Scientific Programming 15 (2007) 137–155
IOS Press

Mixed task and data parallel executions in general linear methods

Thomas Rauber[a,*] and Gudula Rünger[b]
[a]*University Bayreuth, Germany*
[b]*Chemnitz University of Technology, Germany*
E-mail: ruenger@informatik.tu-chemnitz.de

Abstract. On many parallel target platforms it can be advantageous to implement parallel applications as a collection of multiprocessor tasks that are concurrently executed and are internally implemented with fine-grain SPMD parallelism. A class of applications which can benefit from this programming style are methods for solving systems of ordinary differential equations. Many recent solvers have been designed with an additional potential of method parallelism, but the actual effectiveness of mixed task and data parallelism depends on the specific communication and computation requirements imposed by the equation to be solved. In this paper we study mixed task and data parallel implementations for general linear methods realized using a library for multiprocessor task programming. Experiments on a number of different platforms show good efficiency results.

Keywords: General linear methods, ordinary differential equations, task parallelism, data parallelism, orthogonal structures of communication

1. Introduction

Scalability problems associated with large parallel systems can often be avoided by exploiting mixed task and data parallelism with multiprocessor tasks (M-tasks). For an implementation using M-tasks the application program is split into modules which are realized as M-Tasks and which can be executed concurrently. Internally M-tasks can also be executed in parallel: this can be done in a fine-grained SPMD or data parallel way and also in a more general way.

M-task parallelism on top of fine-grained parallelism often results in improved efficiency in cases where collective communication is required for global data exchange; in this case a pure data parallel implementation often imposes a large communication overhead for larger numbers of processors, whereas the mixed task and data parallel implementation restricts communication to a subset of the processors. The improvement of the communication in M-task programming results from the restriction of the communication to smaller groups of processors. This can be achieved when several independent tasks work, independently, on subsets of the entire set of processors; thus collective communication inside an M-task is performed on a smaller set of processors, while the entire application uses the entire set of processors. The reason for this behavior is that the cost of communication depends on the number of participating processors in a linear or logarithmic way. The actual effect of exploiting mixed parallelism on performance improvements, however, depends on the specific application problem and the specific communication behavior of the target machine. A class of applications which can benefit from mixed parallelism are methods for solving ordinary differential equations (ODEs). One-step ODE solvers perform a series of time steps which have to be executed sequentially due to data dependencies. Potential parallelism inside each time step arises when systems of ODEs are solved and collective communication is required when

*Corresponding author. University Bayreuth, Angewandte Informatik II, 95440 Bayreuth, Germany. Tel.: +49 921 555100; Fax: +49 921 555102; E-mail: rauber@uni-bayreuth.de.

ISSN 1058-9244/07/$17.00

exploiting this fine-grain parallelism. In such situations it is worthwhile considering M-task parallelism.

For the integration of initial value problems for first-order ODEs, several new parallel solution methods have been proposed which offer the potential for method parallelism in each time step. Many of these ODE solvers are based on multistage methods, offering potential for method parallelism in the stage vector computation. Method parallelism allows the parallel execution of the solution method using a number of processors coincident with the number of stages in the methods – usually fewer than ten. In the case of systems of ODEs there is an additional source of system parallelism in each stage vector computation corresponding to the size of the system. The exploitation of this source of parallelism allows the employment of a larger number of processors and improves scalability. Examples are iterated Runge-Kutta (RK) methods [17, 24] and diagonal-implicitly iterated Runge-Kutta (DIIRK) methods [18,25]. The efficiency of iterated RK solvers is heavily affected by communication since a global exchange is necessary after each iteration step of the stage vector computation and strongly depends on the characteristics of the ODE system to be solved: for sparse ODE systems, the communication bandwidth of the target platform may limit performance. Experiments on different target platforms also show that a pure data parallel execution often results in lower execution times than parallel execution that exploits method parallelism [19]. On the other hand, parallel DIIRK methods for solving stiff ODEs show good efficiency for sparse and dense ODE systems on a wide variety of target platforms. Moreover, on most platforms, method parallelism results in significantly faster execution than pure data parallel implementations.

In this paper, we explore another class of solution methods which have different communication requirements. In particular, we consider a variant of general linear methods – the parallel Adams methods proposed by van der Houwen and Messina in [23]. Parallel Adams methods have the advantage that the computations in the parallel stages within each time step are completely independent. Strong data dependencies occur only at the end of each time step. We have implemented this method using multiprocessor tasks (M-tasks) for stage vector computations exploiting system parallelism internally within each M-task. We investigate the trade-off between reduced communication between M-tasks and increased communication after each time step. The implementation is designed for arbitrary right hand sides of the ODE. Thus no application specific characteristics are exploited and the pure effect of using M-tasks can be observed. As applications we consider both dense and sparse ODE systems. Experiments have been performed on a Cray T3E, two Beowulf clusters with different interconnection networks, and an IBM Regatta system.

M-task programs are implemented using the Tlib library, which can be used to hierarchically organize M-task programs on top of SPMD modules [21]. The advantage is that different parallel implementations can be easily developed on top of available stage vector SPMD codes without incurring an additional overhead. Several other models have been proposed for mixed task and data parallel executions, see [2,22] for an overview of systems and approaches and see [4] for a detailed investigation of the benefits of combining task and data parallel execution. Many environments for mixed parallelism in scientific computing are extensions of the HPF data parallel language – see [5] for an overview. Examples are HPJava [26], LPARX [11], KeLP [14] and NestStep [10].

The contribution of this paper is to present a parallel programming model for mixed task and data parallelism as well as to investigate parallel implementations of parallel Adams methods using this model. For the parallel Adams methods we show that the potential parallelism within the method can be exploited to achieve efficient parallel implementations. The development of several parallel versions using different programming models shows that a rigorous study of parallelism can facilitate the construction of a program version with good speedups for ODE solvers. On the other hand, the parallel Adams method is a good examplar for demonstrating the applicability of mixed task and data parallelism. We advocate a specific parallel programming model which uses cooperating multiprocessor tasks (M-tasks) to realize the coarse structure of parallel applications. This programming model is suitable for expressing the natural modular structure of many numerical applications, since they are usually built from different cooperating methods. Also, efficiency gains are observed which can be explained by effects on collective communication when reducing the number of participating processors. We also present an approach to exploit orthogonal group arrangements for communication. This approach can be exploited in all numerical software which use several array data structures requiring a component-wise data exchange. The parallel Adams methods in a task parallel version exhibit this kind of data dependence and thus can benefit from orthogonal communication structures.

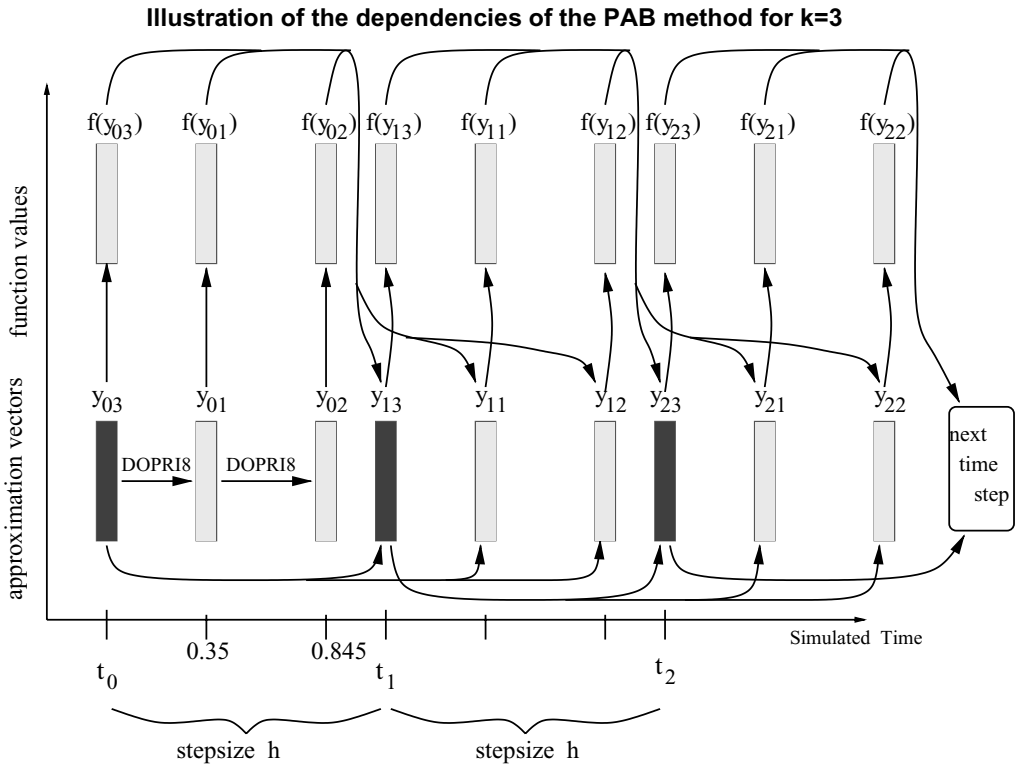

Fig. 1. Illustration of the dependence structure of a PAB method with $k = 3$ stage values. For abscissae values $\mathbf{a} = (a_1, \ldots, a_k)$ with $a_k = 1$ the stage values y_{nk} correspond to the approximations of $y(t_n)$, $n = 0, 1, 2, \ldots$.

The remainder of the paper introduces the parallel Adams method in Section 2. Section 3 discusses the implementation of the parallel Adams methods with mixed parallelism and Section 4 introduces the Tlib library. Section 5 presents runtime results, Section 6 discusses related work, and Section 7 comments on the effectiveness of the proposed approach.

