Technische Universität Chemnitz-Zwickau

Sonderforschungsbereich 393

Numerische Simulation auf massiv parallelen Rechnern

Thomas Apel Uwe Reichel

Partitioning of finite element meshes for parallel computing: A case study

Preprint SFB393/96-18a

Abstract. The realization of the finite element method on parallel computers is usually based on a domain decomposition approach. This paper is concerned with the problem of finding an optimal decomposition and an appropriate mapping of the subdomains to the processors. The quality of this partitioning is measured in several metrics but it is also expressed in the computing time for solving specific systems of finite element equations.

The software environment is first described. In particular, the data structure and the accumulation algorithm are introduced. Then several partitioning algorithms are compared. Spectral bisection was used with different modifications including Kernighan-Lin refinement, post-processing techniques and terminal propagation. The final recommendations should give good decompositions for all finite element codes which are based on principles similar to ours.

AMS(MOS) subject classification. 65Y05, 65Y10, 68M07, 68R10

Key Words. Parallel computation, partitioning of graphs, finite element method, domain decomposition.

Preprint-Reihe des Chemnitzer SFB 393

SFB393/96-18a **January 1997**

Contents

Authors' addresses:

Thomas Apel, Uwe Reichel
TU Chemnitz-Zwickau
Fakultät für Mathematik
D-09107 Chemnitz, Germany

apel@mathematik.tu-chemnitz.de
reichel@mathematik.tu-chemnitz.de

http://www.tu-chemnitz.de/~tap/

1 Introduction

The mathematical treatment of many physical phenomena and engineering problems leads to direct problems as solving (nonlinear) partial differential equations or optimization problems including partial differential equations. On the other hand, the determination of parameters (or the optimal parameters) in such equations (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(ized) partial differential equations 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) [9, 10, 19]. 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 of the nodes on the boundaries between two or more processors, in the following called accumulation.

The time needed for accumulation could be minimized by an optimal distribution of the n finite elements on the p processors. This is by no means a trivial problem:

- The optimal distribution depends on the hardware and on the software, in particular on the data structure. That means the functional to be minimized is not clear.

- The number of possible distributions grows exponentially with n; the optimization problem is NP hard.

Nevertheless, there are many attempts to model key features of the accumulation [11]. It is assumed that the time depends essentially on the quantity of data which has to be exchanged and on the length of the path on which each information has to be transferred. In this sense, the distribution problem can be (but needs not to be) decoupled in a partitioning problem and an assignment problem.

These models and other heuristics led to a variety of algorithms (including recursive coordinate bisection, recursive inertial bisection [23, 24], recursive spectral bi-, quadri-, octasection [14, 15], Kernighan-Lin [17], terminal propagation [6, 8]) and combinations of them. The preference of one of the algorithms depends strongly on the application, for example on the size of p and n, time constraints, and the frequency of redistribution. The aim of this study is to suggest an algorithm for our application, the parallel finite element code *SPC-PMPo3D* [1, 3].

In Version 3, *SPC-PMPo3D* can solve the Poisson equation and the Lamé system of linear elasticity with in general mixed boundary conditions of Dirichlet and Neumann type on a variety of (in general curved) domains. The program has been developed for MIMD computers; it has been tested on Parsytec machines (GCPowerPlus–128 with Motorola Power PC601 processors and GCel–192 on transputer basis) and on workstation clusters. We point out that the implementation is based on a special data structure which allows that all components of the program run with almost optimal performance ($\mathcal{O}(n)$ or $\mathcal{O}(n \ln n)$). The data representation and the accumulation algorithm are described in [2, 3]. Because of their relevance for our study and to keep this paper self-contained, we review some main points in Section 2.

In Section 3 we present the results of our study. For the computation of the element distributions compared here, the package **Chaco** [12] was used. This program contains many of the algorithms mentioned above for the partitioning of finite element meshes. Our report is completed with a short final section where we summarize our results.

The importance of this study is based on two arguments. First, other parallel finite element codes work on similar principles; our results should extend to these codes. Second, the tests carried out are the starting point for our ongoing research into adaptive finite element methods in three dimensions which is still a challenge on a parallel computer.

The authors know about a similar study [18] where dynamic load balancing is considered in the context of the parallel multilevel finite element code *ug* [4, Chapter 4]. This report [18] goes beyond our study because the far more difficult *dynamic* load balancing case is examined. On the other hand, the study is limited to two dimensional problems. But effective accumulation becomes more important for three dimensional problems. To see this, 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.

2 The software environment

2.1 Main algorithms

In the finite element method, we consider a family $\{\mathcal{T}_k\}_{k=0}^{\infty}$ of meshes. A mesh is a subdivision of the domain $\Omega \subset I\!R^3$ into finite elements. A hierarchy of meshes is defined if $\mathcal{T}_{k+1}$ is a refinement of $\mathcal{T}_k$ ($k = 0, 1, 2, \ldots$). $\mathcal{T}_0$ is called coarse mesh. Up to Version 3 of our finite element package *SPC-PMPo3D* the simplest case of a hierarchy is realized where $\mathcal{T}_{k+1}$ is obtained by a uniform subdivision of all elements of $\mathcal{T}_k$ into 8 smaller elements of equal volume.

The main steps of the finite element code include

1. the distribution of the elements of $\mathcal{T}_0$ to the processors,
2. the refinement of the mesh,
3. the assembling the local stiffness matrices K_s and the local right hand sides f_s, $s = 0, \ldots, p-1$, and
4. the solving of the system of equations.

Observe that each element belongs to exactly one processor, but nodes can belong to several processors. Such nodes are called *coupling nodes.*

The steps 2 and 3 are executed in parallel without any communication. This implies that the vector f (the global right hand side) is of *additive type*, that means that the correct value at the coupling nodes would be obtained only after the accumulation (adding) of the partial values which are contributed by the corresponding processors. This can be expressed in mathematical terms by defining f as

$$f = \sum_{s=0}^{p-1} A_s^T f_s,$$

where A_s are Boolean connectivity matrices. In the same way there holds for the global stiffness matrix K

$$K = \sum_{s=0}^{p-1} A_s^T K_s A_s,$$

but neither f nor K are actually accumulated.

