Parallel solution of finite element equation systems: efficient inter-processor communication*

Thomas Apel† Gundolf Haase‡ Arnd Meyer§ Matthias Pester¶

October 14, 2005

Abstract. This paper deals with the application of domain decomposition methods for the parallel solution of boundary value problems for partial differential equations over a domain $\Omega \subset I\!R^d$, $d = 2, 3$. The attention is focused on the conception of efficient communication routines for the data exchange which is necessary for example in the preconditioned cg-algorithm for solving the resulting system of algebraic equations. The paper describes the data structure, different algorithms, and computational tests.

Key Words. Parallel computation, finite element method, domain decomposition.

AMS(MOS) subject classification. 65Y05, 65Y10, 68M07

Contents

*The paper appeared as Preprint SPC95_5 in the series if the Chemnitz research group "'Scientific Parallel Computing"'.

†Technische Universität Chemnitz-Zwickau, Fakultät für Mathematik, D–09107 Chemnitz, Germany; e-mail: apel@mathematik.tu-chemnitz.de

‡Johannes Kepler Universität Linz, Institut für Mathematik, Altenbergerstr. 69, A–4040 Linz, Austria; e-mail: ghaase@miraculix.numa.uni-Linz.ac.at

§Technische Universität Chemnitz-Zwickau, Fakultät für Mathematik, D–09107 Chemnitz, Germany; e-mail: a.meyer@mathematik.tu-chemnitz.de

¶Technische Universität Chemnitz-Zwickau, Fakultät für Mathematik, D–09107 Chemnitz, Germany; e-mail: mspester@mathematik.tu-chemnitz.de

1 Introduction

The mathematical treatment of many physical phenomena and engineering problems leads to direct problems as solving (nonlinear) partial differential equations (p.d.e.) or optimization problems including p.d.e. On the other hand, the determination of parameters (or the optimal parameters) in the p.d.e. (for example material properties) is often the scope of interest. These so called inverse problems require numerous iterations of solving the direct problems above. In any case a fast solver for the applied problems depends on the fast solving of the linear p.d.e. and therefore after a discretization (here via the finite element method) on the availability of a fast solver for large linear systems of equations.

During the last decade various ideas for parallel solving finite element equation systems were developed. Our report is based on parallel iterative solvers using a non-overlapping domain decomposition and parallel computers with distributed memory (distributed data) [3, 4, 5]. After mapping the subdomains to the processors this class of algorithms requires some small amount of a special type of communication for updating the values at the nodes on the boundaries between two or more processors, in the following called accumulation. The particuliarity is that some vectors are stored in such a way that the correct value at coupling nodes must be obtained by adding the partial values which are contributed by the corresponding processors. The scope of this report consists in the description of effective algorithms for this necessary accumulation in the two- and the three-dimensional cases assuming at least a logical hypercube topology, for an introduction to hypercubes see [6, 7]. Notice that physically neighboured subdomains may be placed on processors which are not adjacent physically or logically.

In Section 2 we give a short description of the domain decomposition and the data structures supporting the accumulation. The following three sections contain algorithms for the accumulation which will be partially unified in Section 6 for the three-dimensional case. Section 7 presents some tests for a comparison of two variants of the accumulation.

The algorithms given are written for the logical hypercube topology with respect to the portability of the code, indeed using just the basic routines of our library `libCubecom.a` [2] whose implementation is hardware independent. Nevertheless, some ideas for using direct links between arbitrary processors as provided under PARIX and PVM will be given.

Of course the presented strategies have much less importance for a small number of processors (less or equal 16) than for medium and large numbers (greater or equal 128) because the global amount of data for accumulation will grow with the number of processors. Moreover, let N be the typical problem size (for example the number of unknowns) then the amount of data to be accumulated is $\mathcal{O}(N^{1/2})$ for two-dimensional problems but $\mathcal{O}(N^{2/3})$ in three dimensions. Thus an effective accumulation algorithm becomes more important for three dimensional problems.