2. Parallel Adams method

We consider a system of initial-value problems (IVPs) of first order differential equations (ODEs)

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0 \tag{1}$$

with initial vector $\mathbf{y}_0$ at start time t_0, system size $d \geqslant 1$, and right hand side function $\mathbf{f} : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$. One-step methods for solving ODE systems of the form (1) start with $\mathbf{y}_0$ and generate a sequence of approximations $\mathbf{y}_n$, $n = 1, 2, \ldots$, for the solution $\mathbf{y}(t_n)$ at time t_n, $n = 1, 2, \ldots$. Parallelism can only be exploited within the computations of each time step, since the time steps depend on each other. This can be realized by distributing the computation of components of $\mathbf{y}_n$ or by exploiting specific characteristics of potential method parallelism.

In recent years solvers have been designed that additionally provide a higher degree of method parallelism while guaranteeing good numerical properties. Many of these are based on classical implicit Runge-Kutta methods in which the implicit equation is handled using a predictor-corrector approach or fixed point iterations. These computations include the evaluation of several stage vectors for which method parallelism can be exploited. However, several interactions between different computation parts are required. Another class of parallel solvers is based on implicit multistage methods which are already parallel in the stage vector computation and which can be described by the class of general linear methods [3]. In each time step, general linear methods compute several stage values $\mathbf{y}_{ni}$ (vectors of size d) corresponding to numerical approximation of $\mathbf{y}(t_n + a_i h)$ with abscissa vector (a_i), $i = 1, \ldots, k$ and stepsize $h = t_n - t_{n-1}$. The stage values of one time step are combined in the vector $\mathbf{Y}_n = (\mathbf{y}_{n1}, \ldots, \mathbf{y}_{nk})$ of length $d \cdot k$ and the computation in each step is given by:

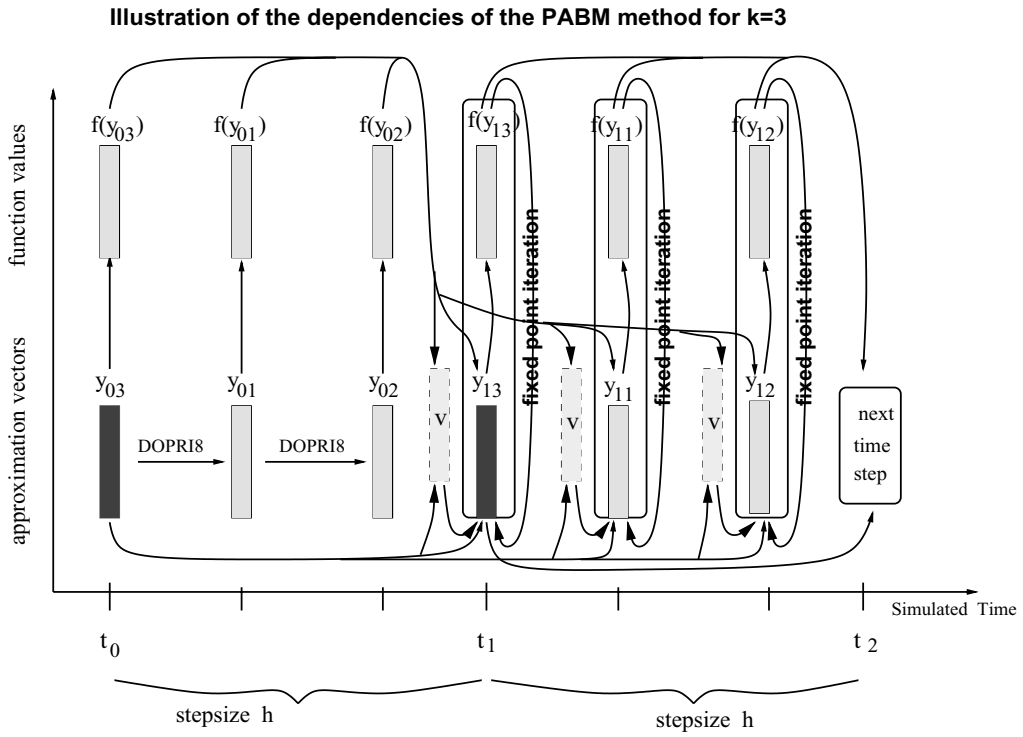

Fig. 2. Illustration of the dependence structure of a PABM method with $k = 3$ stage values. The stage value y_{ni}, $i = 1, \ldots, k$, computed by the PAB method is used as starting vector for the fixed point iteration of the PABM method. For each stage value, an additional vector v_{ni} is needed which constitutes the constant part of the right hand side of the fixed point iteration. The computation of v_{ni} is similar to the computation of y_{ni}, but with a different matrix S.

Pseudocode PAB — one time step:

```
(1)  for ( i=0; i<k; i++ )
(2)      for (e=0;e<d/p;e++ )
(3)          fy_{n+1,i}[l] = f_l(t, y_{n,i});
(4)  for ( i=0; i<k; i++ )
(5)      for (e=0;e<d/p;e++ )
(6)          y_{n+1,i}[l] = sum_{j=1}^{k} r_ij y_nj[l] + h * sum_{j=1}^{k} s_ij fy_{n+1,j}[l];
(7)      exchange components of y_{n+1,i}[l] by multi-broadcast(p,d/p);
```

Fig. 3. Pseudocode for one time step of the PAB method implemented in a data parallel way.

$$\begin{aligned}\mathbf{Y}_{n+1} = (\mathbf{R} \otimes \mathbf{I})\mathbf{Y}_n + h(\mathbf{S} \otimes \mathbf{I})\mathbf{F}(\mathbf{Y}_n) \\ + h(\mathbf{T} \otimes \mathbf{I})\mathbf{F}(\mathbf{Y}_{n+1}), \quad n = 1, 2, \ldots\end{aligned} \tag{2}$$

The matrices $\mathbf{R}$, $\mathbf{S}$ and $\mathbf{T}$ have dimension $k \times k$ and $\mathbf{R} \otimes \mathbf{I}$, for example, denotes the Kronecker tensor product, i.e. the $d \cdot k \times d \cdot k$ dimensional block matrix with $d \times d$ blocks $r_{ij} \cdot \mathbf{I}$ for $i, j = 1, \ldots, k$. $\mathbf{I}$ denotes the $d \times d$ unit matrix. Typical values for k lie in the range 2 to 8.

In this paper, we consider a variant of this method, the parallel Adams method proposed in [23]. The name was chosen because of a similarity of the stage equations with classical Adams formulae. In this paper different computation schemes are realized and evaluated using both system and method parallelism. The method is executed on different collections of processors and the parallel efficiency of the implementations is investigated.

Pseudocode PABM — one data parallel time step:

```
(1)  for ( i=0; i<k; i++ )
(2)      compute ⌈d/p⌉ elements of μ_{n+1,i} = f(t, y_{ni}) ;
(3)  for ( i=0; i<k; i++ ) {     /* PAB predictor */
(4)      compute ⌈d/p⌉ elements of y^{(0)}_{n+1,i} = Σ_{j=1}^{k} r_ij y_nj + h · Σ_{j=1}^{k} s_ij μ_{n+1,j};
(5)      exchange local components of y^{(0)}_{n+1,i} by multi-broadcast(p,d/p);
(6)      for (it=0; it<It; it++) {     /* PAM corrector */
(7)          compute ⌈d/p⌉ elements of μ^{(it+1)}_{n+1,i} = f(t, y^{(it)}_{n+1,i}) ;
(8)          compute ⌈d/p⌉ elements of y^{(it+1)}_{n+1,i} = y^{(0)}_{n+1,i} + h · δ_i · μ^{(it+1)}_{n+1,i} ;
(9)          exchange components of y^{(it+1)}_{n+1,i} by multi-broadcast(p,d/p);
(10)     }
(11) }
```

Fig. 4. Pseudocode for one time step of the PABM method implemented in a data parallel way.

The parallel Adams methods result from (2) by different choices for $\mathbf{R}$, $\mathbf{S}$ and $\mathbf{T}$. A diagonal matrix $\mathbf{T}$ yields an implicit equation system to be solved by fixed point iteration giving the Parallel Adams-Moulton (PAM) method. A zero matrix $\mathbf{T} = 0$ gives the Parallel Adams-Bashforth (PAB) method.

PAB methods First, we consider the PAB methods which result from setting $\mathbf{T} = 0$ and $\mathbf{R} = \mathbf{e}\cdot\mathbf{e_k}^T$ where $\mathbf{e} = (1, \ldots, 1)$ and $\mathbf{e_k} = (0, \ldots, 0, 1)$. With $V_\mathbf{x} = (\mathbf{x}, \mathbf{x}^2, \ldots, \mathbf{x}^k)$, $W_\mathbf{x} = (\mathbf{e}, 2\mathbf{x}, 3\mathbf{x}^2, \ldots, k\mathbf{x}^{k-1})$ and $b_i = a_i - 1$, we define $\mathbf{S} = V_\mathbf{a} W_\mathbf{b}^{-1}$ according to [23]. The abscissae values a_i, $i = 1, \ldots, k$, are determined so that a convergence order of $k + 1$ results. This can be obtained by using the Lobatto points.

To compute the stage vector $\mathbf{Y}_0$ of the initial time step from the given initial value y_0, the stage values y_{0i}, $i = 1, \ldots, k$, are evaluated using another explicit method, such as DOPRI8. Starting from $\mathbf{Y}_n$, time step $n + 1$ computes $\mathbf{Y}_{n+1} = (y_{n+1,0}, \ldots, y_{n+1,k})$ in two stages. First the right hand side function $\mathbf{f}$ is applied to $y_{n0}, \ldots, y_{nk}$ to give $\mathbf{F}(\mathbf{Y}_n)$. Then $\mathbf{Y}_{n+1}$ is obtained by adding $(\mathbf{R} \otimes \mathbf{I})\mathbf{Y}_n$ to $h(\mathbf{S} \otimes \mathbf{I})\mathbf{F}(\mathbf{Y}_n)$. Figure 1 shows the dependence structure of the resulting computation scheme for the case $k = 3$.

