Orthogonal Processor Groups for Message-Passing Programs *

THOMAS RAUBER
Institut für Informatik
Universität Halle-Wittenberg

ROBERT REILEIN   GUDULA RÜNGER
Fakultät für Informatik
Technische Universität Chemnitz

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 in a single parallel implementation. The parallel programming model is appropriate for grid based applications with computations and dependence pattern mainly aligned in horizontal or vertical directions. For those applications we propose a systematic development process for message-passing programs using orthogonal processor groups. The development process starts with an SPMD-like specification of tasks indicating horizontal and vertical sections. A mapping to orthogonal processor groups realizes a group SPMD execution model with different orthogonal processor groups and a final transformation generates the corresponding MPI program. In this paper, we describe the approach and present example applications which can benefit from an implementation with orthogonal processor groups. The runtimes are measured on a Cray T3E.

1 Introduction

Parallel machines with distributed memory organization are still a popular platform for the implementation of applications from scientific computing because it is now well-understood how to get portable efficient programs. The portability of message-passing programs is guaranteed due to widely accepted message-passing standards like MPI or PVM. To reach a high efficiency, the distribution of computations and data to processors has to achieve load balance and a minimal communication overhead. For typical grid-based computation structures of applications from scientific computing, the SPMD programming model usually leads to good results. But there are also applications which can benefit from a parallel programming model that allows more general but still regular communication and dependence patterns. Existing parallel programming models in this direction include extensions to HPF or mixed task and data parallel approaches which we will discuss later. In this paper, we consider a generalization of SPMD programming to orthogonal processor groups.

Therefore we focus on applications which consist of independent disjoint computations or data sets in alternating or changing directions. 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 [16, 19, 20]. Correspondingly, it is useful to arrange the processors in orthogonal partition structures so that each processor belongs to a different group for each of the partitions. Since different directions of computation may be active at different points during the execution, it is suitable to use a data distribution that is not biased towards a specific direction. Examples are blockwise data distributions or data distributions that are cyclic in each direction. The advantage is that collective communication operations performed on smaller processor groups lead to smaller executions times due to the logarithmic or linear dependence of the communication times on the number of processors. The total number of collective communication operations may increase but the message sizes are usually smaller and different communication operations can often be executed concurrently on disjoint processor groups without interference. Load balance is usually also achieved for concurrent SPMD computations. So the advantages might outperform the disadvantages, but only a real implementation can show which programming model leads to the most efficient program.

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. Moreover, the way of using MPI group operations can have an immense influence on the performance of the program. 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.

In this paper, we concentrate on applications with an orthogonal structure of potential parallelism and their realization on distributed memory machines. We propose a systematic approach to build parallel programs comprising hor-

* A shorter version of this paper is published in the proceedings of the HPCN Europe, June 2001, Amsterdam, Springer LNCS.

izontal or vertical SPMD sections in the two-dimensional case. 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. The mapping onto processor groups uses parameterized distributions and 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 contribution of the paper is to identify a class of applications exhibiting orthogonal structures and to propose a transformation approach to develop appropriate MPI-programs in a group SPMD programming model in several steps. The class of applications includes, e.g., applications resulting from a discretization of partial differential equations in the physical space using neighborhood stencils like the five-point-star. In this article, we consider two other two-dimensional problems which exhibit dependencies in the entire horizontal or vertical direction, namely the LU factorization using Gaussian elimination and special Runge-Kutta methods for solving ordinary differential equations.

The rest of the paper discusses orthogonal structures in Sections 2 and shows the mapping onto orthogonal processor groups in Section 3. Section 4 describes the transformation into an MPI program. Typical examples exhibiting orthogonal computation structures are given in Section 5. Runtime tests with orthogonal group structures are given in Section 6. Section 7 discusses related work and Section 8 concludes.

2 Orthogonal structures of computations in scientific applications

We consider scientific applications with an orthogonal internal computation structure. An orthogonal structure requires a two-dimensional (or in the general case a multi-dimensional) organization of computations. We say that an application has an orthogonal structure of computation and communication if the computations and the data dependencies exhibit regular patterns in a horizontal or vertical direction of this organization.