2 Notation and data representation

In the finite element method, we consider a family $\{\mathcal{T}_k\}$ of partitions of the domain $\Omega \subset I\!R^d$, $d = 2, 3$, into finite elements. Such a family can be constructed by the following algorithm:

Algorithm 1

1. Find a computational description of the domain Ω and the data of the boundary value problem.

2. Construct a coarse mesh $\mathcal{T}_0$ which approximates Ω, and distribute the elements to the processors.

3. To create $\mathcal{T}_{k+1}$ ($k = 0, 1, \ldots$) divide all elements of $\mathcal{T}_k$ into 2^d smaller elements of the same volume.

index	contents of the vector at position *index*
1	pointer, smallest number of a node in the *Kette*
2	length of the *Kette*
3	communication identifier *PathId*, see sections 4 and 5
4 – 7	global identification of the *Kette* (*KettenId*)

Table 1: Definition of the vector describing a *Kette*.

index				interpretation
4	5	6	7	
C_1	C_2	0	0	*1D-Kette*, the corresponding edge has the crosspoints C_1 and C_2 as vertices
K	0	0	0	*1D-Kette*, the corresponding edge has the global number K
C_1	C_2	C_3	0	*2D-Kette*, the corresponding face is a triangle with the vertices C_1, C_2, and C_3
C_1	C_2	C_3	C_4	*2D-Kette* corresponding to a quadrilateral
0	0	K	0	*2D-Kette* corresponding to the face with the global number K

Table 2: Identification of a *Kette* by four integers.

Clearly, Step 3 can only be executed until the memory of the computer is exhausted. The programming of the steps is not under consideration in this paper, but we introduce the following notation and conventions.

Consider first the two-dimensional case. All nodes of the coarse mesh $\mathcal{T}_0$ are called *crosspoints* and the edges of $\mathcal{T}_0$ are called *coupling edges*. The numbers of crosspoints and of coupling edges are constant for all meshes of the family. In each mesh $\mathcal{T}_k$ the crosspoints have the same enumeration. After distributing the data over the processors each processor possesses a smaller number of local crosspoints. As a global information there is a vector which maps the local crosspoint numbers to the global crosspoint numbers.

During Step 3 of Algorithm 1, additional nodes are introduced at the coupling edges and in the interior of the elements of $\mathcal{T}_0$. The latter are called *inner nodes*, their number grows with $2^{2k} \sim h^{-2}$. Note that inner nodes belong to only one processor, that means they do not contribute to the communication.

The nodes at the coupling edges may belong to another processor as well, however, contrary to the crosspoints, at most to one other. Because their number is of the order $2^k \sim h^{-1}$ we shall avoid expensive searches during the communication process by demanding from the mesh generator of Step 3 that the nodes of each edge are numbered consecutively. Thus they are identified by a pointer to the first node, the number of nodes at this edge, and a characterization of this edge. We denote such a sequence by *Kette*, the German word for chain. Note that the coupling edges can be characterized by a global edge number (if available) or by their global crosspoint numbers. We remark also that this data structure is convenient for a preconditioner related to the coupling edges as described in [1].

The three-dimensional case is handled in the canonical way. However, there is a new quality, namely the existence of coupling edges and *coupling faces*. Their treatment as in the two-dimensional case leads to different sequences of nodes, denoted by *1D-Kette* and *2D-Kette*, respectively. The number of nodes grows with $2^k \sim h^{-1}$ and $2^{2k} \sim h^{-2}$, respectively.

In the program, each *Kette* is described by a vector of integer type and of dimension 7, see Tables 1 and 2 for an explanation. These vectors are stored in an array `Kette` (in two dimensions) or in two arrays `Kette1D` and `Kette2D`.

3 Accumulation of data at crosspoints

The algorithm needs one auxiliary array `H` of length `NCrossG` and one global communication. `NcrossG` denotes the global number of crosspoints.

Algorithm 2

1. Initialize `H` with 0.
2. Write the values of the local crosspoints to the appropriate places in `H`.
3. Perform a cube sum for `H`.
4. Get the accumulated values for the local crosspoints out of `H`.