PAM methods PAM methods result from (2) by using a diagonal matrix $\mathbf{T}$. The diagonal entries δ_i, $i = 1, \ldots, k$, are determined in such a way that specific consistency conditions are satisfied. In addition, $\mathbf{R} = \mathbf{e}\cdot\mathbf{e_k}^T$ and $\mathbf{S} = (V_\mathbf{a} - \mathbf{R}\cdot V_\mathbf{b} - \mathbf{T}\cdot W_\mathbf{a})W_\mathbf{b}^{-1}$ are used in the PAM methods. Again, the Lobatto points can be used as abscissae values. Since $\mathbf{T}$ is diagonal, the resulting implicit relation is un-coupled and has the form

$$\begin{aligned} &\mathbf{y}_{n+1,i} - h * \delta_i * \mathbf{f}(\mathbf{y}_{n+1,i}) = \mathbf{v}_{ni} \\ &\text{for } i = 1, \ldots, k. \end{aligned} \tag{3}$$

The vectors $\mathbf{v}_{ni}$ are the d-dimensional vector components of $\mathbf{V} = (\mathbf{R} \otimes \mathbf{I})\mathbf{Y}_n + h(\mathbf{S} \otimes \mathbf{I})\mathbf{F}(\mathbf{Y}_n)$. Using fixed point iteration for the solution produces the following computation scheme:

$$\begin{aligned} &\mathbf{y}^{(j)}_{n+1,1} - h * \delta_1 * \mathbf{f}(\mathbf{y}^{(j-1)}_{n+1,1}) = \mathbf{v}_{n1} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ &\mathbf{y}^{(j)}_{n+1,k} - h * \delta_k * \mathbf{f}(\mathbf{y}^{(j-1)}_{n+1,k}) = \mathbf{v}_{nk} \end{aligned} \tag{4}$$

Equation (4) offers method parallelism and defines, for each i, $i = 1, \ldots, k$, an independent task that can be executed on a group of processors. The $\mathbf{v}_{ni}$, $i = 1, \ldots, k$, on the right hand side of (4) are calculated using the stage vectors $\mathbf{y}_{nj}$, $j = 1, \ldots, k$, from the previous time step:

$$\mathbf{v}_{ni} = \sum_{j=1}^{k} r_{ij}\mathbf{y}_{nj} + h * \sum_{j=1}^{k} s_{ij}\mathbf{f}(\mathbf{y}_{nj}) \tag{5}$$

with matrices $R = (r_{ij})_{i,j=1,\ldots,k}$ and $S = (s_{ij})_{i,j=1,\ldots,k}$. The convergence order of the resulting implicit method is $k + 2$. The starting vectors for the iteration can be generated using a PAB method as predictor, since both methods use the Lobatto points as abscissae values. The resulting predictor-corrector method is referred to as PABM below. Usually the PAB method is used as predictor in the PABM method which is a combination of the PAB and the PAM methods. The data dependencies of the PABM methods are illustrated in Fig. 2.

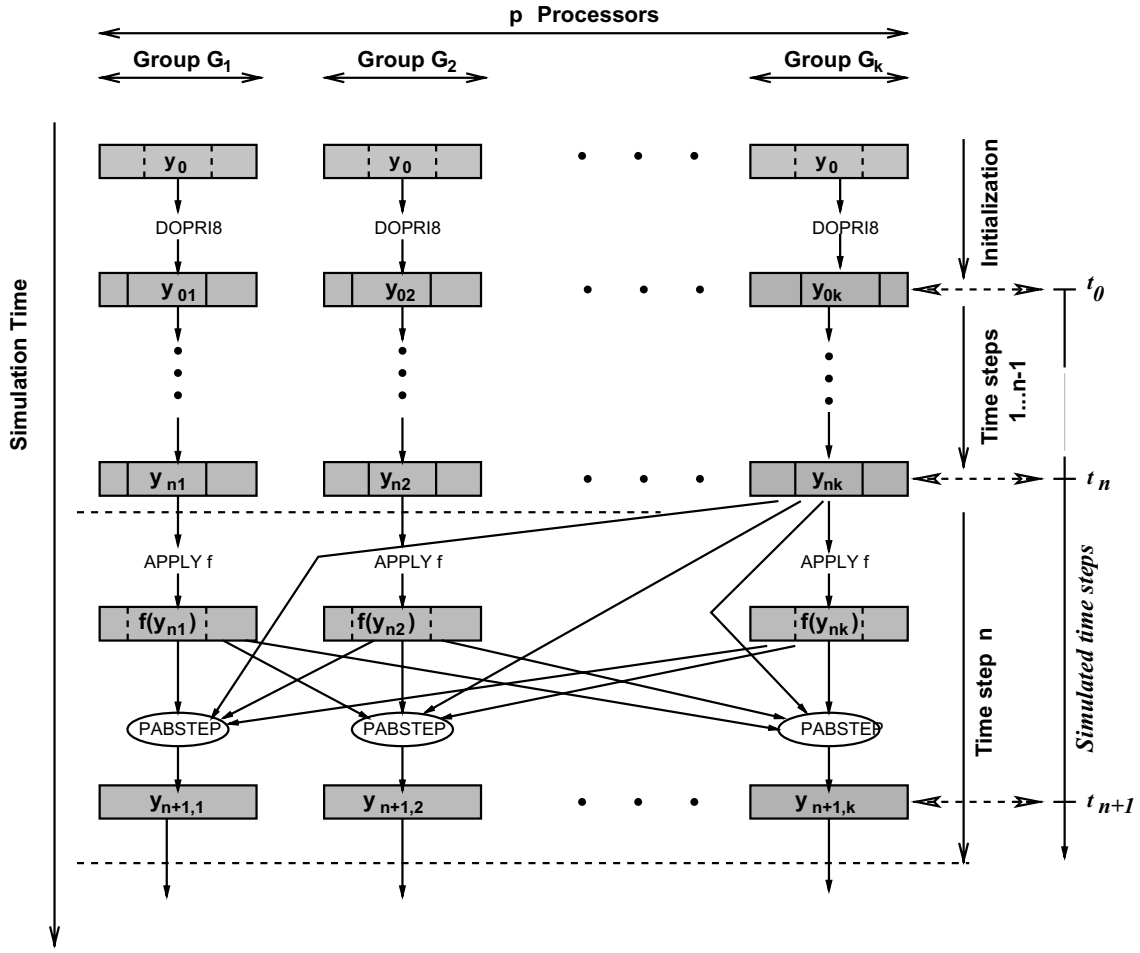

Fig. 5. Illustration of a task parallel implementation of a PAB method with k stage values on k disjoint processor groups. For abscissa values $\mathbf{a} = (a_1, \ldots, a_k)$ with $a_k = 1$ the stage values $y_{n,k}$ correspond to the approximations of $y(t_n)$, $n = 0, 1, 2, \ldots$.

3. Parallel implementation

The PAB and PABM methods offer several possibilities for parallel execution that differ in the order in which the computations in each time step are performed and in the way in which the required data exchange is effected. The data dependencies require that the time steps be executed one after another.

Data parallel implementation of the PAB method In each time step a pure data parallel realization computes the stage values one after another with all processors available. To compute $\mathbf{f}(y_{ni})$, $i = 1, \ldots, k$, each of the p processors evaluates approximately d/p components of $\mathbf{f}$. After this evaluation, each processor uses its local components of $\mathbf{f}(y_{ni})$ to compute the corresponding components of $y_{n+1,i}$ according to Eq. (2). Since $\mathbf{f}$ is considered to be an unknown black-box function, we must assume that all components of $y_{n+1,i}$ are available for the computation of each component of $\mathbf{f}(y_{n+1,i})$ in the next time step. Thus, after the computation of $y_{n+1,i}$, a global multi-broadcast operation must be performed to make the entire stage value $y_{n+1,i}$ available to all processors for the next time step. Figure 3 gives pseudocode for the internal computations of one time step.

Altogether, k global multi-broadcast operations are performed in each time step with each processor contributing approximately d/p components to each of these communication operations, thus effecting a globally replicated distribution of the entire stage vector. The resulting communication overhead per time step is given by

$$C_{PAB,dp}(d,p) = k \cdot T_{\mathrm{mbroad}}(p, d/p), \qquad (6)$$

where $T_{\mathrm{mbroad}}(p,x)$ denotes the time for a multi-broadcast operation executed on p processors with each

Pseudocode PABM — one task parallel time step:

```
(1)  if (me ∈ G_i) {
(2)    compute ⌈d/g_i⌉ components of μ_{n+1,i} = f(t, y_{n,i}) ;
(3)    gather components of μ_{n+1,i} at processor q_{i0} of G_i;
(4)    broadcast μ_{n+1,i} from q_{i0} to all processors;
     }
(5)  if (me ∈ G_i) {
       /* PAB predictor */
(6)    compute ⌈d/g_i⌉ elements of y^{(0)}_{n+1,i} = Σ_{l=1}^{k} r_{i,l} y_{n,l} + h · Σ_{l=1}^{k} s_{i,l} μ_{n+1,i};
(7)    exchange local components of y^{(0)}_{n+1,i} within G_i by multi-broadcast;
(8)    for (it=0; it<It; it++) {    /* PAM corrector */
(9)      compute ⌈d/g_i⌉ elements of μ^{(it+1)}_{n+1,i} = f(t, y^{(it)}_{n+1,i}) ;
(10)     compute ⌈d/g_i⌉ elements of y^{(it+1)}_{n+1,i} = y^{(0)}_{n+1,i} + h · δ_i · μ^{(it+1)}_{n+1,i} ;
(11)     exchange components of y^{(it+1)}_{n+1,i} within G_i by multi-broadcast;
       }
(12)   y_{n+1,i} = y^{(It)}_{n+1,i};
     }
(13) communication to make y_{n+1,k} available to all processors;
```

Fig. 6. Pseudocode for one time step of the PABM method in a task parallel implementation using k processor groups G_i of size $g_i, i = 1, \ldots, k$. All processors work in parallel and `me` denotes the processor Id of the executing processor.

