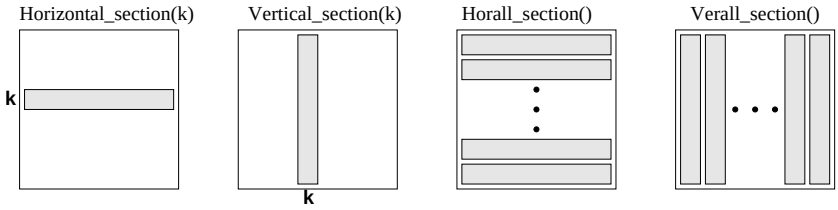

Figure 2. Illustration of two-dimensional orthogonal structures. The gray parts depict active computation parts in the group processor structure.

Three steps are needed to determine the orthogonal processor groups which execute the user function of an `ORT_Section()` call. First the `dim_select` array is used to compute the position in the communicator array. If the i-th element of the selection array is equal to `ORT_ALL` the corresponding bit in the position pointer is set to 1. The position located is used to access the communicator which have to be provided for the user function as parameter. The second step is to check whether or not the communicator for the selected orthogonal processor group already exists. If not, the contents of the selection array are used to construct the corresponding communicator. Finally the executing processor determines its membership to one of the selected orthogonal processor groups. This is done by iterating over all dimensions of the processor grid and testing if either a dimension is included in all selected orthogonal processor groups (`ORT_ALL`) or each processor of that dimension is included in one selected group (`ORT_PARALLEL`) or the given index k selects the executing processor in that dimension. If at least one condition for an arbitrary dimension is not fulfilled the executing processor will not take part in the execution of the user function with the previously located communicator as parameter.

3 Example Application

We demonstrate the use of the ORT library by two example applications, the well-known LU decomposition for solving linear systems of equations and a specific solver for systems of ordinary differential equations. For both examples, we show the main structure of an ORT implementation.

LU decomposition We consider a parallel LU decomposition with a double-cyclic data distribution. A specification within the library is given in Figure 3 (the variable `p1` specifies the number of processors in the rows of the virtual processor grid).
`determine_pivot_lnr()` computes the largest value of the local elements of matrix column k. The global maximum of these values and the corresponding line number are computed by an `MPI_Allreduce` call inside the column group which owns k.
`distribute_pivot_lnr()` distributes the pivot line number within the row groups, thus making the index of the pivot row available to each processor.
`exchange_pivot_line()` exchanges the local elements of the pivot row r and the current row k. If the rows r and k are stored in the same row group, only local exchanges are necessary. If not, the processors of the two row groups involved exchange their local elements of rows r and k with the corresponding processor in the other row group.
`broadcast_pivot_line()` broadcasts the pivot row, thus making it available to all processors for the following elimination step.
`determine_L_factors()` computes the elimination factors for column k by the processors in the corresponding column group. Each processor that has a part of column k participates in the operation.
`distribute_L_factors()` broadcasts the elimination factors computed in the row groups.
`update_U_factors()` performs the actual elimination step of the remaining matrix elements using the elimination factors computed. This function contains no communication.

