Published: Proc. of HPCN'01, S. 363-372, Springer LNCS 2110, 2001.

Orthogonal Processor Groups for Message-Passing Programs

Thomas Rauber[1], Robert Reilein[2], Gudula Rünger[2]

[1] Institut für Informatik, Universität Halle-Wittenberg, 06099 Halle(Saale), Germany,
rauber@informatik.uni–halle.de
[2] Fakultät für Informatik, Technische Universität Chemnitz, 09107 Chemnitz, Germany,
{reilein,ruenger}@informatik.tu-chemnitz.de

Abstract. We consider a generalization of the SPMD programming model to orthogonal processor groups. In this model different partitions of the processors into disjoint processor groups can be exploited simultaneously in a single parallel implementation. The parallel programming model is appropriate for grid based applications working in horizontal or vertical directions as well as and for mixed task and data parallel computations[2]. For those applications we propose a systematic development process for message-passing programs using orthogonal processor groups. The development process starts with a specification of tasks indicating horizontal and vertical sections. A mapping to orthogonal processor groups realizes a group SPMD execution model and a final transformation step generates the corresponding message-passing program.

1 Introduction

Parallel machines with distributed memory organization are a popular platform for the implementation of applications from scientific computing because it is now well-understood how to get portable efficient programs. For typical grid-based computation structures of scientific applications, the SPMD programming model usually leads to good efficiency. But there are also applications which can benefit from a parallel programming model that allows more general but still regular communication and dependence patterns.

In this paper, we consider a group SPMD programming model with multiple processor groups in orthogonal directions where different directions are active at different points of the execution time. This programming model is suitable for applications which consist of independent disjoint computations with varying structure. Theoretical investigations have shown that it can be useful to exploit this kind of parallelism pattern in a parallel program with disjoint processor sets which concurrently execute the program in SPMD mode [7, 8]. The advantage is that 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. Moreover, the message sizes are usually smaller and different communication operations can often be executed concurrently on disjoint processor groups without interference. Disjoint processor groups can be expressed in the communication library MPI, but the direct coding of orthogonal processor groups may lead to intricate and error-prone message-passing programs. Therefore, it seems to be appropriate to provide a way to specify programs with orthogonal processor groups on a more abstract level and to generate the corresponding MPI program by a compiler tool.

The contribution of this paper is to propose a transformation approach to develop appropriate message-passing programs in a group SPMD programming model in several steps. The approach comprises a structuring of the potential parallelism, a mapping onto orthogonal processor groups and a final transformation step into a corresponding MPI program. The structuring is given by a specification program indicating sections of horizontal, vertical, or global interaction structures which are identified by analyzing the potential parallelism of the computations. The mapping onto processor groups leads to a structured parallel program in a group SPMD model with changing or alternating group structure. Different partitions of the processors into disjoint processor groups can be exploited in a single parallel implementation, but at each time of the execution only one partition can be active. The final MPI program realizes the horizontal and vertical sections with group operations. The paper outlines the transformation approach and illustrates the transformation steps for appropriate application programs.

The rest of the paper discusses orthogonal structures and its specification in Sections 2 and shows the mapping onto orthogonal processor groups in Section 3. Runtime tests with orthogonal group structures are given in Section 4. Section 5 discusses related work and Section 6 concludes.

2 Orthogonal structures of computations in scientific applications

An application has an orthogonal structure of computation and communication with varying directions, if the computations and the data dependencies exhibit regular patterns in a horizontal or vertical direction of the task organization.

Describing orthogonal computation structures For the organization of the computations and the assignment to processors, we use the following abstraction: A parallel application program is composed of a set $\mathcal{T}$ of n one-processor tasks which are organized in a two-dimensional way. The tasks are numbered with two-dimensional indices in the form T_{ij}, $i = 1, \ldots, n_1, j = 1, \ldots, n_2$, with $n = n_1 \cdot n_2$. Each single task $T \in \mathcal{T}$ consists of a sequence of computations and communication commands. Data needed from other tasks or required by other tasks form a dependence structure between the tasks.