Algorithm 1

Initialization:

$$r_s := K_s u_s - f_s,$$

calculate $w_s = A_s w$ from $w := C^{-1} r$,

$$\gamma_s := w_s^T r_s, \quad \gamma := \sum_{s=0}^{p-1} \gamma_s, \quad q_s := w_s.$$

Iteration:

$$v_s := K_s q_s, \quad \delta_s := v_s^T q_s, \quad \delta := \sum_{s=0}^{p-1} \delta_s, \quad \alpha := -\gamma/\delta,$$

$$u_s := u_s + \alpha q_s, \quad r_s := r_s + \alpha v_s,$$

calculate $w_s = A_s w$ from $w := C^{-1} r$,

$$\gamma_s := w_s^T r_s, \quad \hat{\gamma} := \sum_{s=0}^{p-1} \gamma_s, \quad \beta := \hat{\gamma}/\gamma, \quad \gamma := \hat{\gamma}, \quad q_s := w_s + \beta q_s.$$

The counterpart of vectors of additive type are vectors of *overlapping type*, that means that the correct values are stored on each processor:

$$u_s = A_s u.$$

With these definitions we are prepared to introduce our method for solving the system of equations, the parallelized preconditioned conjugate gradient method (PPCG) [19] for solving $C^{-1}Ku = C^{-1}f$, see Algorithm 1. The vectors u, w, and q are of overlapping type, and f, r, and v are of additive type. Note that this choice leads to the elegant realization of the scalar products $\delta := v^T q$ and $\hat{\gamma} := w^T r$,

$$v^T q = \left(\sum_{s=0}^{p-1} A_s^T v_s \right)^T q = \sum_{s=0}^{p-1} v_s^T A_s q = \sum_{s=0}^{p-1} v_s^T q_s,$$

and of the matrix vector multiplication,

$$v = Kq = \sum_{s=0}^{p-1} A_s^T K_s A_s q = \sum_{s=0}^{p-1} A_s^T K_s q_s = \sum_{s=0}^{p-1} A_s^T v_s \quad \text{with } v_s := K_s q_s.$$

Each PPCG iteration contains communication in form of two global sums of single numbers ($\delta := \sum_{s=0}^{p-1} \delta_s$, $\gamma := \sum_{s=0}^{p-1} \gamma_s$) and in form of a change of the type of vectors in the preconditioning step $w := C^{-1} r$. Even without preconditioning ($C = I$) this step includes *accumulation*, that means the realization of

$$w_s = A_s w := A_s r = A_s \sum_{i=0}^{p-1} A_i^T r_i.$$

This step is the most expensive part in the PPCG in comparison with a one processor variant. It must be treated with much care, which is reflected in the choice of the data structure and the expense for a favourable distribution of the elements to the processors. Details about the implementation of preconditioners are omitted here; they can be found for example in [3, 19].

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*
4 – 7	global identification of the *Kette* (*KettenId*)

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

2.2 Accumulation

Before we describe the part of the data representation which is necessary to understand the accumulation algorithm we want to clarify some notation which may be used in a slightly different manner by other authors.

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

All nodes at the coupling edges/faces may belong to several processors. Because their number is of the order $2^k \sim h^{-1}$ and $2^{2k} \sim h^{-2}$, respectively, we shall avoid expensive searches during the communication process by demanding from the mesh generator that the nodes of each edge/face are numbered consecutively, and in the same manner on all corresponding processors. Thus these nodes are identified by a pointer to the first node, the number of nodes at this edge/face, and a characterization of this edge/face. We denote such a sequence by *Kette*, the German word for chain. Note that the coupling edges/faces can be characterized by a global edge/face 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/faces as described in [7] for two dimensional problems. In the program, each *Kette* is described by a vector of integer type and of dimension 7, see Table 1 for an explanation. These vectors are stored in an array `Kette`.

The accumulation is divided into two steps, the accumulation of the data at crosspoints and of *Kette* data. For both steps we assume that the parallel computer has at least a logical hypercube topology, for an introduction to hypercubes see [21, 22]. Notice that physically neighboured subdomains may be placed on processors which are not adjacent physically or logically.

The accumulation of data at crosspoints is performed via Algorithm 2. It needs an auxiliary array `H` and one global communication.

It would be easy to modify Algorithm 2 for the accumulation of the data at the coupling edges/faces, but there are strong disadvantages: First, a large auxiliary array is necessary. And second, a large amount of useless information is exchanged, its part is increasing with the number of processors. So we try to use the model 'hypercube' in a specific way. We will explain the idea in an example.

Algorithm 2

Initialize `H` with 0 and write the values of the local crosspoints to the appropriate places.

Perform a cube sum for `H`.

Get the accumulated values for the local crosspoints out of `H`.

Consider a *Kette* that belongs to the three processors with the numbers

$$p_1 = 11 = \mathrm{000L0LL},$$
$$p_2 = 17 = \mathrm{00L000L},$$
$$p_3 = 65 = \mathrm{L00000L}.$$

If we break Link 0 of all processors then the hypercube of dimension `ncube` is split in two sub-hypercubes of dimension `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`

$$\texttt{PathId} = \mathrm{L0LL0L0};$$

L means that the corresponding link is necessary, 0 indicates that a communication via this link is without use. This integer is calculated once and stored in the third column of the array `Kette`, compare Table 1. We remark that 1) *Kette*s that belong to one processor only, and 2) *Kette*s of length 0 need not to be communicated. This is indicated by `PathId` $= 0$.

Our aim is 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* themselves. Algorithm 3 realizes the approach described in the example. It needs three auxiliary buffers `Wait`, `Send`, and `Recv` to store *Kette*s. They are initially empty, `Send` is always empty after step 2, `Recv` always after step 3, and `Wait` at the end of the algorithm. For a simple description we denoted the set of *Kette*s that the processor possesses itself, by `Own`. Note that only the main point is explained in Algorithm 3, namely our use of hypercubes. Indeed, our code contains two modifications to avoid some useless operations:

1. *Kette*s which belong to faces are communicated to only one (at most) processor. That means each intermediate processor sends it only once. So one can reduce the range of the loops in Steps 1 and 3.

2. Step 3 contains some searches (IF `Kette` IN `Own/Wait`). The time for this is reduced by recording the actions in the first iteration and using this record in all subsequent iterations of the PCCG algorithm.