Iterated Runge-Kutta method Our second example is an explicit iterated Runge-Kutta (RK) method for the solution of systems of ordinary differential equations (ODEs) [15, 16]. The solution method performs consecutive time steps to compute approximation vectors of the solution at discrete points of the independent variable, usually denoted as time variable. In each of the time steps the RK method computes a fixed number of stage vectors by fixed point iteration. The time loop and the fixed point loop have to be computed consecutively. The number of fixed point iterations is selected according to the order of the underlying implicit RK method so that a guaranteed convergence order results. Each step of the fixed point iteration offers mixed task and data parallelism as the different stage vectors can be computed concurrently to each other and the computation of each stage vector can be distributed among the processors of one group. Thus, the computation of the

```
typedef struct {
      ORT_Structure ort;
      [...]         /* list of variables */
      } LU_VARIABLES;

int gauss(void *arg) {
   LU_VARIABLES *luv=(LU_VARIABLES *)arg;
   ORT_Structure ort=luv->ort;
   [...]          /* initialization code */
   for(k=0;k<n-1;k++) {
      /* column group determines pivot element */
      ORT_Vertical_section(k%p1, (void *)determine_pivot_lnr, (void *)luv, ort);
      /* broadcast pivot line nr to row groups */
      ORT_Horall_section( (void *)distribute_pivot_lnr, (void *)luv, ort);
      /* local or global exchange pivot line */
      ORT_Vertall_section( (void *)exchange_pivot_line, (void *)luv, ort);
      /* broadcast pivot line */
      ORT_Vertall_section( (void *)broadcast_pivot_line, (void *)luv, ort);
      /* column group determines L factors */
      ORT_Vertical_section(k%p1, (void *)determine_L_factors, (void *)luv,ort);
      /* broadcast L factors to row groups */
      ORT_Horall_section( (void *)distribute_L_factors, (void *)luv, ort);
      /* update U factors */
      update_U_factors(luv);
}  }
```

Figure 3. ORT-program of a group-parallel LU factorization with group operations of the ORT library interface.

stage vectors can be realized using a partition of the processors into subsets of equal size so that one stage vector is uniquely assigned to one group of the partition. The computations of stage vectors are not completely independent but require some regular data exchange after each fixed point step, where every processor of each group holding a part of a vector needs only the corresponding vector elements of other vectors that have been computed by the other processor groups. This structure suggests to use additional orthogonal processor groups reducing the communication to concurrent group communications. The code structure for `m` fixed point steps in each time step is given in the following code fragment:

```
do {
   for (j=0 ; j<m ; j++) {
      ORT_Horall_section( compute vectors );
      ORT_Vertall_section( data exchange );
   }
   compute approximation
} while ( error condition )
```

The iterated RK method can be used to solve arbitrary time-dependent partial differential equations (PDEs) by using the methods of lines, which leads to a system of ODEs in the time domain with an equation for each spatial discretization point.

4 Runtime Experiments

The ORT library and the example applications are implemented on two message-passing machines, a Cray T3E[1] and a cluster of 528 Linux PCs (CLiC - Chemnitz Linux Cluster) [2] connected with switch-coupled 100MBit Ethernet. Figure 4 compares the resulting speedup values of the LU decomposition using a column-cyclic and a row-cyclic distribution of the matrix entries with an implementation using the ORT library. The figure shows that on the Cray T3E, the row-cyclic distribution leads to larger speedups than the column-cyclic distribution and the implementation with the ORT library leads to even larger speedups due to the reduced internal communication overhead.

Figure 5 compares the same versions on the CLiC. The results are similar, but the speedups of the row-cyclic and the column-cyclic implementations have changed their roles. The overall speedup values are smaller than the speedups on the Cray T3E because the T3E has an interconnection network with a much larger bandwidth so that communication operations are executed faster than on the CLiC. Table 1 shows the percentage of the runtime of the different phases in the different versions for 64 processors.

[1] at the NIC supercomputing center at Jülich, Germany
[2] Number 156 on the TOP500 list (June 2001)

function name	orth. proc. groups		row-cyclic distr.		col.-cyclic distr.	
	CLiC	T3E	CLiC	T3E	CLiC	T3E
`determine_pivot_lnr()`	1.36 %	1.10 %	14.03 %	6.47 %	0.07 %	0.26 %
`distribute_pivot_lnr()`	12.42 %	7.81 %			8.29 %	13.73 %
`exchange_pivot_line()`	1.69 %	0.27 %	0.62 %	0.11 %	0.05 %	0.07 %
`broadcast_pivot_line()`	9.16 %	1.90 %	45.71 %	11.08 %		
`determine_L_factors()`	0.11 %	0.56 %	0.09 %	0.42 %	0.06 %	0.27 %
`distribute_L_factors()`	1.47 %	4.59 %			34.17 %	19.05 %
`update_U_factors()`	73.79 %	83.77 %	39.55 %	81.92 %	57.36 %	66.62 %

Table 1. Percentage of the runtime of the orthogonal phases in the LU decomposition.