For applications exhibiting orthogonal interaction structures the horizontal and vertical computations and communications are clearly separated, i.e., at each point in time, a task is either involved in operations in the vertical or the horizontal direction, but not both. This can be formalized using an *interaction matrix* which is the adjacency matrix of the dependence graph of a task array $\mathcal{T}$. An interaction matrix $A_{\mathcal{T}}$ for a task array $\mathcal{T}$ has size $n \times n$ with $n = n_1 * n_2$. The row $r = (i-1)n_2 + j$ of matrix A_T shows the interactions of task T_{ij}. The columns of the matrix A_T are also associated with the tasks of $\mathcal{T}$, where column $s = (k-1)n_2 + l$ is associated with task T_{kl}. The matrix A_T has a nonzero entry in $A_{\mathcal{T}}[r, s]$, $r, s = 1, ..., n$, if task T_{ij} associated with row r has an interaction with task T_{kl} associated with column s. If task T_{ij} has no interaction with task T_{kl} then there is no entry in $A_{\mathcal{T}}[r, s]$ for $r = (i-1)n_2 + j$ and $s = (k-1)n_2 + l$.

Figure 1 shows the pattern of an interaction matrix with horizontal and vertical interactions (in the middle) and two matrices which show the interactions separately. The matrix on the left, depicting horizontal dependencies, consists of n_2 blocks of size $n_1 \times n_1$ where each block indicates the dependencies within one row of the task array

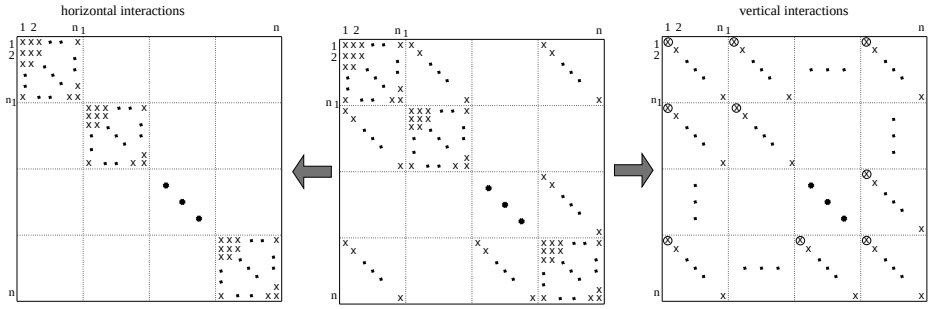


Fig. 1. Interaction matrices of size $n \times n$ for task arrays of size $n_1 \times n_2$, $n = n_1 n_2$, showing the nonzero pattern for horizontal and vertical (middle), horizontal (left) or vertical (right) dependencies.

$\mathcal{T}$. Since there are no nonzero elements outside those blocks the dependency graph of the corresponding task array consists of n_2 independent subgraphs. The matrix on the right, depicting vertical dependencies, has nonzero entries in the main diagonal and the diagonals with distance n_1. The nonzero elements in all rows with distance n_1 correspond to computations within one column of a task array; e.g. the set of items with circles indicate the pattern for vertical dependencies in the first column of the task array. The nonzero pattern of the illustration in Figure 1 express the maximal set of dependencies for a single vertical or horizontal computation structure which means that an orthogonal program part can have less nonzero elements than depicted but not more or in other positions. To express SPMD computations in horizontal or vertical direction, we can define the interaction matrix in a more general way, such that a nonzero element means that the two tasks associated with the corresponding row and column perform similar computations in an SPMD style.

To exploit orthogonal process structures, an application is not required to have a dependence graph with corresponding interaction matrix for the entire program. Rather, we consider a much larger class of applications for which the orthogonal structures are only given for parts of the program and define orthogonal application structures in the following sense: If an application program can be decomposed into program parts and the program parts have either a dependence graph with only horizontal interactions (Figure 1 left) or only vertical interactions (Figure 1 right), then the application exhibits an orthogonal structure with the potential parallelism suitable for a group SPMD program with varying orthogonal groups. Thus, different program parts may have different interaction matrices. Also program parts with general interaction matrix are allowed but cannot benefit from the specific orthogonal group implementation.

