Developing Parallel Programs with Multiprocessor Tasks

Thomas Rauber

Department of Computer Science
University of Bayreuth

Contributors: Gudula Rünger, Sascha Hunold, Robert Reilein, Matthias Kühnemann, Mattias Korch

June 13, 2006

Outline

Introduction

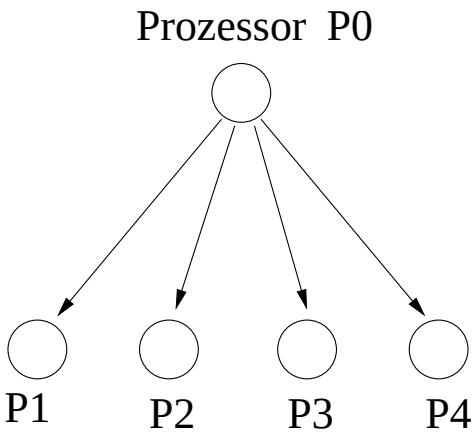
- Many **large applications** from scientific computing have a **modular structure** of cooperating subtasks.
- **Numerical Algorithms** can often be **restructured** such that a **task structure** is generated:
  - if the **inherent degree of parallelism** is large enough, a program-based restructuring is sufficient $\rightsquigarrow$ the **numerical properties** are **not changed**;
  - if the inherent degree of parallelism is too small, an **algorithmic restructuring** has to be performed;
    $\rightsquigarrow$ the numerical properties **may be changed**.
- Task parallelism can be combined with other types of parallelism
  example: **mixed task and data parallel execution**;

Advantages of M-task programming

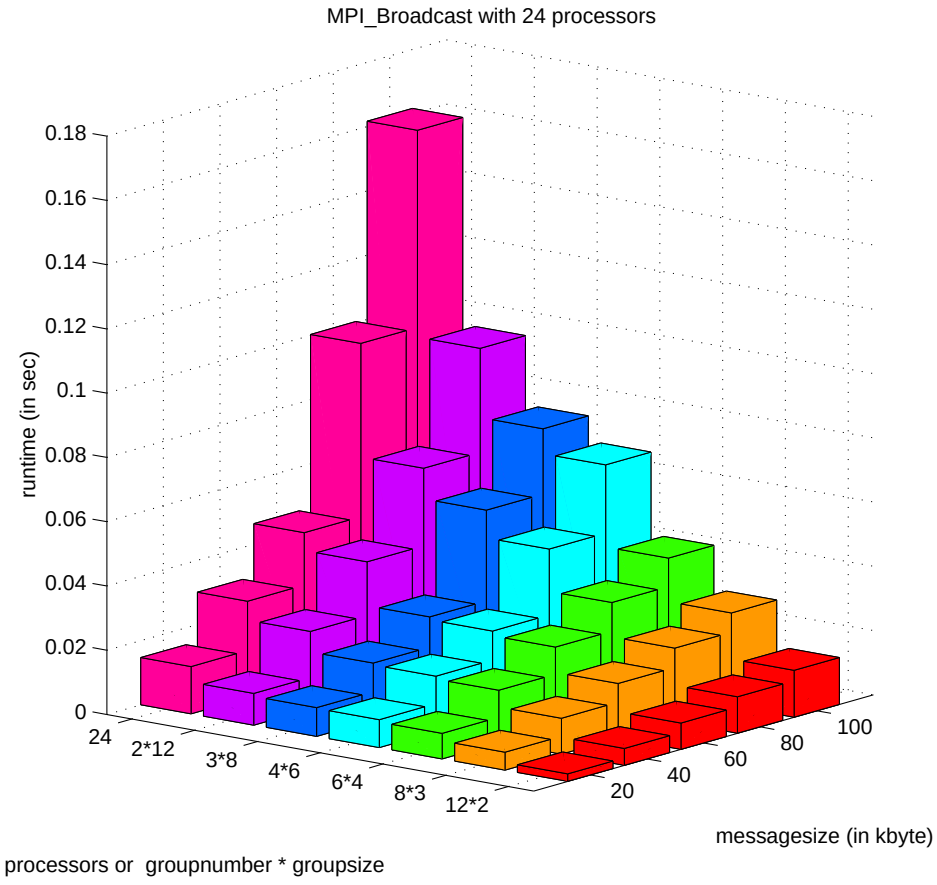
- Realization of task parallelism: **multiprocesor tasks (M-tasks)**;
  An M-task program is a collection of M-tasks that can be executed on an arbitrary number of processors;
- Using M-tasks that are **concurrently executed** and perform mainly **internal communication** can significantly **reduce the communication overhead**;
  main reason: collective communication operations are executed on **subsets of processors**
- The M-task execution can be **adapted to the execution environment**; the **number of executing processors**, the **execution order** (sequentially or concurrently) and the **processor assignment** can be adapted;

Group-based Broadcast Operation

global communication operations:
linear or logarithmic cost functions



Beowulf Cluster CLiC



Multiprocessor-Tasks programming

- An **M-task** can be executed by an **arbitrary number of processors** (malleable task)
- Independent tasks can be executed concurrently by **disjoint processor groups**
- Each M-task can be internally computed in a **data-parallel or SPMD way** (or threads)
  $\rightsquigarrow$ **specific mixed parallel execution**
- each M-task can be activated at different program points with a varying number of processors
- M-tasks can contain point-to-point-communication and collective communication operations

Scheduling and data distribution

- **static** scheduling of M-Tasks **(compiler framework TwoL)**:
  - **Advantage:**
    fixed data distribution and redistribution $\longrightarrow$ data distribution **cost** are known for scheduling
  - Limited class of applications: M-task structure has to be known in advance
- **dynamic** scheduling of M-Task **(runtime library Tlib)**:
  - **Advantage:**
    recursive, hierarchical M-task structure; divide&conquer algorithms
  - **But:** adaptive re-distibutions for different data layouts and group structures is required

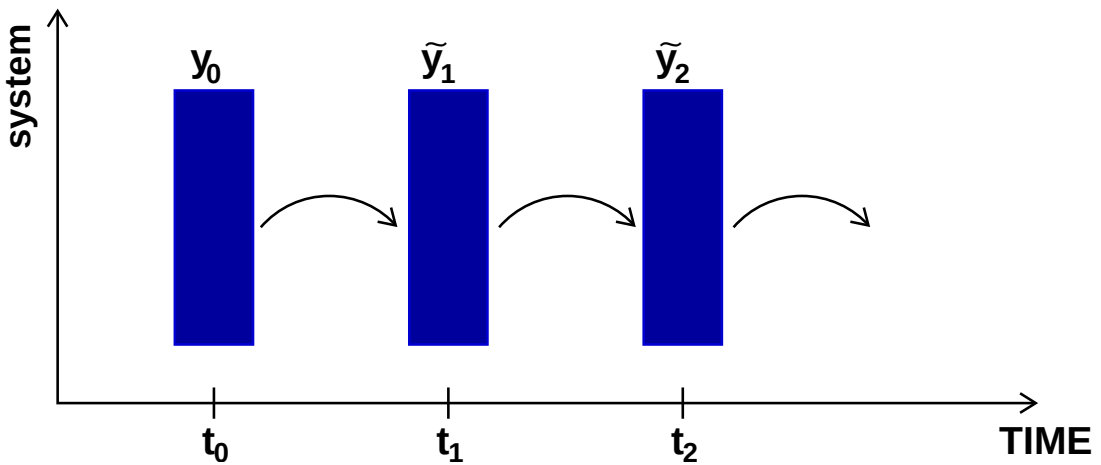
Outline

Ordinary Differential Equations

Initial value problem

$$y'(t) = f(t, y(t)), \quad f : R \times R^{\mathbf{d}} \rightarrow R^{\mathbf{d}}$$
$$y(t_0) = y_0, \quad y_0 \in R^{\mathbf{d}}$$

One-step methods:



Sources of parallelism within one time step:

- parallelism across the method
- parallelism across the system

General linear method

Computation of **k** stage values in time step n at time t_n

$$Y_n = \underbrace{(y_{n\mathbf{1}}, \ldots, y_{n\mathbf{k}})}_{\text{stage values}} \qquad \text{vector of size } \mathbf{d} \cdot \mathbf{k}$$

Stage value y_{ni} is the approximation of $y(t_n + a_i h)$
Computation in each time step

$$Y_{n+1} = \underbrace{(R \otimes I)}_{\text{Kronecker}} \; Y_n + h(S \otimes I)F(Y_n) + h(T \otimes I)F(Y_{n+1})$$

with R, S, T $\mathbf{k} \times \mathbf{k}$ matrices and I $\mathbf{d} \times \mathbf{d}$ matrix
$F(Y_n) = (f(y_{n\mathbf{1}}), \ldots, f(y_{n\mathbf{k}}))$ vector of size $\mathbf{d} \cdot \mathbf{k}$

Kronecker product

$$R \otimes I = \underbrace{\begin{pmatrix} r_{11} & \cdots & r_{1k} \\ \vdots & & \vdots \\ r_{k1} & \cdots & r_{kk} \end{pmatrix}}_{k} \otimes \underbrace{\begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{pmatrix}}_{d}$$

$$= \begin{pmatrix} r_{11} & 0 & r_{12} & 0 & \cdots & r_{1k} & 0 \\ 0 & r_{11} & 0 & r_{12} & & 0 & r_{1k} \\ r_{21} & 0 & & & & & 0 \\ 0 & r_{21} & & & & & \\ & & \vdots & & & \vdots & \\ r_{k1} & 0 & & & & r_{kk} & 0 \\ 0 & r_{k1} & & & \cdots & 0 & r_{kk} \end{pmatrix}$$

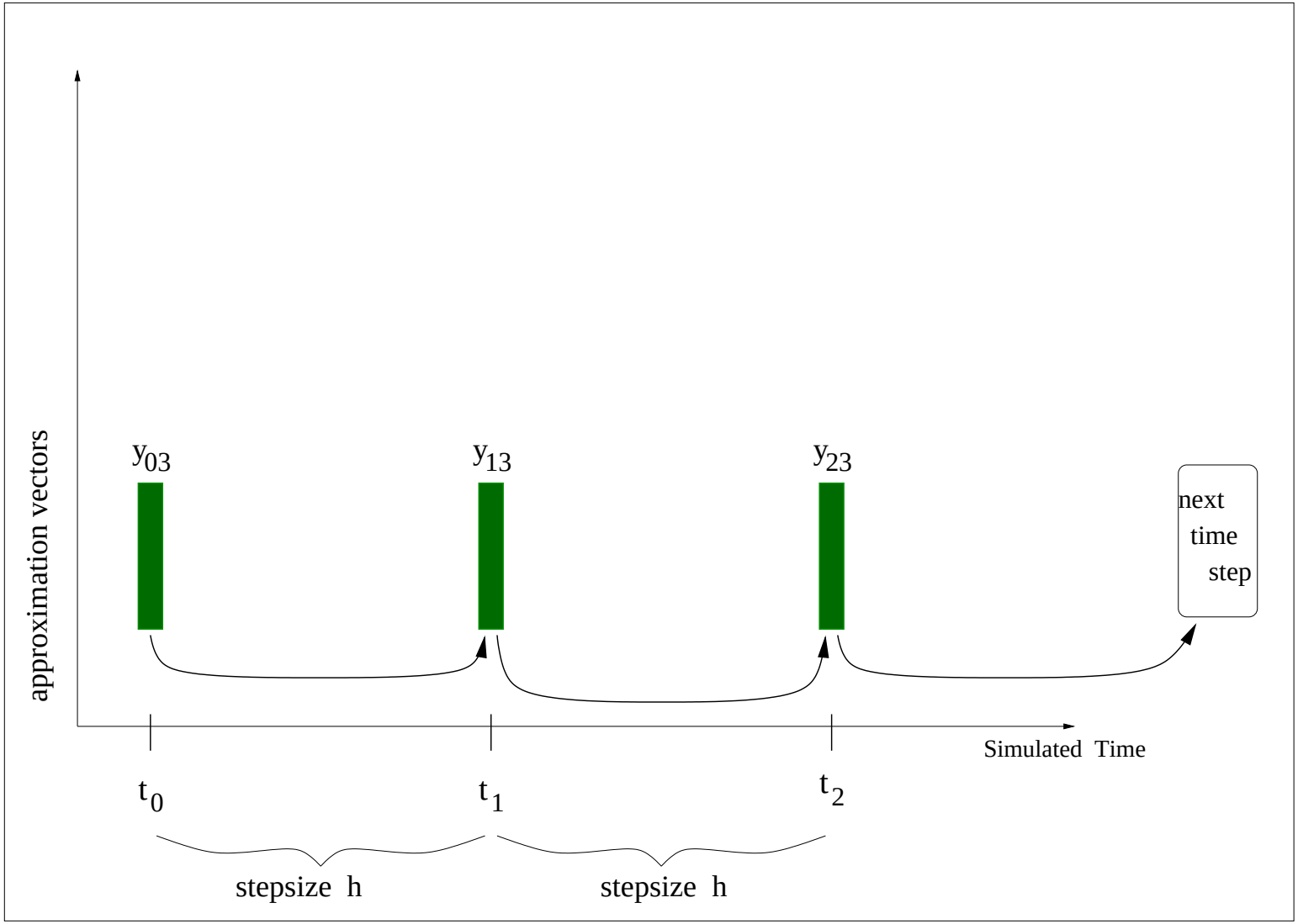
Parallel Adams Bashforth (PAB)