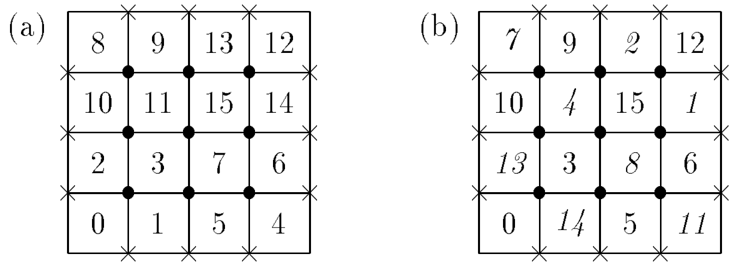
For details see [2].

We will complete this section with the remark that 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

Algorithm 3

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

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

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

 3. FORALL Kette IN Recv DO
       IF (Kette IN Own)
       THEN Accumulate Values in Own.
       ELSE IF (PathId ≥ 2^nrlink)
          THEN IF (Kette IN Wait)
             THEN Accumulate Values in Wait.
             ELSE Add Kette at the end of buffer Wait.
             END IF
          END IF
       END IF
    DONE

 4. Compress Wait by deleting each Kette with PathId < 2^nrlink.

 5. TestBit:=TestBit*2

DONE
```

following two examples of the processor distribution are constructed using the Gray code [21].



× boundary edges with two adjacent subdomains

• inner edges with four adjacent subdomains

In the case (a), the dimension of the subcube is 1 for faces and boundary edges and 2 for inner edges, because the numbers of the processors of adjacent subdomains differ in exactly one bit. In case (b) they differ in (`ncube` − 1) bits. Consequently, the complete hypercube is necessary for the accumulation of each inner 1D-*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 (hardwired) 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.

3 Comparison of several partitioning algorithms

3.1 The test examples and the metrics

In the following we will denote a set of elements assigned to one processor as a *partition*. The process of dividing a mesh into n subsets and mapping it to the processors we will call *partitioning* and its result *decomposition*.

If we consider a graph $(\mathcal{V}, \mathcal{E})$, we mean always the dual graph of the mesh: the elements of the mesh are the vertices of that graph and edges between vertices are introduced if the corresponding elements share a common face.

Our tests are performed on two FE-meshes, *cube768* (Figure 1) and spc3-123 (Figure 2). While the first one is very regular the second one is quite unstructured. [t]

For the evaluation of a computed decomposition **Chaco** provides 7 different metrics. We will use only four of them. For their description let $\mathcal{V}_i$ be a subset of vertices, v_i a vertex, $\mathcal{E}_i$ a subset of edges, e_{ij} the edge between the vertices v_i and v_j, and $\mathcal{E}_{\mathcal{V}_i\mathcal{V}_j} := \{e_{\ell m} : v_\ell \in \mathcal{V}_i, v_m \in \mathcal{V}_j\} \subset \mathcal{E}$ the set of cut edges between subset $\mathcal{V}_i$ and $\mathcal{V}_j$.

Set Size: The total weight of the vertices in a set. Since all our vertices have weight 1 the interesting values are:

$$\min_i |\mathcal{V}_i|, \quad \max_i |\mathcal{V}_i|.$$

In a balanced decomposition the maximal and minimal value should be as close as possible.

Edge Cuts: The weight of those edges which connect a vertex in one set to vertices in

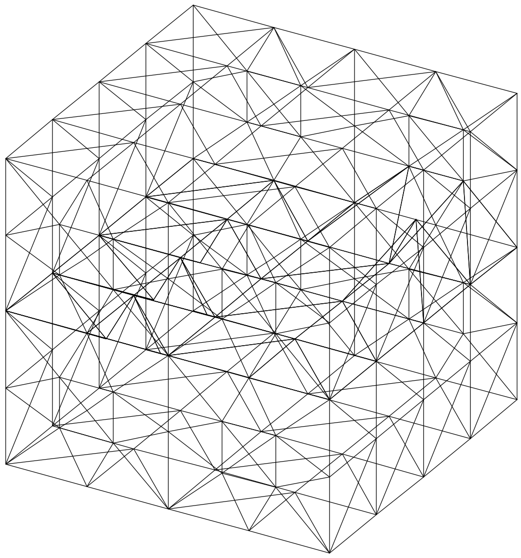


Figure 1: *FE-mesh cube768 with 768 tetrahedral elements.*

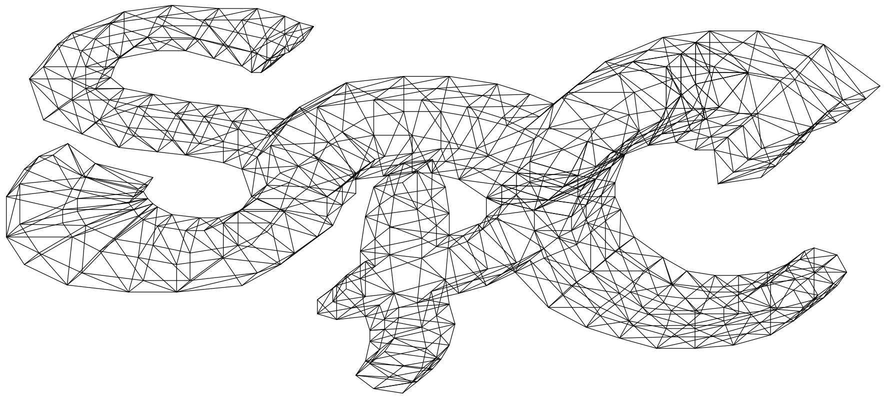


Figure 2: *FE-mesh spc3-123 with 1398 tetrahedral elements.*

another set (cut edges). All our edges have weight 1 and so we are interested in:

$$\sum_{i \neq j} |\mathcal{E}_{\mathcal{V}_i \mathcal{V}_j}|.$$

Hypercube Hops: A measure in which each cut edge is multiplied by the architectural distance between the two processors owning its vertices. This metric models communication time often better than Edge Cuts does because it takes into account network congestion. In our special case this could be represented by:

$$\sum_{i \neq j} h_{ij} |\mathcal{E}_{\mathcal{V}_i \mathcal{V}_j}|,$$

where h_{ij} is the number of bits that are different in the binary representation of i and j.

Internal Vertices: The total weight of all the vertices in a set which have no edges connecting them to vertices in other sets. As discussed in 3.4, the presence of such vertices may allow for an overlap of communication with computation. In a short form this will be represented by:

$$\sum_{i} |\{v_\ell \in \mathcal{V}_i : \forall e_{\ell m} \quad \text{is} \quad v_m \in \mathcal{V}_i\}|.$$

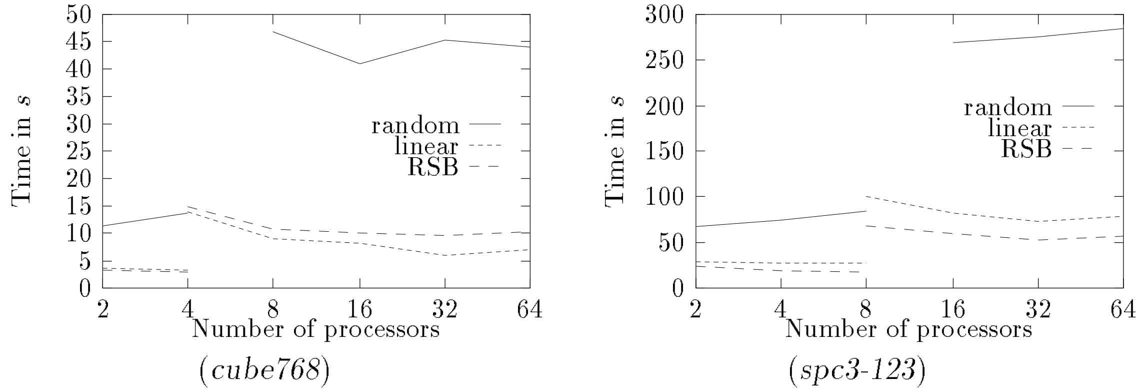


Figure 3: *Computing times of different decompositions for hypercube dimension* $d = 1, \ldots, 6$.