4 Accumulation of data belonging to edges (2D) and faces (3D)

It is easy to modify Algorithm 2 for the accumulation of the data at the coupling edges (2D) and faces (3D), but there are strong disadvantages:

- A large auxiliary array is necessary.
- A large amount of useless information is exchanged, its part is increasing with the number of processors.

On the other hand, we realize that each *Kette* under consideration belongs to at most two processors. The improvement idea is to determine the number of the other processor having the same *Kette*. Therefore the feature of creating virtual topologies (under PARIX) as well as the direct point-to-point communication between arbitrary processors (under PVM) can be used to exchange the data with the corresponding processor. This can be done by creating a list of descriptors *PoP* (*Pairs of Processors*). For each *Kette* there is an entry

`Proc1, Proc2, Level, KettenID`

where `(Proc1,Proc2)` is the pair of the processors sharing this *Kette*, and `KettenID` is a unique number or name to identify *Kette*, for example

$$\texttt{KettenID} := f(C_1, C_2) = C_1 * \texttt{NCrossG} + C_2$$

for edges in two dimensions, for $C1$, $C2$ see Table 2. The index `Level` is just a classification for the `PoP`'s. All the pairs of processors belonging to the same level are mutually exclusive. Hence, all communications for one level can be performed simultaneously between those pairs of processors. This classification is important for avoiding dead-locks in communications.

Algorithm 3

1. Prepare the local contribution to the `PoP` list:

   `Proc1 := Ich` (this processor's number)
   `Proc2 := -1`
   `Level := 0`
   `KettenID :=` $f(C_1, C_2\,[, C_3, C_4])$

   A *Kette* of length 0 is not added to this list.

2. Broadcast the local lists to all processors (global exchange).

3. Find the missing `Proc2` entry for the local lists by searching for the same `KettenID` in the “foreign” lists. If no `Proc2` is found for a certain *Kette* (which can be determined by a remaining `Proc2=-1`), this one is either internal for the processor or it belongs to the boundary, and therefore it can be ignored for communications.

4. Select all *PoP*s with `Ich = Proc1 < Proc2`.
(Now each `KettenID` occurs just once among all local lists.)

5. Send all local lists to processor 0 (Tree_Up communication).
Generate a subdivision into classes of disjunct *PoP*s (set the value for `Level`).
Broadcast the resulting list `PoP` to all processors.

6. Sort the local list `Kette` (or `Kette2D`) of coupling edges (or faces) corresponding to `Level` and `KettenId` and keep the number of the destination processor in this list as communication identifier.

7. When running under PARIX, create the virtual topology with one link for each pair of processors.

Algorithm 3 is used only once to prepare the communication for the accumulation during the conjugate gradient algorithm. This accumulation is done by running-down the sorted list of coupling edges (or faces):

Algorithm 4

<u>FORALL</u> `Kette` <u>IN</u> `Kette2D` <u>DO</u>
  Exchange `Kette` with destination processor `PathId`
  and accumulate it in both processors.
<u>DONE</u>

The advantage of this processing is that the amount of communicated data during the conjugate gradient algorithm is minimized, moreover, the communication is done in a few steps between certain groups (levels) of pairs of processors.

Unfortunately, this algorithm is hardly to extend to the case of *1D-Kette*s in three dimensions because, generally, they belong to more than two processors.

5 Accumulation of data belonging to edges (3D)

Because the application of Algorithm 2 is too expensive and an efficient extension of algorithm 3 to this case is hardly to manage, we try to use the model “hypercube” in a specific way. To explain our idea we start with an example.

Example 1 Consider a *1D-Kette* that belongs to the three processors with the numbers

$$\begin{aligned} p_1 &= 11 = 000\mathrm{L}0\mathrm{LL}, \\ p_2 &= 17 = 00\mathrm{L}000\mathrm{L}, \\ p_3 &= 65 = \mathrm{L}00000\mathrm{L}. \end{aligned}$$

If we break Link 0 of all processors then the hypercube of dimension `ncube` is split in two sub-hypercubes of dimension $\texttt{ncube} - 1$. The last bit in the binary representation of the number of the processor indicates the sub-hypercube the processor belongs to. In our example, all three processors belong to the same sub-hypercube, that means the data exchange via Link 0 is useless. Obviously, the same is valid for Links 2 and 5.

