# EXPLOITING MULTIPLE LEVELS OF PARALLELISM IN SCIENTIFIC COMPUTING

Thomas Rauber
*Computer Science Department*
*University Bayreuth, Germany*
rauber@uni–bayreuth.de

Gudula Rünger
*Computer Science Department*
*Chemnitz University of Technology, Germany*
ruenger@informatik.tu–chemnitz.de

**Abstract** Parallelism is still one of the most prominent techniques to improve the performance of large application programs. Parallelism can be detected and exploited on several different levels, including instruction level parallelism, data parallelism, functional parallelism and loop parallelism. A suitable mixture of different levels of parallelism can often improve the performance significantly and the task of parallel programming is to find and code the corresponding programs,

We discuss the potential of using multiple levels of parallelism in applications from scientific computing and specifically consider the programming with hierarchically structured multiprocessor tasks. A multiprocessor task can be mapped on a group of processors and can be executed concurrently to other independent tasks. Internally, a multiprocessor task can consist of a hierarchical composition of smaller tasks or can incorporate any kind of data, thread, or SPMD parallelism. Such a programming model is suitable for applications with an inherent modular structure. Examples are environmental models combining atmospheric, surface water, and ground water models, or aircraft simulations combining models for fluid dynamics, structural mechanics, and surface heating. But also methods like specific ODE solvers or hierarchical matrix computations benefit from multiple levels of parallelism. Examples from both areas are discussed.

**Keywords:** Task parallelism, multiprocessor tasks, orthogonal processor groups, scientific computing.

## 1. Introduction

Applications from scientific computing often require a large amount of execution time due to large system sizes or a large number of iteration steps. Often the execution time can be significantly reduced by a parallel execution on a suitable parallel or distributed execution platform. Most platforms in use have a distributed address space, so that each processor can only access its local data directly. Popular execution platforms are cluster systems, cluster of SMPs (symmetric multiprocessors), or heterogeneous cluster employing processors with different characteristics or sub-interconnection networks with different communication characteristics. Using standardized message passing libraries like MPI [Snir et al., 1998] or PVM [Geist et al., 1996], portable programs can be written for these systems. A satisfactory speedup is often obtained by a data parallel execution that distributes the data structures among the processors and lets each processor perform the computations on its local elements.

Many applications from scientific computing use collective communication operations to distribute data to different processors or collect partial results from different processors. Examples are iterative methods which compute an iteration vector in each iteration step. In a parallel execution, each processor computes a part of the iteration vector and the parts are collected at the end of each iteration step to make the iteration vector available to all processors. The ease of programming with collective communication operations comes for the price that their execution time shows a logarithmic or linear dependence on the number of executing processors. Examples are given in Figure 1 for the execution time of an MPI_Bcast() and an MPI_Allgather() operation on 24 processors of the cluster system CLIC (Chemnitzer Linux Cluster). The figure shows that an execution on the global set of processors requires a much larger time than a concurrent execution on smaller subsets. The resulting execution time is influenced by the specific collective operation to be performed, the implementation in the specific library, and the performance of the interconnection network used. The increase of the execution time of the communication operations with the number of processors may cause scalability problems if a pure data parallel SPMD implementation is used for a large number of processors. There are several techniques to improve the scalability in this situation:

(a) Collective communication operations are replaced by single-transfer operations that are performed between a single sender and a single receiver. This can often be applied for domain decomposition methods where a data exchange is performed only between neighboring processors.

(b) Multiple levels of parallelism can be exploited. In particular, a mixed task and data parallel execution can be used if the application provides task parallelism in the form of program parts that are independent of each other and that can therefore be executed concurrently. Depending on the application, multiple levels of task parallelism may be available.

(c) Orthogonal structures of communication can be exploited. In particular, collective communication operations on the global set of processors can be reorganized such that the communication operations are performed in phases on subsets of the processors.

Each of the techniques requires a detailed analysis of the application and, starting from a data parallel realization, may require a significant amount of code restructuring and rewriting. Moreover, not all techniques are suitable for a specific application, so that the analysis of the application also has to determine which of the techniques is most promising.

In this paper, we give an overview how multiple levels of parallelism can be exploited. We identify different levels of parallelism and describe techniques and programming support to exploit them. Specifically, we discuss the programming with hierarchically structured multiprocessor tasks (M-tasks) [Rauber and Rünger, 2000; Rauber and Rünger, 2002]. Moreover we give a short overview how orthogonal structures of communication can be used and show that a combination of M-task parallelism and orthogonal communication can lead to efficient implementations with good scalability properties. As example we mainly consider solution methods for ordinary differential equations (ODEs). ODE solvers are considered to be difficult to parallelize, but some of the solution methods provide task parallelism in each time step. The rest of the paper is organized as follows. Section 2 describes multiple levels of parallelism in applications from scientific computing. Section 3 presents a programming approach for exploiting task parallelism. Section 4 gives example applications that can benefit from using task parallelism. Section 5 shows how orthogonal structures of communication can be used. Section 6 demonstrates this for example applications. Section 7 concludes the paper.