cube768				*spc3-123*			
	random	linear	RSB		random	linear	RSB
Set Size (max./min.)	12/12	12/12	12/12	Set Size (max./min.)	22/21	22/21	22/21
Edge Cuts	1390	544	584	Edge Cuts	2232	655	526
Hypercube Hops	4181	736	1130	Hypercube Hops	6778	1834	926
Internal Vertices	0	68	118	Internal Vertices	0	470	682

Table 2: *Metrics of the random and linear decomposition and of the recursive spectral bisection for a distribution among 64 processors.*

For a performance test of the decompositions we have measured the total execution time of the PPCG. This is the critical part in our code with most of the communication, other expensive parts like assembling the stiffness matrix are fully parallel without communication. Note that the measured time is the total time, a composition of user and system time and the maximum over all processors.

3.2 Spectral bisection

The spectral bisection from Pothen, Simon, and Liou [20] is probably the best known partitioning algorithm of spectral type. It provides good results but it is expensive. Therefore the spectral bisection and their recursive application (RSB) is suited only for small meshes (about 1000 elements). Our meshes are from the right size (recall that we distribute only $\mathcal{T}_0$) and so we first test the RSB against random and linear partitioning.

In the linear scheme, vertices are assigned to the processors according to their numbering in the original graph. In our case of an unweighted graph with n vertices divided into p sets, the first n/p vertices would be assigned to set 0, the next n/p to set 1, etc. This mostly produces surprisingly good results because data locality is often implicit in the vertex numbering. In the random scheme, vertices are assigned randomly to sets in a way that preserves balance.

Our tests start with a comparison between a random decomposition as the worst case, the linear decomposition as the simplest case, and the recursive spectral bisection. The computing times are shown in Figure 3. In Table 2 we give the metrics for the case $d = 6$.

As expected we obtain an extreme computation time for the random scheme. For the linear algorithm we get an interesting result. The left part of Figure 3 shows a better time for the linear decomposition than for the recursive spectral bisection. But this result seems to be mesh specific and not the general case as shown clearly in the right part of Figure 3. The very good outcome of the linear decomposition on partitioning *cube768* results first from the number of elements as a multiple of 64 and second from the regular mesh and its advantageous numbering. On a very irregular mesh like *spc3-123* the recursive spectral

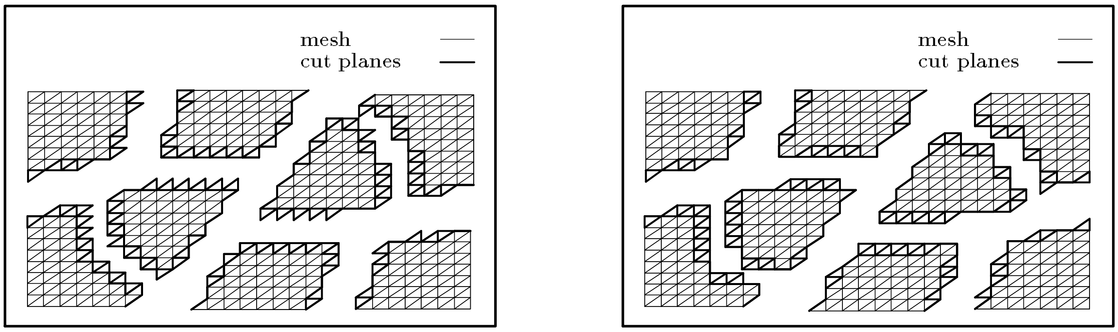


Figure 4: *Tetrahedral mesh with 1800 elements partitioned into 8 parts by recursive spectral bisection. Left: Normal RSB. Right: RSB with local KL refinement.*

	RSB	RSB+KL
Edge Cuts	165	142
Hypercube Hops	244	217

Table 3: *Metrics for a tetrahedral mesh with 1800 elements.*

bisection shows its superiority because of its independence from the shape of the domain and numbering of the mesh.

Hendrickson and Leland [13] have generalized the recursive spectral bisection to a recursive spectral quadri- and octasection. We also made tests with these algorithms but we could not find improvements against the bisection so we will not mention the quadri- and octasection case further.

3.3 Kernighan-Lin refinement

The heuristic of Kernighan and Lin [17] (KL) is an iterative algorithm that tries to reduce the number of edges cut by the decomposition by moving vertices between sets. It is mainly a local optimization strategy and so a start distribution must be given. Doing this in the original way with a random distribution at start the algorithm can not really satisfy. On the other hand, the spectral bisection introduced in 3.2 computes very good global results but its local performance is rather poor. Boundaries between two sets are usually rough and not optimal so it seems to be advantageous to use the KL for a local refinement of every bisection.

As an illustration we show in Figure 4 an example with a nearly two dimensional domain, in the third direction we use only one layer of elements. In the left part we see the partitioning result of the standard RSB with very rough boundary. The right decomposition was found via RSB and local KL refinement and has a much smoother boundary. This observation is also proved by Table 3.