v.d.Houwen, Messina 99

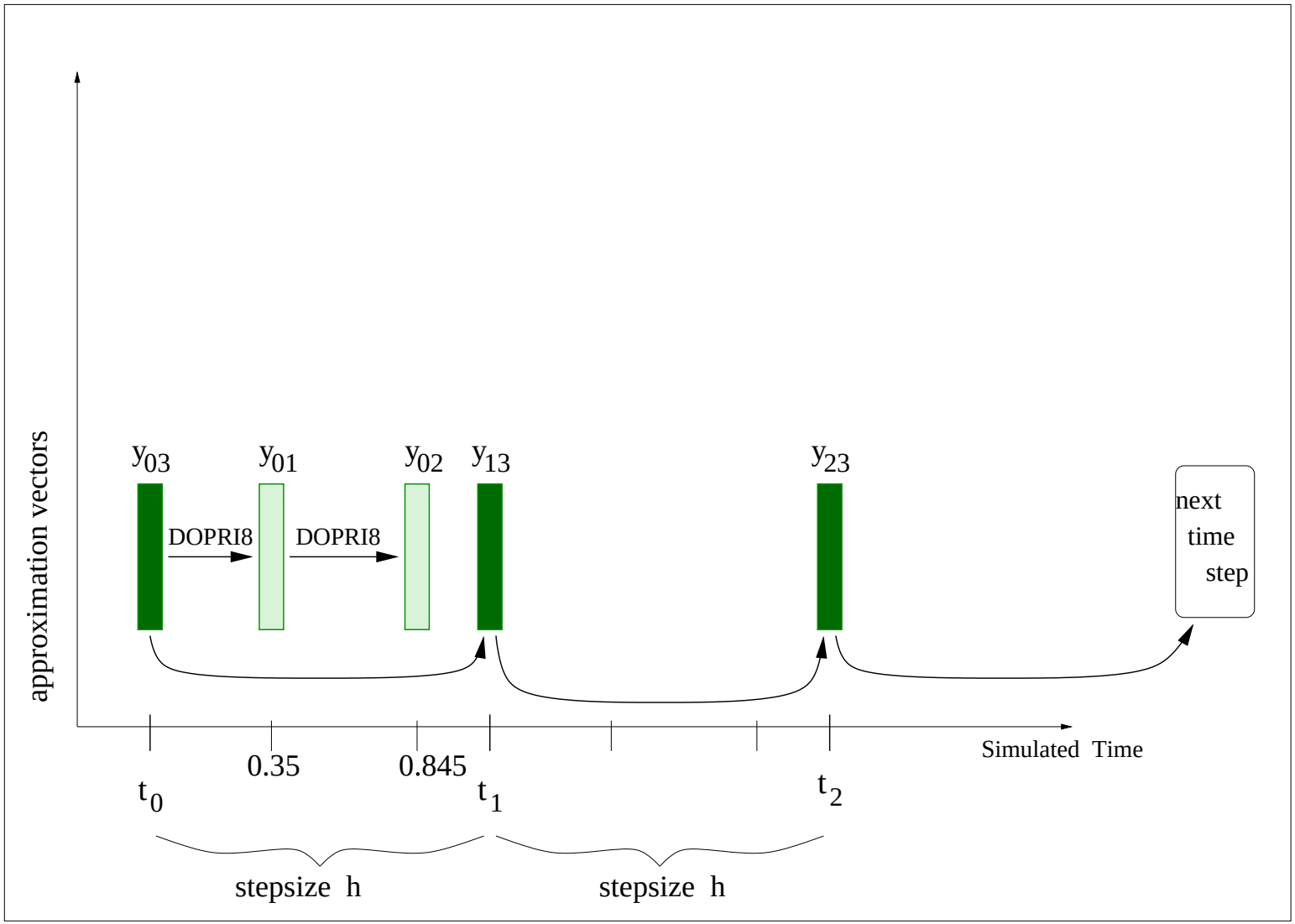
- $T = 0$ (explicit method)
- $R = e \cdot e_k^T \;\;,\;\; e = (1, \ldots 1),\;\; e_k = (0, \ldots, 0, 1)$
- $S = V_a W_b^{-1} \;\;,\;\; S = (s_{ij}) i, j = 1, \ldots k$
- abscissa vector $a = (a_1, \ldots, a_k)$ Lobatto points
- explicit method (DOPRI8) to compute $Y_0 = (y_{01}, \ldots y_{0k})$

$$y_{n+1,\mathbf{i}} = \underbrace{y_{n,\mathbf{k}}}_{\substack{\mathbf{k}-th \\ \text{stage value}}} + h \sum_{l=1}^{k} s_{il} \underbrace{f(y_{nl})}_{\substack{\text{previous} \\ \text{stage} \\ \text{values}}} \;\;,\;\; i = 1, \ldots k$$

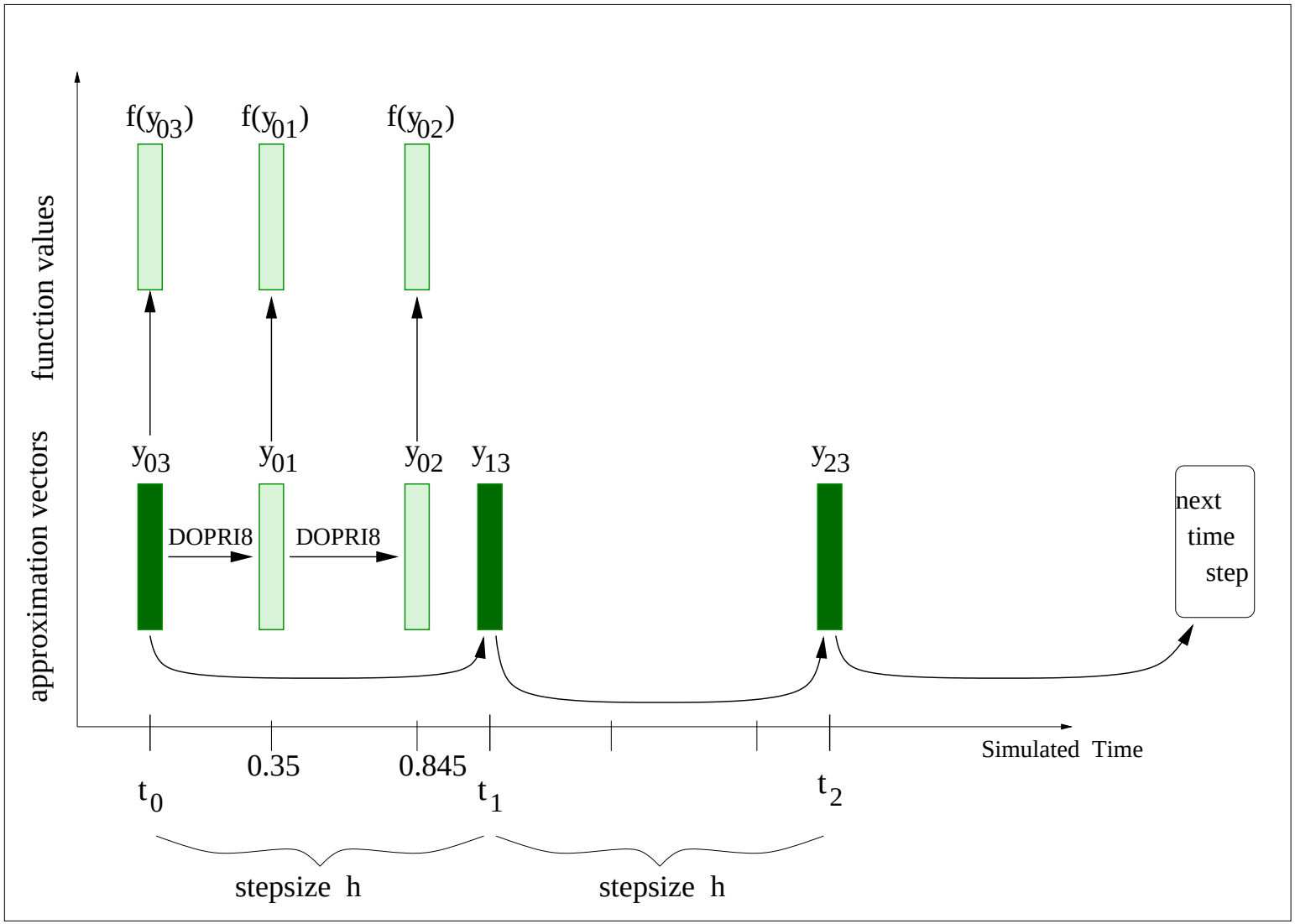
Parallel Adams Bashforth (PAB) $k = 3$ stage values



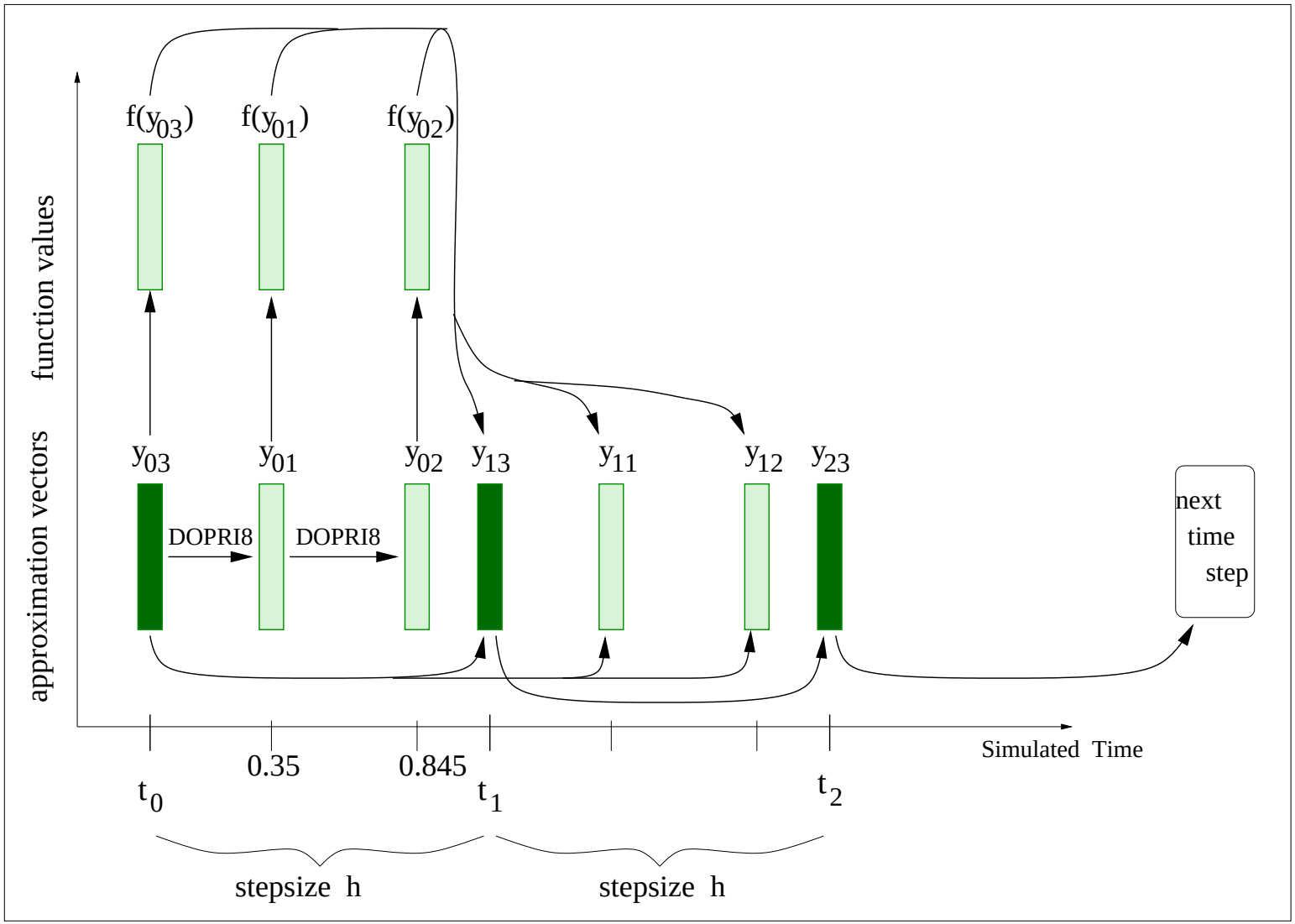
Parallel Adams Bashforth (PAB) $k = 3$ stage values



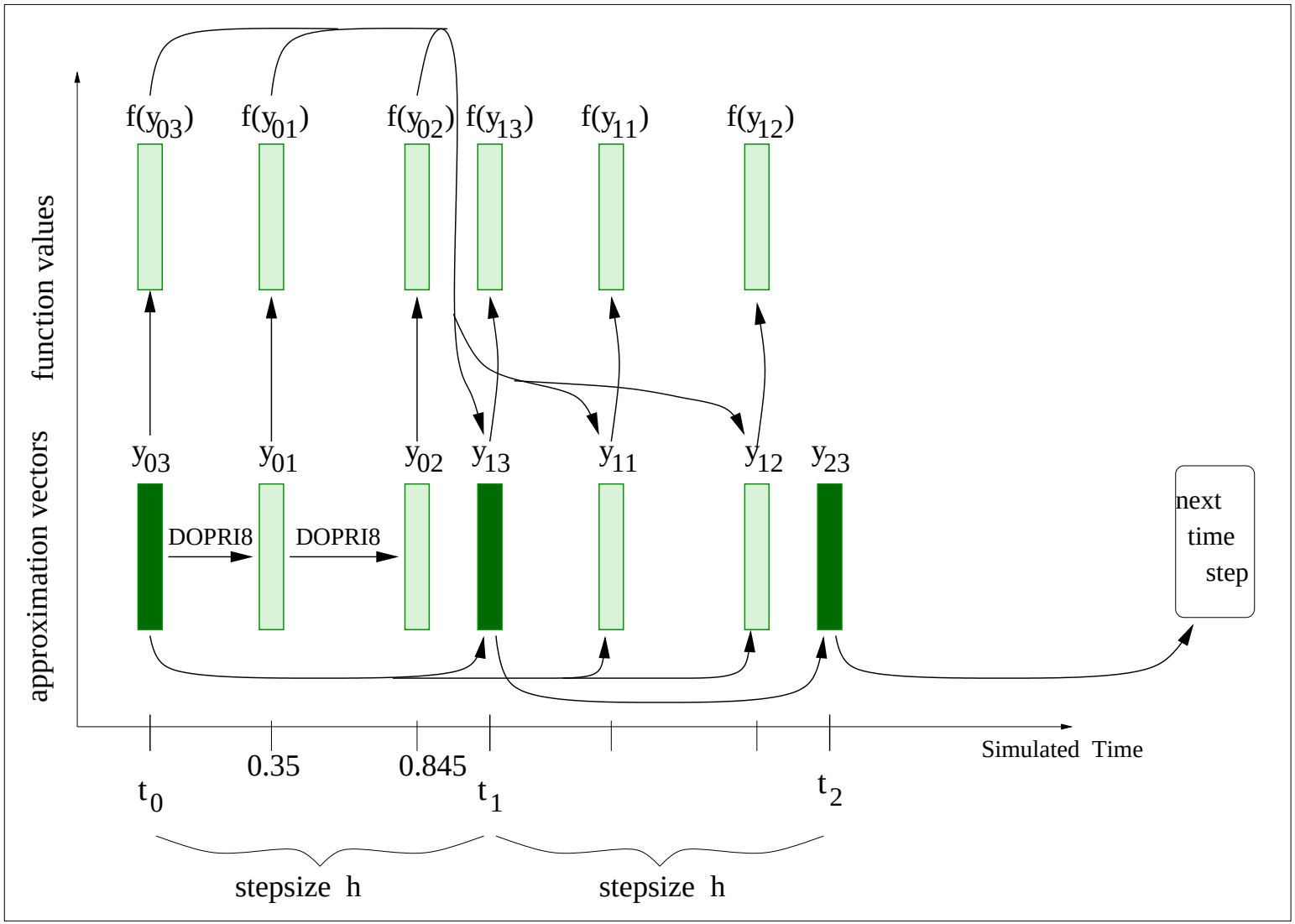
Parallel Adams Bashforth (PAB) $k = 3$ stage values