However, Links 1, 3, 4, and 6 cannot be broken, otherwise the processors would belong to different subcubes. The minimal sub-hypercube for our example can be characterized by an integer `PathId`

$$\texttt{PathId} = \mathrm{L}0\mathrm{LL}0\mathrm{L}0;$$

L means that the corresponding link is necessary, 0 indicates that a communication via this link is without use.

Our aim is to to realize the accumulation in a way that the communication of the *Kette* is performed by a specific subcube sum, here in a four-dimensional sub-hypercube. Note that 13 of the 16 processors which are engaged in the communication, do not possess the *Kette* theirselves.

In the following we shall describe the two algorithms related to the accumulation. Algorithm 5 realizes the initialization and must be executed once before Algorithm 6 is called the first time.

Algorithm 5

1. Distribute the set T of pairs (`KettenId`, `Ich`) to all processors using the global exchange communication `cube_cat` [2]. For `KettenId` see Table 2, the integer `Ich` is the number of the processor.
   *Kette*s of length 0 are not included.

2. <u>DO</u> `k:=1` <u>TO</u> `NKetteLoc`
    `AndBit :=` $\overline{0}$
    `OrBit :=` 0
    <u>IF</u> (Length_of_Kette(`k`) > 0) <u>THEN</u>
      <u>DO</u> `p:=1` <u>TO</u> `nproc`
        <u>IF</u> (`KettenId(k)`, `p`) $\in T$ <u>THEN</u>
          `AndBit := AndBit` $\wedge$ `p`
          `OrBit := OrBit` $\vee$ `p`
        <u>END IF</u>
      <u>DONE</u>
    <u>END IF</u>
    `PathId :=` $\overline{\texttt{AndBit}} \wedge$ `OrBit`
   <u>DONE</u>

`NKetteLoc` denotes the local number of *Kette*s, `nproc` is the number of processors, the symbols $\wedge$ and $\vee$ denote the logical AND and OR, respectively, performed bitwise, and $\overline{a}$ is the bitwise NOT of the integer a.

Lemma 1 *Algorithm 5 computes the integer* `PathId` *as defined in Example 1.*

Proof When the inner loop is finished, the binary representation of the integer `AndBit` has an L exactly at those positions where all relevant numbers p of processors have an L in their binary representation. On the other hand, `OrBit` has an 0 exactly at those positions where all relevant numbers p of processors have an 0 in their binary representation. Consequently, $\overline{\texttt{AndBit}} \wedge \texttt{OrBit}$ has an 0 exactly at those positions where all relevant numbers p of processors have the same value. □

In Example 1 it is `AndBit` = 000000L and `OrBit` = L0LL0LL.

Remark 1 If any *Kette* belongs to only one processor `Ich`, then

$$\texttt{AndBit} = \texttt{Ich}, \quad \texttt{OrBit} = \texttt{Ich} \quad \Rightarrow \quad \texttt{PathId} = 0.$$

If any *Kette* has length 0 then

$$\texttt{AndBit} = \overline{0}, \quad \texttt{OrBit} = 0 \quad \Rightarrow \quad \texttt{PathId} = 0.$$

That means there is no communication for such *Kette*s.

Algorithm 6 needs three auxiliary buffers `Wait`, `Send`, and `Recv` to store *Kette*s. For a simple description denote the set of *Kette*s that the processor possesses itself, by `Own`.

Algorithm 6

`TestBit:=1`
DO `nrlink:=1` TO `ncube`

1. FORALL `Kette` IN `Own` DO
   IF `PathId` $\wedge$ `TestBit` THEN Copy `Kette` to `Send`.
   DONE

2. FORALL `Kette` IN `Wait` DO
   IF `PathId` $\wedge$ `TestBit` THEN Copy `Kette` to `Send`.
   DONE

3. Send buffer `Send` to the neighbouring processor via link `nrlink` and store the data received from the same processor in the buffer `Recv`.