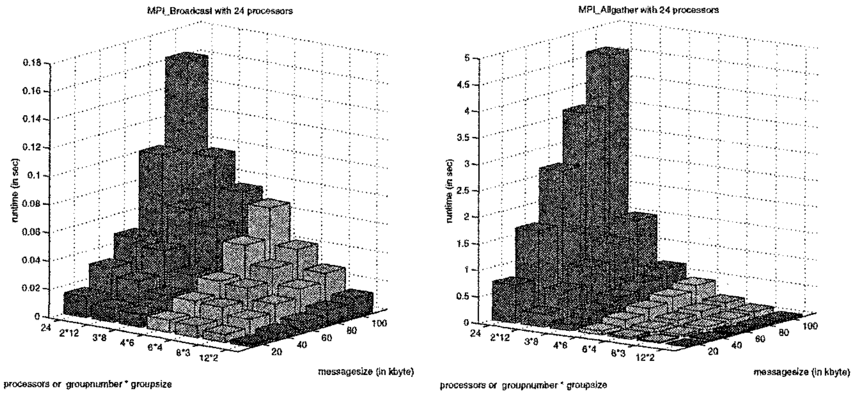


*Figure 1.* Execution time of MPI_Bcast() and MPI_Allgather() operations on the CLIC cluster. The diagrams show the execution time for different message sizes and groups organizations. The execution time for, e.g., group organization $2 * 12$ denotes the execution time on two groups of 12 processors, that work concurrently to each other.

## 2. Multiple levels of parallelism in numerical algorithms

Modular structures of cooperating subtasks often occur in large application programs and can be exploited for a task parallel execution. The tasks are often complete subprograms performing independent computations or simulations. The resulting task granularity is therefore quite coarse and the applications usually have only a small number of such subtasks. Numerical methods on the other hand sometimes provide potential task parallelism of medium granularity, but this task parallelism is usually limited. These algorithms can often be reformulated such that an additional task structure results. A reformulation may affect the numerical properties of the algorithms, so that a detailed analysis of the numerical properties is required. In this section, we describe different levels of parallelism in scientific applications and give an overview of programming techniques and support for exploiting the parallelism provided.

### 2.1 Instruction level parallelism

A sequential or parallel application offers the potential of instruction level parallelism (ILP), if there are no dependencies between adjacent instructions. In this case, the instructions can be executed by different functional units of the microprocessor concurrently. The dependencies that have to be avoided include true (flow) dependencies, anti dependencies and output dependencies [Hennessy and Patterson, 2003]. ILP results in fine-grained parallelism and is usually exploited by the instruction scheduler of superscalar processors. These schedulers perform a dependency analysis of the next instructions to be executed and assign the instructions to the functional units with the goal to keep the functional units busy. Modern microprocessors offer several functional units for different kinds of instructions like integer instructions, floating point instructions or memory access instructions. However, simulation experiments have shown that usually only a small number of functional units can be exploited in typical programs, since dependencies often do not allow a parallel execution.

ILP cannot be controlled explicitly at the program level, i.e., it is not possible for the programmer to restructure the program so that the degree of ILP is increased. Instead, ILP is always implicitly exploited by the hardware scheduler of the microprocessor.

### 2.2 Data parallelism

Many programs contain sections where the same operation is applied to different elements of large regular data structure like vectors or matrices. If there are no dependencies, these operations can be executed concurrently by different processors of a parallel or distributed system (data parallelism). Potential data parallelism is usually quite easy to identify and can be exploited by distributing the data structure among the processors and let each processor perform only the operation on its local elements (owner-computer rule). If the processors have to access elements that are stored on neighboring processors, a data exchange has to be performed before the computations to make these elements available. This is often organized by introducing ghost cells for each processor to store the elements sent by the neighboring processors. The data exchange has to be performed explicitly for platforms with a distributed address space by using a suitable message passing library like MPI or PVM, which requires an explicit restructuring of a (sequential) program.

Data parallelism can also be exploited by using a data parallel programming language like Fortran90 or HPF (High-Performance Fortran) [Forum, 1993] which use a single control flow and offer data parallel operations on portions of vectors or matrices. The communication operations to exchange data elements between neighboring processors do not need to be expressed explicitly, but are generated by the compiler according to the data dependencies.

## 2.3 Loop parallelism

The iterations of a loop can be executed in parallel if there are no dependencies between them. If all loop iterations are independent from each other, the loop is called a *parallel loop* and provides loop-level parallelism. This source of parallelism can be exploited by distributing the iterations of the parallel loop among the processors available. For a load balanced parallel execution, different loop distributions may be beneficial.

If all iterations of the loop require the same execution time, the distribution can easily be performed by a static distribution that assigns a fixed amount of iterations to each processor. If each iteration requires a different amount of execution time, such a static distribution can lead to load imbalances. Therefore dynamic techniques are used in this case. These techniques distribute the iterations in chunks of fixed or variable size to the different processors. The remaining iterations are often stored in a central queue from which a central manager distributes them to the processors. Non-adaptive techniques use chunks of fixed size or of a decreasing size that is determined in advance. Examples of non-adaptive techniques are FSC (fixed size chunking) and GSS (guided self scheduling) [Polychronopoulos and Kuck, 1987]. Adaptive techniques use chunks of variable size whose size is determined according to the number of remaining iterations. Adaptive techniques include factoring or weighted factoring [Banicescu and Velusamy, 2002; Banicescu et al., 2003] and allow also an adaptation of the chunk size to the speed of the executing processors. A good overview can be found in [Banicescu et al., 2003].

Loop-level parallelism can be exploited by using programming environments like OpenMP that provide the corresponding techniques or by implementing a loop manager that employs the specific scheduling technique to be used. Often, sequential loops can be transformed into parallel loops by applying loop transformation techniques like loop interchange or loop splitting, see [Wolfe, 1996] for an overview.