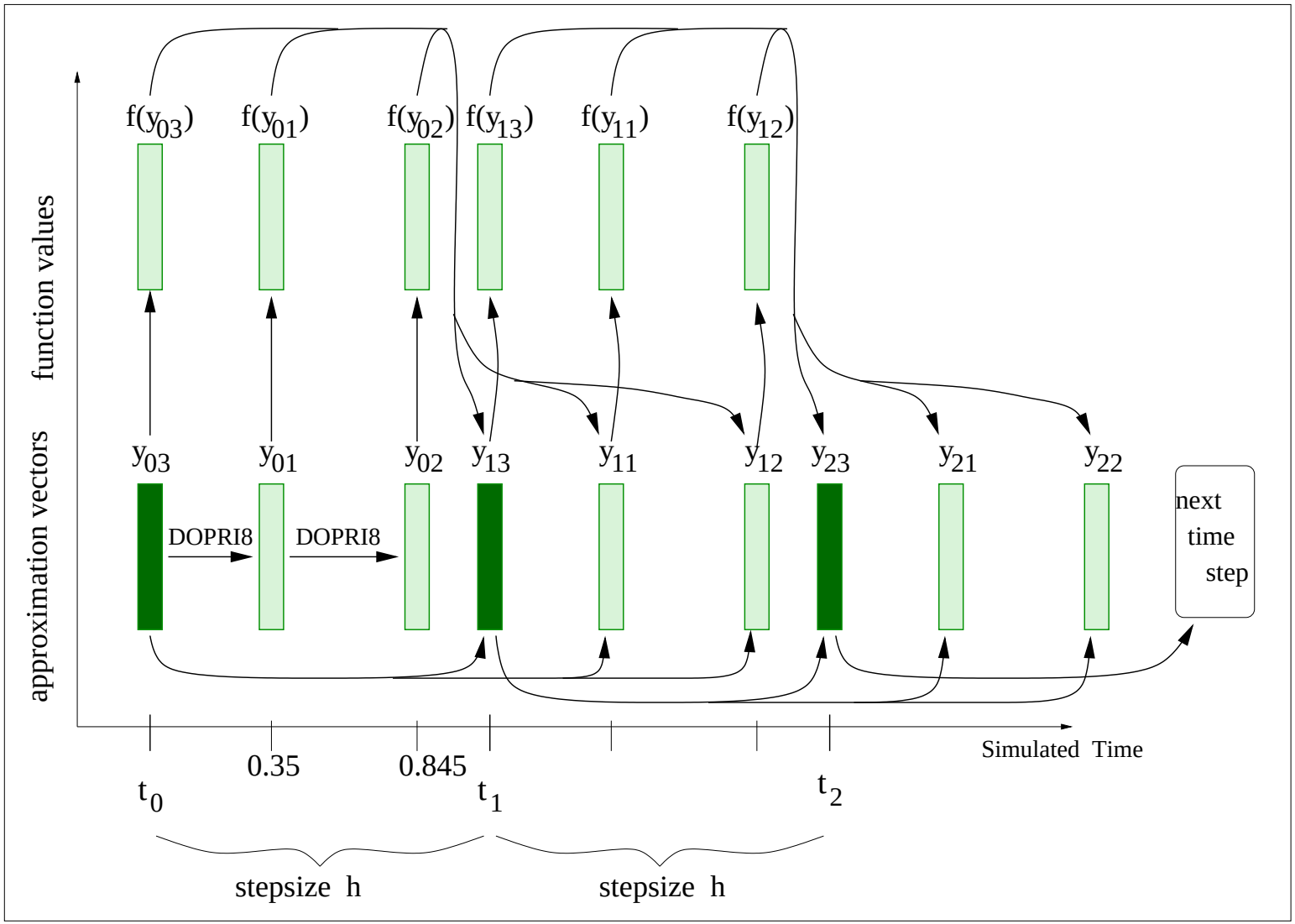
Parallel Adams Bashforth (PAB) $k = 3$ stage values



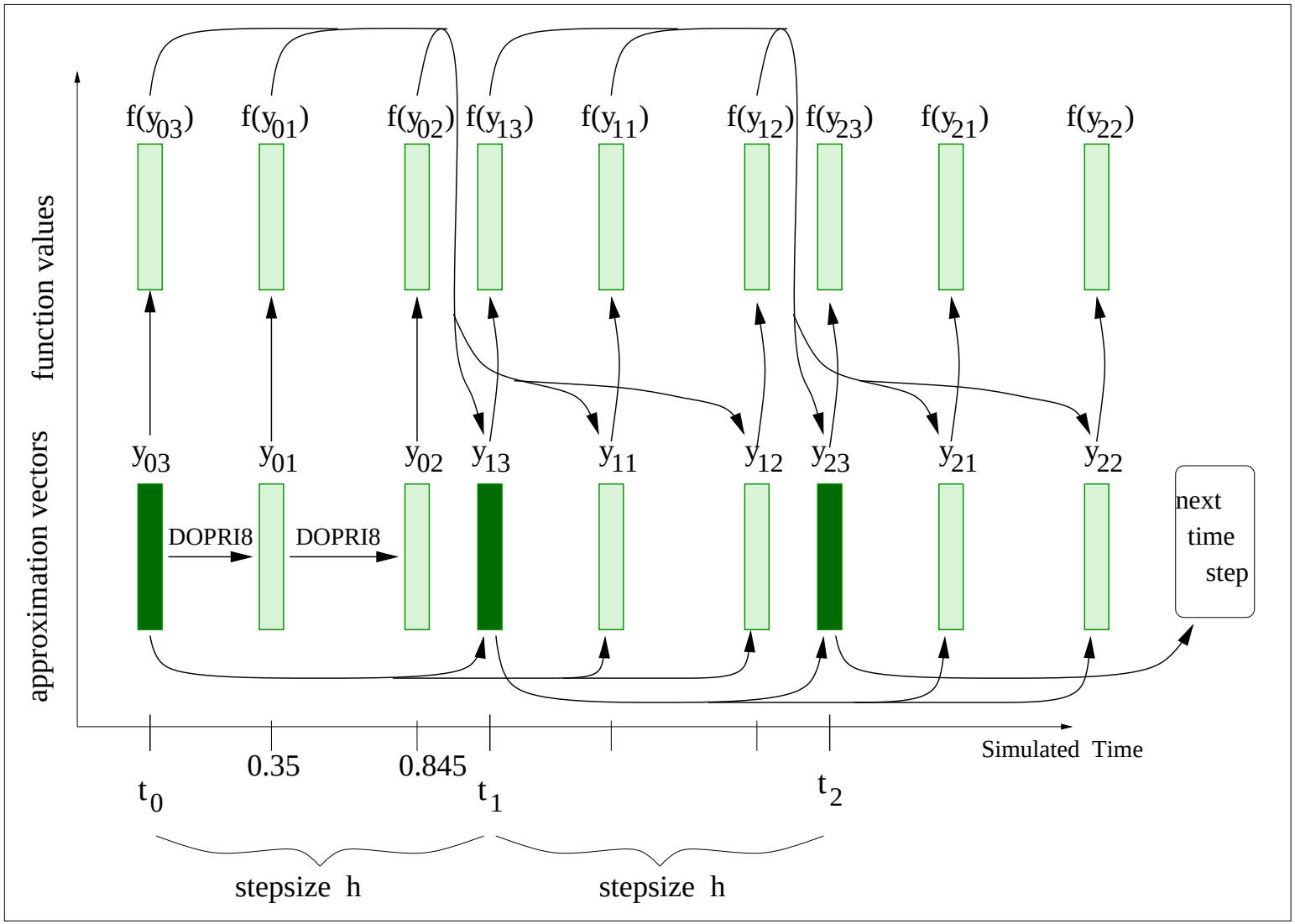
Parallel Adams Bashforth (PAB) $k = 3$ stage values



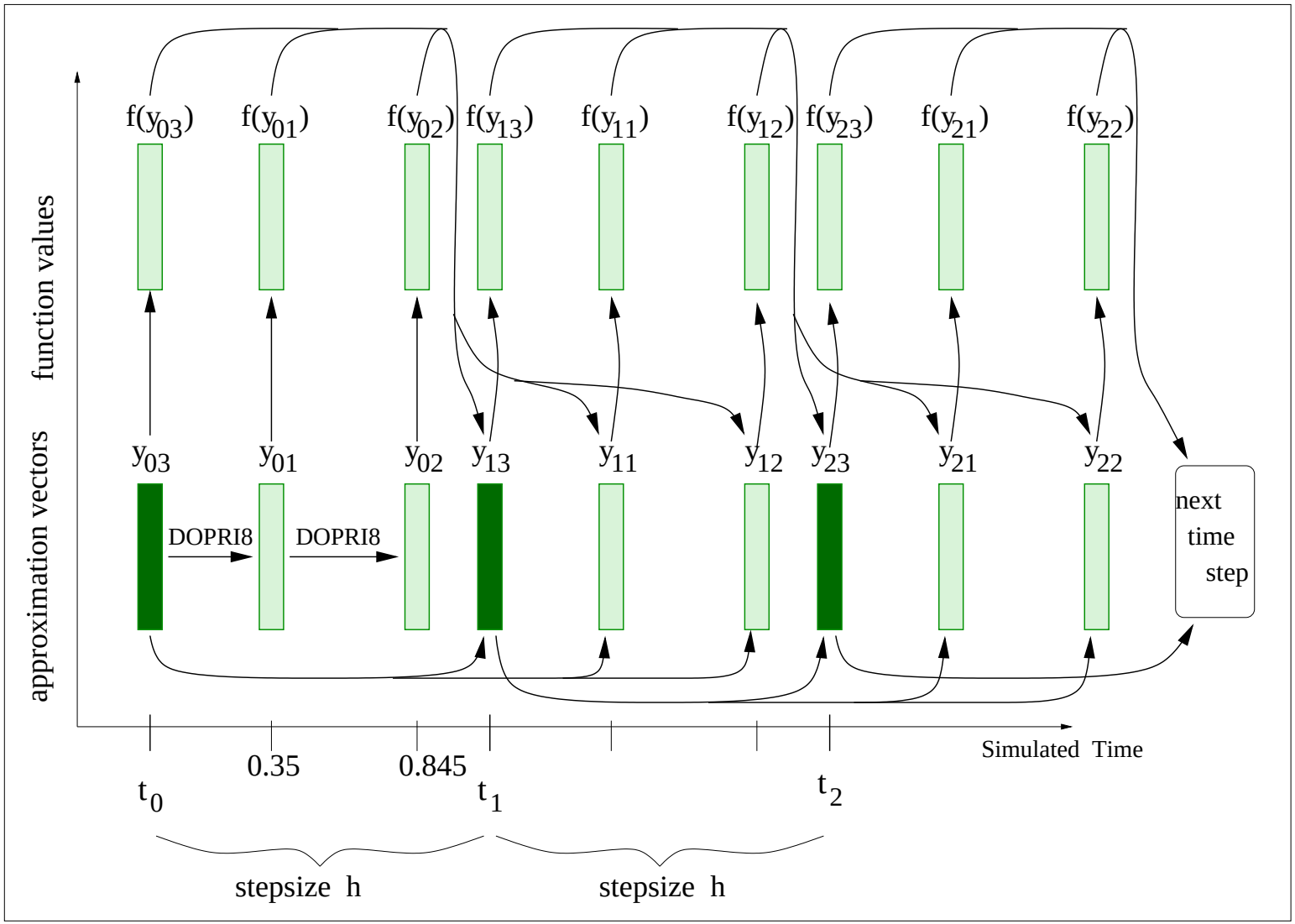
Parallel Adams Bashforth (PAB) $k = 3$ stage values



Parallel Adams Bashforth (PAB) $k = 3$ stage values



Parallel Adams Bashforth (PAB) $k = 3$ stage values



Parallel Adams Moulton (PAM)

- T diagonal matrix (implicit method)
  specific value $\delta_i, i = 1 \ldots k$
- $R = e \cdot e_k^T$
- $S = (V_a + RV_b - TW_a)W_b^{-1}$ (see v.d.Houwen, Messina)
- $a = (a_1, \ldots, a_k)$ Lobatto points

$$\underbrace{y_{n+1,\mathbf{i}} - h * \delta_{\mathbf{i}} * f(y_{n+1,\mathbf{i}})}_{\text{implicit}} = \underbrace{v_{n\mathbf{i}}}_{\text{contains } y_{n\mathbf{i}}} \quad i = 1, \ldots, k$$

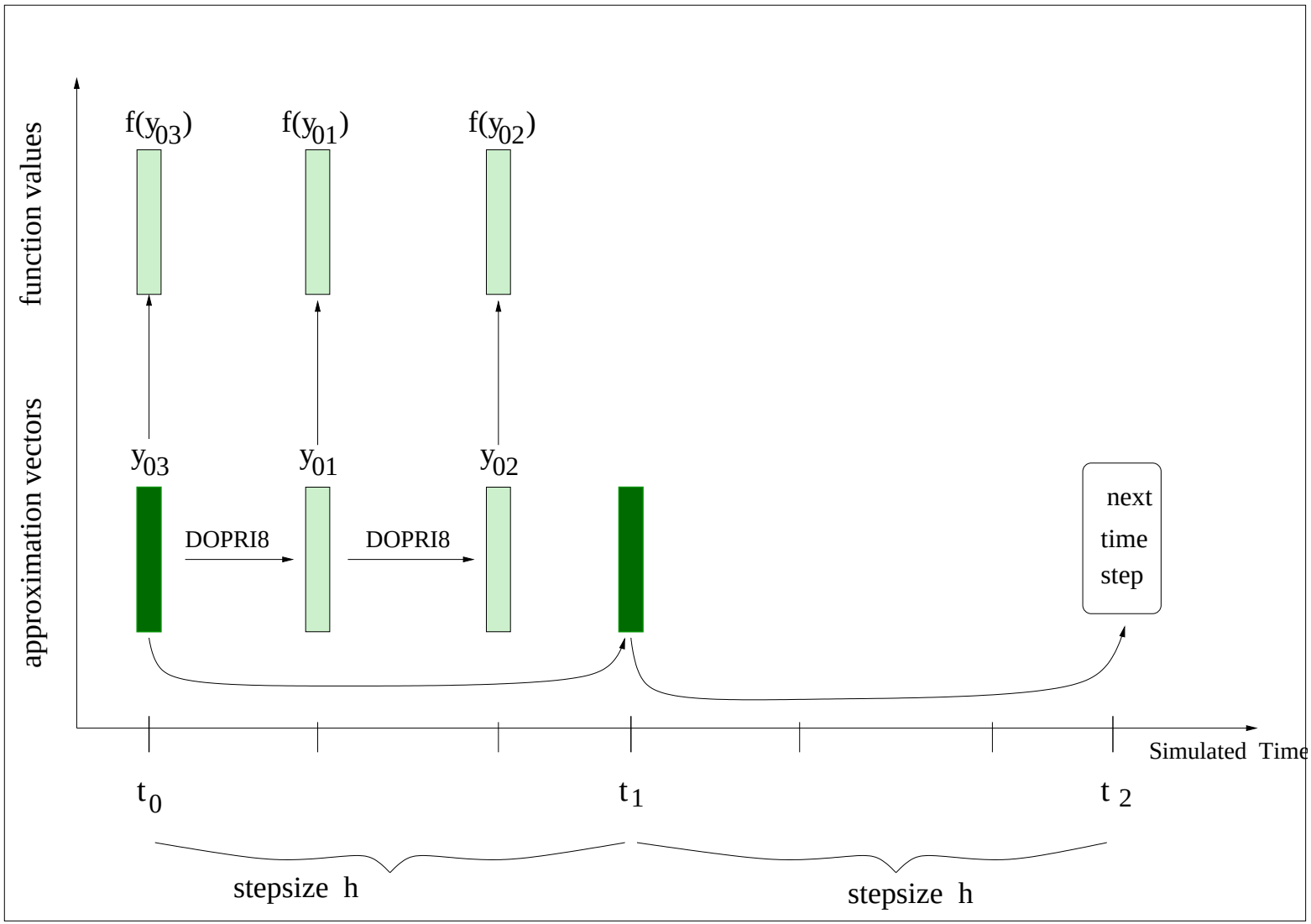
$\rightsquigarrow$ fixed point iteration

$$y^{(j)}_{n+1,\mathbf{i}} - h * \delta_{\mathbf{i}} * f(y^{(j-1)}_{n+1,\mathbf{i}}) = v_{n\mathbf{i}} \quad i = 1, \ldots, k$$