Figure 6 shows speedup values for different parallel versions of an iterated RK method [21] on a Cray T3E. As basic RK method, a Lobatto IIIC6 method with four stages has been used which results in a method of convergence order 6. Each time step performs five iterations. The method has been used for the solution of the Brusselator equation, a time-dependent 2D reaction-diffusion equation using the method of lines [7]. Performing a spatial discretization with a uniform grid with n grid points in each dimension leads to an ODE system of size $2n^2$. The figure shows the results for a 32×32 grid. The resulting right hand side function F of the ODE system is a sparse function, i.e., the evaluation of each component of F depends only on a fixed number of components of the argument vector, thus leading to a linear dependence of the total evaluation time of F on the size of the ODE system. Figure 6 compares a pure data parallel realization of the iterated RK method, a general task parallel implementation which performs the computations of the approximations of the stage vectors concurrently by different groups of processors, and a task parallel implementation which is optimized by using orthogonal processor groups. For this example, the pure data parallel implementation is much faster than the mixed task and data parallel implementation. This is also the case for the implementation with the orthogonal groups for a small number of processors. But the scalability properties are improved by using orthogonal processor groups and for more than 32 processors, this implementation is getting better than the data parallel implementation. For 64 processors, it is twice as fast as the data parallel implementation. Figure 7 shows the results for an iterated RK method that is based on a Radau IIA method with four stages to the Brusselator equation on a T3E. The results are similar than for the Lobatto IIIC6 based method, but the implementation with orthogonal processor groups is now competitive also for a small number of processors.

The overall speedup values of the iterated RK method are not very large when applied to sparse ODE systems. On the CLiC, for sparse ODE systems no speedup is reached since the computational effort is small compared to the communication overhead. This changes, when applying iterated RK methods to dense ODE systems, which arise, e.g., from Galerkin solutions of PDEs. A dense ODE system is characterized by the fact that the evaluation time of each component of F depends on all or nearly all components of its argument vector, i.e., the evaluation time of the entire function F depends quadratically on the size of the ODE system. Applying an iterated RK method to a dense ODE system leads to nearly ideal speedup values on the T3E for both a pure data parallel realization and a group-based implementation with orthogonal processor groups.

Figures 8 and 9 show that on the CLiC, speedup values can be reached for dense ODE systems. In particular, it can be seen that the implementation with orthogonal processor groups reaches speedup values that are quite impressive compared to the speedup values of a pure data parallel realization. The large difference is caused by the fact that the execution time of collective communication operations like broadcast operations increases significantly with the number of executing processors, so that the use of smaller processor groups is especially valuable.

5 Comparison with Related Work

Related work comes from different research directions, including programming models and software support for scientific computing, parallel languages and libraries, and mixed task and data parallelism [2, 18].

Several models have been proposed to support the programmer in writing efficient programs without dealing too much with the underlying communication and coordination details of a specific parallel machine, see [18] for a good overview. A lot of research has been invested in the development of the BSP (bulk synchronous parallelism) model and there exists a programming library (Oxford BSP library) that allows the formulation of BSP programs in an SPMD style [8, 11].