## 2.4 Task parallelism

A program exhibits task parallelism (also denoted as functional parallelism) if it contains different parts that are independent of each other and can therefore be executed concurrently. The program parts are usually denoted as tasks. Depending on the granularity of the independent program parts, the tasks can be executed as single-processor tasks (S-tasks) or multiprocessor tasks (M-tasks). M-tasks can be executed on an arbitrary number of processors in a data-parallel or SPMD style whereas each S-task is executed on a single processor.

A simple but efficient approach to distribute executable S-tasks among the processors is the use of (global or distributed) task pools. Tasks that are ready for execution are stored in the task pool from which they are accessed by idle processors for execution. Task pools have originally been designed for shared address spaces [Singh, 1993] and can provide good scalability also for irregular applications like the hierarchical radiosity method or volume rendering [Hoffmann et al., 2004]. To ensure this the task pools have to be organized in such a way that they achieve load balance of the processors and avoid bottlenecks when different processors try to retrieve executable tasks at the same time. Bottlenecks can usually be avoided by using distributed task pools that use a separate pool for each processor instead of one global pool that is accessed by all processors. When using distributed task pools, load balancing can be achieved by allowing processors to access the task pools of other processors if their local pool is empty (task stealing) or by employing a task manager that moves tasks between the task pools in the background. In both cases, each processor has to use synchronization also when accessing its local pool to avoid race conditions. The approach can be extended to distributed address spaces by including appropriate communication facilities [Hippold and Rünger, 2003].

The scheduling of M-tasks is more difficult than the scheduling of S-tasks, since each M-task can in principle be executed on an arbitrary number of processors. If there are several tasks that can be executed concurrently, the available processors should be partitioned into subsets such that there is one subset for each M-task and such that the execution of the concurrent M-tasks is finished at about the same time. This can be achieved if the size of the subsets of processors is adapted to the execution time of the M-tasks. The execution time is usually not known in advance and heuristics are applied.

M-tasks may also exhibit an internal structure with embedded M-tasks, i.e., the M-tasks may be hierarchically organized, which is a typical situation when executing divide-and-conquer methods in parallel.

## 3. Basics of M-task programming

This section gives a short overview of the Tlib library [Rauber and Rünger, 2002] that has been developed on top of MPI to support the programmer in the design of M-task programs. A Tlib program is an executable specification of the coordination and cooperation of the M-tasks in a program. M-tasks can be library functions or user-supplied functions, and they can also be built up from other M-tasks. Iterations and recursions of M-task specifications is possible; the parallel execution might result in a hierarchical splitting of the processor set until no further splitting is possible or reasonable. Using the library, the programmer can specify the M-tasks to be used by simply putting the operations to be performed in a function with a signature of the form

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

where the parameter `arg` comprises the arguments of the M-task, `comm` is the MPI communicator that can be used for the internal communication of the M-task and `pdescr` describes the current (hierarchical) group organization.

The Tlib library provides support for (a) the creation and administration of a dynamic hierarchy of processor groups, (b) the coordination and mapping of M-tasks to processor groups, (c) the handling and termination of recursive calls and group splittings and (d) the organization of communication between M-tasks. M-tasks can be hierarchically organized, i.e., each function of the form above can contain Tlib operations to split the group of executing processors or to assign new M-tasks to the newly created subgroups. The current group organization is stored in a group descriptor such that each processor of a group can access information about the group that it belongs to via this descriptor. The group descriptor for the global group of processors is generated by an initialization operation. Each splitting operation subdivides a given group into smaller subgroups according to the specification of the programmer. The resulting group structure is stored in the group descriptors of the participating processors. An example is the Tlib operation

```
int T_SplitGrp (T_Descr *pdescr, T_Descr *pdescr1,
                          float p1, float p2)
```

with $0 \leq p1 + p2 \leq 1$. The operation generates two subgroups with a fraction `p1` or `p2` of the number of processors in the given processor group described by `pdescr`. The resulting group structure is described by group descriptor pdescr1. The corresponding communicator of a subgroup can be obtained by the Tlib operation

```
MPI_Comm T_GetComm (T_Descr *pdescr1).
```

Each processor obtains the communicator of the subgroup that it belongs to. Thus, group-internal communication can be performed by using this communicator. The Tlib group descriptor contains much more information including the current hierarchical group structure and explicit information about the group sizes, group members, or sibling groups.

The execution of M-tasks is initiated by assigning M-tasks to processor groups for execution. For two concurrent processor groups that have been created by T_SplitGrp(), this can be achieved by the Tlib operation

```
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 *pdescr1).
```

Here, f1 and f2 are the M-task functions to be executed, parg1 and parg2 are their arguments and pres1 and pres2 are their possible results. The last argument pdescr1 is a group descriptor that has been returned by a preceeding call of T_SplitGrp(). The activation of an M-tasks by T_Par() automatically assigns the MPI communicator of the group descriptor, provided as last argument, to the second argument of the M-task. Thus, each M-task can use this communicator for group-internal MPI communication. The group descriptor itself is automatically assigned to the third argument of the M-task. By using this group descriptor, the M-task can further split the processor group and can assign M-tasks to the newly created subgroups in the same way. This allows the nesting of M-tasks, e.g. for the realization of divide-and-conquer methods or hierarchical algorithms.