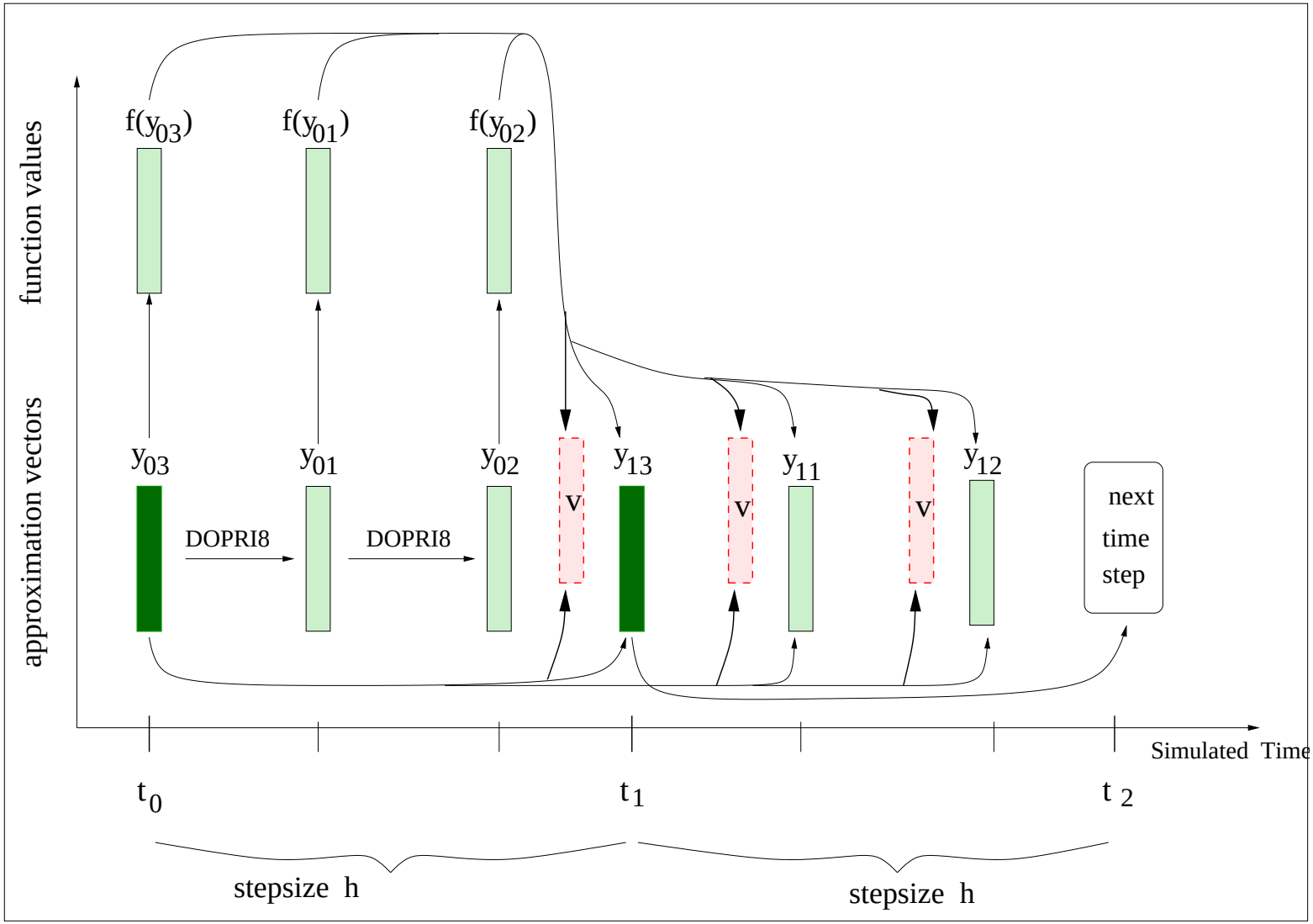
Parallel Adams Bashforth Moulton

- $k = 3$ stage values
- Stage values y_{ni}, $i = 1, \ldots, k$, are computed by the PAB method are as starting vector for the fixed point iteration of the PABM method
- Additional vector v_{ni} to hold constant part of the right hand side of the fixed point iteration
- Similar computation for v_{ni} and y_{ni} with different matrix S