The organization of the computation may result from an underlying grid data structure or an array but may also result from a task structure with regular dependencies. In both cases, the description of the orthogonal structure can be interpreted as potential parallelism: Similar computations can be executed in parallel on different processors and the orthogonal data dependencies have to be realized by explicit

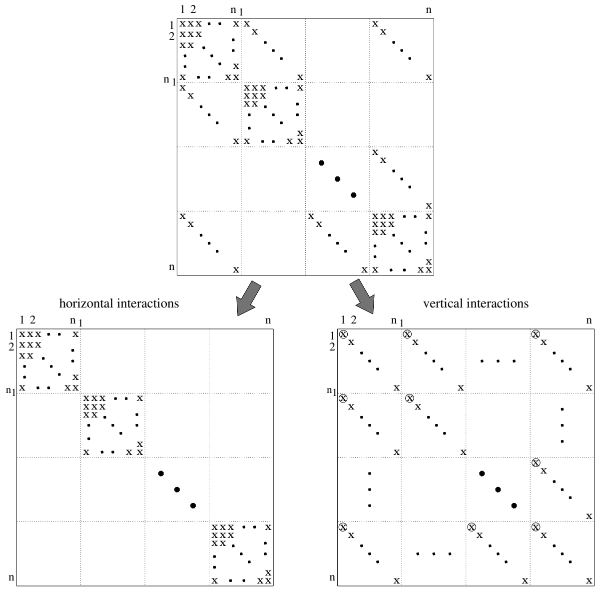

Figure 2. Interaction matrices A_T of size $n \times n$ for a task array of size $n_1 \times n_2$ with $n = n_1 \cdot n_2$, showing mixed horizontal and vertical computation structures (top) and pure horizontal and pure vertical computation structures (bottom). For the horizontal interaction matrix, the submatrices are clearly visible as n_2 blocks. For the vertical interaction matrix, the circles show items comprising one of the submatrices.

communication operations. The transformation approach of orthogonal computation structures into MPI programs is structured into three phases as described in Figure 1. In the next subsection we describe the specification of tasks with orthogonal structures for the two-dimensional case.

2.1 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 $n = n_1 * n_2$. Each single task $T \in \mathcal{T}$ consists of a sequence of computations which may need data from other tasks or which may produce data needed by other tasks.

We consider applications exhibiting orthogonal interaction structures in which 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. Those structures can be described by an *interaction matrix* A_T of size $n \times n$ with $n = n_1 * n_2$ which has one row $r = (i-1)n_2 + j$ for each task T_{ij} . The matrix A_T

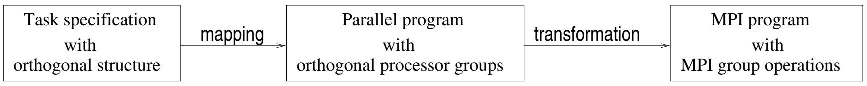

Figure 1. Transformation of orthogonal structures into MPI programs.

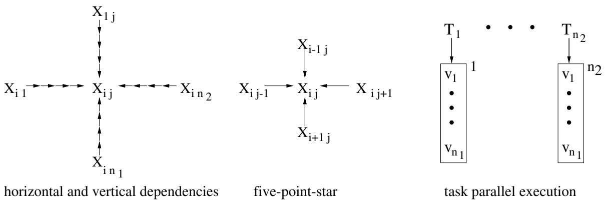

Figure 3. Data dependencies for grid based computations. The left part illustrates computations with horizontal and vertical dependencies. The five point star is a special case of computations with horizontal and vertical structures involving only the direct neighbors in both dimensions. The right part shows a task parallel execution producing vectors.

has an entry in $A_T[r,s]$, $r,s = 1,...,n$, if task T_{ij} associated with row r has a interaction with task T associated with column s. Figure 2 shows the pattern of an interaction matrix with horizontal and vertical interactions and two submatrices which show the interactions separately. Both submatrices show that the corresponding interaction graphs form n_1 or n_2 independent subgraphs. To express SPMD computations in horizontal or vertical direction, we can define the interaction matrix in a more general way, so that a filled element in the matrix can also mean that the two tasks associated with the corresponding row and column perform similar computations in an SPMD style.
Examples for orthogonal interaction structures can be found in grid-based computations or applications with mixed task and data parallelism:

a) Grid-based computation on grid points x_{ij}, $i = 1,\ldots,n_1, j = 1,\ldots,n_2$: Each grid point x_{ij} gives rise to a task T_{ij} computing a new value x_{ij}^{new} at the grid point. Figure 3 (left) illustrates horizontal and vertical data dependencies.

b) Task parallelism for tasks on multidimensional data: An application with task parallelism consists of tasks T_j, $j = 1,...n_2$, which are independent and can be executed in parallel. Each of the tasks executes on data or produces data $v_1,\ldots,v_{n_1}$, so that also a two-dimensional computation structure results, see Figure 3 (right) for an illustration of a corresponding vertical computation structure.

The entire task program is expressed in an SPMD style with explicit constructs 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 $\{T_{i'j} | i' = 1,...,n_1, i' \neq i\}$.