In addition to the local group communicator, an M-task can also use other communicators to perform MPI communication operations. For example, an M-task can access the communicator of the parent group via the Tlib operations

```
parent_descr = T_GetParent (pdescr);
parent_comm = T_GetComm (Parent_descr);
```

and can then use this communicator to perform communication with M-tasks that are executed concurrently.

## 4. Examples for M-task programming

In this section, we describe some example applications that can benefit from an exploitation of task parallelism. In particular, we consider extrapolation methods for solving ordinary differential equations (ODEs) and the Strassen method for matrix multiplication.

### 4.1 Extrapolation methods

Extrapolation methods are explicit one-step solution methods for ODEs. In each time step, the methods compute $r$ different approximations for the same point in time with different stepsizes $h_1, \ldots, h_r$ and combine the approximations at the end of the time step to a final approximation of higher order [Hairer et al., 1993; Deuflhard, 1985; van der Houwen and Sommeijer, 1990b]. The computations of the $r$ different approximations are independent of each other and can therefore be computed concurrently as independent M-tasks. Figure 2 shows an illustration of the method.

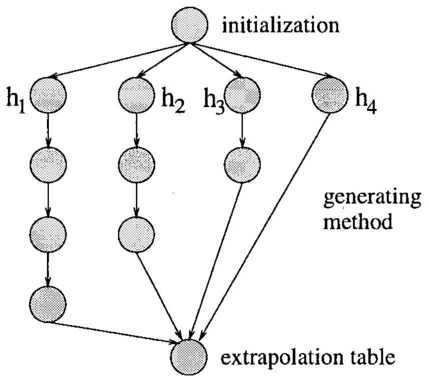


*Figure 2.* One time step of extrapolation method with $r = 4$ different stepsizes. The steps to perform one step with a stepsize $h_i$ is denoted as microstep. We assume that stepsize $h_i$ requires $r - i + 1$ microsteps of the generating method which is often a simple Euler method. The $r$ approximations computed are combined in an extrapolation table to the final approximation for the time step.

The usage of different stepsizes corresponds to different numbers of microsteps, leading to a different amount of computation for each M-task. Thus, different group sizes should be used for an M-task version to guarantee that processor groups finish the computations of their approximations at about the same time. The following two group partitionings achieve good load balance:

(a) Linear partitioning: we use $r$ disjoint processor groups $G_1, \ldots, G_r$ whose size $g_1, \ldots, g_r$ is determined according to the computational effort for the different approximations. If we assume that stepsize $h_i$ requires $r - i + 1$ microsteps, a total number of $\sum_{i=1}^{r} (r - i + 1) = (1/2)r \cdot (r + 1)$ microsteps have to be performed. Processors group $G_i$ has to perform $i$

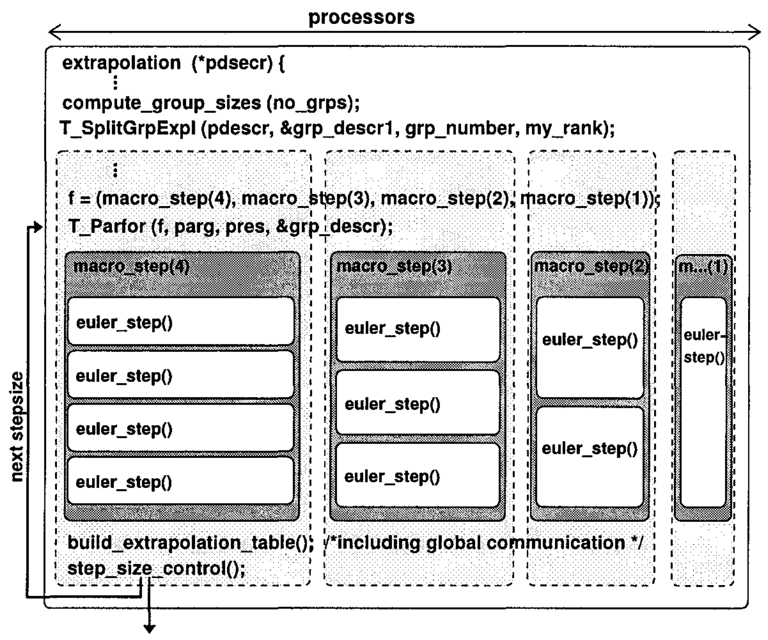

processors
extrapolation (*pdsecr) {
compute_group_sizes (no_grps);
T_SplitGrpExpl (pdescr, &grp_descr1, grp_number, my_rank);
f = (macro_step(4), macro_step(3), macro_step(2), macro_step(1));
T_Parfor (f, parg, pres, &grp_descr);
macro_step(4)
macro_step(3)
macro_step(2)
m...(1)
euler_step()
euler-step()
build_extrapolation_table(); /*including global communication */
step_size_control();
next stepsize

*Figure 3.* M-task structure of extrapolation methods expressed as Tlib coordination program.

steps of those. Thus, for $p$ processors in total, group $G_i$ should contain $g_i = p \cdot \frac{2 \cdot i}{r \cdot (r+1)}$ processors.

(b) Extended partitioning: instead of $r$ groups of different size, $\lceil r/2 \rceil$ groups of the same size are used. Group $G_i$ performs the microsteps for stepsizes $h_i$ and $h_{r-i+1}$. Since stepsize $h_i$ requires $r - i + 1$ microsteps and stepsize $h_{r-i+1}$ requires $r - (r - i + 1) + 1$ microsteps, each group $G_i$ has to perform a total number of $r + 1$ microsteps, so that an equal partitioning is adequate.