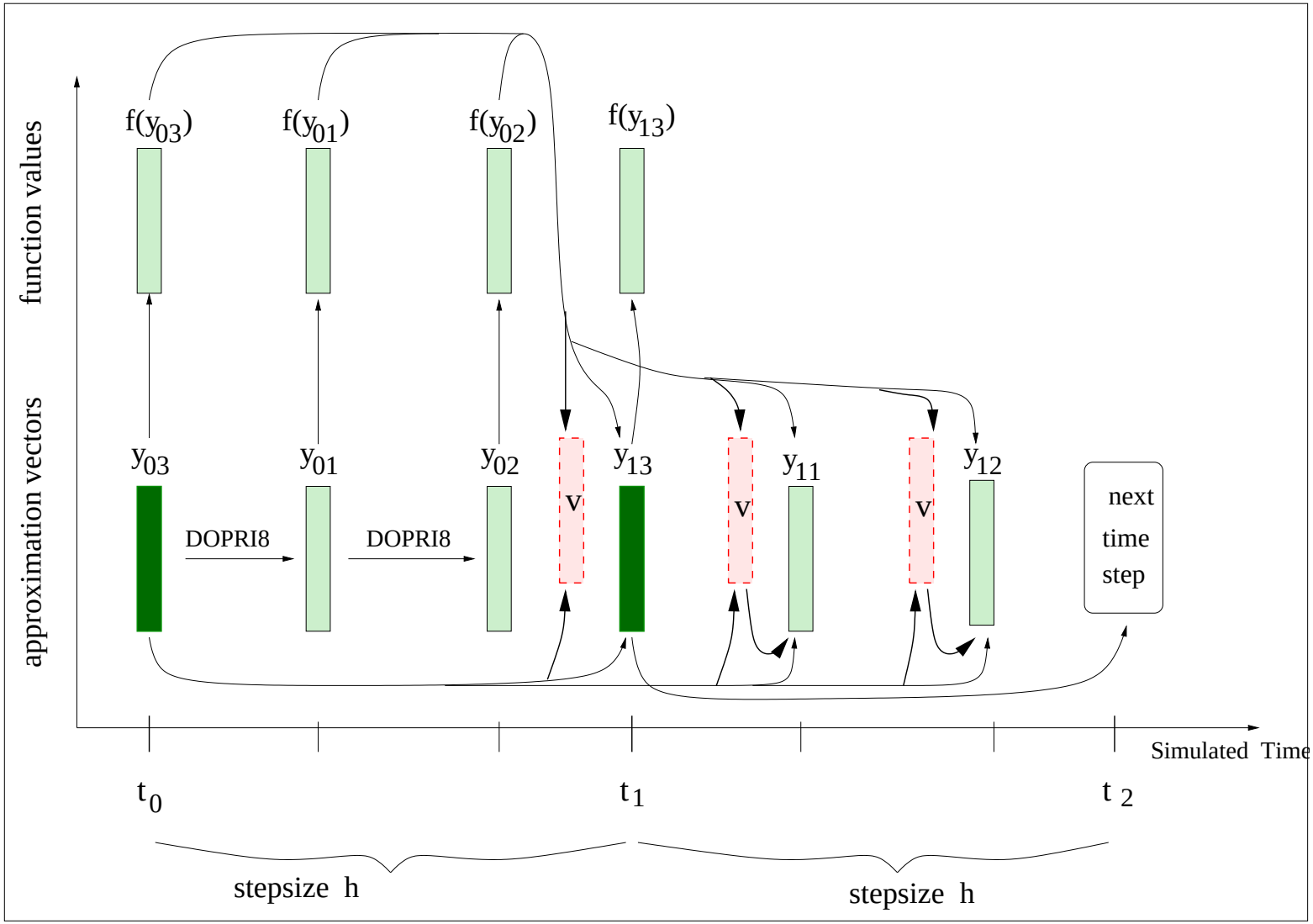
Parallel Adams Moulton



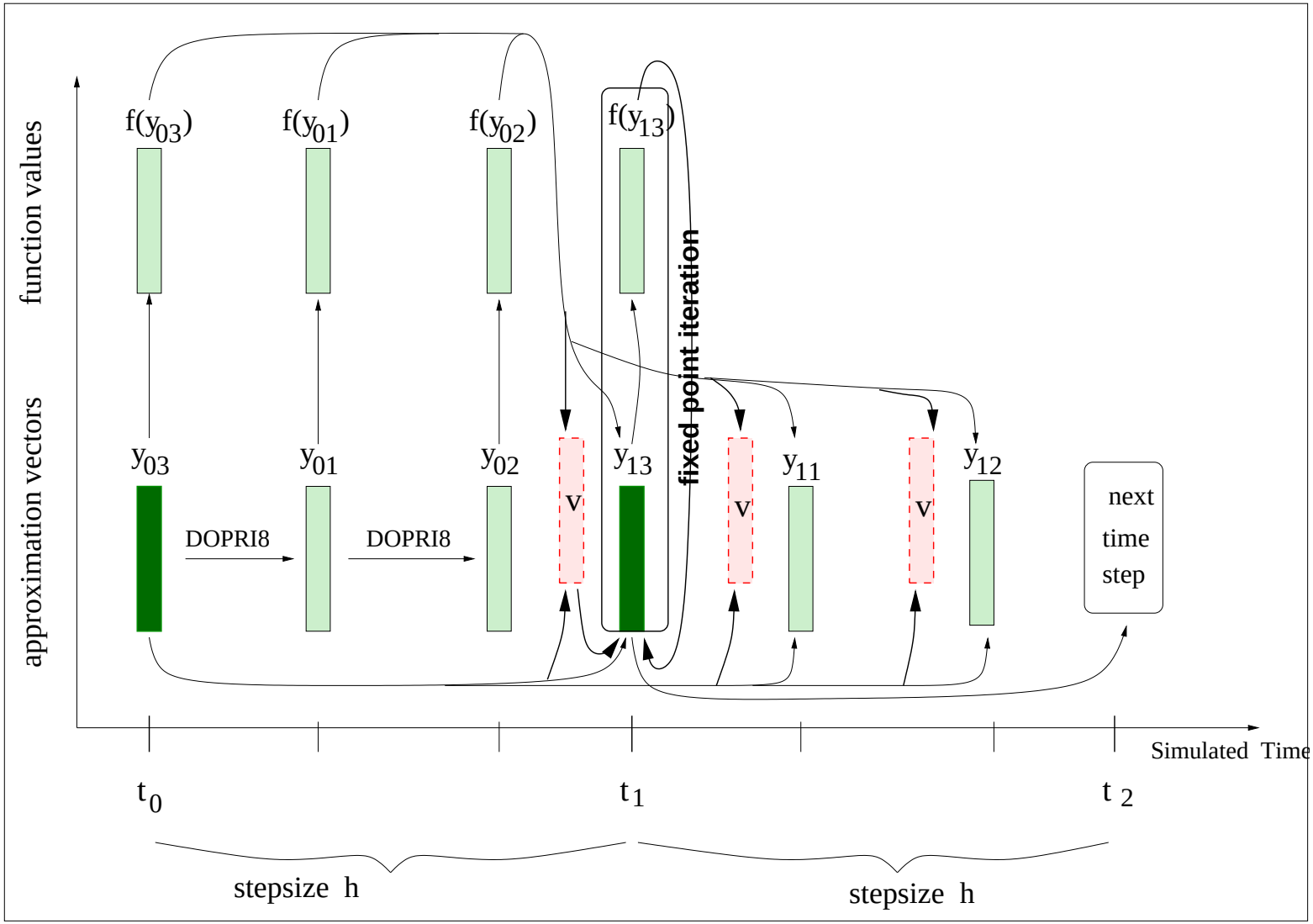
Parallel Adams Moulton



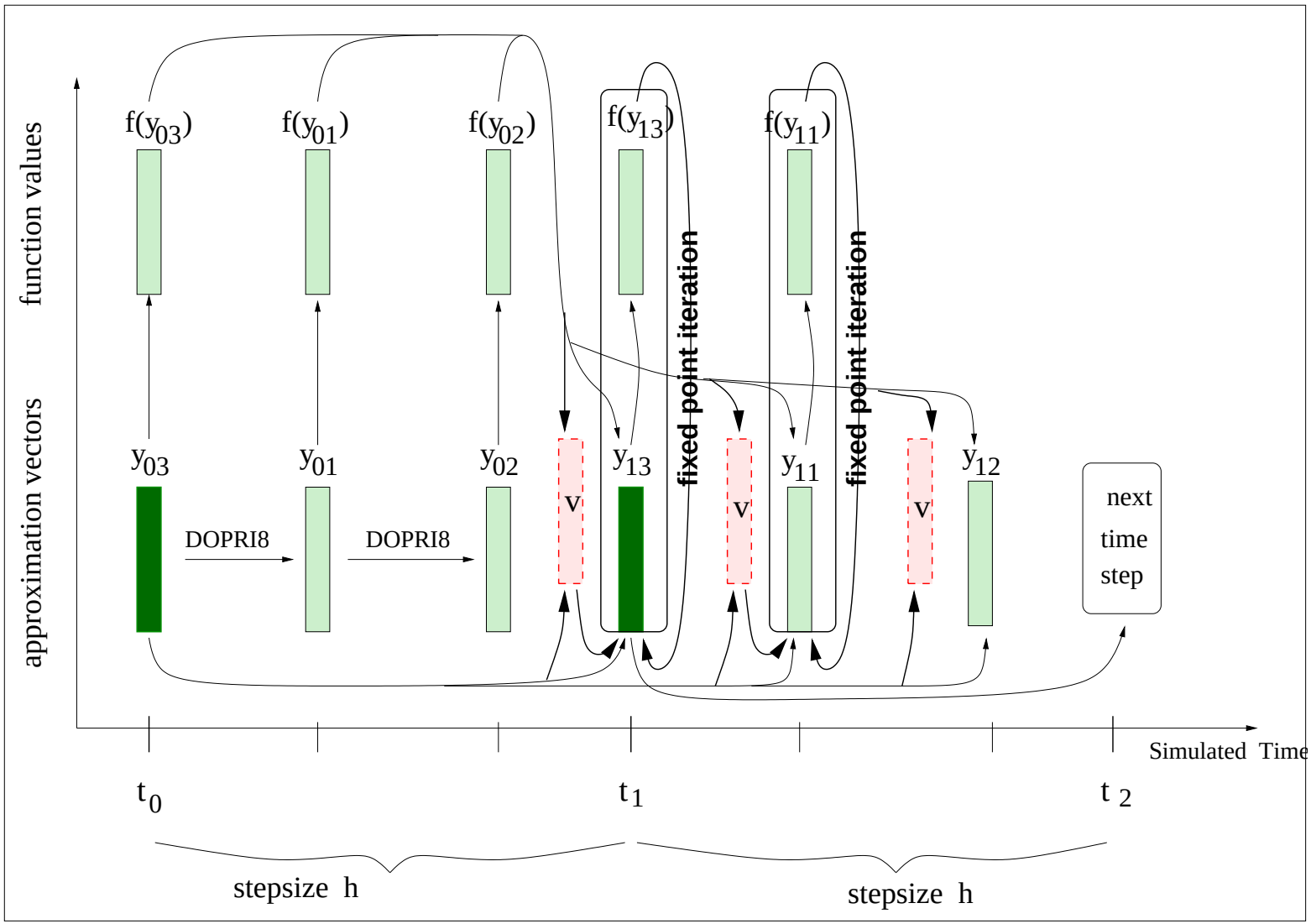
Parallel Adams Moulton



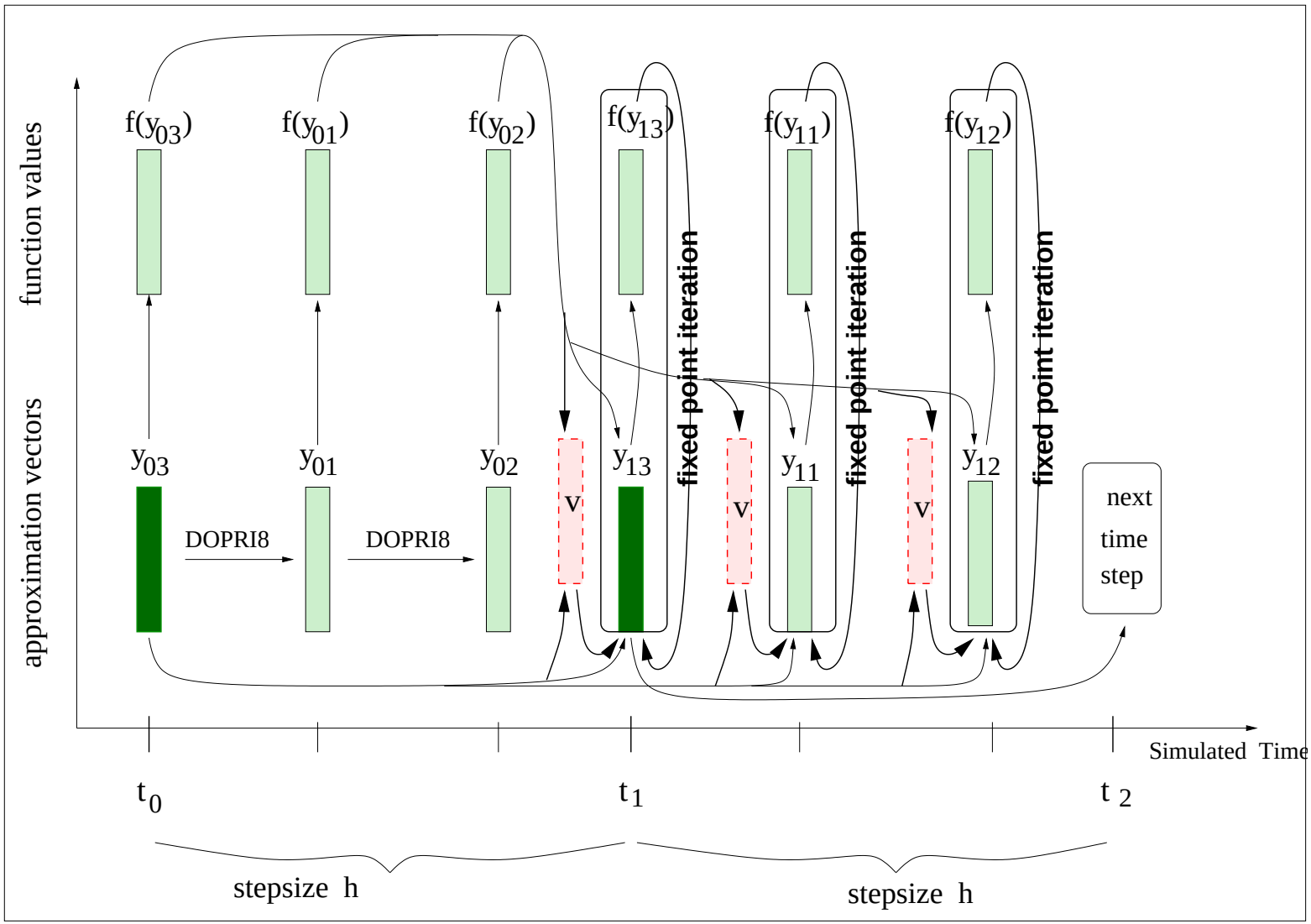
Parallel Adams Moulton



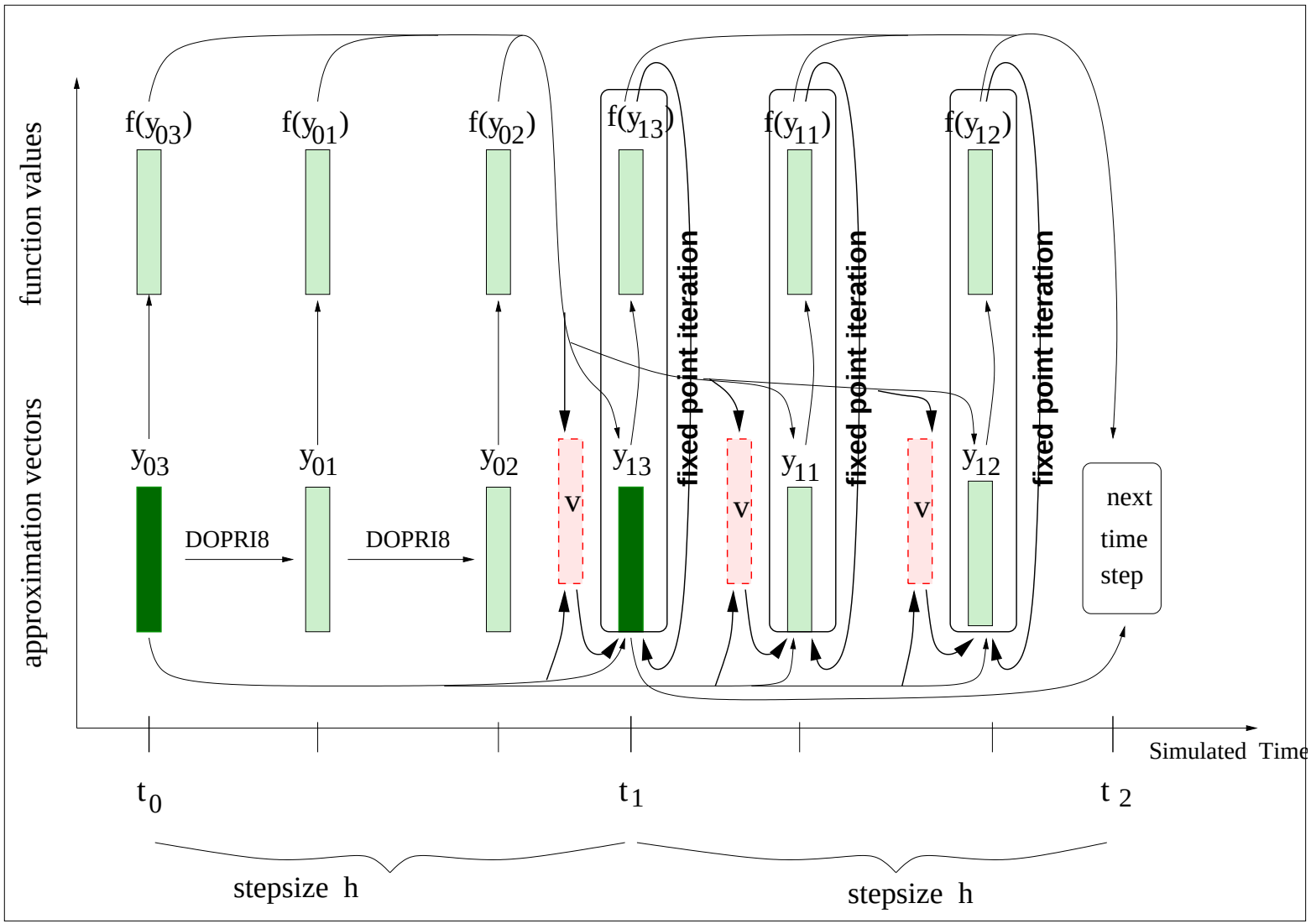
Parallel Adams Moulton



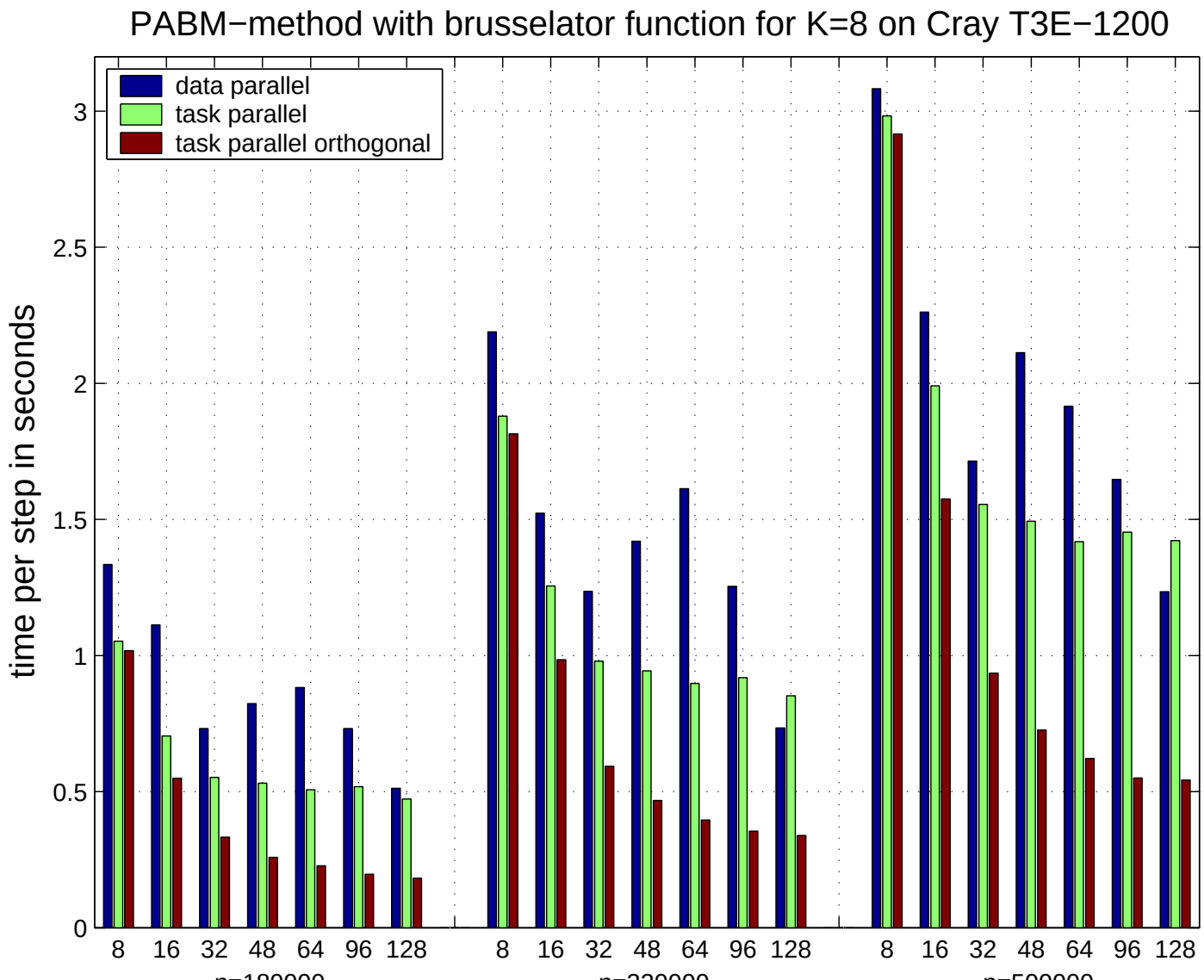
Parallel Adams Moulton



Parallel Adams Moulton



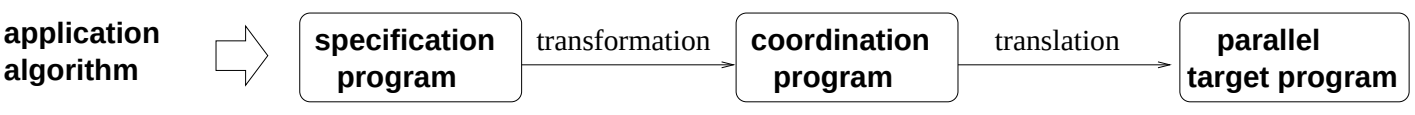
Experiments k=8 [Europar 04]



Outline

Compiler framework TwoL – (Two-Level Parallelism)

Mixed task and data parallelism:

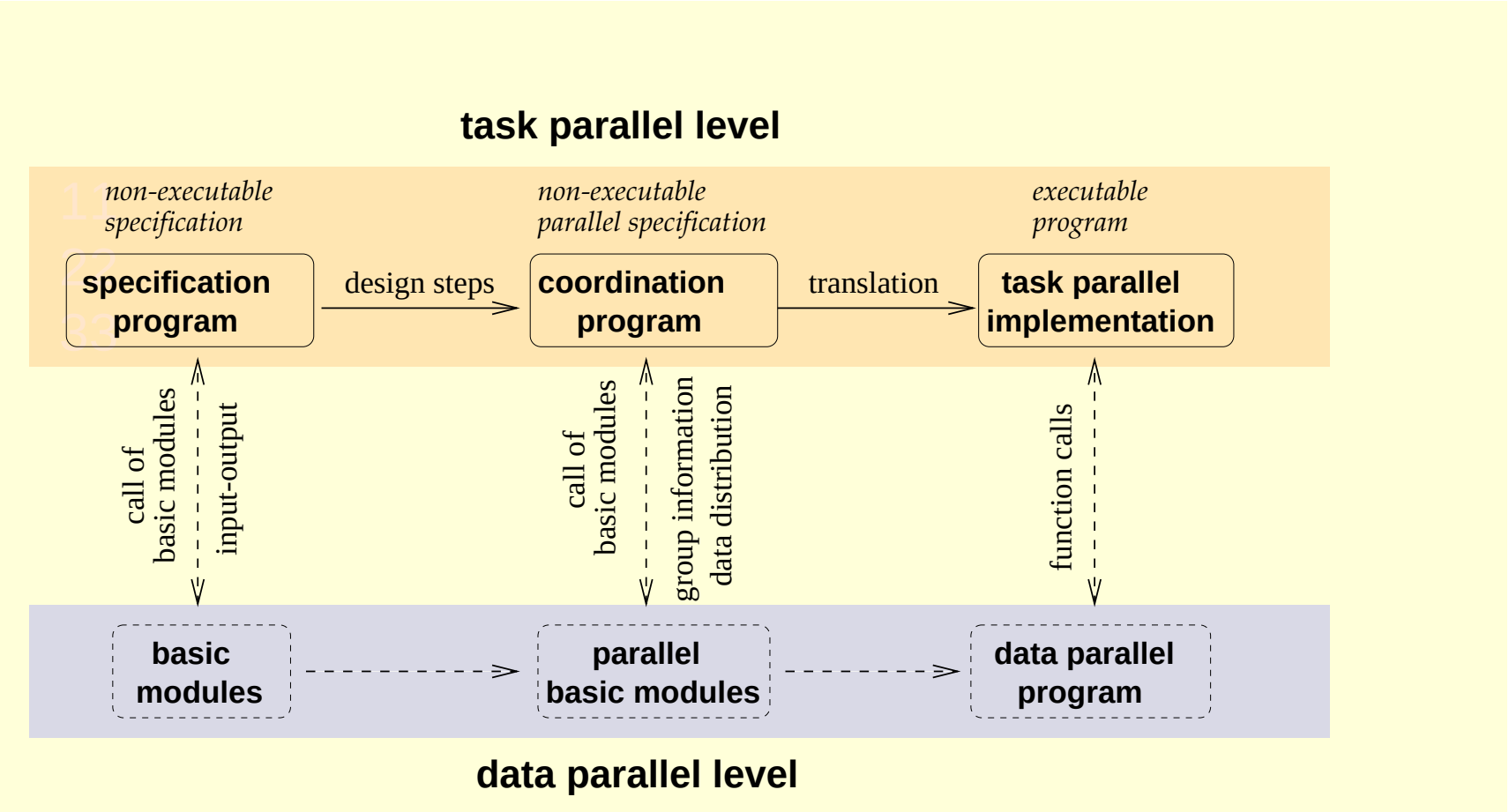


specification program maximum degree of parallelism

- composed modules → medium grain task parallelism (coordination language)
- basic modules → fine grain data parallelism