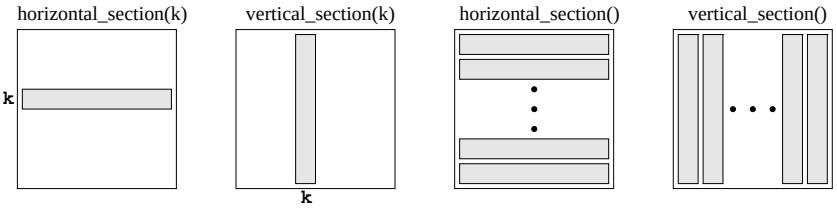
Specification of orthogonal structures The entire task program is specified in a group SPMD style with explicit constructions for horizontal or vertical executions, which we call *horizontal sections* and *vertical sections*, respectively. In horizontal sections, a task T_{ij} has interactions to a set of tasks $\{T_{ij'} | j' = 1, ..., n_2, j' \neq j\}$. In vertical sections, a task T_{ij} has interactions to a set of tasks $\{T_{i'j} | i' = 1, ..., n_1, i' \neq i\}$. To indicate that a task participates in an SPMD-like operation together with other tasks in horizontal or vertical direction we use the commands:

– vertical_section(k) { statements } : Each task in column k executes statements in an SPMD-like way together with the other tasks in column k; statements may

contain computations as well as collective communication and reduction operations involving tasks $T_{1k}, \ldots, T_{n_1,k}$. Tasks T_{ij} with $j \neq k$ perform a `skip`-operation.

- horizontal_section(k) { statements } : Similar to vertical_section(k), but using a horizontal organization.
- vertical_section( ) { statements } : Each task executes statements in an SPMD-like way together with the other tasks in the same column. A task in column k may perform computations as well as collective communication and reduction operations involving the tasks $T_{1k}, \ldots, T_{n_1,k}$ in the same column. Computations in a specific column of the task array are executed in parallel with the other columns in a group SPMD programming model. Thus, vertical_section( ) corresponds to a parallel loop over all columns k where each iteration executes vertical_section(k).
- horizontal_section( ) { statements } : Similar to vertical_section( ), but using a horizontal organization.

The gray parts in the following illustration depict active computation parts in the task array:



Commands outside of horizontal or vertical sections are executed in SPMD style which also includes task specific operations. The approach can be generalized to more than two dimensions in an analogous way by introducing orthogonal_section() commands specifying subspaces of the task grid in which the communication is performed.

Example: LU factorization For an illustration we use the well-known LU factorization for the solution of linear equation systems $Ax = b$ which has been investigated in great detail in the past. The optimal computational structure for the LU decomposition results from a double-cyclic distribution that distributes the rows and columns of the coefficient matrix $A \in R^{n_1 \times n_1}$ cyclically among the processors [7, 8]. Each entry a_{ij} of the coefficient matrix is assigned to a task T_{ij}, $i, j = 1, \ldots, n_1$, that is responsible for the computations of this entry in the elimination steps. Each elimination step k, $1 \leq k < n_1$, consists of the following computation phases:

1. vertical_section(k): The tasks in column k of the task array cooperate to determine the global pivot element $a_{rk} = max_{k \leq j \leq n_1} |a_{jk}|$.
2. horizontal_section(): The entries of the pivot row r are distributed in the columns of the task array such that element a_{rj} is distributed to all tasks T_{ij} with $i > r$. Moreover, if $r \neq k$ the tasks T_{kj} and T_{rj} exchange elements a_{kj} and a_{rj}.
3. vertical_section(k): The tasks in column k of the task array computes the elimination factors l_k.
4. horizontal_section(): The tasks T_{jk} in column k distribute the elimination factors l_k in the corresponding rows j, $k + 1 \leq j < n_1$.
5. SPMD section: The tasks T_{ij} for $k + 1 \leq j < n_1$ compute new values for their associated entries of the coefficient matrix.