Also in our two test examples we get a good improvement of the decomposition. Figure 5 shows the reduction of computing time by the local KL refinement and Table 4 contains the corresponding metrics.

3.4 Post-processing

In this section we want to discuss three post-processing strategies. Post-processing means that a full decomposition is already given and we only want to improve it. Contrary to this the local KL refinement is applied in each step of the (spectral) bisection; therefore it was described separately.

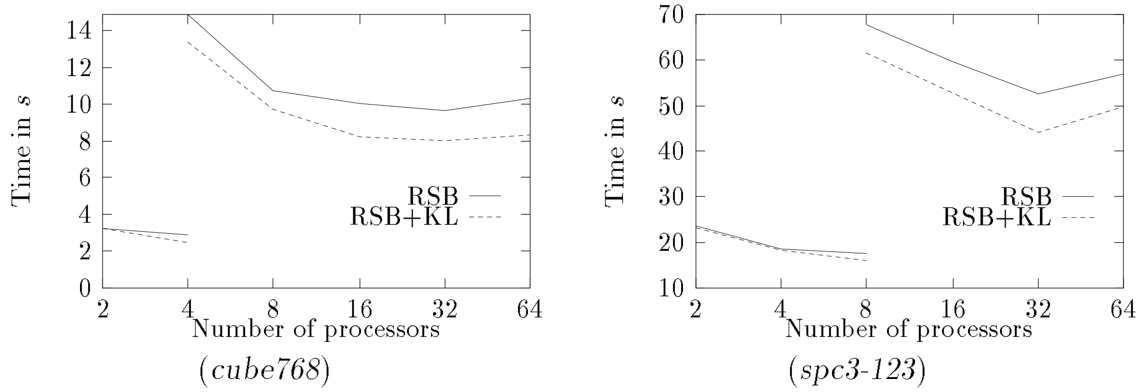


Figure 5: *Computation times for recursive spectral bisection with and without local KL refinement on hypercube dimension $d = 1, \ldots, 6$.*

cube768		
	RSB	RSB+KL
Edge Cuts	584	516
Hypercube Hops	1130	978
Internal Vertices	118	115

spc3-123		
	RSB	RSB+KL
Edge Cuts	526	458
Hypercube Hops	926	708
Internal Vertices	682	659

Table 4: *Metrics for the recursive spectral bisection with and without local KL refinement for a distribution to 64 processors.*

The first strategy is to try to improve (*refine*) the *mapping* (RM) of the sets (partitions) to the processors. The sets themselves remain unchanged. To achieve this the code determines how the hypercube hop metric would change if we swap any two sets which are adjacent in the hypercube. The swap with the maximal improvement is performed and the process goes on until no further improvement is possible. Note that all other metrics remain constant during this algorithm.

The positive practical result is shown in Figure 6 and Table 5. The computation time is shortened with a decreasing number of hypercube hops, not proportionally however. The reason for the positive influence is surely our realization of the communication, see Subsection 2.2. Thus we recommend to apply this post-processing always.

Note that partitioning and mapping can also be interwoven, but this is postponed to Subsection 3.5.

The second post-processing strategy consists in *increasing* the number of *internal vertices* (IV). This can make sense in two ways: There is no need for communication across internal vertices and only local data are required to process them. This can reduce the total amount of communication and may allow for overlapping communication and computation since the computation associated with an internal vertex can be performed while waiting for data from other processors to arrive (but this is not exploited yet).

To accomplish this the procedure first determines the number of internal vertices in each set. Then the set with the fewest internal vertices receives vertices from other sets to make some of its own vertices to internal ones, and it gives back other vertices to preserve the balance.

The results of our tests are shown in Figure 7 and in Table 6. The success of this post-processing clearly depends on the implementation of the communication and it is obvious that our test application *SPC-PMPo3D* is not able to profit from this idea. For other applications this result may be completely different depending on the possibility to overlap communication and computation.

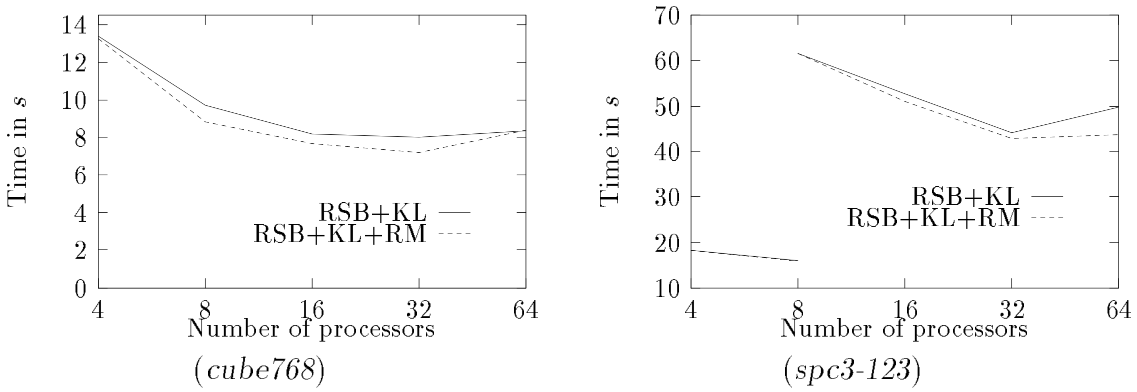


Figure 6: *Computation time of the recursive spectral bisection with local KL refinement with and without improved mapping (RM) for hypercube dimension* $d = 2, \ldots, 6$.

cube768			*spc3-123*		
Proc.	RSB+KL	RSB+KL+RM	Proc.	RSB+KL	RSB+KL+RM
4	76	76	4	52	52
8	180	150	8	93	93
16	384	317	16	206	177
32	660	561	32	395	331
64	978	875	64	708	615

Table 5: *Hypercube-Hop-Metrics of the recursive spectral bisection with local KL refinement with and without improved mapping.*

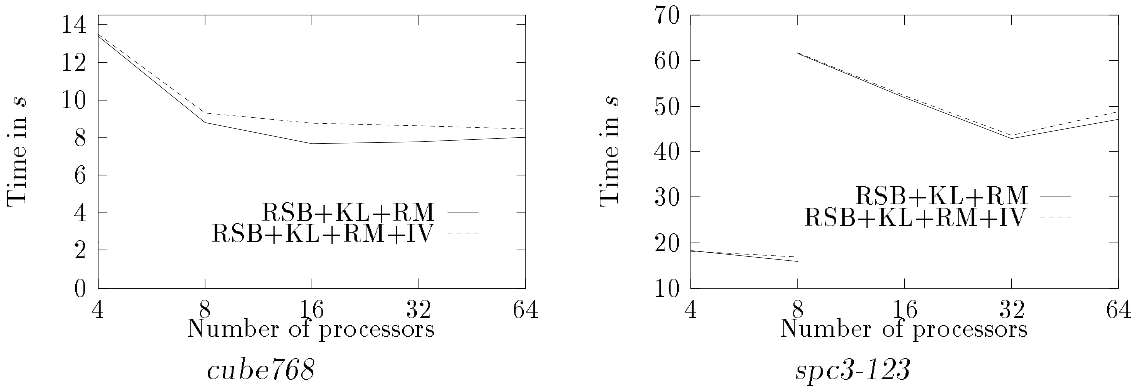


Figure 7: *Computation times for the local KL refined recursive spectral bisection with and without increasing the number of internal vertices.*

cube768			*spc3-123*		
Proc.	RSB+KL+RM	RSB+KL+RM+IV	Proc.	RSB+KL+RM	RSB+KL+RM+IV
4	628	631	4	1303	1305
8	522	525	8	1236	1238
16	342	360	16	1100	1098
32	213	245	32	903	908
64	115	148	64	659	679

Table 6: *Number of internal vertices for a local KL refined recursive spectral bisected partitioning with improved mapping with and without increasing internal vertices.*

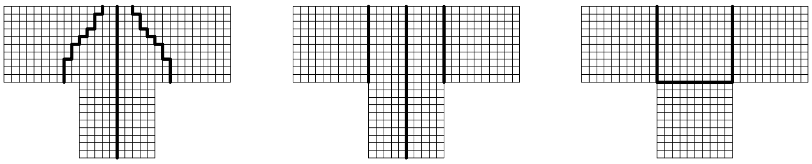