A single task T_{ij} is composed of communication and computation commands. To indicate that a task participates in a SPMD-like operation together with other tasks in horizontal or vertical direction, respectively, we introduce specific commands with the following meaning:

- 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 uses 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 other 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. Thus, vertical_section( ) corresponds to a parallel loop over all columns k with each iteration executing vertical_section(k).

- horizontal_section( ) { statements } :
  Similar to vertical_section( ), but uses a horizontal organization.

Figure 4 shows an illustration. 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 commands orthogonal_section() specifying subspaces of the task grid in which the communication is performed.

2.2 Example: LU factorization

LU factorization is a well-known method which can be used for the solution of linear equation systems $Ax = b$ and which has been investigated in great detail in the past, see

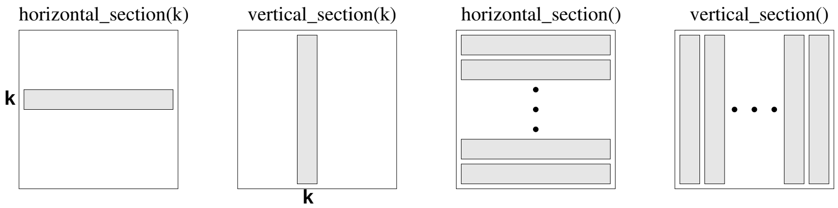

Figure 4. Illustrations of the commands *horizontal_section(k)*, *vertical_section(k)*, *horizontal_section()*, and *vertical_section()*.

the book of Van de Velde [20] and the references therein. This example is useful for two reasons: First, an optimal computational structure has already been derived by other approaches, see, e.g., the book of Fox et al. [8] and the work of Van de Velde [19], and has been verified by experiments on various parallel machines. Second, finding an optimal computational structure for LU decomposition is not trivial as both the communication overhead and load balancing issues have to be taken into consideration.

The optimal computational structure for LU decomposition has been found to result from a double-cyclic distribution that distributes the rows and columns of the coefficient matrix $A \in R^{n \times n}$ cyclically among the processors [16, 19]. Each entry a_{ij} of the coefficient matrix is assigned to a task T_{ij} that is responsible for the computations of this entry in the different elimination steps $(i, j = 1, \ldots, n)$. Each elimination step k, $1 \leq k < n$, 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 $a_{rk} = max_{k \leq i \leq n} |a_{ik}|$.
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_{ik} in column k distribute the elimination factors l_k in the corresponding rows i, $k+1 \leq i, j < n$.
5. SPMD section: The tasks T_{ij} for $k + 1 \leq j < n$ compute new values for their associated entries of the coefficient matrix.

3 Mapping to orthogonal processor groups

In this section, we describe the mapping of the tasks to the processors. For the mapping, we assume that the number of tasks is smaller than the number of processors. In order to exploit the potential parallelism of orthogonal computation structures, we map the computations to a two-dimensional processor grid for which we provide two different partitions that exist simultaneously and can be exploited to reduce the communication overhead. For the assignment of tasks to processors, we use parameterized mappings similar to parameterized data distributions [5, 16] which describe the data distribution for arrays of arbitrary dimension. This mechanism is used to define disjoint row and column groups such that each row and column of the task array is assigned to a single row and column group respectively. A mapping uniquely assigns tasks to processors for the entire program, i.e., a task is assigned to exactly one processor, but each processor might have several tasks assigned to it. In the following we describe the mechanism for two-dimensional task-arrays.

3.1 Assignment of tasks to processors

For the parameterized mapping, the processors are logically arranged in a two-dimensional processor grid that can be described by the number p_1 and p_2 of processors in each dimension. A double-cyclic mapping of the two-dimensional task array to the processor grid is specified by blocksizes 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

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

with $|R_q| = p_2$ for $1 \leq q \leq p_1$ and p_2 column groups

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

with $|C_q| = p_1$ for $1 \leq q \leq p_2$. The row and column groups build separate orthogonal partitions of the set of processors

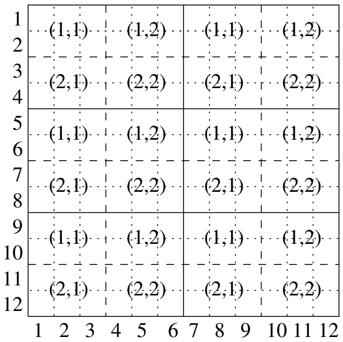

Figure 5. Illustration of a mapping of a task array to processors in the two-dimensional case.