4. FORALL `Kette` IN `Recv` DO
   IF (`Kette` IN `Own`)
   THEN Accumulate Values in `Own`.
   ELSE IF (`Kette` IN `Wait` AND `PathId` $< 2^{\texttt{nrlink}}$)
   THEN Accumulate Values in `Wait`.
   ELSE Add `Kette` at the end of buffer `Wait`.
   END IF
   END IF
   DONE

5. Compress `Wait` by deleting each `Kette` with `PathId` $< 2^{\texttt{nrlink}}$.

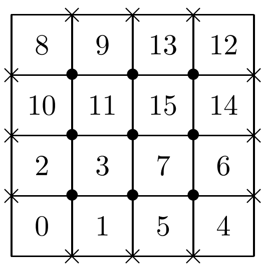
6. `TestBit:=TestBit*2`

DONE

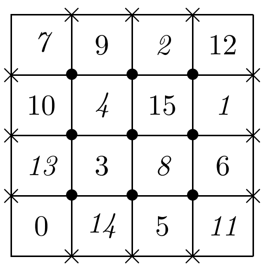
Remark 2 The buffer `Recv` is empty before and after performing the loop in Algorithm 6.

Remark 3 The dimensions of the sub-hypercubes depend on an intelligent distribution of the subdomains to the processors. Consider a quadratic 4×4 grid with edges at the vertices, directed in the third dimension. The following two examples of the processor distribution are constructed using the Gray code [6].

(a)



(b)



$\times$ boundary edges with two adjacent subdomains

$\bullet$ inner edges with four adjacent subdomains

In the case (a), the dimension of the subcube is 2 for inner edges and 1 for boundary edges, because the numbers of the processors of adjacent subdomains differ in exactly one bit. In case (b) they differ in (`ncube` $- 1$) bit. Consequently, the complete hypercube is necessary for the accumulation of each inner *Kette*. Note that the example easily extends to a higher hypercube dimension.

That means, an intelligent distribution of the subdomains is achieved when the numbers of adjacent subdomains differ in few bits only. These considerations are sufficient for (hard-wired) hypercubes. In the case of other topologies (for example under PARIX) one should also keep in mind that the data exchange via links with low numbers `nrlink` may be faster than via higher links depending on the mapping of hypercube links to the PARIX grid. For the PVM workstation cluster the only restriction will be to avoid any useless communication with respect to the large setup time.

6 Unification of the accumulation of *Kette*s in two and three dimensions

Algorithm 6 is not restricted to *1D-Kette*s, the *2D-Kette*s could easily be included. However, Algorithm 4 has the advantage that only a minimal set of data is communicated. On the other hand, if *2D-Kette*s are to be transferred between two processors then some *1D-Kette*s have to be transferred as well, that means the times for the startup in Algorithm 4 can be saved when they are transferred together.

Consequently, the aim is to construct an algorithm which combines the advantages of Algorithms 4 and 6. The following one saves the time for additional startups and avoids the distribution of the *2D-Kette*s on the whole sub-hypercube. In before, a determination of *PathId* as in Algorithm 5 has to be performed for the *2D-Kette*s as well.

Algorithm 7

```
FORALL 2D-Kette IN Own DO
   IF PathId > 0 THEN Copy 2D-Kette to Wait
DONE
TestBit := 1
DO nrlink:=1 TO ncube

  1. FORALL 1D-Kette IN Own DO
        IF PathId ∧ TestBit THEN Copy Kette to Send.
     DONE

  2. FORALL 1D-Kette IN Wait DO
        IF PathId ∧ TestBit THEN Copy 1D-Kette to Send.
     DONE

  3. FORALL 2D-Kette IN Wait DO
        IF PathId ∧ TestBit THEN Copy 2D-Kette to Send.
        Set PathId for 2D-Kette in Wait to 0.
     DONE

  4. Send buffer Send to the neighbouring processors via link nrlink and
     store the data received from the same processor in the buffer Recv.

  5. FORALL 1D-Kette IN Recv DO
        IF (1D-Kette IN Own)
        THEN Accumulate Values in Own.
        ELSE IF (1D-Kette IN Wait AND PathId < 2^nrlink)
           THEN Accumulate Values in Wait.
           ELSE Add 1D-Kette at the end of buffer Wait.
           END IF
        END IF
     DONE
```