Figure 3 illustrates the group partitioning and the M-task assignment for the linear partitioning using the Tlib library. The groups are built using the Tlib function `T_SplitGrpExpl()` which uses explicit processor identifications and group identifications provided by the user-defined function compute_group_sizes(). The group partitioning is performed only once and is then re-used in each time step. The library call `T_Parfor()` is a generalization of `T_Par()` and assigns the M-tasks with the corresponding argument vectors to the subgroups for execution.

Figure 4 compares the execution time of the two task parallel execution schemes with the execution time of a pure data parallel program version on a Cray T3E with up to 32 processors. As application, a Brusselator ODE has been used. The figure shows that the exploitation of task parallelism is worth the

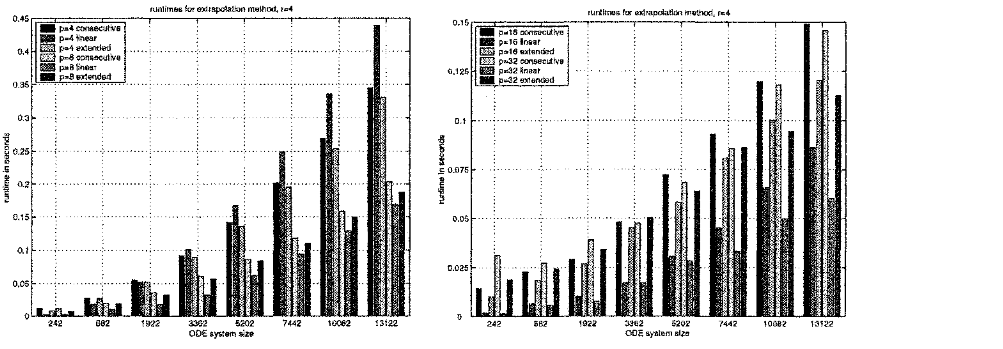


*Figure 4.* Execution time of an extrapolation method on a Cray T3E.

effort in particular for a large number of processors. When comparing the two task parallel execution schemes, it can be seen that exploiting the full extent of task parallelism (linear partitioning) leads to the shortest execution times for more than 16 processors.

## 4.2 Strassen matrix multiplication

Task parallelism can often be exploited in divide-and-conquer methods. An example is the Strassen algorithm for the multiplication of two matrices $A, B$. The algorithm decomposes the matrices $A, B$ into square blocks of size $n/2$:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

and computes the submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ separately according to

$$\begin{aligned} C_{11} &= Q_1 + Q_4 - Q_5 + Q_7 \\ C_{12} &= Q_3 + Q_5 \\ C_{21} &= Q_2 + Q_4 \\ C_{22} &= Q_1 + Q_3 - Q_2 + Q_6 \end{aligned}$$

where the computation of the matrices $Q_1, \ldots, Q_7$ require 7 matrix multiplications for smaller matrices of size $n/2 \times n/2$. The seven matrix products can be computed by a conventional matrix multiplication or by a recursive application of the same procedure resulting in a divide-and-conquer algorithm with the following subproblems:

$$\begin{aligned} Q_1 &= \mathit{strassen}(A_{11} + A_{22}, B_{11} + B_{22}); \\ Q_2 &= \mathit{strassen}(A_{21} + A_{22}, B_{11}); \\ Q_3 &= \mathit{strassen}(A_{11}, B_{12} - B_{22}); \end{aligned}$$

$$\begin{aligned}
Q_4 &= strassen(A_{22}, B_{21} - B_{11});\\
Q_5 &= strassen(A_{11} + A_{12}, B_{22});\\
Q_6 &= strassen(A_{21} - A_{11}, B_{11} + B_{12});\\
Q_7 &= strassen(A_{12} - A_{22}, B_{21} + B_{22});
\end{aligned}$$

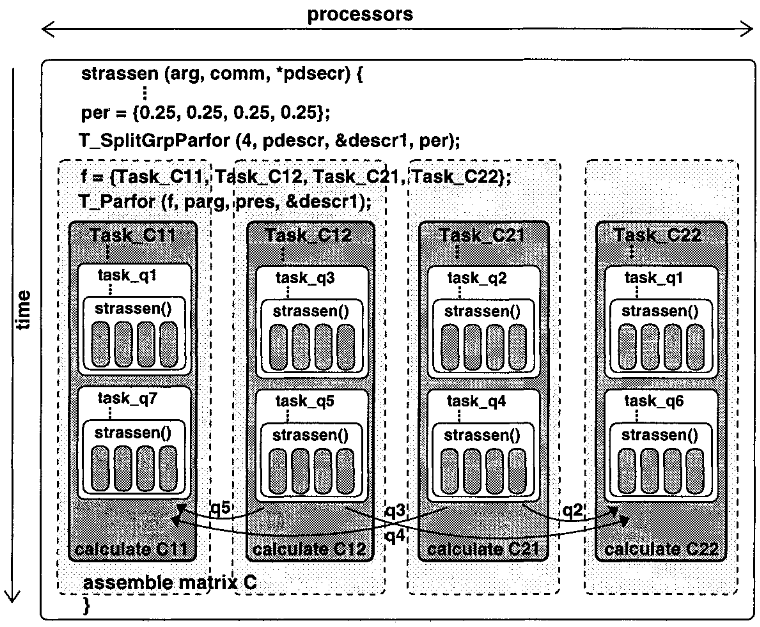


*Figure 5.* Task structure of the Strassen method using the Tlib library.