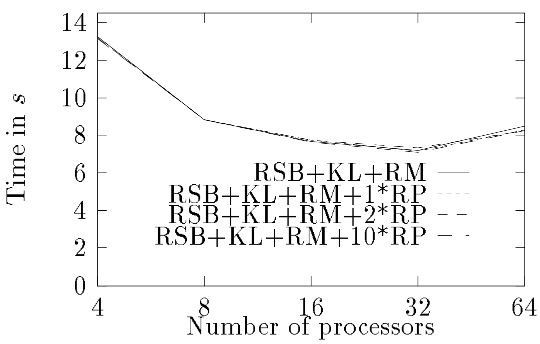
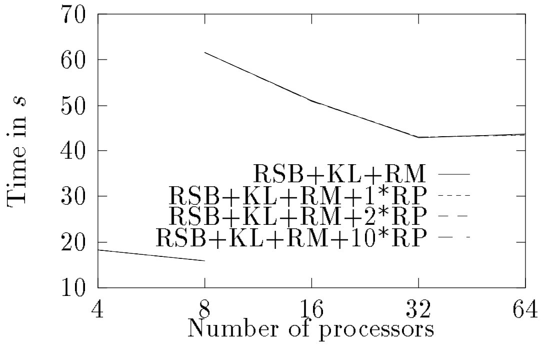


Figure 8: *Illustration of global KL refinement. Left: Initial decomposition. Middle: One step of global refinement. Right: Two steps of global refinement.*



(*cube768*)



(*spc3-123*)

Figure 9: *Computation times for the recursive spectral bisection local KL refined with improved mapping and with different number of iteration steps on hypercube dimensions* $d = 2, \ldots, 6$.

The last post-processing strategy is called *refine partition* (RP). The idea is the following: In the recursive generation of a decomposition, some information is lost with each recursion level. For example, a KL refinement is performed between only a fraction of the total number of adjacent sets. That why **Chaco** provides the possibility to perform a local refinement between *all* pairs of sets. To do so, first the number of edges between each pair of sets is determined. Kernighan-Lin refinement is then performed between each pair in descending order from the pair with the largest number of edges to that with the smallest.

The result of such a global refinement is shown in Figure 8. The recursive spectral bisection does not provide an optimal result (left picture). The number of edge cuts is here 50. In the middle we see the result of recursive spectral bisection with one step global refinement. The quality is clearly better, namely only 40 edge cuts. The picture on the right hand side shows the result of recursive spectral bisection with two steps global refinement. Now the optimal decomposition with 30 edge cuts is reached.

The practical use of this strategy is shown in Figure 9. Obviously the success is very little. The decrease in computation time is barely visible. Because the algorithm is cheap in comparison with the whole partitioning and it may improve the decomposition it makes sense to use it. The number of steps should depend on the number of mesh elements. Up to 1000 elements 2 or 3 steps are enough, for more elements some further steps could be advantageous. But we doubt that more that 10 steps are useful.

3.5 Terminal propagation

As we have seen in Subsection 3.4, not only the number of edge cuts is important for the quality of a decomposition but also an intelligent mapping. Post-processing is suited to improve our decomposition, but the result may still be not optimal. For an illustration consider Figure 10: A mesh is distributed to eight processors. We connected those processors by a line which contain adjacent subdomains. The dotted lines represent links in the

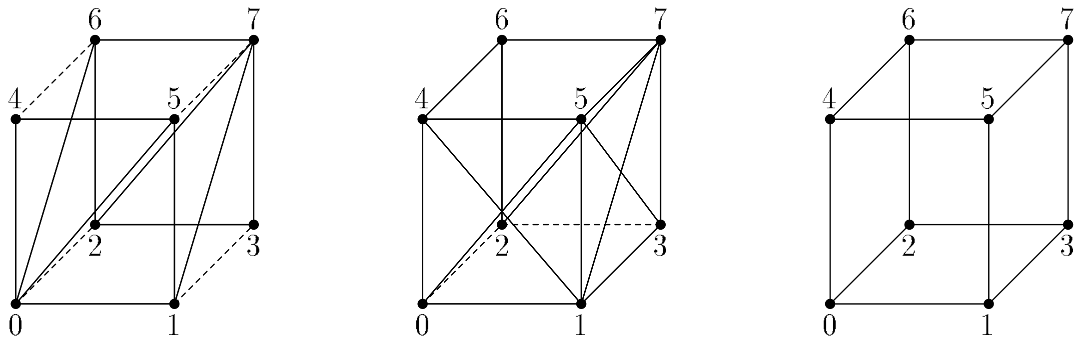


Figure 10: *Inter processor relations for a distribution of a $4 \times 4 \times 2$ hexahedral mesh to 8 processors. Left: standard RSB, Middle: RSB with KL refinement and improved mapping. Right: RSB with terminal propagation.*

hypercube. The left picture shows the result with standard RSB, the direct links are badly used. Improved mapping (RM) can help (middle). But if some subdomains have more than three (the hypercube dimension) adjacent ones, post-processing will never be able to create a decomposition which matches perfectly the hypercube, as in the right picture. Better distributions can only be found by a coupling of the dividing and the mapping. A technique which is able to do this is the *terminal propagation* [5, 16].

Assume we are in an intermediate stage of a recursive bisection algorithm. We have just bisected the first half of the graph and will now process the other half. The idea behind terminal propagation is to enhance the graph model of the actual subproblem by introducing two new vertices called *terminals* representing the two partitions of the first half and restricted to $\mathcal{V}_0$ and $\mathcal{V}_1$ respectively. Now we introduce new *terminal edges* reflecting edges between the parts of the first half and the vertices of the actual subproblem. The terminal edges are weighted with a value of *preference*. This is the preference of the normal vertex to be assigned to either $\mathcal{V}_0$ or $\mathcal{V}_1$. Now the problem is solved for this enhanced graph. There are several ways to do this, in the implementation of **Chaco** Hendrickson and Leland use an extended eigenproblem $Ax = \lambda x + g$ with a preference vector g to model the minimal problem for the enhanced graph.