6. FORALL `2D-Kette` IN `Recv` DO
   IF (`2D-Kette` IN `Own`)
   THEN Accumulate Values in `Own`.
   ELSE Add `2D-Kette` at the end of buffer `Wait`.
   END IF
   DONE

7. Compress `Wait` by deleting each `Kette` with `PathId` $< 2^{\texttt{nrlink}}$.

8. `TestBit:=TestBit*2`

DONE

7 Numerical comparison of the algorithms

Our test example is to solve

$$\begin{aligned} -\Delta u &= 0 \quad \text{in } \Omega = (0,2)^3, \\ u &= xy \quad \text{on } \partial\Omega, \end{aligned}$$

via the finite element method with linear shape functions on tetrahedral meshes. We generated coarse meshes with 48, 96, 192, 384, and 768 elements and refined them uniformly until the memory was exhausted. Note that the coarse meshes with 384 and 768 elements are different from those after one refinement step of the meshes with 48 and 96 elements, respectively.

These examples were solved on a parsytec GC/PP-128 machine using 8 to 128 processors and as variants 1 and 2 processors per node. The times for the communication were comparable in the 5 examples, here we present the results for 192 coarse terahedra. In Figures 1 and 2 the times for communication and for accumulation (communication plus some arithmetics) are shown.

We draw the following conclusions:

- For small and medium numbers of processors Algorithm 7 is better than the combination of Algorithms 4 and 6 regarding the pure communication time and the time for accumulation. The time for arithmetics is much higher of course.

- The situation changes for a high processor number and a large number of unknowns (*2D-Kette*s dominate the *1D-Kette*s). It is very likely that in Algorithm 7 a large number of the quite long *2D-Kette*s has to be communicated in several steps. But in Algorithm 4 they are communicated just once. Here, the possibility of direct links under PARIX is very favourable.

  At the moment, the distribution of the elements to the processors is done in the order that the coarse mesh generator produced. An optimization in the direction to minimize the communication and to minimize the subhypercube dimension (see Remark 3) is expected to improve the behaviour of Algorithm 7.

- Comparing the same number of processors in the two variants of one or two processors per node, we see that arithmetic time and communication time show controverse behaviour. This influences the comparison of the two algorithms in a very machine dependent way.

Acknowledgement. The first author was supported by DFG (German Research Foundation), No. La 767-3/1.