P, i.e.,

$$\bigcup_{q=1}^{p_1} R_q = \bigcup_{q=1}^{p_2} C_q = P$$

and $R_q \cap R_{q'} = \emptyset = C_q \cap C_{q'}$ für $q \neq q'$. The row groups R_q and the column groups C_q are orthogonal processor groups. Figure 5 shows an example for a mapping of a task array of size 12×12 to a processor grid of size 2×2 using block sizes $b_1 = 2$ and $b_2 = 3$.. The row and column processor groups are $R_1 = \{(1,1),(1,2)\}$, $R_2 = \{(2,1),(2,2)\}$, $C_1 = \{(1,1),(2,1)\}$, and $C_2 = \{(1,2),(2,2)\}$.

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.

3.2 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 leads to the fact that a single processor may execute 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. To perform a reduction operation, for example, each processor performs the reduction for its local tasks and then participates in a communication with the other processors in its row or column group, respectively, assuming an associative reduction operation.

The resulting execution model for orthogonal processor groups has the following characteristics:

- Processors are organized in a two- or multi-dimensional grid structure depending on the mapping.
- Processor groups are supported in each direction of the processor grid, i.e., if processors are described by s-tuples then up to $s-1$ values of the tuple can be fixed to build one processor group.
- Multiple program execution controls, one for each group, can exist, i.e., one or all groups of a single partition can be active.
- Interactions between concurrently working groups are possible and easy to realize.
- The connection between the group structure and the task structure is preserved also after the mapping to the processor groups. The indices of the tasks are retained in the final program and the execution of the processors is directed by appropriate functions to identify the row and column groups corresponding to the tasks to be executed.
- The entire program is executed in an SPMD style, but the section statements pick subgroups to work and communicate together within a part of the program. For those parts, the execution model is a group SPMD model. The structure of the groups is predefined by orthogonal processor groups.

4 Transformation to MPI programs

We now consider the translation of the orthogonal sections into MPI programs using the mapping of the tasks to the processors from the previous section. Each orthogonal section is translated separately according to its compositional structure.

4.1 Support of orthogonal groups in MPI

MPI supports the organization of the communication in orthogonal groups by the concept of communicators and group objects. Starting from the default communicator, row and column group communicators can be constructed by using `MPI_Split()`. The groups are indexed separately from 0 ...p1−1 and 0 ... p2−1, respectively. Each processor calculates its own index variables `MyRowGroup` and `MyColGroup` and calls `MPI_Split()` with it to construct the corresponding communicators `MyRowComm` and `MyColComm`. In addition the MPI processor id `me` is used as the "color" parameter to determine the rank inside the group.

The row and column group for a row i and column j of the task array T can be obtained by $Ro(i)$ = `RoGroup[i/b1 % p1]` and $Co(j)$ = `CoGroup[j/b2 % p2]` assuming that the arrays `RoGroup` and `CoGroup` contain all row and column groups as opaque objects, respectively, and blocksizes 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. The communication partner of a specified processor q is the processor in a given row group $Ro(k)$ that belongs to the same column group as q. This partner can be computed by calculating the group intersection of $Ro(k)$ and `MyColGroup` yielding the requested processor. An alternative way is to determine the rank of q in its own row group `MyRowGroup` which is identical with the rank of the communication partner in $Ro(k)$.

4.2 Translation of horizontal and vertical sections into MPI

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. We consider vertical sections as example, the translation of horizontal sections can be done analogously.

A `vertical_section(k)` has to be executed only by the processors of $Co(k)$. This can be achieved by a conditional statement which tests whether the executing processor belongs to $Co(k)$. For the communication operation within the vertical section, the corresponding communicator is used, thus restricting the communication to one column group.

A `vertical_section()` is executed by all processors but the collective communication operation are executed concurrently using the assigned column group communicators. This is achieved by letting each processor execute the section and by restricting the collective communication operation in which each processor participates to that column group to which the processor belongs.