Figure 5 shows the structure of a task parallel execution of the Strassen algorithm and sketches a Tlib program. Four processor groups of the same size are formed by `T_SplitGrpParfor()` and `T_Parfor()` assigns tasks to compute $C_{11}$, $C_{12}$, $C_{21}$ and $C_{22}$, respectively, to those groups. Internally, those M-tasks call other M-tasks to perform the subcomputations of $Q_1, \ldots, Q_7$, including recursive calls to Strassen. Communication is required between those groups and is indicated by annotated arrows. Such a task parallel implementation can be used as a starting point for the development of efficient parallel algorithms for matrix multiplication. In [Hunold et al., 2004b] we have shown that a combination of a task parallel implementation of the Strassen method with an efficient basic matrix multiplication algorithm tpMM [Hunold et al., 2004a] can lead to very efficient implementations, if a fast sequential algorithm like Atlas [Whaley and Dongarra, 1997] is used for performing the computations on a single processor.

## 5. Exploiting orthogonal structures of communication

For many applications, the communication overhead can also be reduced by exploiting orthogonal structures of communication. We consider the combination of orthogonal communication structures with M-task parallel programming and demonstrate the effect for iterated Runge-Kutta methods.

### 5.1 Iterated Runge-Kutta methods

Iterated Runge-Kutta methods (RK methods) are explicit one-step solution methods for solving ODEs, which have been designed to provide an additional level of parallelism in each time step [van der Houwen and Sommeijer, 1990a]. In contrast to embedded RK methods, the stage vector computations in each time step are independent of each other and can be computed by independent groups of processors. Each stage vector is computed by a separate fixed point iteration. For $s$ stage vectors $v_1, \ldots, v_s$ and right-hand side function $f$, the new approximation vector $y_{k+1}$ is computed from the previous approximation vector $y_k$ by:

$$\begin{aligned}
v^l_{(0)} &= f(y_\kappa), \qquad l = 1, \ldots, s \\
v^l_{(j)} &= f(y_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} v^i_{(j-1)}), \quad l = 1, \ldots, s, \quad j = 1, \ldots, m \\
y^{(m)}_{\kappa+1} &= y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, v^l_{(m)} \\
y^{(m-1)}_{\kappa+1} &= y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, v^l_{(m-1)}
\end{aligned}$$

After $m$ steps of the fixed point iteration, $y^{(m)}_{(k+1)}$ is used as approximation for $y_{k+1}$ and $y^{(m-1)}_{k+1}$ is used for error control and stepsize selection.

A task parallel computation uses $s$ processor groups $G_1, \ldots, G_s$ with sizes $g_1, \ldots g_s$ for computing the stage vectors $v_1, \ldots, v_s$. Since each stage vector requires the same amount of computations, the groups should have about equal size. An illustration of the task parallel execution is shown in Figure 5.1. For computing stage vector $v_l$, each processor of $G_l$ computes a block of elements of the argument vector $\mu(l, j) = y_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} v^i_{(j-1)}$ (where $j$ is the current iteration) by calling `compute_arguments()`. Before applying the function $f$ to $\mu(l, j)$ in `compute_fct()` to obtain $v^l_{(j)}$, each component of $\mu(l, j)$ has to be made available to all processors of $G_l$, since $f$ may access all components of its argument. This can be done in `group_broadcast()` by a group-internal MPI_Allgather() operation. Using an M-task for the stage vector computation,

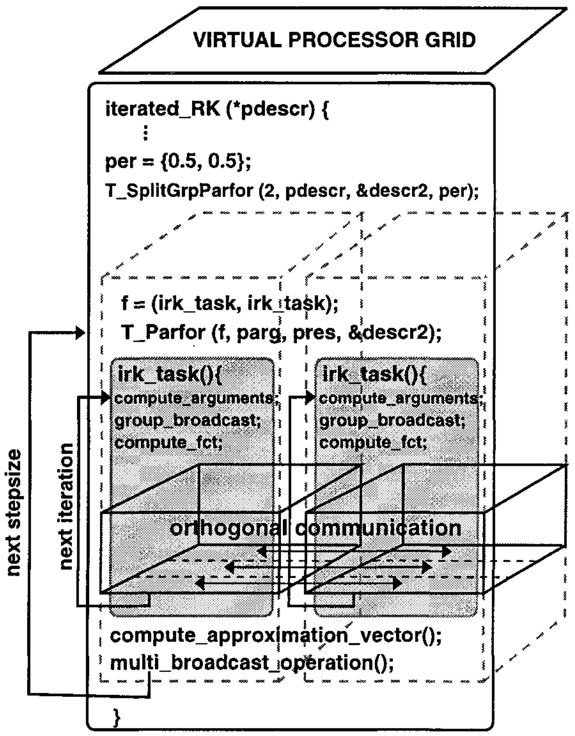


*Figure 6.* Exploiting orthogonal communication structures for iterated RK methods.

an internal communication operation is performed. Before the next iteration step $j$, each processor needs those parts of the previous iteration vectors $v^i_{(j-1)}$ to compute its part of the next argument vector $\mu(l,j)$. In the general case where each processor may have stored blocks of different size, the easiest way to obtain this is to make each processor the entire vectors $v^i_{(j-1)}$ available. This can be achieved by a global MPI_Allgather() operation. After a fixed number of iteration steps, the next approximation vector $y_{\kappa+1} = y^{(m)}_{\kappa+1}$ is computed and is made available to all processors by a global MPI_Allgather() operation.