For a test of the effectiveness of this method we compare the RSB with terminal propagation with the best method hitherto. The results are shown in Figure 11 and Table 7. As we have guessed the application of terminal propagation produces in general better results than all other methods and post-processings till now.

At last we want to test if we can improve the result by using the possibility to scale the preference vector in order to model the relative importance of generating a new edge cut versus increasing the interprocessor distance associated with an existing edge cut. This controls the tradeoff between importance of communication volume and communication locality. In **Chaco** the vector is scaled by setting the parameter `CUT_TO_HOP_COST` to a real value. The default is 1.0.

Figure 12 shows the results. It is clearly to be seen that local minima in the hypercube hop curve lead to local minima in the computing time. But it is also shown that this kind of tuning is very sensitive and the optimal parameter is mesh specific. It has to be determined for every mesh by making a test series. But it is questionable if such a high expense for relative small improvements against the default value 1.0 makes sense.

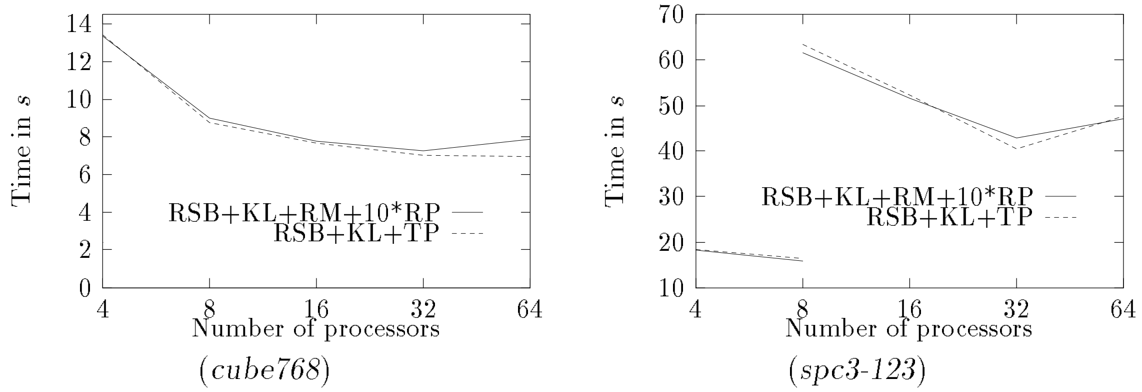


Figure 11: *Comparison of the computing times for the decomposition with recursive spectral bisection with terminal propagation or with recursive spectral bisection with improved mapping and 10 iterations global KL refinement. Both methods with local KL refinement.*

cube768			*spc3-123*		
	RSB+KL +RM+10*RP	RSB+KL +TP		RSB+KL +RM+10*RP	RSB+KL +TP
Edge Cuts	501	555	Edge Cuts	454	508
Hypercube Hops	873	661	Hypercube Hops	625	544

Table 7: *Comparison between the metrics for a recursive spectral bisection with local KL refinement,* `REFINE_MAP` *and* `REFINE_PARTITION=10` *and that for a RSB with terminal propagation for a distribution among 64 processors.*

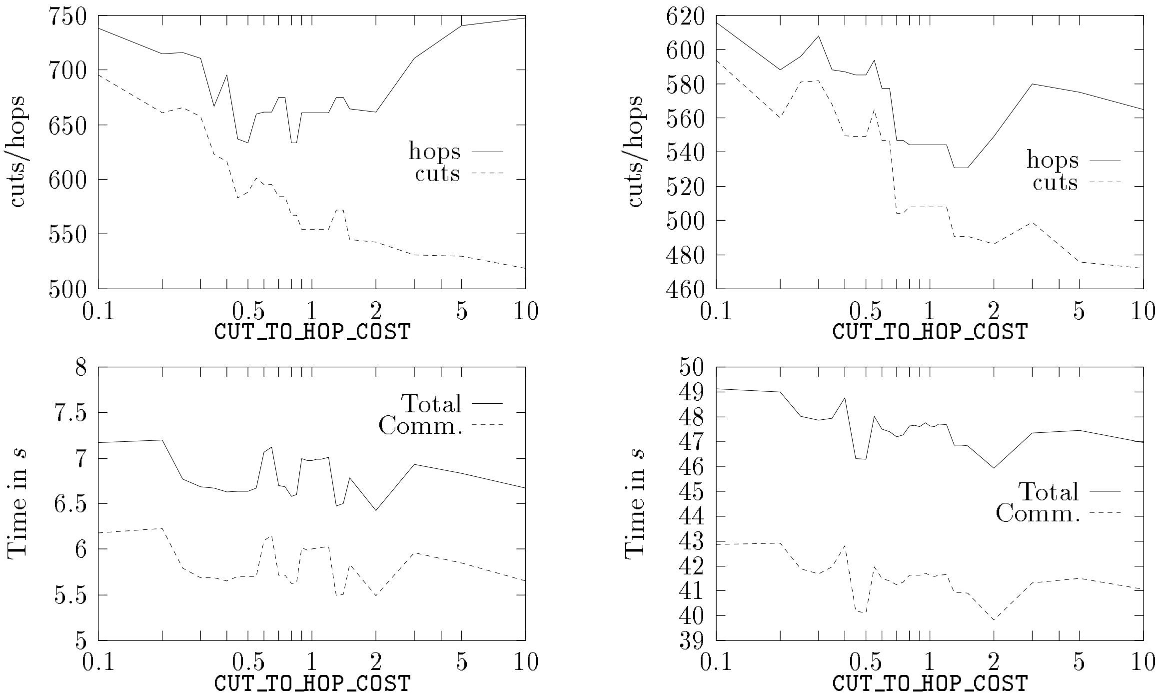


Figure 12: *Effect of the parameter* `CUT_TO_HOP_COST`. *Above: Metrics, Below: Communication and computation times. Left: Example cube768, Right: Example spc3-123.*

4 Summary

Before we summarize our tests we have to note two things.

First, the measured times are showing only a tendency, they reflect the approximate computing time. An exact time measurement up to a hundredth of a second is unrealistic and meaningless in practice. The execution times for one distribution can vary between some tenth of a second because our application is no stand alone version but it works with the Parix operating system which needs more or less time too. Therefore time differences less than 1% between distributions have to be neglected in this context.

Second, all tested partitioning algorithms were computed on a MC68040 based machine. Comparisons with other systems like Sun, HP, or Pentium-PCs have shown that the results from **Chaco** are system dependent. On request Bruce Hendrickson confirmed that the computations in **Chaco** are influenced by machine dependent values like the machine epsilon. Also the generation of random numbers is important for the algorithms in **Chaco** and this is machine dependent too. With respect to this it may possible to obtain slightly different results on other machines. An individual test on at least two different machines and a comparison of the results by the obtained metrics could make sense.

Nevertheless, neglecting this limitation, we can derive some conclusions from the tests. First there is no doubt that in general the recursive spectral bisection computes much better results than the random and linear scheme, see 3.2. Furthermore, it was obvious that the local refinement with Kernighan-Lin results always in better decompositions. The time saving runs to an average of 18%.

The improvement of the mapping to fit the hypercube was very useful, see 3.4. We obtain a further decrease in the computing time between 5 and 7 percent.

The improvement procedure is not necessary if we use terminal propagation. This enhancement of the ordinary spectral bisection results always in the best decompositions, see 3.5. Against the spectral bisection with local KL refinement and improved mapping the computing time consumption could be reduced by 8–15%. A further tuning of this algorithm is possible by variation of the parameter `CUT_TO_HOP_COST`, but this is expensive and probably not worth the expense. Nevertheless, we could measure in the test an improvement up to 7.8%.