Many environments for scientific computing are extensions to the HPF data parallel language. A good overview can be found in [4]. 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 logi-

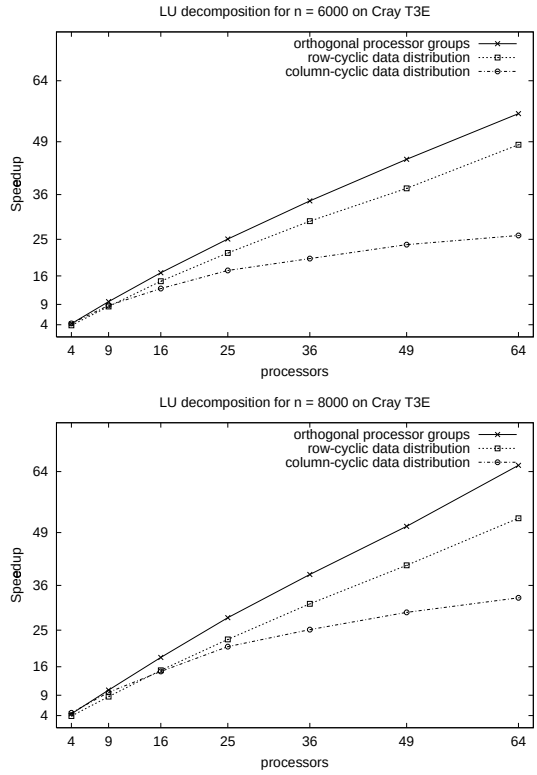

Figure 4. Speedup values for LU decomposition on Cray T3E.

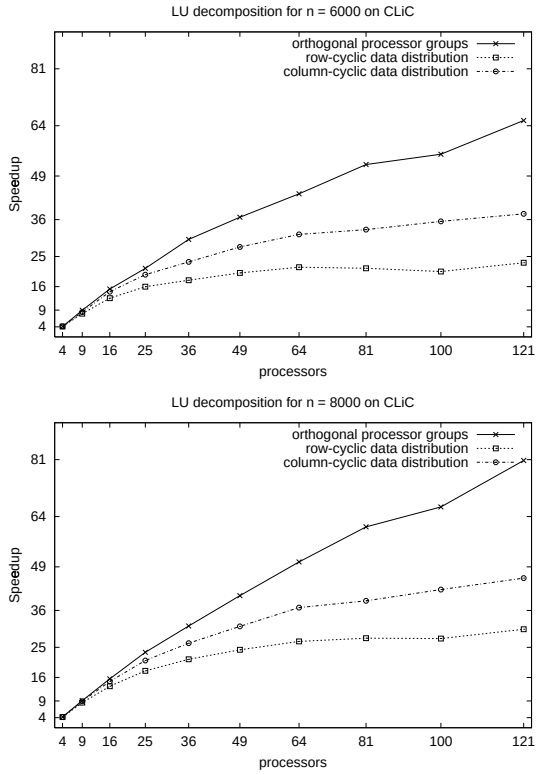

Figure 5. Speedup values for LU decomposition on CLiC.

cal control threads and includes collective communication operations encapsulated in a communication library. A language description is given in [22]. The concept of processor groups is supported in the sense that global data distributed over one process group can be defined and that the program execution control can choose one of the process groups to be active. In contrast, our approach provides processor groups which can work simultaneously and, thus, can exploit the potential parallelism of the application and the machine resources allocated more efficiently. Hence, orthogonal processor groups seem to provide the right level for applications with medium or fine-grained potential parallelism.

LPARX is a parallel programming system for the development of dynamic, nonuniform scientific computations supporting block-irregular data distributions [10]. KeLP extends LPARX to support the development of efficient programs for hierarchical parallel computers such as clusters of SMPs [1, 4]. In comparison to our approach, LPARX and KeLP are more directed towards the realization of irregular grid computations whereas our approach considers the mapping of regular task grids onto different partitions of the same set of processors. KeLP has been extended to KeLP-HPF which uses an SPMD program to coordinate multiple HPF tasks and, thus, combines regular data parallel computations in HPF with a coordination layer for irregular block-structured features on one grid [12].