## 5.2 Orthogonal Communication for Iterated RK Methods

For the special case that all groups have the same size $g$ and that each processor stores blocks of the iteration vectors of the same size, orthogonal structures can be exploited for a more efficient communication. The orthogonal communication is based on the definition of orthogonal processor groups: For groups $G_1, \dots, G_s$ with $G_l = \{q_{l1}, \dots, q_{lg}\}$, we define orthogonal groups $Q_1, \dots, Q_g$ with $Q_k = \{q_{lk} \in G_l, l = 1, \dots, s\}$, see Figure 7. Instead of making all components of $v^i_{(j-1)}$ available to each processor, a group-based MPI_Allgather()

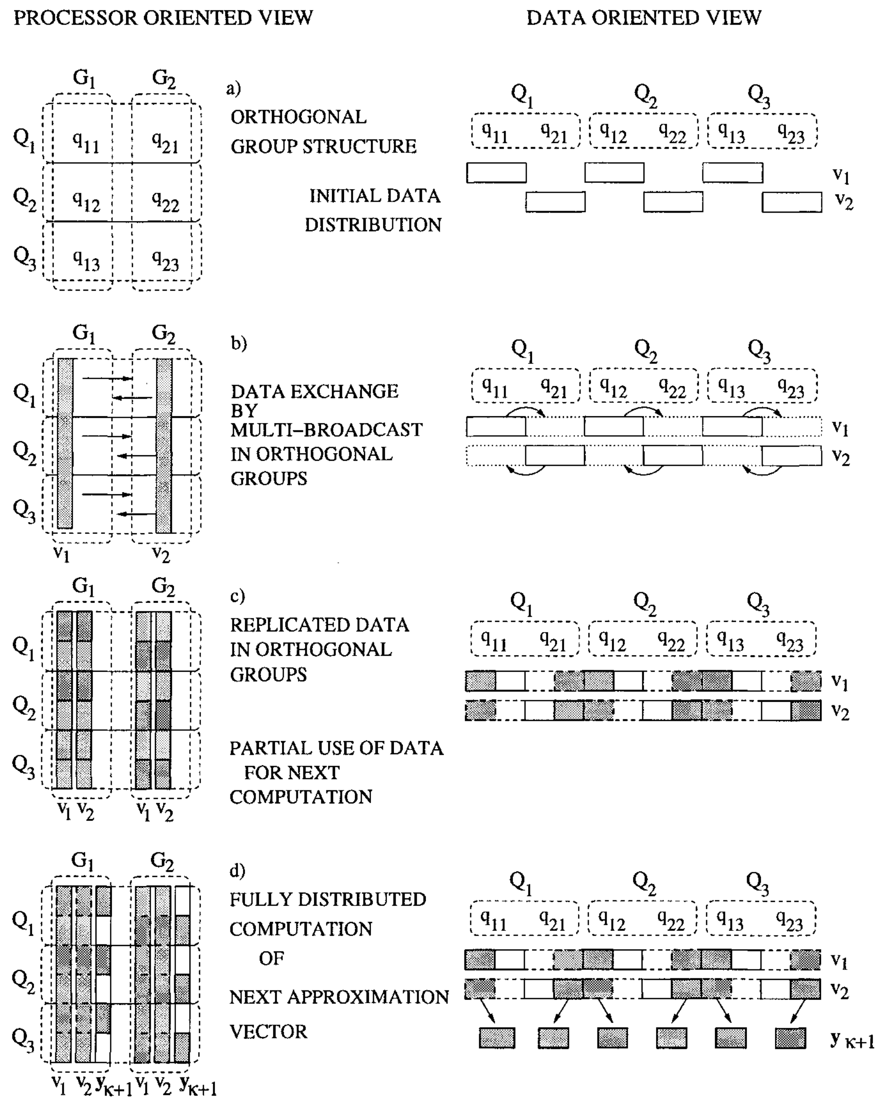
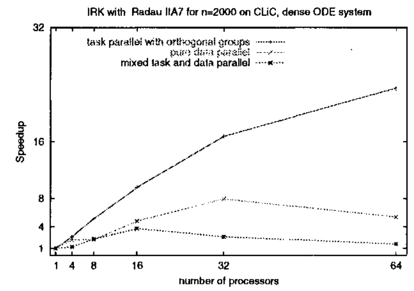

PROCESSOR ORIENTED VIEW
DATA ORIENTED VIEW
a)
ORTHOGONAL GROUP STRUCTURE
INITIAL DATA DISTRIBUTION
b)
DATA EXCHANGE BY MULTI-BROADCAST IN ORTHOGONAL GROUPS
c)
REPLICATED DATA IN ORTHOGONAL GROUPS
PARTIAL USE OF DATA FOR NEXT COMPUTATION
d)
FULLY DISTRIBUTED COMPUTATION OF NEXT APPROXIMATION VECTOR
G1
G2
Q1
Q2
Q3
q11
q21
q12
q22
q13
q23
v1
v2
y K+1

*Figure 7.* Exploiting orthogonal communication structures for the iterated RK method using two stage vectors $v_1$ and $v_2$ and corresponding processor groups $G_1 = \{q_{11}, q_{12}, q_{13}\}$ and $G_2 = \{q_{21}, q_{22}, q_{23}\}$. Part (a) illustrates the group structure with orthogonal groups $Q_1, Q_2, Q_3$ (left) and the distribution of $v_1$ and $v_2$ among the processors of $G_1$ and $G_2$. Part (b) illustrates the data exchange within the orthogonal groups after the computation of the argument vectors, leading to a replication of the corresponding blocks of $v_1$ and $v_2$ in the orthogonal groups, see part (c). Part (d) illustrates the usage of the stage vector blocks for the computation of the next approximation vector $y_{\kappa+1}$. Each processor uses only a part of the replicated data blocks.