3 Mapping to orthogonal processor groups

For the mapping of the tasks to the processors, we assume that the number of processors is smaller than the number of tasks. In order to exploit the potential parallelism of orthogonal computation structures, we map the computations onto a two-dimensional processor grid of size $p_1 \times p_2$ for which we provide two different partitions that exist simultaneously. For the assignment of tasks to processors, we use parameterized mappings similar to parameterized data distributions [3, 7] which describe the data distribution for arrays of arbitrary dimension, i.e., a task is assigned to exactly one processor, but each processor might have several tasks assigned to it.

Assignment of tasks to processors A double-cyclic mapping of the two-dimensional task array to the processor grid is specified by block-sizes b_1 and b_2 in each dimension, which determine the number of consecutive rows and columns that each processor obtains of each cyclic block. For a total number of p processors, the distribution of the task array $\mathcal{T}$ of size $n_1 \times n_2$ is described by an assignment vector of the form

$$((p_1, b_1), (p_2, b_2)) \tag{1}$$

with $p = p_1 \cdot p_2$ and $1 \leq b_i \leq n_i$ for $i = 1, 2$. For simplicity we assume $n_i/(p_i \cdot b_i) \in \mathbb{N}$. To describe a double-cyclic distribution, we logically arrange the processors in a two–dimensional grid of size $p_1 \times p_2$ by specifying a grid function $\mathcal{G} : P \to \mathbb{N}^2$ where P is the set of available processors. This defines p_1 row groups R_q, $1 \leq q \leq p_1$, and p_2 column groups C_q, $1 \leq q \leq p_2$,

$$R_q = \{Q \in P \mid \mathcal{G}(Q) = (q, \cdot)\} \qquad C_q = \{Q \in P \mid \mathcal{G}(Q) = (\cdot, q)\}$$

with $|R_q| = p_2$ and $|C_q| = p_1$. The row and column groups build separate orthogonal partitions of the set of processors P, i.e.,

$$\bigcup_{q=1}^{p_1} R_q = \bigcup_{q=1}^{p_2} C_q = P \text{ and } R_q \cap R_{q'} = \emptyset = C_q \cap C_{q'} \text{ for } q \neq q'.$$

The row groups R_q and the column groups C_q are orthogonal processor groups. Using distribution vector (1), row i of the tasks $\mathcal{T}$ is assigned to the processors of a single row group $Ro(i) = R_k$ with $k = \left\lfloor \frac{i-1}{b_1} \right\rfloor \bmod p_1 + 1$. Similarly, column j is assigned to the processors of a single column group $Co(j) = C_k$ with $k = \left\lfloor \frac{j-1}{b_2} \right\rfloor \bmod p_2 + 1$. A program mapped to processors refers to the row and column groups by $Ro(i)$ and $Co(i)$, i.e., it uses the original task indices. Thus, after the mapping, the task structure is still visible and the orthogonal processor groups according to the given mapping are known implicitly.

Execution model for orthogonal processor groups The mapping of the tasks to the processors defines the computations that each processor has to perform. Computations of tasks that are outside a vertical or horizontal section are executed in SPMD style. Horizontal and vertical sections require the coordination of the participating processors. A vertical_section(k) operation is performed by all processors in column group

$Co(k)$. Similarly, a horizontal_section(k) operation is performed by all processors in $Ro(k)$. A vertical_section( ) operation is performed by all processors, but each processor is only exchanging information with the processors in the same column group. A horizontal_section() is executed analogously.

Mapping the tasks to processors may cause that a single processor executes more than one task of a specific row or column of the tasks array. Therefore, the communication operations within a horizontal or vertical section have to be transformed from a row or column oriented communication to a mixed local and global communication on the assigned processor set. For a reduction operation, for example, each processor performs the reduction for its local tasks and then participates in a global communication with the other processors in its row or column group, respectively, assuming an associative reduction operation.