Other approaches have been proposed to combine task and data parallelism. Language approaches include Braid, Fortran M, Fx, Opus, and Orca, see [2] for a good overview and comparison. Fortran M [6, 5] 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 Fx [19, 20] approach allows the expression of task parallelism by the definition of *parallel sections* which may contain subroutine calls and loops with constant bounds in there body. 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. This is expressed by grouping subroutine calls to modules if they are executed by the same processor group. An exploitation of task and data parallelism in the context of a parallelizing compiler can be found in the Paradigm compiler [9, 13, 14]. The Paradigm compiler provides a framework that expresses task parallelism by a

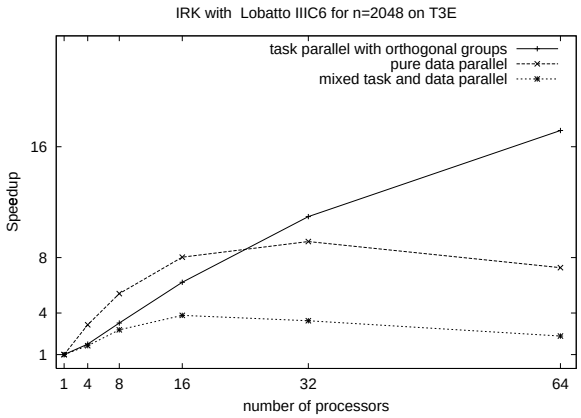

Figure 6. Speedup values for different parallel versions of the Lobatto IIIC6 based iterated RK method applied to a sparse ODE system on a Cray T3E-1200.

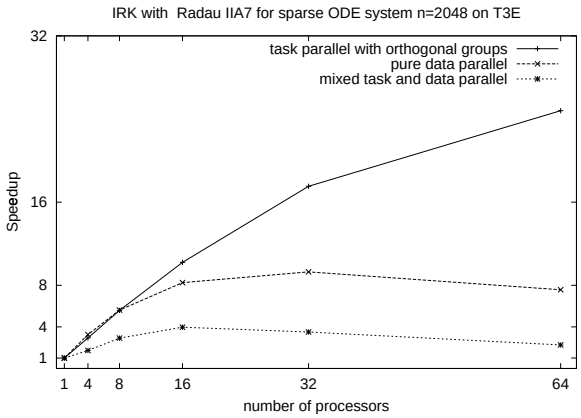

Figure 7. Speedup values for different parallel versions of the Radau IIA7 based iterated RK method applied to a sparse ODE system on a Cray T3E-1200.

macro data-flow graph which has been derived from the hierarchical task graphs used in the Parafrase compiler [3]. Nodes in the macro data-flow graph correspond to basic parallel tasks or loop constructs, edges correspond to precedence constraints that exist between tasks. The nodes and edges are weighted with processing and data transfer costs both of which depend on the number of processors used for the execution. [13] describes scheduling and allocation algorithms for macro data-flow graphs where allocation decides on the number of processors to use for each node and scheduling decides on a scheme of execution for the allocated nodes. None of these approaches provides an automated support for the creation of processor groups that can be used in an alternating or changing way without explicit coding by the application programmer.

6 Conclusions and Future Work

Writing efficient message-passing programs for a larger number of processors is often difficult since the influence of the communication overhead increases with an increasing number of processors, especially if collective communication operations are involved. In this article, we have proposed a library on top of MPI to support the application programmer in writing efficient parallel programs without dealing with all details of the processor management. Using the library releases the programmer from organizing the processors into orthogonal structures and substructures and from generating and managing the corresponding communication operations. Instead, the application programmer can concentrate on the parallel algorithm design.

We have shown that an exploitation of orthogonal structures can be useful for application programs like LU factorization or an iterated Runge-Kutta method. In both cases, the runtime of the resulting program is considerably reduced compared to the implementations that do not exploit an orthogonal structure, but already have shown reasonable speedups. It can be observed that orthogonal processor groups improve the scalability behavior.