For asymmetric communication operations with a root processor, it might be necessary to determine a root processor q which belongs to a specific row group $Ro(\cdot)$. This can be achieved by intersecting $Ro(\cdot)$ with the column group of

```
struct {double val;int pvtline;} z,y;

for(k=0;k<n-1;k++) {
  if(Co(k) == MyColGroup) {
    r=max_col_loc(a,k);
    z.pvtline=r;z.val=abs(a[r][k]);
    MPI_Allreduce(&z,&y,1,MPI_DOUBLE_INT
      ,MPI_MAXLOC,MyColComm);
  }
  q=intersect(MyRowGroup,Co(k));
  MPI_Bcast(&y,1,MPI_DOUBLE_INT,q
    ,MyRowComm);
  r=y.pvtline;
  if(Ro(k) == Ro(r)) {
    /* pivot row and row k
       in same row group */
    if(Ro(k) == MyRowGroup) {
      if(r != k)
        exchange_row_loc(a[r],a[k]);
      copy_row_loc(a[k],pvt_buf);
    }
  } else
    /* pivot row and row k
       in different row groups */
    if (Ro(k) == MyRowGroup) {
      q = compute_partner(Ro(r),me);
      MPI_Send(a[k],n,MPI_DOUBLE,q
        ,tag,MyColComm);
    } else
      if(Ro(r) == MyRowGroup) {
        q=compute_partner(Ro(k),me);
        MPI_Recv(pvt_buf,n,MPI_DOUBLE
          ,q,tag,MyColComm);
        exchange_row_loc(a[r],pvt_buf);
      }
  q=intersect(MyColGroup,Ro(r));
  MPI_Bcast(pvt_buf,n,MPI_DOUBLE,q
    ,MyColComm);
  if(Ro(k)!=Ro(r) && Ro(k)==MyRowComm)
    copy_row_loc(pvt_buf,a[k]);
  if(Co(k) == MyColGroup)
    l_buf=l_fact_loc(k,a,buf);
  q=intersect(MyRowGroup,Co(r));
  MPI_Bcast(l_buf,lbuf_size,MPI_DOUBLE
    ,q,MyRowComm);
  transform_rows(k,a,l_buf,pvt_buf);
}
```

Figure 6. MPI-program of a group-oriented LU factorization with group operations.

the executing processor. A more efficient implementation which avoids the intersection of groups can be based on the computation of row and column groups with functions in closed form, see Section 3.

5 Examples

As examples, we consider the LU factorization from Subsection 2.2 and iterated Runge-Kutta methods for the solution of non-stiff ordinary differential equations.

5.1 Double-cyclic realization of LU factorization

Figure 6 shows the structure of the group-based MPI program which results from a double-cyclic mapping of the computations to a processor grid and from a translation of the computation phases from Section 2.2. In step k, column k of A is distributed among the processors of a column group $Co(k)$ which compute the global pivot element $a_{rk}^{(k)}$ by a group-based single-accumulation operation with a maximum reduction. If necessary, row r and k are exchanged between the corresponding processors of row groups $Ro(r)$ and $Ro(k)$. The pivot row is broadcasted to all processors by group-based single-broadcast operations on the column groups. The elimination factors l_{ik} are computed by the processors in column group $Co(k)$ by `l_fact_loc()`. The factor l_{ik} is distributed to the processors in $Ro(i)$ by a group-based single-broadcast operation. Each processor computes the new values for its local elements of $A^{(k)}$.

The program uses the following functions. The function `compute_partner(Ro(r),me)` computes the communication partner of the executing processor in `Ro(r)` for the exchange of the pivot row. This is the processor in `Ro(r)` that belongs to the same column group as `me`. The function `l_fact_loc(k)` computes the elimination factors for all elements of the executing processor. The computation of the matrix elements is performed by `transform_rows()`.

5.2 Iterated Runge-Kutta methods

Iterated Runge-Kutta (RK) [21] methods are explicit one-step methods for the solution of systems of non-stiff ordinary differential equations (ODEs) of the form

$$\frac{dy(x)}{dx} = f(x, y(x)), \quad y(x_0) = y_0, \quad x_0 \leq x \leq x_{end} \quad (2)$$

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 non-linear 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. The solution method computes approximation vectors $y_\kappa = \tilde{y}(x_\kappa)$ for the exact values $y(x_\kappa)$ at discrete x-values $x_\kappa = x_{\kappa-1} + h$ one after another. The value h is the *stepsize*. RK methods are useful for the solution of time-dependent partial differential equations since a spatial discretization leads to a system of ODEs. There is a large variety of RK methods that can be used for ODEs with different characteristics. We consider iterated RK methods in the following, since, compared to traditional RK methods, they have the advantage to offer different levels of parallelism that can be exploited for a parallel implementation [15, 21]. The methods are derived from s-stage implicit RK methods by iterating the resulting non-linear equation system for a fixed number of times m producing approximations $\mu_{(j)}^l$, $j = 1, \ldots, m$ of the stage vectors. The last approximations $\mu_{(m)}^l$ are used for the computation of the next approximation of the solution function (denoted by $y_{\kappa+1}^{(m)}$) according to the following computation scheme:

$$\begin{aligned} \mu_{(0)}^l &= f(y_\kappa), \\ \mu_{(j)}^l &= f(y_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mu_{(j-1)}^i), \qquad (3) \\ y_{\kappa+1}^{(m)} &= y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, \mu_{(m)}^l . \end{aligned}$$

for $l = 1, \ldots, s$ and $j = 1, \ldots, m$. Vector $b = (b_1, \ldots, b_s)$ and matrix $a = (a_{li})_{i,l=1,\ldots,s}$ contain the coefficients of the specific RK method used. Embedded solutions result when using approximations $\mu_{(j)}^l$ of the stage vectors k_l with $j < m$:

$$y_{\kappa+1}^{(m-1)} = y_\kappa + h_\kappa \sum_{l=1}^{s} b_l \, \mu_{(m-1)}^l .$$