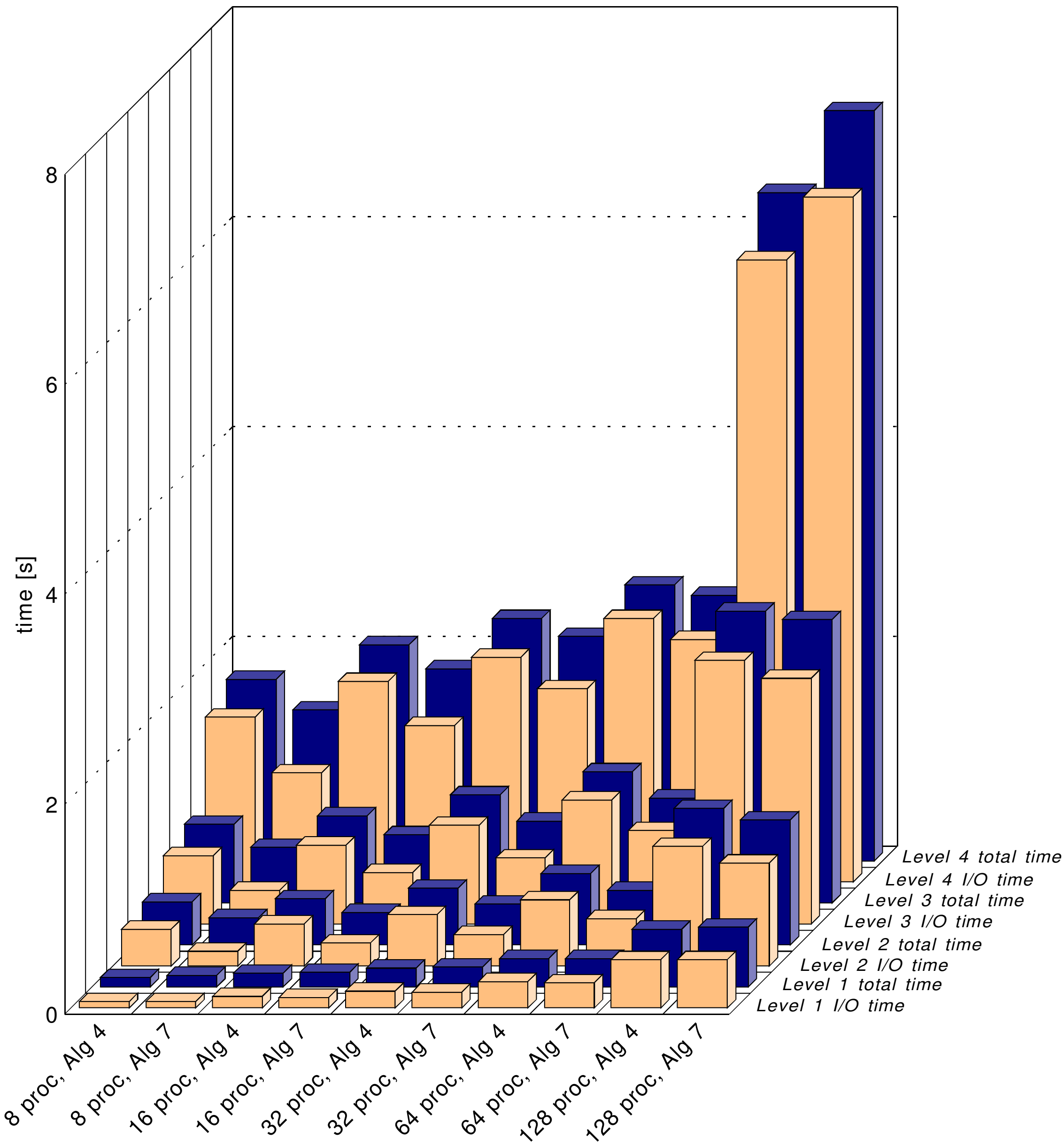


Figure 1: Comparison of Algorithms 4 and 7 varying the number of processors and the problem size, (run on ParsytecGC with 2 tasks per node)