Another example for the use of orthogonal structures are implementations of implicit Runge-Kutta methods for stiff ODE equations which require the solution of nonlinear equation systems in each time step. For diagonally-implicit iterated RK methods, each time step requires the solution of independent non-linear equation systems exhibiting a similar structure as the iterated RK method considered in the paper with the difference that instead of evaluating functions, nonlinear equation systems have to be solved. Using a Newton method with internal LU factorization leads to an orthogonal computation structure with an outer and an inner level requiring an organization of the processors in a three-dimensional grid.

Acknowledgement

We thank the NIC Jülich for providing access to the Cray T3E and Matthias Kühnemann for performing measurements of the iterated RK method on the T3E and the CLiC.

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.

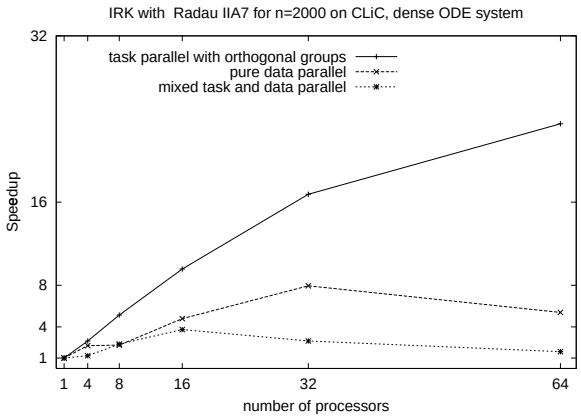

Figure 8. Speedup values for different parallel versions of the Radau IIA7 based iterated RK method applied to a dense ODE system of size $n = 2000$ on the CLiC.

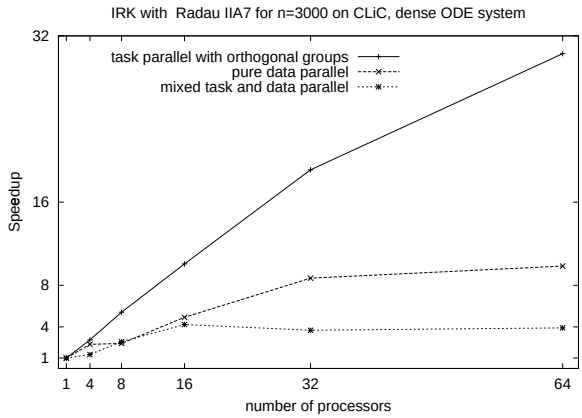

Figure 9. Speedup values for different parallel versions of the Radau IIA7 based iterated RK method applied to a dense ODE system of size $n = 3000$ on the CLiC.

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

[5] 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.

[6] 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.

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

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

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

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

[11] W.F. McColl. Universal Computing. In *Proceedings of the EuroPar'96*, Springer LNCS 1123, pages 25–36, 1996.

[12] J. Merlin, S.Baden, St. Fink, and B. Chapman. Multiple data parallelsim with HPF and KeLP. *J. Future Generation Computer Science*, 15(3):393–405, 1999.

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

[14] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed-Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, 1997.

[15] T. Rauber and G. Rünger. Parallel Iterated Runge–Kutta Methods and Applications. *International Journal of Supercomputer Applications*, 10(1):62–90, 1996.

[16] T. Rauber and G. Rünger. Parallel Execution of Embedded and Iterated Runge–Kutta Methods. *Concurrency: Practice and Experience*, 11(7):367–385, 1999.

[17] T. Rauber and G. Rünger. A Transformation Approach to Derive Efficient Parallel Implementations. *IEEE Transactions on Software Engineering*, 26(4):315–339, 2000.

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

[19] J. Subhlok. Automatic Mapping of task and data Parallel programs for Efficient Execution on Multiprocessors. Technical Report CMU-CS-93-212, Carnegie Mellon University, 1993.

[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 and B.P. Sommeijer. Parallel Iteration of high–order Runge–Kutta Methods with stepsize control. *Journal of Computational and Applied Mathematics*, 29:111–127, 1990.

[22] 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.