Pseudocode PABM — one time step with orthogonal groups:

```
(1)  if (me ∈ G_i)
(2)    compute ⌈d/g_i⌉ components of μ_{n+1,i} = f(t, y_{n,i}) ;
(3)    exchange local components of μ_{n+1,i} within group Q_1, ..., Q_g by multi-broadcast;
rest as for the task parallel version;
```

Fig. 7. Pseudocode for one time step of a task parallel implementation of the PABM method using k processor group G_i, $i = 1 \ldots, k$, with orthogonal processor groups $Q_1, \ldots, Q_g$.

processor contributing x data values. For the special case $\mathbf{R} = \mathbf{e} \cdot \mathbf{e_k}^T$, see Section 2, the computation time for a single time step is given by

$$T_{PAB,dp}(d,p) = k \cdot (d/p \cdot T_{eval}(f) + (2k+1) \cdot d/p \cdot t_{op}), \quad (7)$$

where $T_{eval}(f)$ is the time to evaluate one component of $\mathbf{f}$ and t_{op} is the time for an arithmetic operation such as addition or multiplication.

Data parallel implementation of the PABM method: The data parallel implementation of the PABM method uses the data parallel implementation of the PAB method as predictor to yield the starting vector $y^{(0)}_{n+1,i}$, $i = 1, \ldots, k$, for each stage value of the PABM corrector in a replicated data distribution. For each processor, this requires k function evaluations and $k \cdot (2k+1)$ operations for d/p elements, see formula (7). A fixed number, It, of corrector iterations is performed where, in each iteration, each processor first computes d/p components of $f(y^{(0)}_{n+1,i})$ and then uses its local components to compute the corresponding d/p components of $y^{(it+1)}_{n+1,i} = y^{(0)}_{n+1,i} + h \cdot \delta_i \cdot f(y^{(it)}_{n,i})$, see Fig. 4. This leads to the following computation time:

$$T_{PABM,dp}(d,p) = k \cdot ((It+1) \cdot d/p \cdot T_{eval}(f) + (2k+1) \cdot d/p \cdot t_{op} + It \cdot 3 \cdot d/p \cdot t_{op}) \quad (8)$$

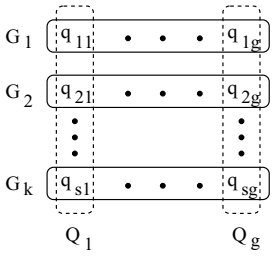

Fig. 8. Disjoint processor groups $G_1, \ldots, G_k$ and orthogonal processor groups $Q_1, \ldots, Q_g$.

Since only the local components are needed for this computation no data exchange is required. For the next iteration, the vector $y_{n+1,i}^{(it+1)}$ is needed as argument vector for the function evaluation. Thus a global multi-broadcast operation is required to make all of the components of $y_{n+1,i}^{(it+1)}$ available to all processors for $it = 0, \ldots, It-1$. Figure 4 gives pseudocode for the main computation and communication operations of one time step of the PABM method.

This implementation strategy results in the following communication overhead within one time step:

$$C_{\mathrm{PABM,dp}}(d,p) = k \cdot (It+1) \cdot T_{\mathrm{mbroad}}(p, d/p) \tag{9}$$

M-task implementation with internal data parallelism for PAB An M-task realization exploiting method parallelism employs k disjoint processor groups $G_1, \ldots, G_k$ of approximately equal size $g = p/k$ – see the diagram in Fig. 5. In each time step n, processor group G_i is responsible for computing stage value $y_{n+1,i}$, $i = 1, \ldots, k$. The computation of $y_{n+1,i}$ requires access to $\mathbf{f}(y_{n1}), \ldots, \mathbf{f}(y_{nk})$ for $i = 1, \ldots, k$ which are computed independently. Thus, by data exchange, each processor gets exactly those components that it needs for its local computation. This can, for example, be realized by first collecting the components of $\mathbf{f}(y_{ni})$ on a specific processor, q_{ij}, of G_i, $i = 1, \ldots, k$, using a group-local gather operation and then broadcasting them to the processors of all other groups. The group-local gather operations can be performed concurrently by the different groups, whereas the single-broadcast operations have to be performed one after another because all processors are involved, thus leading to k single-broadcast operations. Since $\mathbf{f}$ is an unknown function, $y_{n+1,i}$ has to be distributed among all the processors of G_i by a multi-broadcast operation to ensure a group-replicated distribution of the stage values and to enable a group internal evaluation of $\mathbf{f}(y_{n+1,i})$ in the next time step. In addition, the last stage vector y_{nk} has to be sent to all other groups G_i, $i = 1, \ldots, k$, by a single-broadcast operation involving all processors for the computation of the PAB step in the next time step.

Compared to the pure data parallel execution scheme, more communication operations are necessary, but most of these are group-local. Since the group-specific communications can be executed concurrently, the communication time per time step can be expressed as:

$$C_{\mathrm{PAB,tp}}(d,p) = T_{\mathrm{gather}}(g, d/p) + (k+1) \cdot T_{\mathrm{sbroad}}(p,d) + T_{\mathrm{mbroad}}(g, d/g) \tag{10}$$

The computation time is identical to the data parallel case, if d is a multiple of p and g and if p is a multiple of g.

M-task implementation with internal data parallelism for PABM Using the PAB method as predictor, the PABM method performs It corrector steps, where each processor group G_i operating independently, iteratively computes $y_{n+1,i}^{(It)}$, $i = 1, \ldots, k$. After the computation of $y_{n+1,i}^{(it+1)}$, this vector is made available by a multi-broadcast to all other processors of the same group G_i where it is required for the computation of $\mu_{n+1,i}^{(it+2)}$ in the next corrector iteration. The final vector of the iteration $y_{n+1,i}^{It}$ is replicated on all processors of G_i, so the next time step can use this value as $y_{n+1,i}$ for the predictor step without further communication. Thus, compared with the PAB method, the PABM method additionally requires It group-based multi-broadcast operations. The communication time for the task parallel PABM methods is therefore:

$$C_{\mathrm{PABM,tp}}(d,p) = T_{\mathrm{gather}}(g, d/p) + (k+1) \cdot T_{\mathrm{sbroad}}(p,d) + (It+1) \cdot T_{\mathrm{mbroad}}(g, d/g) \tag{11}$$

Figure 6 gives pseudocode for the task parallel PABM method. The computation time is identical to the data parallel case, if d is a multiple of p and g and if p is a multiple of g.

Exploiting orthogonal structures of communication The implementation based on M-tasks uses two communication phases, one to make available to each processor those components of $\mathbf{F}(\mathbf{Y}_n)$ that it needs for the computation of its local components of $\mathbf{Y}_{n+1}$, and one to establish a group-replicated distribution of $y_{n+1,i}$ in group G_i. The second communication phase is based on group-local communication, but the first

```
(1)  void* tpar_pab(void *arg, MPI_Comm comm, T_Descr *pdescr1) {
(2)    T_Descr *pdescr2;
(3)    int i;
(4)    for (i=0; i<k; i++) per[i]= 1.0/k;
(5)    T_Split_GrpParfor (k, pdescr1, pdescr2, per);
(6)    /* allocate arrays mfevals, pargs1, pres, pabs, pargs2 */
(7)    for (i=0; i<k; i++) { mfevals[i]= feval; pabs[i]= pabstep;}
(8)    /* initialize data structures */
(9)    t = t_0;
(10)   while (t <= t_0 + H) {
         /* M-task parallel evaluation of f(y_{n0}), ..., f(y_{n,k-1}): */
(11)     T_Parfor(mfevals, pargs1, pres, pdescr2);
(12)     /* send function results to all processors */
         /* M-task parallel computation of PAB step */
(13)     T_Parfor(pabs, pargs2, pres, pdescr2);
(14)     /* send new stage vectors y_{n+1,k-1} to all processors */
(15)     t = t + h;
(16)   }
(17) }
(18) void *feval (void * arg, MPI_Comm comm, T_Descr *pdescr){
(19)   /* data parallel function evaluation */
(20)   /* create a group replicated distribution of the result */
(21) }
(22) void *pabstep (void * arg, MPI_Comm comm, T_Descr *pdescr){
(23)   /* data parallel PAB step for one stage vector */
(24)   /* create a group replicated distribution of the result */
(25) }
```

Fig. 9. Tlib program for task parallel PAB method with k parallel M-tasks independently computing $\mathbf{y}_{n0}, \ldots, \mathbf{y}_{n,k-1}$. The numbering in the program starts with 0 instead of 1 according to the numbering conventions in C.

uses a global broadcast operation. We now re-organize the first communication phase so that it also uses group-local communication only.