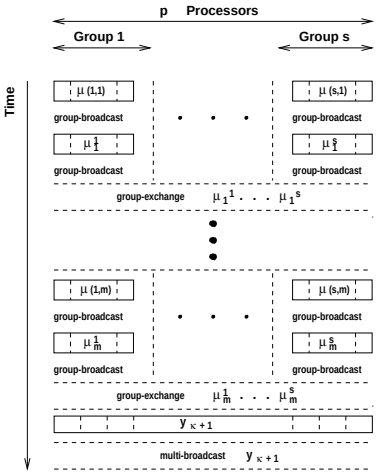
Transformation to MPI programs Each orthogonal section is translated separately into a corresponding MPI program fragment. This is possible since the specification of orthogonal sections is compositional. MPI supports the organization of the communication in orthogonal groups by the concept of communicators and group objects. The row and column group for a row i and column j of the task array $\mathcal{T}$ can be obtained by $Ro(i)$ = `RoGroup[i/b1 % p1]` and $Co(j)$ = `CoGroup[j/b2 % p2]`, if block-sizes b_1 and b_2 are used. The row (column) group of an arbitrary processor with a global rank q can be computed efficiently by defining a virtual two-dimensional process topology of size $p_1 \times p_2$ and by using `MPI_Cart_coords()` to obtain the corresponding grid position (x, y). Then `RoGroup[x]` is the corresponding row group and `CoGroup[y]` is the corresponding column group. Based on predefined row and column groups and their corresponding communicators, the definition of horizontal and vertical sections can be translated into executable MPI programs.

Example: Iterated RK methods Iterated Runge-Kutta (RK) [5, 9] methods are explicit one-step methods for the solution of systems of ordinary differential equations (ODEs) of the form $\frac{dy(x)}{dx} = f(x, y(x)), y(x_0) = y_0, x_0 \leq x \leq x_{end}$ where $y : \mathbb{R} \to \mathbb{R}^n$ is the unknown solution function and $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is an application-specific function which is nonlinear in the general case; n is the size of the ODE-system. The predefined vector $y_0 \in \mathbb{R}^n$ specifies the initial condition at x_0. An s-stage iterated RK method performs a fixed number m of iterations per time step to compute an approximation of the solution function (denoted by $y_{\kappa+1}^{(m)}$) according to the following computation scheme:

$$
\begin{aligned}
\mu_{(0)}^{l} &= f(y_\kappa) \quad \text{for } l = 1, \ldots, s \\
\mu_{(j)}^{l} &= f(y_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mu_{(j-1)}^{i}) \quad \text{for } \; l = 1, \ldots, s, j = 1, \ldots, m \qquad (2) \\
y_{\kappa+1}^{(m)} &= y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, \mu_{(m)}^{l}.
\end{aligned}
$$

The advantage of iterated RK methods for parallel execution is that the iteration system (2) of size $s \cdot n$ consists of s independent systems to determine the approximations $\mu_{(j)}^{l}$, $j = 1, \ldots, m$. These function evaluations can be performed in parallel by s independent, disjoint processor groups $G_1, \ldots, G_s$ in a programming model with mixed task

and data parallelism [6]. Group $G_l = \{q_{l,1}, \ldots, q_{l,g_l}\}$ with g_l processors is responsible for the computation of one sub-vector $\mu^l_{(j)}$, $l \in \{1, \ldots, s\}$. Figure 2 illustrates the computations for one time step.



initialize $\mu^l_{(0)} = f(y_\kappa), l = 1, \ldots, s$;
for $j = 1, \ldots, m$ do sequentially {
  forall groups $G_l, l \in \{1, \ldots, s\}$ do in parallel {
    forall processors $q \in G_l$ do in parallel {
      compute $\lceil n/g_l \rceil$ elements of
        $\bar{\mu}(l,j) = y_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mu^i_{(j-1)}$;
      make $\bar{\mu}(l,j)$ available to all processors of G_l;
      compute $\lceil n/g_l \rceil$ elements of $\mu^l_{(j)} = f(\bar{\mu}(l,j))$; }}
  forall processors q do
    exchange the computed components of $\mu^l_{(j)}$;
}
forall processors q do {
  compute $\lceil n/p \rceil$ el. of $y_{\kappa+1} = y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, \mu^l_{(m)}$;
  compute $\lceil n/p \rceil$ el. of $\hat{y}_{\kappa+1} = y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, \mu^l_{(m-1)}$;
}
make $y_{\kappa+1}$ available to all processors;
compute new stepsize;

Fig. 2. Parallel execution of a time step of the iterated RK method.