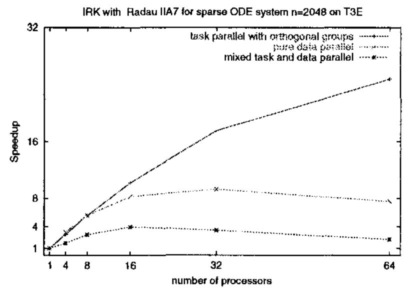


*Figure 8.* Execution times of iterated RK method based on Radau IIA7 on CLiC for dense ODE systems (left) and on Cray T3E for sparse ODE systems (right).

operation can be used on $Q_1, \ldots, Q_g$ concurrently where each processor of $Q_k$ contributes a block of $v^l_{(j-1)}$ to make the other processor of $Q_k$ exactly those blocks available that it needs for the computation of the argument vectors. Thus, each time step contains group-local communication operations only. Figure 8 compares the resulting execution times for a specific iterated RK method on two different platforms, a Cray T3E and a Beowulf Cluster (CLIC).

## 6. Conclusions

Exploiting multiple levels of parallelism often leads to parallel programs that show a better scalability than parallel programs that rely on a single source of parallelism. This can be observed for many examples from scientific computing. In particular, many algorithms provide a source for a multiprocessor task parallel execution that can be used for a group-based execution. In this context, it is often beneficial to use orthogonal communications to exchange data between the different subgroups in such a way that global communication operations are avoided whenever possible.

## References

Banicescu, I. and Velusamy, V. (2002). Load balancing highly irregular computations with the adaptive factoring. In *Proc. of the IEEE - International Parallel and Distributed Processing Symposium (IPDPS 2002) - Heterogeneous Computing Workshop*. IEEE Computer Society Press, Fort Lauderdale.

Banicescu, I., Velusamy, V., and Devaprasad, J. (2003). On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring. *Cluster Computing, The Journal of Networks, Software Tools and Applications*, 6(3):215–226.

Deuflhard, P. (1985). Recent progress in extrapolation methods for ordinary differential equations. *SIAM Review*, 27:505–535.

Forum, H. P. F. (1993). High Performance Fortran Language Specification. *Scientific Programming*, 2(1).

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1996). *PVM Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA.

Hairer, E., Norsett, S., and Wanner, G. (1993). *Solving Ordinary Differential Equations I: Nonstiff Problems.* Springer–Verlag, Berlin.

Hennessy, J. and Patterson, D. (2003). *Computer Architecture — A Quantitative Approach.* Morgan Kaufmann, 3nd edition.

Hippold, J. and Rünger, G. (2003). Task Pool Teams for Implementing Irregular Algorithms on Clusters of SMPs. In *Proc. of the IPDPS (International Parallel and Distributed Processing Symposium),* Nice, France. IEEE.

Hoffmann, R., Korch, M., and Rauber, T. (2004). Using Hardware Operations to Reduce the Synchronization Overhead of Task Pools. In *Proc. of the Int. Conference on Parallel Processing (ICPP),* pages 241–249.

Hunold, S., Rauber, T., and Rünger, G. (2004a). Hierarchical Matrix-Matrix Multiplication based on Multiprocessor Tasks. In Bubak, M., van Albada, G., Sloot, P. M., and Dongarra, J. J., editors, *Proc. of the International Conference on Computational Science ICCS 2004, Part II,* LNCS 3037, pages 1–8. Springer.

Hunold, S., Rauber, T., and Rünger, G. (2004b). Multilevel Hierarchical Matrix Multiplication on Clusters. In *Proc. of the 18th Annual ACM International Conference on Supercomputing, ICS'04,* pages 136–145.

Polychronopoulos, C. and Kuck, D. (1987). Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers,* C-36(12):1425–1439.

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

Rauber, T. and Rünger, G. (2002). Library Support for Hierarchical Multi-Processor Tasks. In *Proc. of the Supercomputing 2002,* Baltimore, USA. ACM/IEEE.

Singh, J. (1993). *Parallel Hierarchical N-Body Methods and their Implication for Multiprocessors.* PhD thesis, Stanford University.

Snir, M., Otto, S., Huss-Ledermann, S., Walker, D., and Dongarra, J. (1998). *MPI: The Complete Reference, Vol.1: The MPI Core.* MIT Press, Camdridge, MA.

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

van der Houwen, P. and Sommeijer, B. (1990b). Parallel ODE Solvers. In *Proc. of the ACM Int. Conf. on Supercomputing,* pages 71–81.

Whaley, R. C. and Dongarra, J. J. (1997). Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee.

Wolfe, M. (1996). *High Performance Compilers for Parallel Computing.* Addison Wesley.

II

# DISTRIBUTED COMPUTING