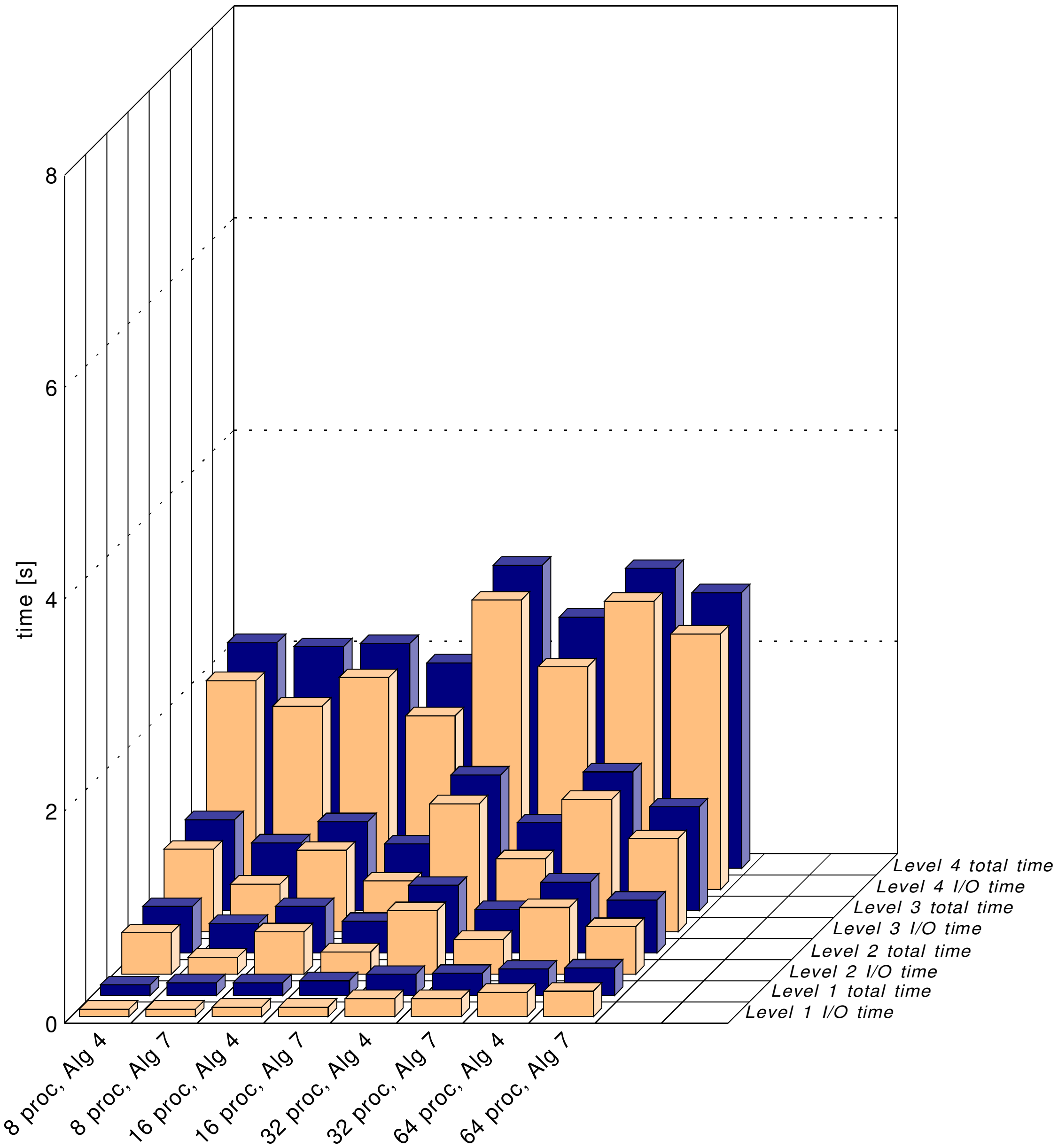


Figure 2: Comparison of Algorithms 4 and 7, varying the number of processors and the problem size (run on ParsytecGC with 1 task per node)

References

[1] M. Dryja. A finite element capitance method for elliptic problems on regions partitioned into subdomains. *Numer. Math.*, 44:153–168, 1984.

[2] G. Haase, Th. Hommel, A. Meyer, and M. Pester. Bibliotheken zur Entwicklung paralleler Algorithmen. Preprint SPC 93_4, TU Chemnitz–Zwickau, 1994.

[3] G. Haase, U. Langer, and A. Meyer. The approximate Dirichlet domain decomposition method. Part I: An algebraic approach. Part II: Applications to 2nd-order elliptic boundary value problems. *Computing*, 47:137–151 (Part I), 153–167 (Part II), 1991.

[4] G. Haase, U. Langer, A. Meyer, and S. V. Nepomnyaschikh. Hierarchical extension and local multigrid methods in domain decomposition preconditioners. *East–West J. Numer. Math.*, 2:173–193, 1994.

[5] A. Meyer. A parallel preconditioned conjugate gradient method using domain decomposition and inexact solvers on each subdomain. *Computing*, 45:217–234, 1990.

[6] Y. Saad and M. H. Schultz. Topological properties of hypercubes. Research report 389, Yale University, Dept. Computer Science, 1985.

[7] Y. Saad and M. H. Schultz. Data communication in hypercubes. *Journal of parallel and distributed computing*, 6:115–135, 1989.