One time step of an s-stage iterated RK method requires $s \cdot m + 1$ evaluations of f. The advantage of iterated RK methods for parallel execution is that the iteration system (3) 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 $G_1, \ldots, G_s$. Group $G_l = \{q_{l,1}, \ldots, q_{l,g_l}\}$ with g_l processors is responsible for the computation of one subvector $\mu_{(j)}^l$, $l \in \{1, \ldots, s\}$. Figure 7 illustrates the computations for one time step. Because each function evaluation requires the same amount of computational work, all groups should have about the same number of processors to guarantee a good load balance. This can be achieved best, if s divides the number of processors p.

A sketch of a parallel implementation is given in Figure 8. In each iteration step j, $j = 1, \ldots, m$, each processor $q \in G_l$ computes $\lceil n/g_l \rceil$ components of the argument vector $\tilde{\mu}(l, j)$ and evaluates $\lceil n/g_l \rceil$ components of the function f. Between these two steps, $\tilde{\mu}(l, j)$ has to be made available to all processors of G_l by a group-based multi-broadcast operation, because the evaluation of f may require all components of the argument vector. After the computation of $\mu_{(j)}^l$ by the processors of G_l, the approximations computed have

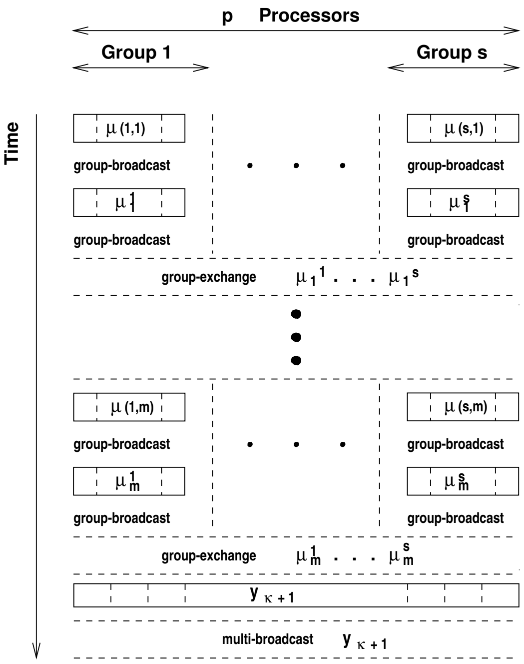

Figure 7. Schematic view of one time step of the parallel iterated RK method using s groups.

initialize $\mu^l_{(0)} = f(y_\kappa), l = 1, \ldots, s$;
for $\quad j = 1, \ldots, m$ 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
        $\tilde{\mu}(l, j) = y_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mu^i_{(j-1)}$;
      make $\tilde{\mu}(l, j)$ available to all processors of G_l;
      compute $\lceil n/g_l \rceil$ elements of $\mu^l_{(j)} = f(\tilde{\mu}(l, j))$; }}
  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;

Figure 8. Pseudo-code structure of the parallel iterated RK method using a group distribution.

to be made available to all processors by a global multi-broadcast operation, since the corresponding components have to be available for the computation of the argument vector in the next iteration step.

For the special case that all groups have exactly the same size $g = p/s$ and that p divides n, an efficient implementation can be obtained by using orthogonal groups. The implementation is based on the definition of orthogonal groups $Q_1 \ldots, Q_g$ with $|Q_k| = s$ and

$$Q_k = \{q_{l,k} \in G_l \mid l = 1, \ldots, s\},$$

see Figure 9. For the initialization, each processor computes n/p components of $f(y_\kappa)$ such that the processors of Q_k compute exactly those components that are needed for the initialization of their local components of $\mu^1_{(0)}, \ldots, \mu^s_{(0)}$. Thus, each group Q_k computes n/g components of $f(y_\kappa)$. Since each processor of Q_k needs all these components, they are made available by a group-multi-broadcast operation that is performed on $Q_1 \ldots, Q_g$ in parallel.