Below, we assume that the number of stage vectors, k, is a multiple of the number of processors, p, so that all groups, G_i, have the same number of processors, $g = p/k$. Moreover, we assume that the dimension d of the stage values is a multiple of g. Based on the processor groups $G_1, \ldots, G_k$, we define orthogonal groups $Q_1, \ldots, Q_g$ with $|Q_j| = k$ and $Q_j = \{q_{lj} \in G_l \mid l = 1, \ldots, k\}$, see Fig. 8. For the computation of $\mathbf{F}(\mathbf{Y}_n)$, each processor q_{ij} of G_i computes d/g components of $\mathbf{f}(y_{ni})$. The corresponding components of $\mathbf{f}(y_{nl})$ for $l \neq i$ are computed by the processors q_{lj} of G_l. These components are just what are needed by q_{ij} for the computation of $y_{n+1,i}$. Thus, the required components can be made available to each processor by a group-oriented multi-broadcast operation on the orthogonal group Q_j. These communication operations can be executed concurrently on all orthogonal groups $Q_1, \ldots, Q_g$. The stage vector $y_{n+1,k}$ of the last group G_k is made available to the other groups by g concurrent single-broadcast operations on the orthogonal groups $Q_1, \ldots, Q_g$. Thus, all group-oriented communication operations can be executed concurrently and the communication time within

one time step is given by

$$C_{\mathrm{PAB,ort}}(d,p) = T_{\mathrm{mbroad}}(k, d/g) + T_{\mathrm{sbroad}}(k, d/g) + T_{\mathrm{mbroad}}(g, d/g). \quad (12)$$

The computation time is the same as for the task parallel case without orthogonal structure.

Exploiting orthogonal structures of communication for the PABM method The PAM corrector is implemented identically to the M-task parallel version, since disjoint processor groups are already exploited by the method. The PAB predictor also exploits the orthogonal communication structure, see Fig. 7 for an illustration. The communication time is reduced to:

$$C_{\mathrm{PABM,ort}}(d,p) = T_{\mathrm{mbroad}}(k, d/g) + T_{\mathrm{sbroad}}(k, d/g) + (It+1) \cdot T_{\mathrm{mbroad}}(g, d/g). \quad (13)$$

Again, the computation time is identical to the M-task version without orthogonal groups.

4. Tlib library for expressing task parallelism

Tlib is a runtime library to support programming with hierarchically structured M-tasks. Tlib provides an API comprising several library functions that are implemented on top of MPI. A parallel program using the library comprises a collection of basic SPMD tasks and a set of *coordination functions* which embody M-tasks. In the coordination functions, concurrent execution is effected by calling the corresponding *library functions* with a description of the (hierarchical) decomposition of the processor set. This description is stored in a *group descriptor* that is created before the concurrent execution of tasks is initiated.

The library's API provides separate functions for the structuring of processor groups and for the coordination of concurrent and nested M-tasks. The task structure can be nested arbitrarily, which means that a coordination function can assign other coordination functions to subgroups, which can, in turn, split the subgroup and assign other coordination functions. Tlib library functions are designed to be called in an SPMD manner which results in multi-level group SPMD programs. The entire management of groups and M-tasks at execution time is done using the library. This includes the creation and administration of the structure of the processor groups, the mapping of tasks to groups and their coordination, the handling and termination of recursive calls and group splittings, and the organization of communication between groups in the hierarchy.

Internally, the library exploits distributed information stored in distributed group descriptors, which are hidden from the user. The first group descriptor `pdescr` of type `T_Descr` of a program is initialized by

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

which needs an MPI communicator and creates a descriptor `pdescr`. The splitting of an existing group of processors that is represented by a group descriptor `pdescr` into two new groups is performed by

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

`per1` and `per2` denote fractional values, with $per1 + per2 \leqslant 1$. The two resulting groups contain a fraction $per1$ or $per2$ of the processors of the original group and are both represented by the group descriptor `pdescr1`. More general splitting operations are available. After a splitting operation, the concurrent execution of two independent tasks can be initiated by calling the library function

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

The M-tasks to be executed are supplied as arguments `f1` and `f2`, the arguments of the tasks are `parg1` and `parg2`, and the result values of the executed M-tasks are returned in `pres1` and `pres2`, respectively; `pdescr` represents the two groups on which the two M-tasks provided are executed.

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

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

The separation of the creation and the use of the descriptors is useful in iterative methods where the same descriptor can be efficiently reused several times.

We have used Tlib to implement the task parallel versions of the PAB and PABM methods. The Tlib library makes it easy to implement different task parallel

```
(1)  void *pabmstep (void * arg, MPI_Comm comm, T_Descr *pdescr){
     /* data parallel computation of PABM step for one stage vector */
(2)  for (it = 0; it < lt; it++) {
(3)    /* data parallel fixed point step for one stage vector computation */
(4)    /* communication to create a group replicated distribution of the result vector */
(5)    }
(6)  }
```

Fig. 10. Tlib function for task parallel PABM method.

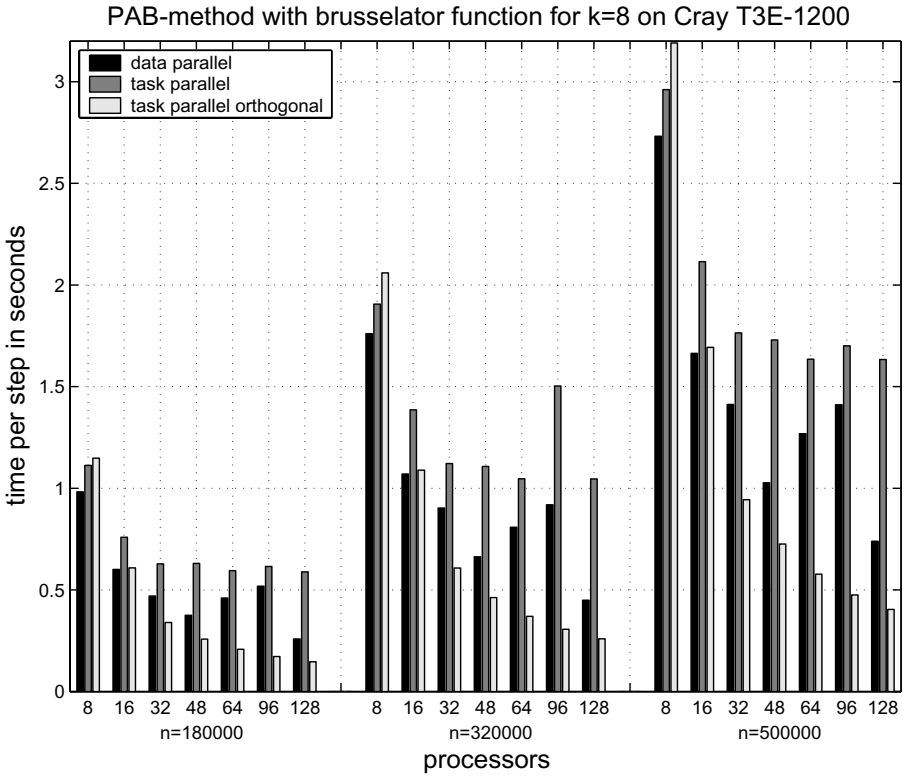

Fig. 11. Runtimes of the PAB method for Brusselator on Cray T3E with $k = 8$.

versions quickly without incurring a perceptible implementation overhead. The use of the library requires the definition of the parallel procedures as M-Tasks by mapping them to the arguments f1 and f2 of the library function `T_Par` and by packing the corresponding arguments of these procedures in the arguments parg1 and parg2 of `T_Par`.

The task parallel implementations of the PAB and the PABM methods use a generalization of the function `T_Par` to define k parallel M-tasks (k is the number of stage values defining the degree of task parallelism). Each M-task is one of the fixed point iterations, exploiting the fact that these computations are completely independent within each time step. After each time step communication between M-tasks is required to exchange vectors as described earlier. The call of the parallel M-tasks using the Tlib function is repeated in each step. The underlying splitting of the processors into k groups of processors is done once only at the outset using a generalized version of the library function `T_SplitGrp` and this partition is reused in each step.

Figure 9 shows essential parts of the Tlib program for the task parallel implementation of the PAB method.

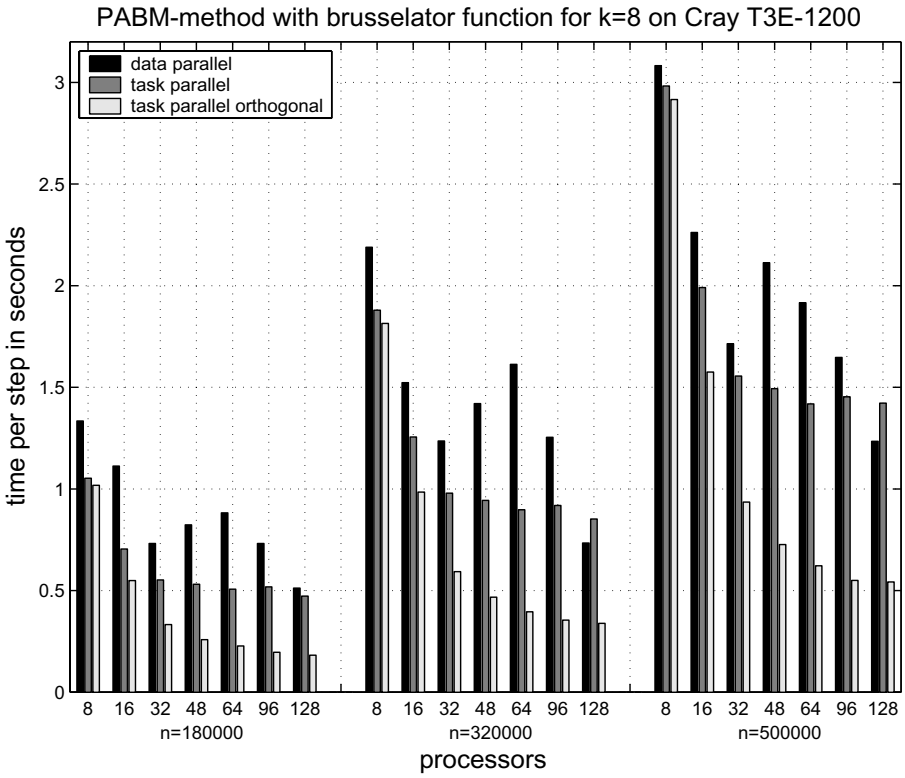

Fig. 12. Runtimes of the PABM method for Brusselator on Cray T3E with $k = 8$.