coordination program exploited degree of parallelism

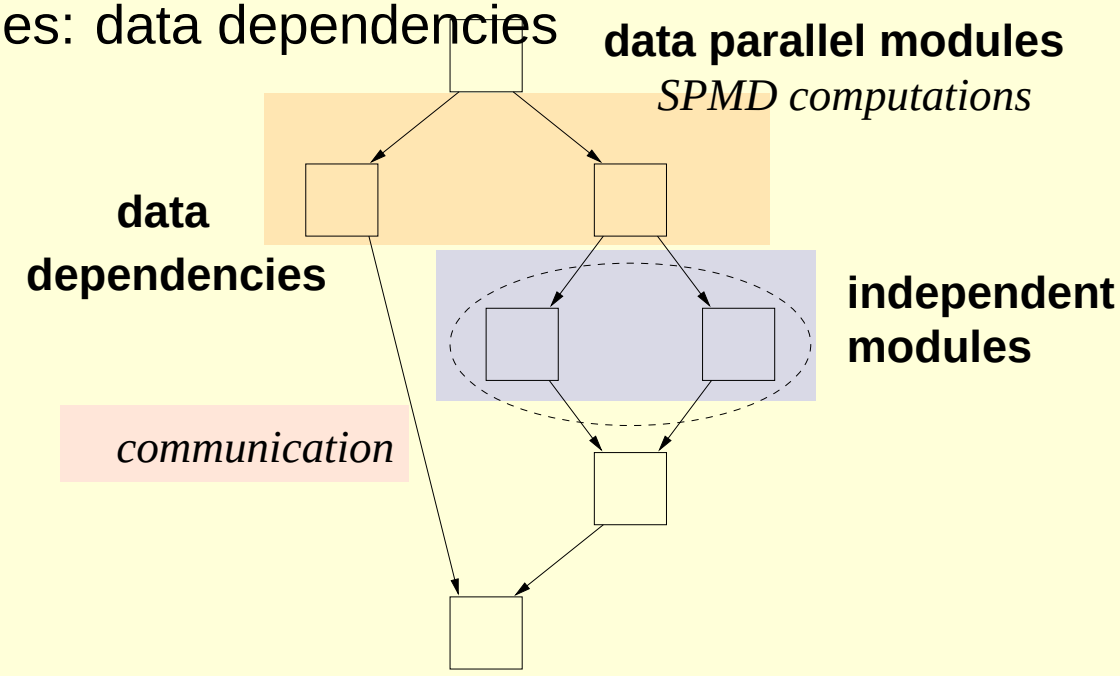
Interaction between task and data parallel level



Task graph

dependencies between modul activations

- nodes: activations of basic modules
- edges: data dependencies



difference to normal flow graphs: equivalent representation as module tree

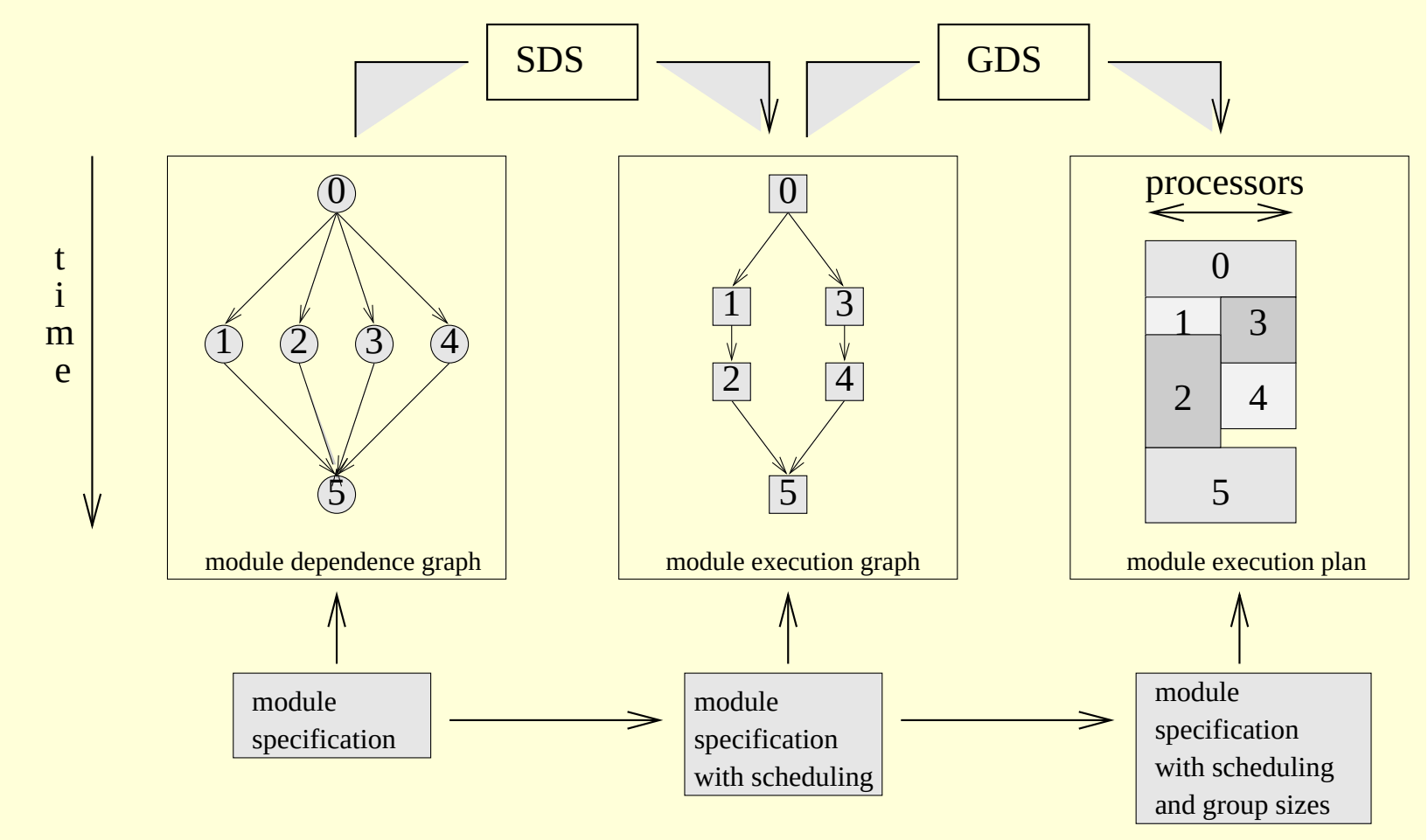
Design decisons

- - load balance – sizes of processor subgroups
  - communication – internal collective communication operations
  - data redistribution between data parallel modules
- $\longrightarrow$ multiprocessor task scheduling with dependencies of data parallel modules
- cost model: function of application and machine specific parameters

$$TIME = F_{parallel}(n, p, maschineparameters)$$

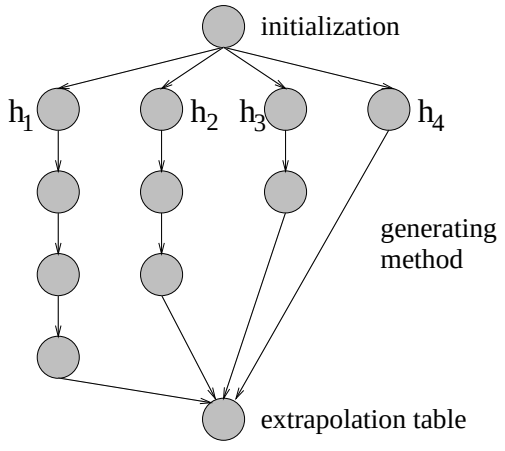
for the execution time computation, communication and data redistribtion times

Overview of the transformation steps



Extrapolation methods for ODEs

one-step method for solving ODEs:
in each time step: r different approximations with different step-sizes $h_0, \ldots, h_{r-1}$ and combination to approximation of higher order

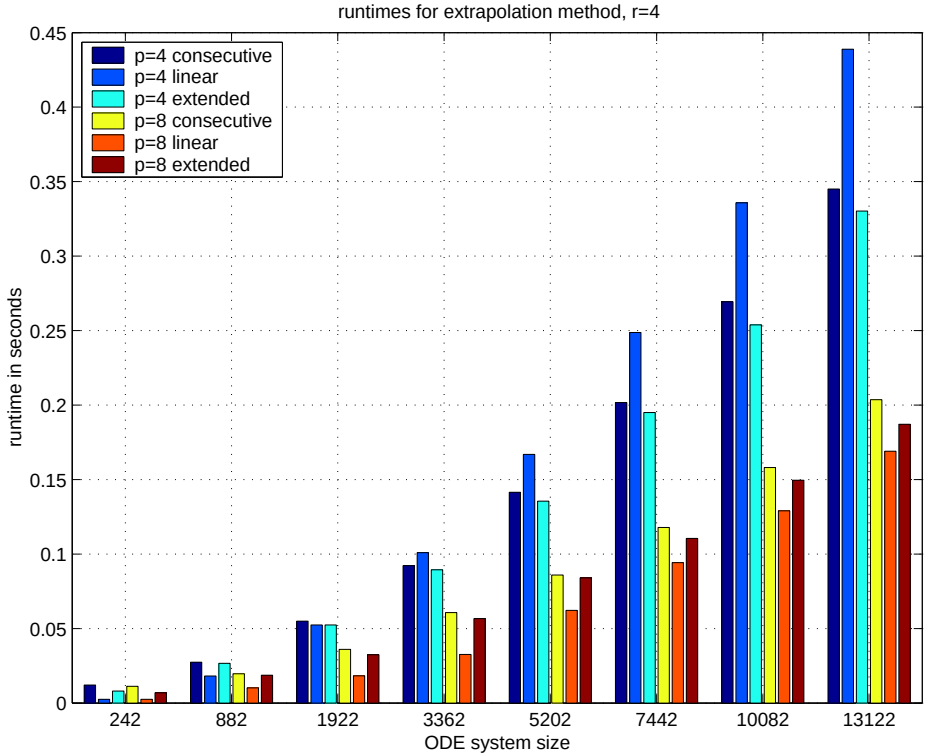


possible task organizations:

- **data parallel** execution
- **linear** group partitioning: use r processor groups and adapt size of the groups to the computational effort: group i contains $g_i = p \cdot \frac{i+1}{1/2 \cdot r(r+1)}$ processors
- **extended** group partitioning: use $\lceil r/2 \rceil$ groups; group i executes the computations for step-sizes h_i and h_{r-i}

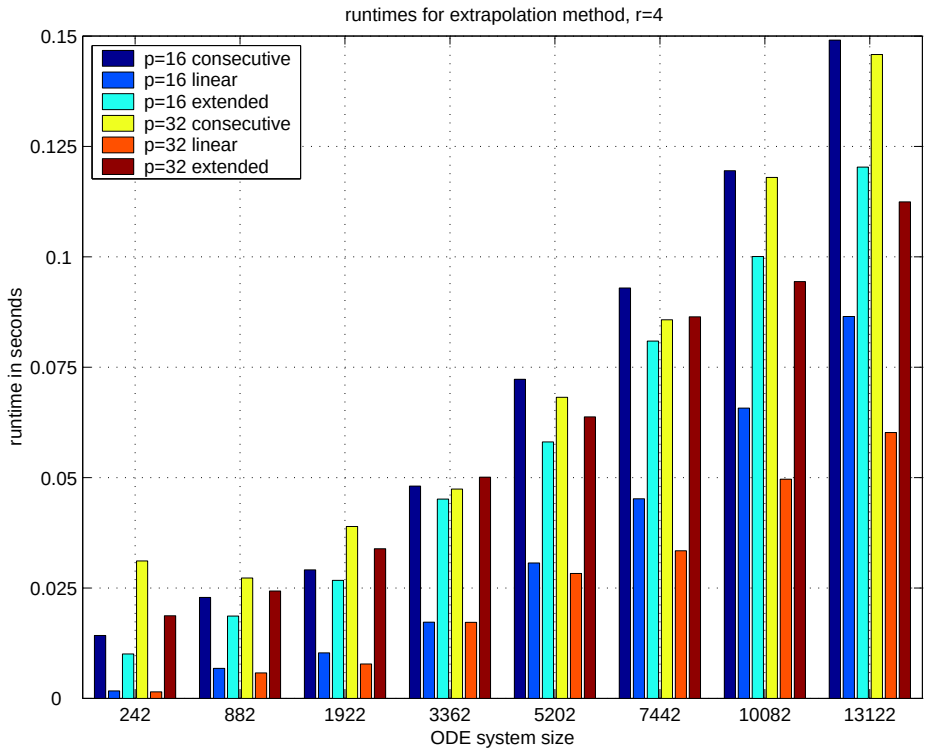
Extrapolation method on the Cray T3E

runtime per time step for sparse non-stiff ODE system; $p = 4$ and $p = 8$ processors of a Cray T3E



Extrapolation method on the Cray T3E — cont.

runtime per time step for sparse non-stiff ODE system
$p = 16$ and $p = 32$ processors of a Cray T3E



Outline

Tlib library – Basic steps

- The **programmer** has to identify the M-tasks;
  an automatic identification by a compiler tool is difficult;
- The **runtime library Tlib** allows the realization of **hierarchically structured M-tasks** programs;
  [SC02, JPDC05];
  the **execution order** of the M-tasks and the **group splittings** are specified statically by the programmer;
  the **number of executing processors** and the **recursion control** can be adapted **automatically**.
- **outlook**: dynamic adaptation of the execution order during M-task execution;

Tlib API

- Tlib is based on **MPI** and allows the use of arbitrary MPI functions.
- The Tlib API provides support for
  - the **creation** and **administration** of a **dynamic hierarchy of processor groups**;
  - the **coordination** and **mapping** of nested M-tasks;
  - the **handling** and **termination** of recursive calls and group splittings;
  - the organization of **communication between M-tasks**;
- The splitting and mapping can be adapted to the execution platform.

Interface of the Tlib library

- Type of all **Tlib tasks** (basic and coordination):

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

  - `void * arg` packed arguments for F;
  - `comm` MPI communicator for internal communication;
  - `pdescr` pointer to a descriptor of executing processor group;

  The function F may contain calls of Tlib functions to **create sub-groups** and to **map sub-computations** to sub-groups.