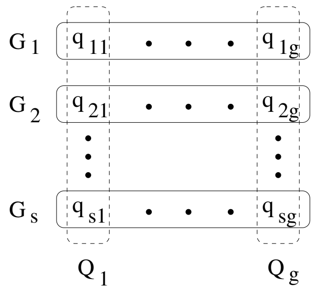

Figure 9. Disjoint processor groups $G_1, \ldots, G_s$ and orthogonal processor groups $Q_1, \ldots, Q_g$. Processors $q \in P$ can be arranged in a processor grid defined by indices q_{lk}, $l = 1, \ldots, s$, $k = 1, \ldots, g$. Processor q_{lk} has position (l, k) in the grid.

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 needed for the execution of the next iteration. After the last iteration, each processor of Q_k has n/g corresponding components of $\mu^1_{(j)}, \ldots, \mu^s_{(j)}$, i.e., each of the s processors of Q_k can compute a different part of size n/p of $y_{\kappa+1}$ and $\hat{y}_{\kappa+1}$ without preceding communication, see Figure 10. In particular, part i can be computed by processor $q_{i \bmod (s+1), \lceil i/g \rceil}$ of $Q_{\lceil i/g \rceil}$.

6 Runtime tests on Cray T3E

Figure 11 shows runtime results for the LU decomposition on a Cray T3E-1200. The diagrams compare three versions which are based on a row or column cyclic data distribution on the global group of processors with an implementation which results from the use of orthogonal groups. The

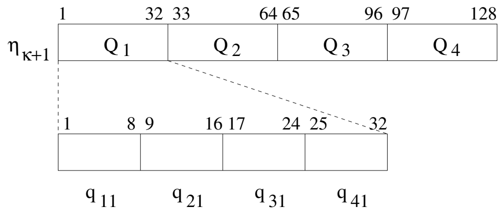

Figure 10. Distribution of $y_{\kappa+1}$ for $n = 128$, $p = 16$ und $s = g = 4$. Each group Q_k computes blocks of size 32, each single processor computes blocks of size 8.

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 a 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 blocksize in each dimension has been set to 1. Runtime tests have shown that larger blocksizes lead to (slightly) larger execution times.

Figure 11 shows that for a larger number of processors, the implementation with orthogonal processor groups shows the best runtime results. 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 11 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 [17]. 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 20 %.

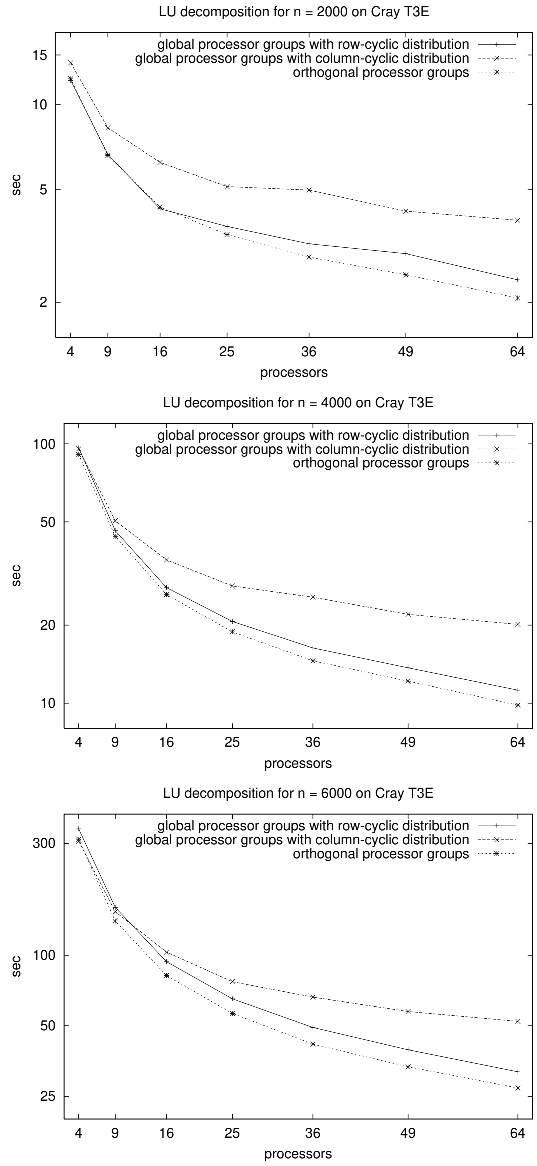

Figure 11. *Runtimes for the LU decomposition on a Cray T3E-1200.*

Figure 12 compares the runtimes of different versions of the iterated RK method on a Cray T3E-1200. As basic RK method, a LobattoIIIC6 method with four stages has been used. This results in a method of order six and 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 [9]. 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