For simplicity, many details, like the allocation of data structures, are omitted; allocations and initializations are indicated by comments. Variables, like the number `k` of stage values, `t_0` for the start time of the time interval, `H` for the size of the time interval, and `h` for the stepsize are considered to be global variables. Initially, the library function `T_SplitGrpParfor` creates k disjoint groups of processors in line (5). In line (7), for each i, $i = 0, \ldots, k-1$, the array `mfeval` is filled with references to function `feval`; analogously, the array `pabs` is filled with references to function `pabstep`. These functions perform the evaluation of **f** (feval) and the computation of the PAB method (pabstep), see also Fig. 5 for the task parallel execution of the PAB method. The library call `T_Parfor` in line (11) ensures that k parallel function calls are issued and executed on the processor groups previously created by `T_SplitGrpParfor`. The second call of `T_Parfor` in line (13) initiates the task parallel execution of the function `pabstep`. In general, the function argument array for `T_Parfor` can be filled with different functions resulting in different groups executing different M-tasks concurrently; however, for the PAB method, all groups execute the same M-task on different stage vectors.

The Tlib function for PABM has the same structure as the function `tpar_pabstep` but uses the assignment `pabs[i] = pabmstep` for $i = 0, \ldots k-1$ instead of `pabs[i] = pabstep` in line (7) of Fig. 9. The function `pabs[i] = pabmstep` is a data parallel implementation of the fixed point iteration for computing one of the stage vectors $y_{n+1,i}$, $i = 0, \ldots k-1$, see Fig. 10.

5. Runtime experiments

For the experiments, the PAB and PABM methods have been implemented for $k = 2, 4, 8$. According to [23], a fixed stepsize strategy is used to facilitate clear observation of the algorithmic features. The implicit relations are solved by a fixed point iteration with a fixed number of iterations for all stages.

As test problems two classes of ODE systems are considered:

Sparse ODE systems are characterized by each component of the right-hand side function **f** of the ODE

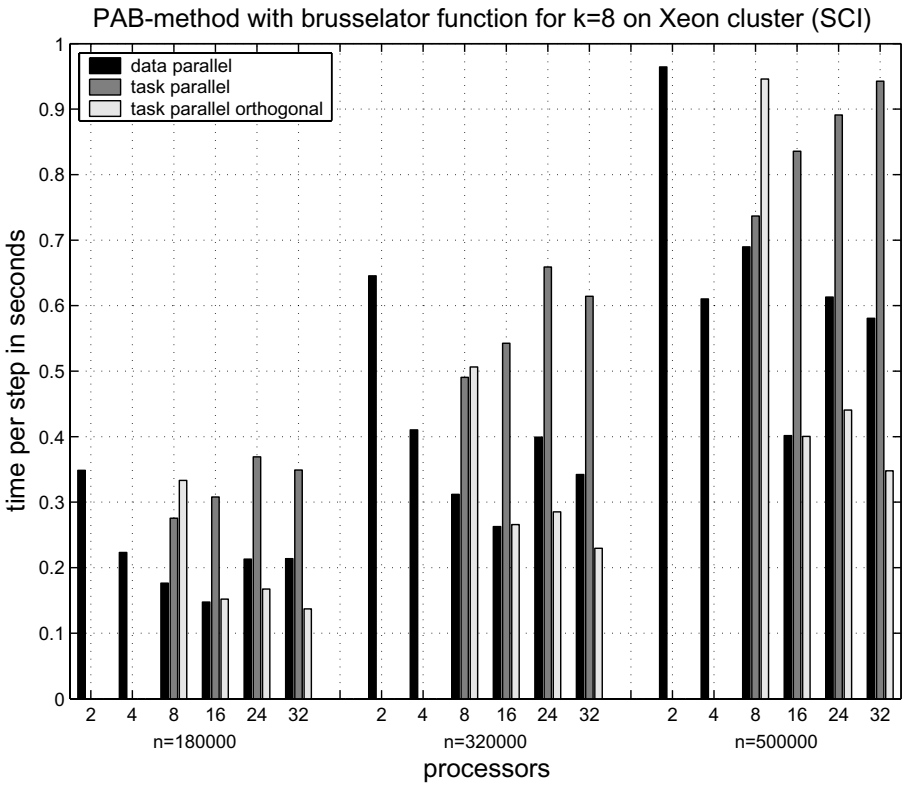

Fig. 13. Runtimes of the PAB method for Brusselator on SCI Xeon cluster with $k = 8$.

system having a fixed evaluation time that is independent of the size of the ODE system, i.e., the evaluation time for the entire function **f** increases linearly with the size of the ODE system. Such systems arise, for example, from a spatial discretization of a time-dependent partial differential equation (PDE) [6]. In particular, an ODE system resulting from the spatial discretization of a 2D PDE describing the reaction with diffusion of two chemical substances (Brusselator equation) is considered [7,17].

Dense ODE systems are characterized by each component of **f** having an evaluation time that increases linearly with the system size, i.e., the evaluation time for the entire function **f** increases quadratically with the size of the ODE system. In particular, a spectral decomposition of a time-dependent Schrödinger equation which results in a dense ODE system is considered [16].

The parallel implementations have been tested on four machines: a Cray T3E-1200, two Beowulf clusters and an IBM Regatta system. One of the Beowulf clusters has 16 nodes, where each node consists of two, 2.0 GHz, Xeon processors. The nodes are connected by a SCI interconnection network. The second Beowulf Cluster CLiC ('**C**hemnitzer **Li**nux **C**luster') is built of 528, 800 MHz, Pentium III processors. The processors are connected by two different networks, the communication network and the service network. Both of these are based on the fast-Ethernet-standard. The IBM Regatta system uses 32, 1.7 GHz, Power4 processors per node and has 34 nodes.

Figures 11 and 12 show the execution times in seconds of one time step of the PAB and PABM methods, with $k = 8$ stage vectors, for the Brusselator equation using system sizes 180000, 320000 and 500000. For $k = 8$, task parallelism can only be implemented for at least 8 processors, because each group needs at least one processor. The figures show that, for both methods, the orthogonal parallel version with M-tasks is considerably faster than both the data parallel version and the standard M-task version, since the former uses group-based communication only. For the PAB method, the data parallel version is faster than the standard M-task version, since the M-task version uses more communication operations and a global gather operation in each time step. In particular, the orthogonal version shows much better scalability than the other two.

In Fig. 11, it can be observed that the data parallel version has a local minimum for $p = 48$ processors.

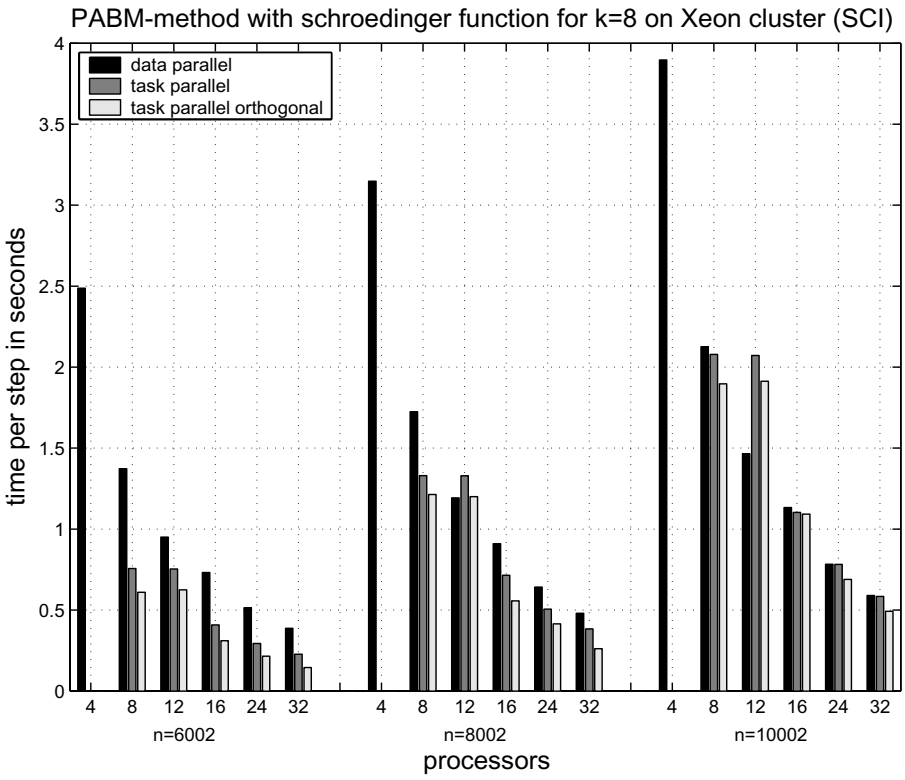

Fig. 14. Runtimes of the PABM method for Schrödinger on SCI Xeon cluster with $k = 8$.

This behavior can be explained by considering the runtime formulae (6) and (7) from Section 3. According to formula (6), the communication time is determined by a global multi-broadcast operation for which each processor contributes a data block with d/p elements. The execution time of a multi-broadcast operation is given by

$$T_{mb}(p, n) = \tau_1 + \tau_2 \cdot p + t_c \cdot p \cdot n \qquad (14)$$

where τ_1, τ_2 and t_c are architectural parameters describing the communication system of the target platform [20]. These parameters can be determined by benchmark programs. The parameter n is the size of the data block on each processor, i.e., for the PAB method it is $n = d/p \cdot$ sizeof(double). Thus, the last term, $t_c \cdot p \cdot n$, is constant for a fixed system size, independently of the number p of processors. But because of the second term $\tau_2 \cdot p$, the communication time of the PAB method increases with the number of processors. The computation time, on the other hand, decreases with increasing numbers of processors according to formula (7). For up to 48 processors, this decrease is larger than the increase in the communication time, so that the overall execution time decreases with the number of processors. Starting with 64 processors, the increase in the communication time is larger than the decrease in the computation time as the number of processors increases. Hence, the overall execution time starts to increase with the number of processors.