- **Initialization** of the Tlib library:

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

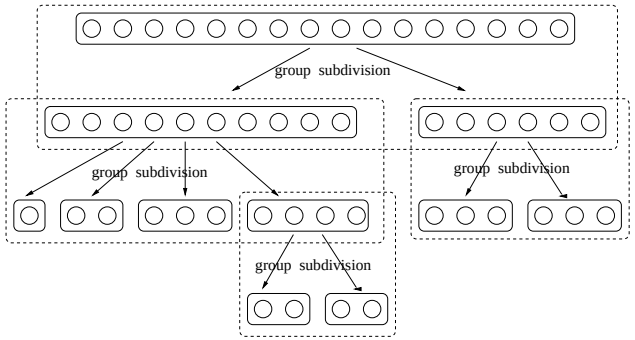
  - `argc , *argv[]`: arguments of main();
  - `comm`: initial MPI communicator;
  - `pdescr`: returns initial group descriptor;

Splitting operations for processor groups

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

- `pdescr`: current group descriptor
- `pdescr1`: new group descriptor for two new subgroups
- `per1` relative size of first subgroup
- `per2` relative size of second subgroup

```
int T_SplitGrpParfor( int n,
                      T_Descr *pdescr,
                      T_Descr *pdescr1,
                      float p[])
```




```
int T_SplitGrpExpl( T_Descr * pdescr,
                    T_Descr * pdescr1,
                    int color,
                    int key)
```

Concurrent execution of M-tasks

Mapping of two concurrent M-tasks to processor groups:

```
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)
```

- `f1` function for first M-Task
- `parg1`, `pres1` packed arguments and results for `f1`
- `f2` function for second M-Task
- `parg2`, `pres2` packed arguments and results for `f2`
- `pdescr` pointer to current group descriptor describing two groups

Concurrent execution of M-tasks

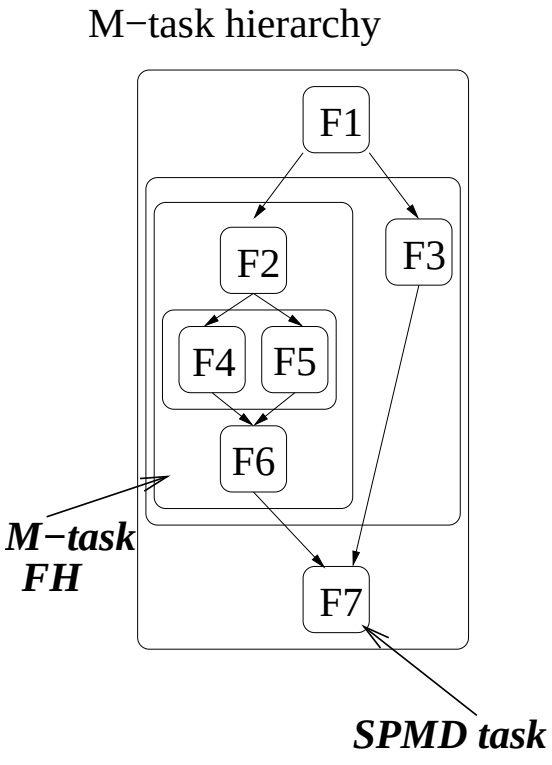
Mapping of more than two independent M-tasks to processor groups:

```
int T_Parfor( void * (*f[])(void *, MPI_Comm, T_Descr),
              void * parg[], void * pres[],
              T_Descr * pdescr)
```

- `f` array of *n* pointers to functions for M-tasks;
- `pdescr` descriptor of group partition into *n* disjoint groups;

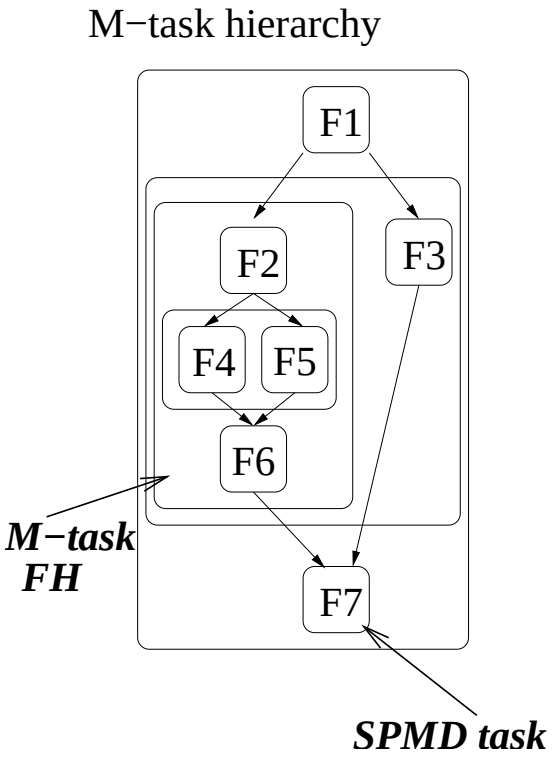
This corresponds to the execution of a **parallel loop** with *n* independent iterates;

Programming example using Tlib




```
main( int argc, char * argv[] ) {
  T_Descr descr, descr1;
  void *arg_f1, *arg_fh, *arg_f3, *arg_f7;
  void *res_fh, *res_f3;
  T_Init (argc,argv,MPI_COMM_WORLD,&descr);
  /* pack argument arg_f1 */
  F1(arg_f1,MPI_COMM_WORLD, &descr);
  T_SplitGrp ( &descr, &descr1, 0.7, 0.3);
  /* pack argument arg_fh in G^a */
  /* pack argument arg_f3 in G^b */
  T_Par (FH, &arg_fh, &res_fh,
         F3, &arg_f3, &res_f3, &descr1);
  /* possible redistribution of data from G^a and G^b */
  /* to G = G^a ∪ G^b; */
  F7(arg_f7,MPI_COMM_WORLD, &descr)
}
```

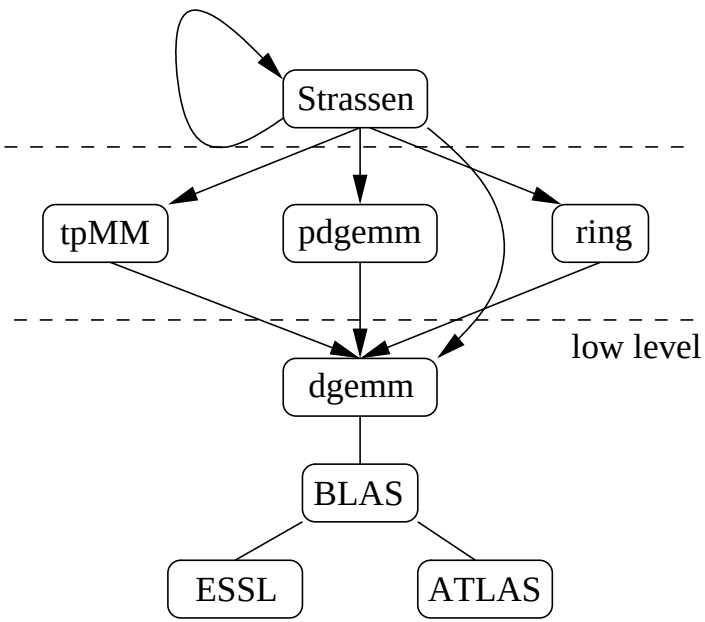
Programming example using Tlib -2-




```
void * FH(void * arg, MPI_Comm comm,
                      T_Descr * pdescr){
   T_Descr descr2;
   void *arg_f2, *arg_f4, *arg_f5, *arg_f6;
   void *res_f2, *res_f4, *res_f5, *res_f6;
   res_f2 = F2( arg_f2, comm, pdescr);
   T_SplitGrp ( pdescr, &descr2, 0.5, 0.5);
   /* pack argument arg_f4 in G^c */
   /* pack argument arg_f5 in G^d */
   T_Par ( F4, &arg_f4, &res_f4,
           F5, &arg_f5, &res_f5, &descr2);
   /* possible redistribution of data from G^c and G^d */
   /* to G^a = G^c ∪ G^d; */
   res_f6 = F6( arg_f6, comm, pdescr);
   /* build result of FH in res_fh; */
   return res_fh;
}
```

Fast Multilevel Hierachical Matrix Multiplication

- A **combination** of different algorithms for matrix multiplication can lead to very competitive implementations:
  Strassen method, tpMM, Atlas; [ICCS'04,ICS'04]
- Illustration:



Recursive splitting – Strassen algorithm

- **matrix multiplication** $C = AB$ with $A, B, C \in IR^{n \times n}$: decompose A, B into square blocks of size $n/2$:

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

- compute the **submatrices** $C_{11}, C_{12}, C_{21}, C_{22}$ separately according to

$$C_{11} = Q_1 + Q_4 - Q_5 + Q_7 \qquad C_{12} = Q_3 + Q_5$$
$$C_{21} = Q_2 + Q_4 \qquad C_{22} = Q_1 + Q_3 - Q_2 + Q_6$$

- compute $Q_1, \ldots, Q_7$ by **recursive calls**:

$$\begin{aligned} Q_1 &= strassen(A_{11} + A_{22}, B_{11} + B_{22}); & Q_5 &= strassen(A_{11} + A_{12}, B_{22}); \\ Q_2 &= strassen(A_{21} + A_{22}, B_{11}); & Q_6 &= strassen(A_{21} - A_{11}, B_{11} + B_{12}); \\ Q_3 &= strassen(A_{11}, B_{12} - B_{22}); & Q_7 &= strassen(A_{12} - A_{22}, B_{21} + B_{22}); \\ Q_4 &= strassen(A_{22}, B_{21} - B_{11}); \end{aligned}$$

Recursive splitting – Strassen algorithm

- organization into four tasks `Task_C11()`, `Task_C12()`, `Task_C21()`, `Task_C22()`:

Task C11 on group 0	Task C12 on group 1	Task C21 on group 2	Task C22 on group 3
compute Q_1	compute Q_3	compute Q_2	compute Q_1
compute Q_7	compute Q_5	compute Q_4	compute Q_6
receive Q_5	send Q_5	send Q_2	receive Q_2
receive Q_4	send Q_3	send Q_4	receive Q_3
compute C_{11}	compute C_{12}	compute C_{21}	compute C_{22}

- realization with Tlib: `Task_C11()` and `Task_q1()`

Strassen algorithm – main coordination function

```
void * Strassen (void * arg, MPI_Comm comm,
                 T_Descr *pdescr){
   T_Descr descr1;
   void * (* f[4])(), *pres[4], *parg[4];
   float per[4];

   per[0]=0.25; per[1]=0.25; per[2]=0.25; per[3]=0.25;
   T_Split_GrpParfor(4, pdescr, &descr1, per);
   f[0] = Task_C11; f[1] = Task_C12;
   f[2] = Task_C21; f[3] = Task_C22;
   parg[0] = parg[1] = parg[2] = parg[3] = arg;
   T_Parfor(f, parg, pres, &descr1);
   assemble_matrix (pres[0], pres[1], pres[2], pres[3]);
}
```

Strassen algorithm – Task_C11()

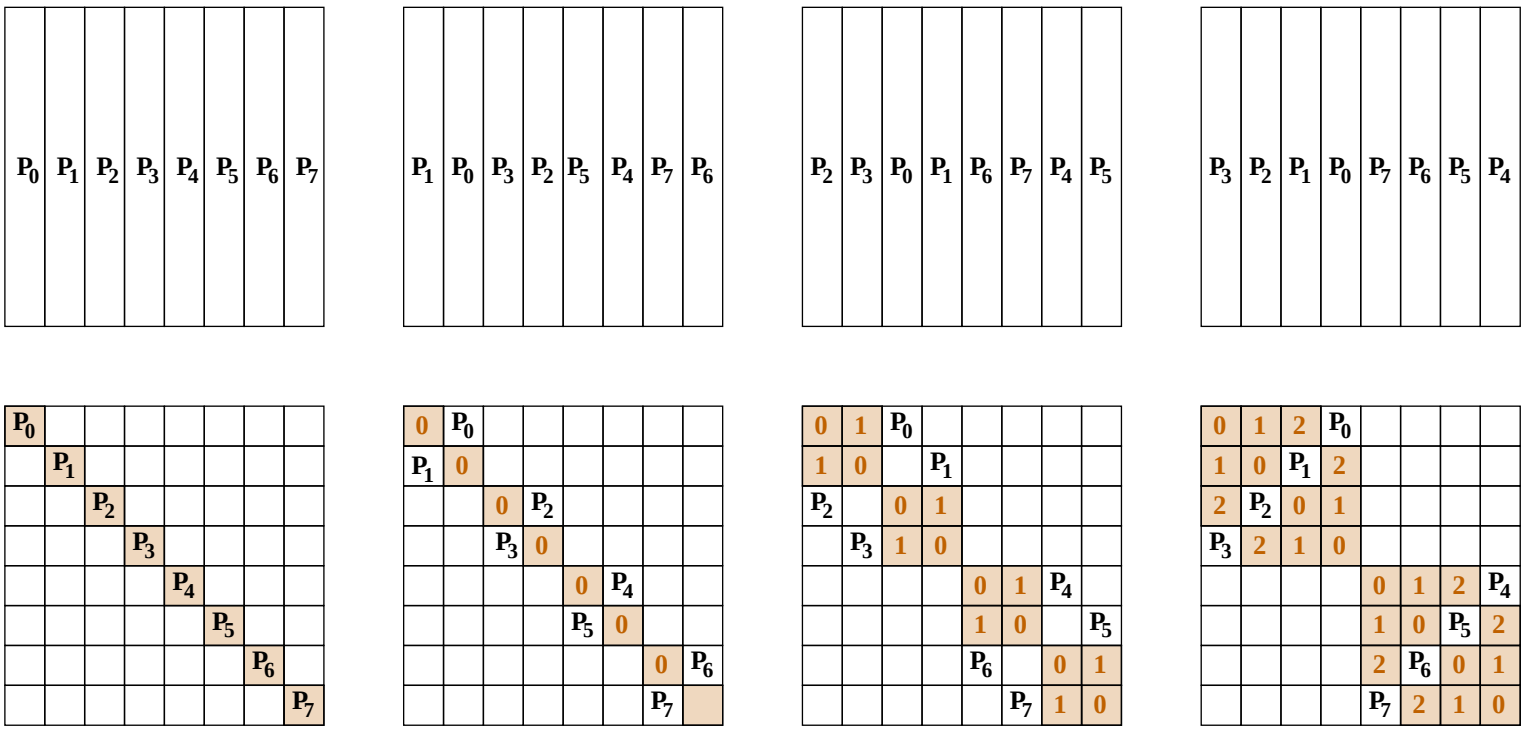
```
void * Task_C11 (void * arg, MPI_Comm comm,
                 T_Descr *pdescr){
   double **q1, **q4, **q5, **q7, **res;
   int i,j,n2;

   /* extract arguments from arg including matrix size n */
   n2 = n/2;
   q1 = Task_q1 (arg, comm, pdescr);
   q7 = Task_q7 (arg, comm, pdescr);
   /* allocate res, q4 and q5 as matrices of size n/2 times n/2 */
   /* receive q4 from group 2 and q5 from group 1 using parent comm. */
   for (i=0; i < n2; i++)
    for (j=0; j < n2; j++)
     res[i][j] = q1[i][j]+q4[i][j]-q5[i][j]+q7[i][j];
   return res;
}
```