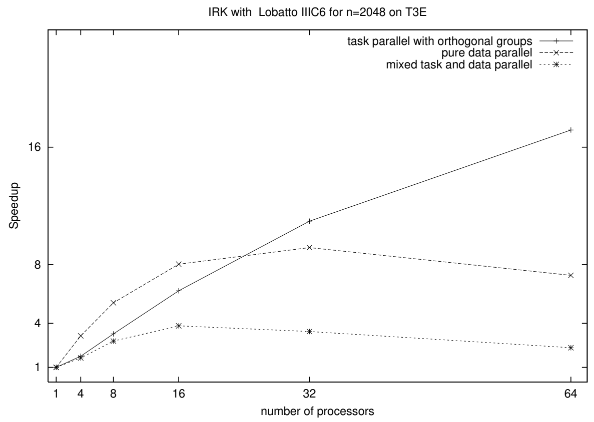

Figure 12. Runtimes of the iterated RK method on a Cray T3E-1200.

hand side function f of the ODE system is a sparse function, i.e., the solution 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 based on the definition of orthogonal groups. The figure shows that 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 starting with 32 processors, the orthogonal processor groups implementation is getting better than the data parallel implementation and for 64 processors, it is twice as fast.

7 Comparison to related work

Related work to orthogonal processor groups come from different research directions, including programming models and software support for scientific computing, parallel languages and libraries, and mixed task and data parallelism.

Many environments for scientific computing are extensions to the HPF data parallel language. A good overview can be found in [6]. An example is HPJava which adopts the data distribution concepts of HPF but uses a high level SPMD programming model with a fixed number of logical control threads and includes collective communication operations encapsulated in a communication library. A language description is given in [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, non-uniform scientific computations supporting block-irregular data distributions [11]. KeLP extends LPARX to support the development of efficient programs for hierarchical parallel computers such as clusters of SMPs [1, 6]. A complete 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 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]. Processor groups are not considered explicitly and the programmer is responsible for organizing the communication structure between the HPF tasks.

Data parallel regular parts and a more irregular coordination level are separated in approaches for mixed task and data parallelism like Fx [18], Paradigm [3, 10, 13], Fortran M [7], and TwoL [14], see [2, 4] for an overview of other approaches. In models with mixed task and data parallelism, a program is described in the form of a task dependence graph or a corresponding coordination program. Internally tasks are realized in an SPMD or data parallel way. Tasks execute independently to other tasks but can have input-output relations with other tasks for multidimensional data. At a fixed point in the execution time, the processors execute in a group SPMD model.

Mixed task and data parallelism is suited for application with coarse to medium grained potential parallelism as the overhead for redistributions of data, multiprocessor task scheduling and building groups is too large otherwise. In the group SPMD execution model for orthogonal processor groups, the groups are predefined for the entire program and are easy to access, so that the resulting overhead is comparably small.

8 Conclusions and future work

In this article we have outlined a new strategy 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 organized in such a way that they can be performed automatically by a compiler system, i.e., the programmer can concentrate on the structure of the algorithm to be implemented. We have shown that an exploitation of orthogonal interaction structures can be useful for LU factorization and for iterated Runge-Kutta methods. In both cases, the runtime of the resulting program is considerably reduced compared to the implementations that do not exploit the orthogonal interaction structure, but already have shown good speedups.

Another example for the use of orthogonal interaction structures is the implementation of implicit Runge-Kutta methods for stiff ODE equations which require the solution of non-linear 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, non-linear 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. For this kind of applications the approach can be generalized to more than two dimensions or to nested orthogonal processor groups.

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] P. Banerjee, J. Chandy, M. Gupta, E. Hodge, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, 1995.

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

[5] A. Dierstein, R. Hayer, and T. Rauber. The ADDAP System on the iPSC/860: Automatic Data Distribution and Parallelization. *Journal of Parallel and Distributed Computing*, 32(1):1–10, 1996.

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

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

[8] G. Fox, R. Williams, and P. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, 1994.

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

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

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

[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, 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.

[14] T. Rauber and G. Rünger. The Compiler TwoL for the Design of Parallel Implementations. In *Proc. 4th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'96)*, IEEE, pages 292–301, 1996.

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

[16] T. Rauber and G. Rünger. Deriving Array Distributions by Optimization Techniques. *Journal of Supercomputing*, 15:271–293, 2000.

[17] T. Rauber and G. Rünger. Modelling the Runtime of Scientific Programs on Parallel Computers. In *Proc. Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA-00)*, Toronto, Kanda, August 2000.

[18] 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.

[19] E. van de Velde. Data Redistribution and Concurrency. *Parallel Computing*, 16:125–138, 1990.

[20] E. van de Velde. *Concurrent Scientific Computing*. Springer, 1994.

[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.