The global KL refinement as post–processing could also improve the decomposition, see 3.4. The test shows a shortening of the computing time by about 2% which is very close to the measurement precision. Since the procedure is less expensive some iteration steps as post–processing may be useful.

The increase of the number of internal vertices was rather counterproductive in the test, see 3.4. But as already explained this might be different for other applications.

To summarize our case study we could give the following recommendations: As the best algorithm we found the recursive spectral bisection with terminal propagation and local KL refinement. Also 2–3 iterations global KL refinement as post-processing are reasonable. This should in general result in very good decompositions. Improvements, resulting from other algorithms or further post-processing, are certainly mesh specific, therefore a more detailed recommendation for general meshes is not possible.

Acknowledgement. The work of the authors is supported by DFG (German Research Foundation), Sonderforschungsbereich 393. Subsection 2.2 is an extract of the paper [2] which was written by the first author together with G. Haase, A. Meyer and M. Pester.

Also we wish to thank B. Hendrickson and R. Leland for licensing **Chaco** which made this chase study possible.

References

[1] Th. Apel. *SPC-PMPo 3D* — User's Manual. Preprint SPC95_33, TU Chemnitz-Zwickau, 1995.

[2] Th. Apel, G. Haase, A. Meyer, and M. Pester. Parallel solution of finite element equation systems: efficient inter-processor communication. Preprint SPC95_5, TU Chemnitz-Zwickau, 1995.

[3] Th. Apel, F. Milde, and M. Thess. *SPC-PMPo 3D* — Programmer's Manual. Preprint SPC95_34, TU Chemnitz-Zwickau, 1995.

[4] P. Bastian. *Parallele adaptive Mehrgitterverfahren.* Teubner, Stuttgart, 1996.

[5] R. Van Driessche. *Algorithms for Static and Dynamic Load Balancing on Parallel Computers.* PhD thesis, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, November 1995. ISBN 90-5682-007-9.

[6] R. Van Driessche and D. Roose. A spectral algorithm for constrained graph partitioning I: The bisection case. TW Report 216, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, October 1994.

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

[8] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Trans.*, CAD(CAD-4):92–98, 1985.

[9] 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.

[10] 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.

[11] S. Hammond. *Mapping unstructured grid computations to massively parallel computers.* PhD thesis, Rensselaer Polytechnic Institute, Dept. of Computer Sience, Troy, NY, 1992.

[12] B. Hendrickson and R. Leland. *The Chaco User's Guide – Version 2.0.* Sandia National Laboratories, Albuquerque, NM. SAND94-2692.

[13] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. Technical Report SAND92-1460, Sandia National Laboratories, Albuquerque, NM, September 1992.

[14] B. Hendrickson and R. Leland. An improved spectral load balancing method. In *6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 953–961. SIAM, March 1993.

[15] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.*, 16, 1995.

[16] B. Hendrickson, R. Leland, and R. Van Driessche. Enhancing data locality by using terminal propagation. To be presented at the minitrack on Partitioning and Scheduling at the HICSS-29 Conference, Maui, Hawaii, January 3–6, 1996.

[17] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–308, 1970.

[18] S. Lang. Lastverteilung für parallele, adaptive numerische Mehrgitterberechnungen. Diplomarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1994.

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

[20] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparce matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, 1990.

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

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

[23] H. D. Simon. Partitioning of unstructured problems for parallel processing. In *Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Pergammon Press, 1991.

[24] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency*, 3:457–481, 1991.