Strassen algorithm – `Task_q1()`

```
void *Task_q1(void *arg, MPI_Comm comm, T_Descr *pdescr){
 double **res, **q11, **q12, **q1;
 struct struct_MM_mul *mm, *arg1;
 *mm = (struct_MM_mul *) arg;
 n2 = (*mm).n /2;
 /* allocate q1, q11, q12 as matrices of size n2 times n2 */
 for (i=0; i < n2; i++)
  for (j=0; j < n2; j++) {
   q11[i][j] = (*mm).a[i][j]+(*mm).a[i+n2][j+n2]; /* A11+A22 */
   q12[i][j] = (*mm).b[i][j]+(*mm).b[i+n2][j+n2]; /* B11+B22 */
  }
 arg1 = (struct_MM_mul *) malloc (sizeof(struct_MM_mul));
 (*arg1).a = q11; (*arg1).b = q12; (*arg1).n = n2;
 q1 = Strassen(arg1, comm, pdescr);
 return q1;
}
```

New hierarchical algorithm

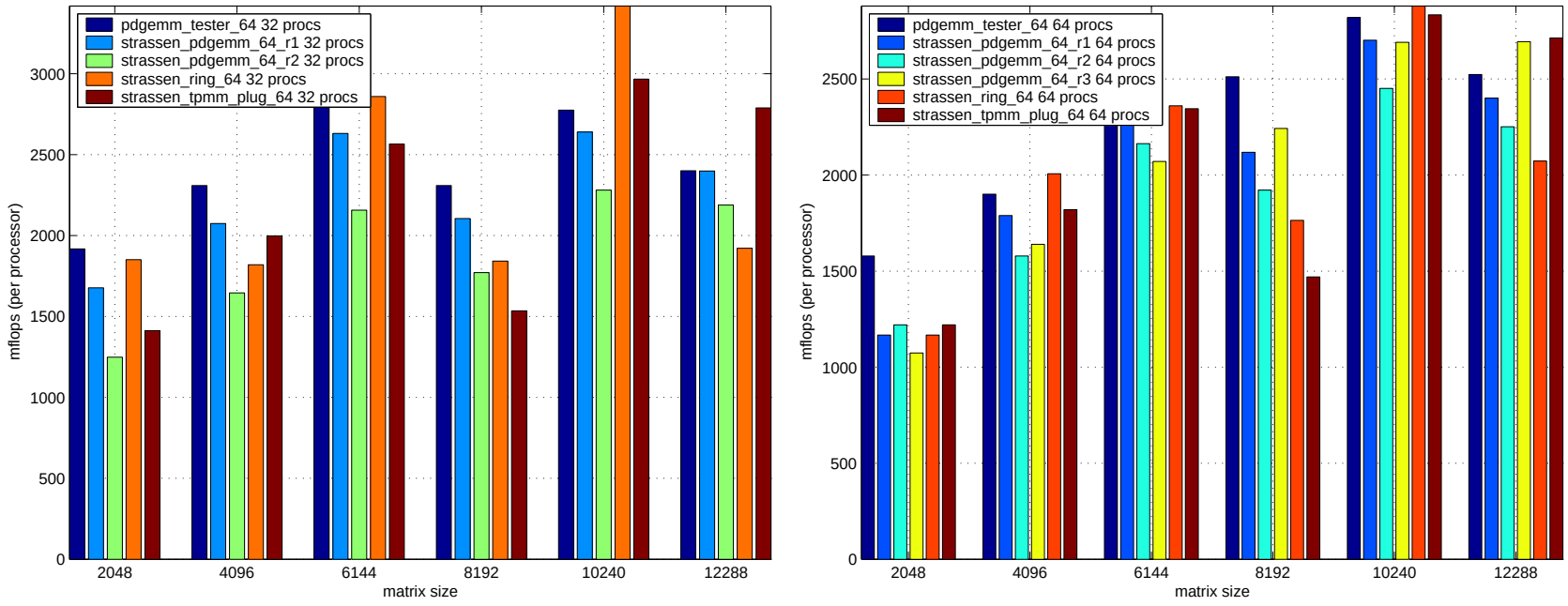
intermediate level: new algorithm tpMM: computation order for 8 processors:



bottom level: **Atlas** for one-processor matrix multiplication

Experimental evaluation: IBM Regatta p690

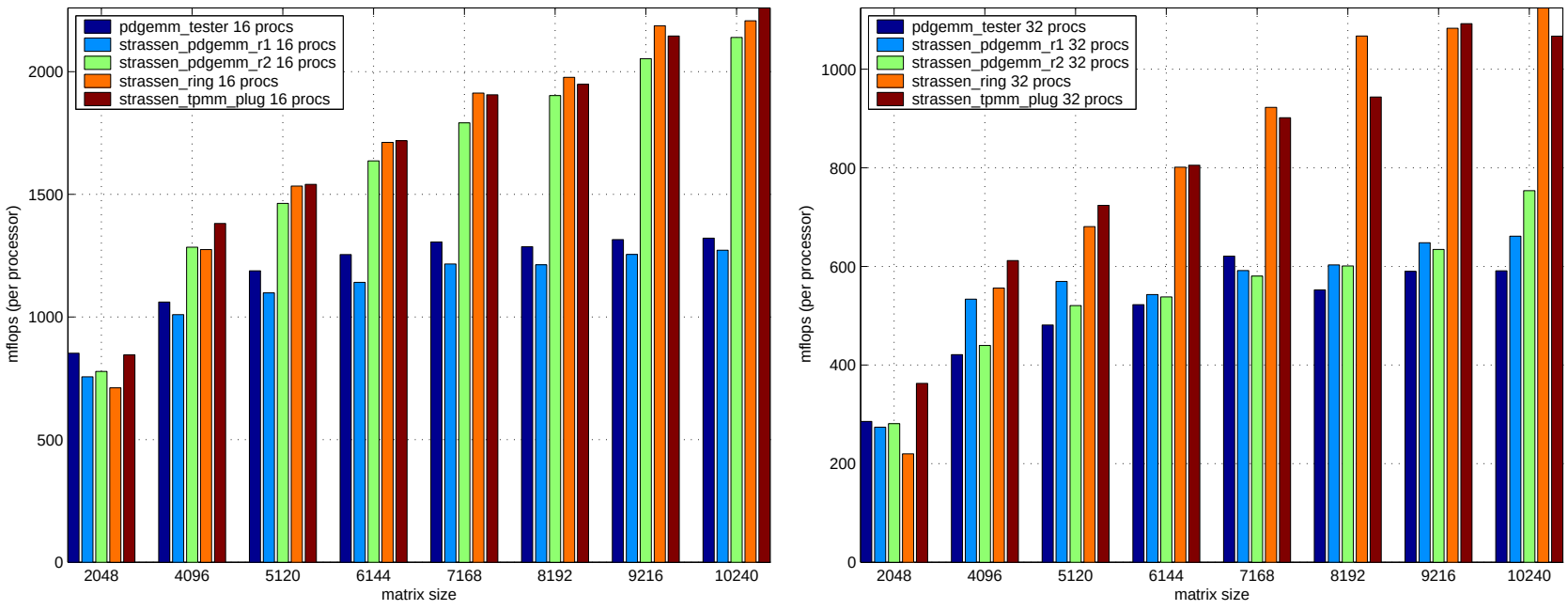
MFLOPS for different algorithmic combinations [ICS'04]:



MFLOPS per processor for 32 and 64 processors

Experimental evaluation: Dual Xeon Cluster 3 GHz

MFLOPS for different algorithmic combinations:



MFLOPS per processor for 16 and 32 processors

DRD-Lib

– a library for **d**ata **red**istribution between M-tasks

- distributed array-based data structures in M-Task parallel context
- data distribution **format**
- set of data **re-distribution** operations for M-task programming (especially Tlib)
  $\longrightarrow$ dynamic character of redistribution· M-task coordination and processor group layout are not known in advance
- information about data location is required:
  - distributed bookkeeping about location of distributed data **or**
  - mechanism to determine data distribution information only when required

DRD-Lib Format

- distributed, opaque data distribution object
- information contained
  - data distribution type (e.gblock-cyclic]
  - parameters for data distribution
  - specifics about storage (e.g. ghost cells)
  - Tlib group descriptor – information about specific processor group
  - size of processor group

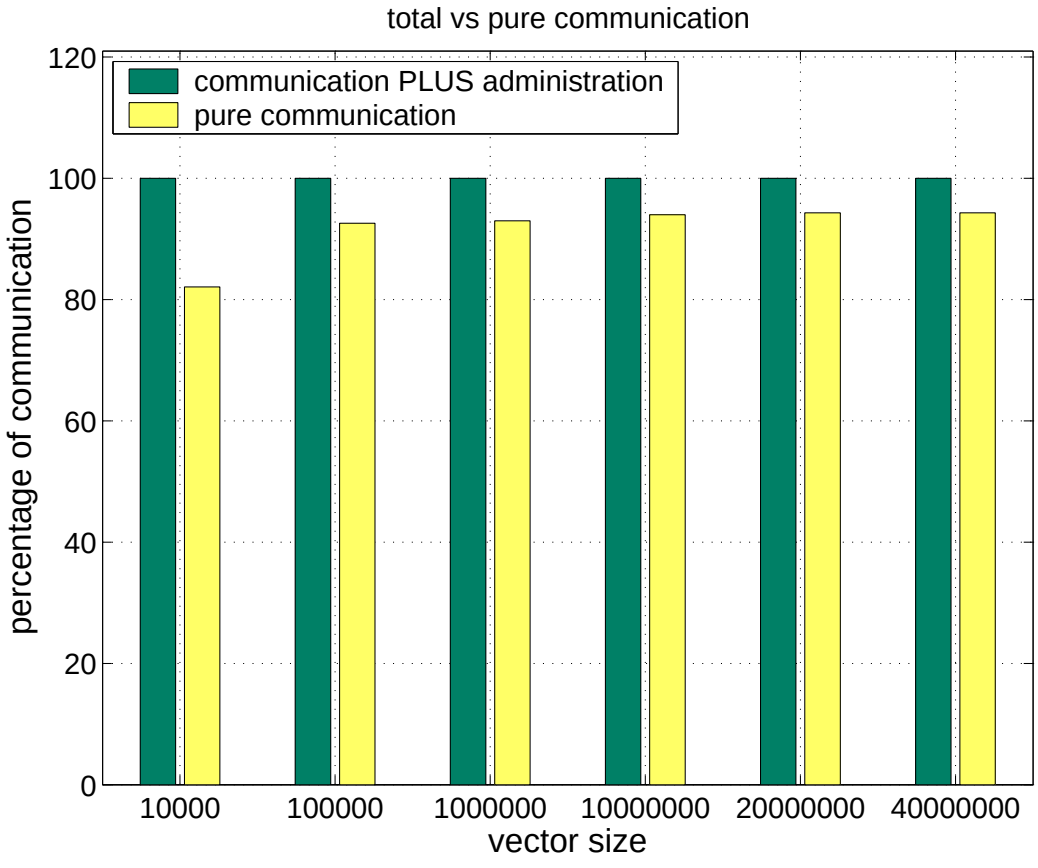
Interface for generating a distributed structure – runtime routines called collectively by a processor group executing an M-Task

DRD-Lib Re-Distributions

data distribution type	processor groups
$T1 = T2$	$G1 = G2$
$T1 = T2$	$G1 \subset G2$
$T1 = T2$	$G1 \supset G2$
$T1 = T2$	$G1 \cap G2 = \emptyset$
$T1 \neq T2$	$G1 = G2$
$T1 \neq T2$	$G1 \subset G2$
$T1 \neq T2$	$G1 \supset G2$
$T1 \neq T2$	$G1 \cap G2 = \emptyset$

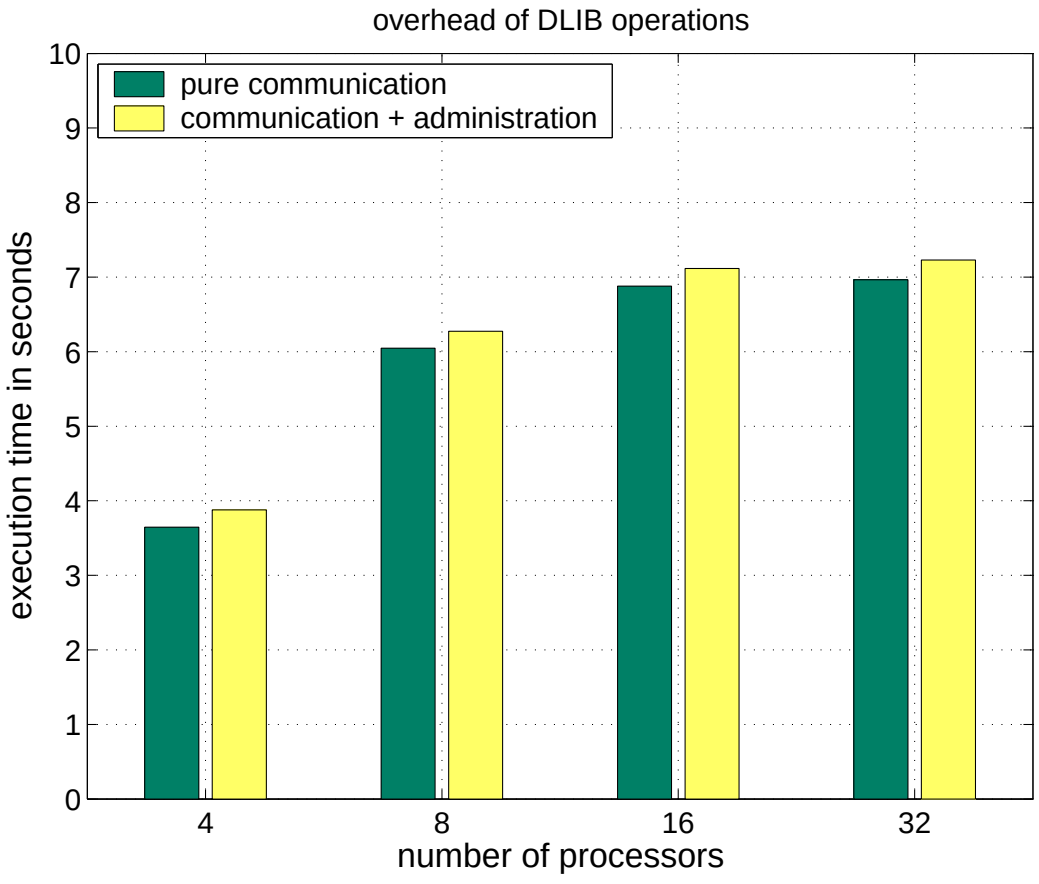
automatic detection of group situation and **adaptive re-distribution** (with distributed algorithm executed in SPMD) [IDPDS workshop APDPM 2005]

Percentage of the communication overhead



Opteron cluster with Gigabit Ethernet for the PVA example and $p = 4$ processors.

Communication and administration overhead



Opteron cluster with Gigabit Ethernet for the PVA example, 10000000 elements.

Outline

Conclusion and Future Work

- Scalability improvement: **Multiprocessor Task programming** $\rightsquigarrow$ scheduling of M-Tasks (or malleable tasks) with dependencies
  $\rightsquigarrow$ static scheduling with scheduling algorithm
  or dynamic scheduling
- **Future work**: dynamic scheduling module for the **automatic re-organization** of task assignement to processor groups; $\rightarrow$ application for **grid environments**;
- further information:
  `ai2.uni-bayreuth.de`
  `rauber@uni-bayreuth.de`