For the PABM method, see Fig. 12, the standard M-task version is faster than the data parallel version, since the latter uses a global multi-broadcast operation in each iteration step whereas the former uses a global gather operation only once in each time step. Orthogonal communication structures further improve the performance. Similar results are obtained for $k = 4$. Figure 13 shows the execution times for the PAB method applied to the Brusselator equation on the Xeon cluster. The results are similar to those for the T3E. The Xeon cluster has only 32 processors; thus, measurements are only shown for up to 32 processors. Again, for 2 and 4 processors, no task parallel version is shown, because this requires at least 8 processors for $k = 8$ stage vectors.

Figure 14 shows the execution times for the PABM method for the Schrödinger equation on the Xeon cluster. Here all versions show good scalability, since the evaluation time for **f** accounts for a large portion of the

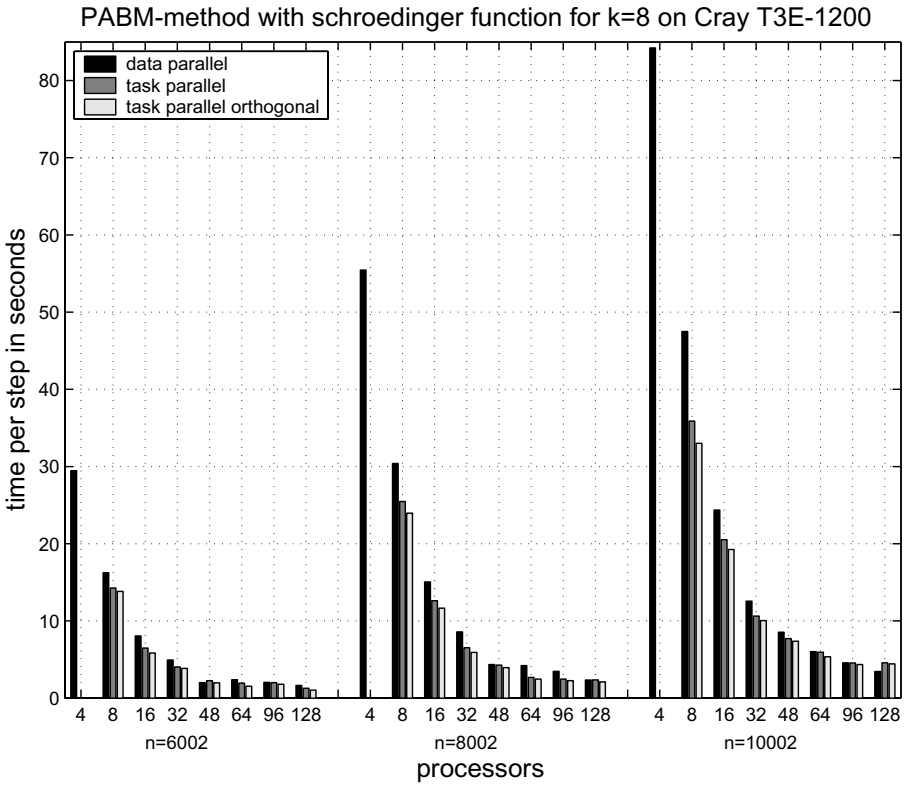

Fig. 15. Runtimes of the PABM method for Schrödinger on Cray T3E with $k = 8$.

parallel runtime. However, the orthogonal task parallel version has the least execution time in most cases. The difference between the M-task version and the data parallel version is most significant for smaller ODE systems. Similar results are obtained for the PAB methods on both hardware platforms for $k = 4$ and $k = 8$. Figure 15 shows the execution times for the PABM method for the Schrödinger equation on the Cray T3E, and again shows similar results because of the dominance of computation over communication.

Figure 16 shows the execution times of the PABM method for the Brusselator equation on the CLiC. In this case, the task parallel implementation is significantly faster than the data parallel implementation which is further improved by exploiting orthogonal communication structures. The impressive decrease in runtime when using concurrent multiprocessor tasks instead of data parallelism can be explained by the large communication overhead for collective communication operations on the CLiC due to its interconnection network. Figure 17 shows the execution times for the PABM method for the Brusselator equation on one node of the IBM Regatta system. Here, the data parallel version has the fastest execution times and shows good scalability because of the shared memory used within each node. Comparing Figs 16 and 17, it can be seen that the advantage of an orthogonal task parallel execution of the PABM methods is most significant on machines with a slow interconnection network and that, for parallel machines with a fast network or a shared memory, data parallel implementations can be competitive with and even be faster than the orthogonal task parallel versions. But for cluster systems with a slower network, such as the Beowulf cluster CLiC, optimizations such as orthogonal task parallel versions are required to achieve competitive performance results.

Orthogonal realization of communication operations The improvement of the communication in M-task programming arises from the restriction of the communication to smaller groups of processors. This is realised when several independent tasks operate independently on subsets of the entire set of processors: thus collective communication inside an M-task is performed on a smaller set of processors, while the entire application uses the complete set of processors. The reason for this behavior is that the costs of communication depends on the number of processors in either a linear or logarithmic way.

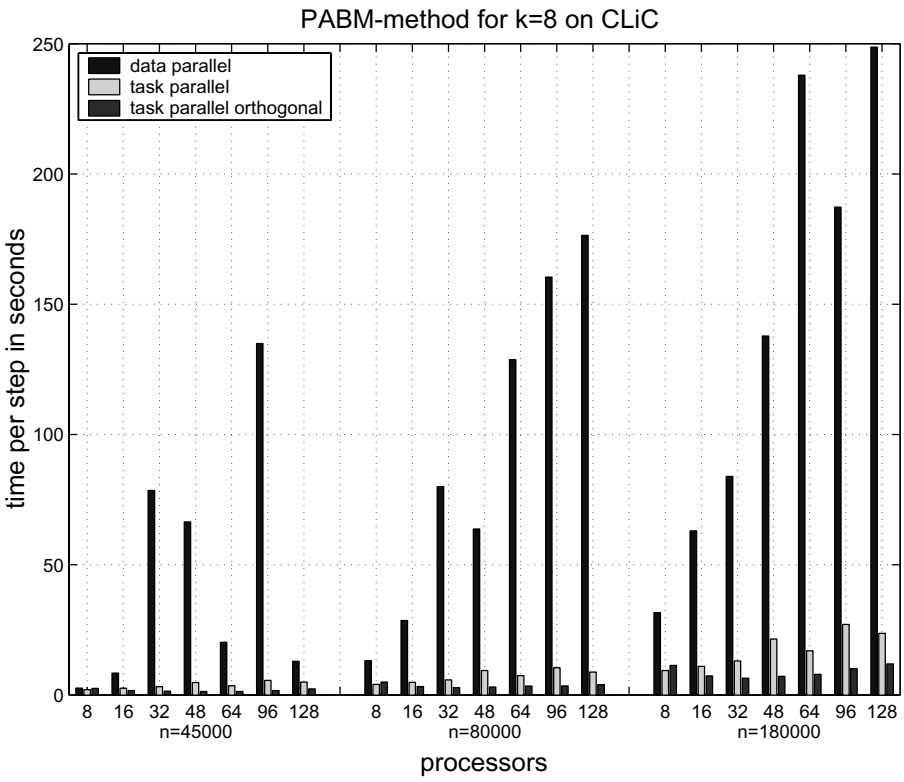

Fig. 16. Runtimes of the PABM method for Brusselator on CLiC with $k = 8$.

There is also the possibility to internally restructure collective communication so that orthogonal structures are used internally. This issue has been addressed in [12]; this paper establishes that orthogonal restructuring can have a significant effect, if the MPI library is not efficiently implemented. In particular, this is the case for the MPI implementation on the CLiC cluster. In addition, for the MPI implementations on the SCI Xeon cluster, the T3E and the IBM Regatta improvements can be obtained. However, these are less impressive than those obtained by restructuring the communication in the algorithm as reported upon here. The improvements for the parallel Adams methods obtained using orthogonal communication structures are also due to the fact that fewer data values need to be exchanged because of a judicious arrangement of the stage vectors.

Runtime prediction The runtime functions from Section 3 can be used to estimate the execution time of the PAB or PABM methods on a given target machine. To do this, architectural parameters of the target machine need to be considered. To estimate the computation time, the average time for performing an arithmetic operation is required. This can be determined using benchmark programs. Based on the time for an arithmetic operation, the time for performing an evaluation of the function **f** describing the specific ODE system (1) to be solved can be determined. To estimate the communication time, functions for the communication operations used are required. An expression for the execution time of a multi-broadcast operation, for example, has been given in Formula (14).