For the *special case* that all groups have exactly the same size $g = p/s$ and that p divides n, the implementation can be optimized by using orthogonal groups $Q_1, \ldots, Q_g$ with $|Q_k| = s$ and $Q_k = \{q_{l,k} \in G_l \mid l = 1, \ldots, s\}$.

The iteration steps are performed in the same way as in the general case. But since each processor computes the same number of components, the exchange of the elements of $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$ can be performed more efficiently: Instead of making all components available to each processor, group-multi-broadcast operations can be performed on $Q_1, \ldots, Q_g$ in parallel with each processor $q_{l,k}$ of Q_k contributing n/g components of $\mu^l_{(j)}$, thus making for each processor exactly those components of $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$ available that are needed for the execution of the next iteration.

4 Runtime tests on Cray T3E

Figure 3 shows speedup results for the parallel LU decomposition on a Cray T3E-1200. The diagrams compare three versions which are based on a row cyclic or column cyclic data distribution on the global group of processors with an implementation which results from the use of orthogonal groups. The orthogonal groups are used for computing the pivot element on a single column group, for making the pivot row available to all processors by parallel executions on all column groups, for computing the elimination factors on a single column group, and for broadcasting the elimination factors in the row groups. Each processor only allocates the elements of the array that it has to compute. This establishes spatial locality when a processor accesses its local elements of one row of the coefficient matrix in storage order. The processor grid is chosen such that there is

an equal number of processors in each dimension, i.e., the row and column groups have equal size. All versions result in a good load balance. The block-size in each dimension has been set to 1. Runtime tests have shown that larger block-sizes lead to (slightly) larger execution times.

For a larger number of processors, the implementation with orthogonal processor groups shows the best runtime results and Figure 3 shows that this implementation has also the best speedup values. For 16 processors and more, the global column-cyclic distribution leads to larger execution times than the global row-cyclic distribution, since for the column-cyclic distribution, both the pivot element and the elimination factors are computed by only one processor, whereas for the row-cyclic distribution, their computation is distributed among all available processors, thus leading to smaller computation times. The additional advantage of the use of orthogonal processor groups shown in Figure 3 comes from the replacement of the global communication operations by group-based operations with fewer participating processors leading to smaller execution times. This advantage increases with the number of processors because of the logarithmic dependence of the execution time of broadcast operations on the number of processors on the T3E [7]. For smaller numbers of processors, the column-cyclic distribution is competitive, because the percentage of idle processors during the computation of the pivot element and the elimination factors is relatively small and because the column-cyclic distribution requires less communication operations and therefore less startup time for these operations than the other variants. For the largest number of processors, the percentage difference of the runtime of the version with orthogonal processors groups to the second best version is about 32%.