With this additional information, it is possible to give a coarse estimate for the execution time of the PAB and PABM methods on a specific target platform. As example, Fig. 18 compares the measured and the predicted execution times of data-parallel implementations of the PAB method (left) and the PABM method (right) on a Cray T3E system. The measurements are based on a particular MPI implementation that reduces the communication time by using an internal orthogonal realization. The predicted execution times match the measured execution times reasonably well. For the T3E system, the deviations are usually below 12%. An exact fitting is not possible using the runtime functions derived, since these ignore important details of the target platform like the memory hierarchy, the availability of multiple instruction units or the pipelining of instruc-

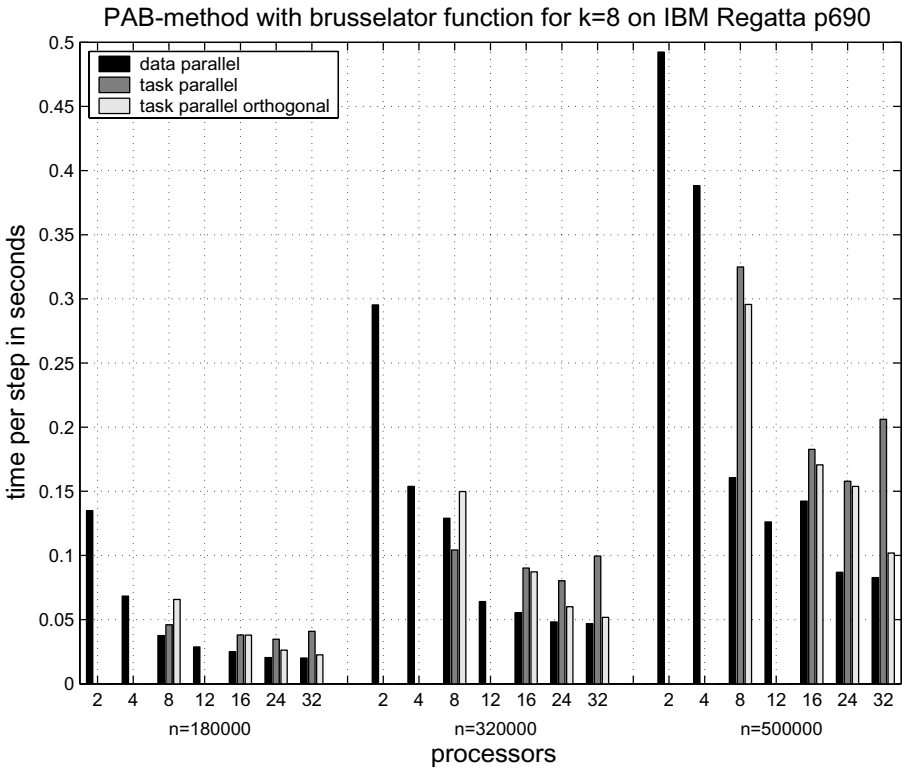

Fig. 17. Runtimes of the PABM method on IBM Regatta with $k = 8$.

tion units. Nevertheless, the runtime functions provide a rough estimate and allow a coarse comparison of different parallel versions.

6. Related work

Several research groups have proposed models for mixed task and data parallel executions with the goal of obtaining parallel programs with faster execution time and better scalability properties, see [2,22] for an overview of systems and approaches and see [4] for a detailed investigation of the benefits of combining task and data parallel execution.

Many environments for mixed parallelism in scientific computing are extensions of the HPF data parallel language: see [5] for an overview. Examples of parallel programming systems are HPJava [26], LPARX [11] and KeLP [14]. HPJava 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. The concept of processor groups is supported in the sense that global data to be distributed over one process group can be defined and that the program execution control can choose one of the process groups to be active. LPARX is a parallel programming system for the development of dynamic, nonuniform scientific computation 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 [1,5]. 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 [14].

Much research has been devoted to the development of the BSP (bulk synchronous parallelism) model and a programming library (Oxford BSP library) is available that allows the formulation of BSP programs in an SPMD style [8,13]. NestStep extends the BSP model by supporting group-oriented parallelism, through nesting of supersteps, and a hierarchical processor group concept [10]. NestStep is defined as a set of extensions to programming languages like C or Java and is designed for a distributed address space. It also supports

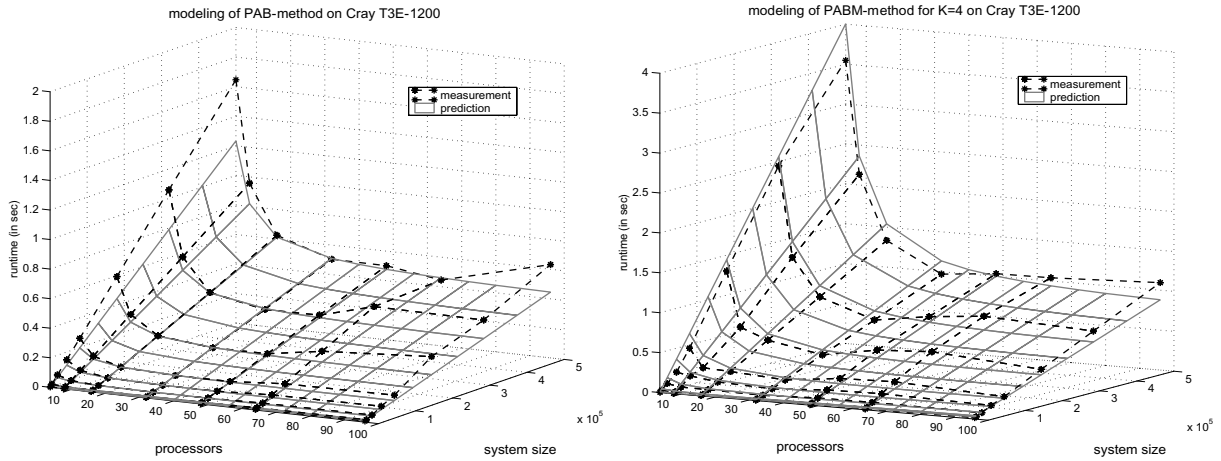

Fig. 18. Measured and predicted runtimes of the PAB-method (left) and PABM-method for $K = 4$ on the Cray T3E-1200.

the use of distributed shared arrays. Since it would be too expensive to exchange all elements of such an array at the end of each superstep, the compiler and runtime system try to arrange prefetching of elements by those processors that use these elements in the next superstep. The group concept of NestStep is similar to that of Tlib. However, the BSP-based programming model of NestStep is quite different from the Tlib programming model in which there is no concept of supersteps. In particular, each communication operation takes effect as soon as the data arrives at the receiving processors.

7. Conclusions

The realization of parallel applications using pure data parallelism can result in efficient programs. However, applications that rely on information exchange with collective communication operations may exhibit scalability problems, when executed on parallel systems with larger numbers of processors. The reason is that the execution time of collective communication operations increases logarithmically or linearly with the number of participating processors. In such situations it can be beneficial to employ a parallel processing approach based on a separation of different program parts which can then be assigned to different groups of processors to be executed concurrently as multiprocessor tasks. Such an execution scheme is attractive for applications having inherent task parallelism potential.

The parallel Adams methods discussed in this paper offer potential method parallelism in the form of independent stage vector computation in each iteration step and can benefit from the M-task programming style. However, additional collective communication is required since information has to be exchanged between processor groups. By reordering the communication using an orthogonal organization of disjoint processor groups for data exchange, the communication overhead can be significantly reduced. For the parallel Adams methods the combination of an implementation as an M-task program with orthogonal processor groups for the communication phases results in the most efficient implementation on a number of parallel platforms having different interconnection networks. The approach also exhibits good scalability. The experiments on different parallel platforms show that the implementation as an M-task program is especially effective on parallel platforms having small bandwidth.

The approach used for the parallel Adams method can also be applied in other applications, particularly in numerical mathematics. For example, parallel iterated Runge-Kutta methods, also used for solving ODEs and also based on the computation of independent stage vectors, can be implemented using the same approach of combining M-task programming with orthogonal communication. Other applications that can benefit from orthogonal communication are matrix-based computations like LU factorization where an orthogonal structuring of the communication leads to increased scalability [15].

The runtime library Tlib offers convenient support to implement M-task programs by providing operations to create and manage processor groups and to map independent computations to these groups. Tlib also supports hierarchical subdivisions of processor groups into smaller groups. Thus, Tlib can be used for the implementation of divide-and-conquer algorithms where

a new level of processor groups is used for each level of recursion. This approach has been used to develop fast multi-level algorithms for matrix multiplication that use a fixed number of recursions of the Strassen algorithm at the top level [9].

Acknowledgment

We thank the anonymous referees for their helpful comments. We also thank the NIC Jülich for access to its parallel machines.

References

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

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

[3] J.C. Butcher, *The Numerical Analysis of Ordinary Differential Equations*, Runge-Kutta and General Linear Methods. Wiley, New York, 1987.

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

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

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

[7] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II*, Springer, 1991.

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

[9] S. Hunold, T. Rauber and G. Rünger, *Multilevel Hierarchical Matrix Multiplication on Clusters*, in Proceedings of the 18th Annual ACM International Conference on Supercomputing, ICS'04, June 2004, 136–145.

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

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

[12] M. Kühnemann, T. Rauber and G. Rünger, Optimizing MPI Collective Communication by Orthogonal Structures. to appear: Cluster Computing – The Journ. of Networks, Software Tools and Applications, *Special Issue on Cluster Computing in Science and Engineering* **8**(4) (2005), 257–279.

[13] W.F. McColl, *Universal Computing*, In Proceedings of the EuroParfl96, Springer LNCS 1123 1996, 25–36.

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

[15] T. Rauber and G. Rünger, *Optimal Data Distribution for LU Decomposition*, In Proc. of the EuroPar'95, Springer LNCS 966, 1995, 391–402.

[16] T. Rauber and G. Rünger, *Parallel Solution of a Schrödinger-Poisson system*, In International Conference on High-Performance Computing and Networking, Springer LNCS 919, 1995, 697–702.

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

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

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

[20] T. Rauber and G. Rünger, Modelling the Runtime of Scientific Programs on Parallel Computers, in: *Proc. ICPP-Workshop on High Performance Scientific and Engineering Computing with Applications* (*HPSECA-00*), Toronto, Kanada, August 2000, 307–314.

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

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

[23] P.J. van der Houwen and E. Messina, Parallel Adams Methods, *J of Comp and App Mathematics* **101** (1999), 153–165.

[24] P.J. van der Houwen and B.P. Sommeijer, Iterated Runge-Kutta Methods on Parallel Computers, *SIAM Journal on Scientific and Statistical Computing* **12**(5) (1991), 1000–1028.

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

[26] 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.