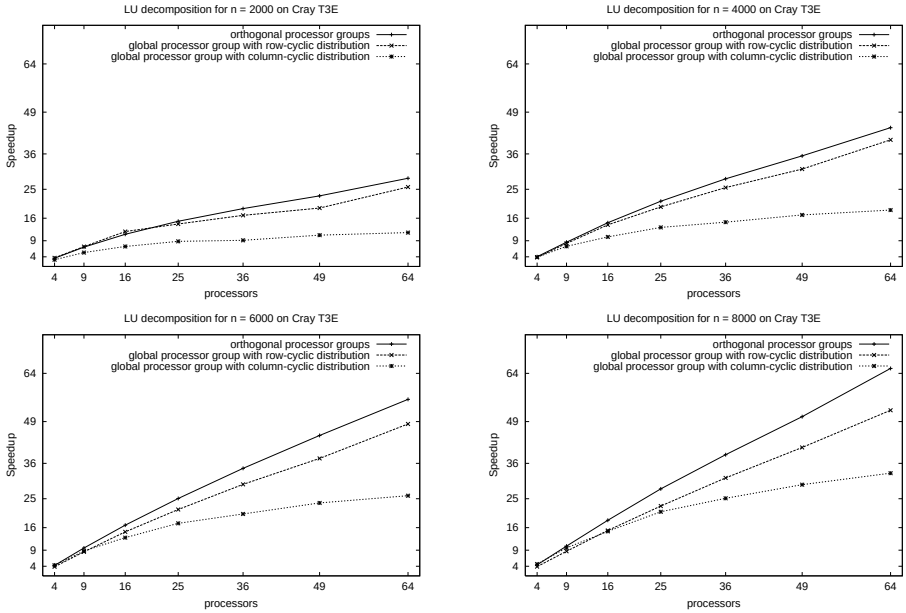


Fig. 3. Speedup values for the LU decompositions on a Cray T3E-1200.

Figure 4 shows speedup values for different parallel versions of the iterated RK method on a Cray T3E-1200. As basic RK method, a LobattoIIIC6 method with four stages has been used which results in a method of 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. 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. The figure compares a pure data parallel realization of the iterated RK method with 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 groups. The pure data parallel implementation is in general 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.

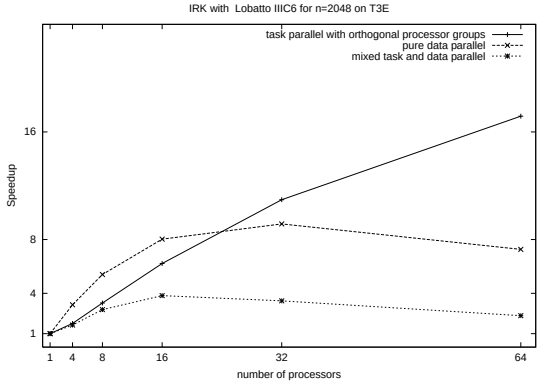


Fig. 4. Speedup values for the iterated RK method on a Cray T3E-1200.

5 Comparison to related work

Many environments for scientific computing are extensions to the HPF data parallel language. An example is HPJava[10] 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 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 programming system for the development of dynamic, nonuniform scientific computations supporting block-irregular data distributions [4]. KeLP extends LPARX to support the development of efficient programs for hierarchical parallel computers such as clusters of SMPs [1]. A KeLP program contains three programming levels: a collective level executing on the entire parallel machine, a node level that manages parallelism between SMP nodes, and a processor level capturing parallelism within a single SMP node. 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 varying partitions of the same set of processors.

6 Conclusions and future work

We have outlined a transformation approach for developing efficient parallel programs. The idea is to give the programmer the possibility to structure a program into horizontal and vertical interaction sections according to the dependencies given by the algorithm to be realized. Given this interaction specification, we propose a mapping step which introduces orthogonal groups of processors and assigns the tasks defined by the application algorithm to the processors according to a parameterized distribution that can be chosen by the programmer. The mapping and the translation to the final MPI program are designed in such a way that they can be performed automatically by a compiler system, so the programmer can concentrate on the structure of the algorithm to be implemented.

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. A. Dierstein, R. Hayer, and T. Rauber. The ADDAP System on the iPSC/860: Automatic Data Distribution and Parallelization. *JPDC*, 32(1):1–10, 1996.
4. S.R. Kohn and S.B. Baden. Irregular Coarse-Grain Data Parallelism under LPARX. *Scientific Programming*, 5:185–201, 1995.
5. T. Rauber and G. Rünger. Parallel Execution of Embedded and Iterated Runge–Kutta Methods. *Concurrency: Practice and Experience*, 11(7):367–385, 1999.
6. T. Rauber and G. Rünger. A Transformation Approach to Derive Efficient Parallel Implementations. *IEEE Transactions on Software Engineering*, 26(4):315–339, 2000.
7. T. Rauber and G. Rünger. Deriving Array Distributions by Optimization Techniques. *Journal of Supercomputing*, 15:271–293, 2000.
8. E. van de Velde. Data Redistribution and Concurrency. *Parallel Computing*, 16:125–138, 1990.
9. P.J. van der Houwen and B.P. Sommeijer. Parallel Iteration of high–order Runge–Kutta Methods with stepsize control. *J. Comp. Applied Mathematics*, 29:111–127, 1990.
10. 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 Univ., 1998.