Domain Decomposition

A. Introduction

In general, there are three common classes of parallelism [1, 2]

1. Farming parallelism

   This is the most obvious form of parallelism. Here the same problem is run with different values of data.

2. Geometric parallelism

   This type of parallelism is characterized by domain decomposition or data parallelism. The same algorithm is executed on different processors, that may run asynchronously, with each processor working on a different part of the data set.

3. Algorithmic or Functional parallelism

   This is a fine grained parallelism compared to the event and geometric type of parallelism. Typically, different parts of an algorithm are executed in parallel on different processors. The data then ‘‘flows” through the system requiring elaborate communications and control structures in the code. For large applications with many data, this type of parallel implementation would incur communication overheads.

We concentrate on the second type of parallelism, to be more precisely on domain decomposition methods, as these are interesting for many scientific and engineering applications. The advantage of a data parallel application is that for problems for which already a sequential solver exists, it requires in general not much effort to change the sequential version into a parallel one.

A.1. An Example

Let us consider an example in which data decomposition can be applied. Suppose there is some computer program that can detect traffic jams in our country, based on the position and velocities of cars participating in traffic. For this purpose all cars have been equipped with a transmitter that sends from time to time the position and velocity of the car to a satellite. The satellite in turn sends the collected positions to a computer. The computer keeps track of recent – and previous positions and velocities of cars and based on these data and the current data the program determines the state of the car traffic. High concentrations of cars indicate the presence or origin of traffic jams in which case appropriate action can be undertaken. For low concentrations no action is needed.

If all data from the satellite is to be processed by a single computer it might take a long time before the results have been computed. In the mean time the actual traffic situation could have changed significantly. To avoid this a parallel computer can be used, where the processors compute the results for only part of the country. The question is then which part(s) of the country is (are) assigned to which processor? A naive solution would be to divide the area of the country by the number of processors and to assign all traffic in a sub-area to a processor. In this case, the processors that take care of sub-areas with a relatively small amount of cars would have to accomplish much less work than the processors that compute the results for sub-areas with lots of cars. The workload per processor will then be very uneven.

Another approach is to assign an equal number of cars to each processor. However, neither this is a guarantee for an even distribution of the work among the processors. The amount of work per car may differ for different cars. The computations for cars that are not involved in situations with a lot of other cars will definitely be less complicated than for cars that are involved in heavy traffic.

Another situation that should be avoided whenever possible, is that in areas with a lot of cars two or more processors decide upon the same traffic situation. The detection of traffic problems is based upon the current and previous positions and velocities of the cars in a certain area. If the data of a specific car is needed on processor A, while it actually resides on processor B, the data of the car (the current and previous positions and velocities) have to be transferred from processor B to processor A. The transfer of many data from a processor to other processors will slow down the performance of the whole system. The only way to prevent an excessive communication overhead is to take care that all information needed by a processor is already there. This can be attained by pursuing both *temporal* and *spatial* locality. Temporal locality is based upon the observation that a car that is involved in a situation with lots of traffic will probably be in a similar situation in the next time step. To perform the computations associated with the car efficiently for the next time step the (recent) "history" data of the car should be directly available to the processor that performed the computations in the previous time step. Spatial locality is based on the observation that cars in the neighborhood of a car involved in heavy traffic are most likely to be there also in the next time step. Also in this case, the data of the cars in the neighborhood of the car should be directly available to the processor that performed the computations for the car in the previous time step.

In this example a number of aspects of domain decomposition methods can be found. Generally, the data set (the data associated with the cars) is large or very large. Second, the result of a good decomposition should make sure that the amount of work per processor is the same and that the data transfer between processors is minimized. This goal may be hampered by the different weights of the elements in the data set. Cars in situations with lots of other traffic require more (complicated) computations than cars with little traffic in their neighbourhood. In the third place, applications may require that domain decompositions are performed at regular intervals due to changing circumstances. In the example, the traffic situation changes dynamically: traffic jams may resolve, new jams may arise etc.

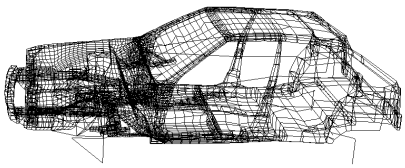

Figure 1: A finite element mesh for a car body.

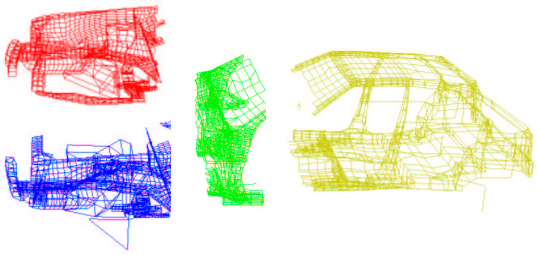

Figure 2: A possible decomposition of the mesh of Figure 1.

As another example consider the simulation of car crashes. Let us however take a closer look at it. Just like in the two-dimensional diffusion problem, we have to discretize the body of the car in small computational elements, and using some numerical technique, the forces and deformations of the computational elements are computed. A possible discretization of a car body is shown in Figure 1. The method that is used to solve the equations that model the car crash is called the *finite element method*, and the corresponding discretization, as shown in Figure 1, is called a *finite element mesh*. Although the differences between the finite element method for the car crashes and the finite difference schemes for the solution of the diffusion equation are huge, a few important similarities can be observed. In most cases the forward integration in time in

the car crash simulation is based on explicit schemes. So, there is no need for solving equations. Furthermore, the update procedures are also based on stencil operations. This means that for updating the state of a computational element to the next time step, only information from neighboring elements is required. Therefore, parallelization can be achieved by domain decomposition, much like the strip wise decomposition of the two-dimensional domain in the diffusion equation. Clearly, decomposing the mesh of Figure 1 such that a good loadbalance and minimal communication overhead is achieved, is far from trivial. A possible decomposition is shown in Figure 2, but many more can off course be found. In this chapter we will discuss methods to find such (sub-) optimal decompositions.

A.2. Data Sets

The data sets can come from many scientific or engineering problems. The data can be galaxies in astrophysics problems, neurons in artificial neural networks, pictures in image processing, grid points in problems that are solved by Finite Difference Methods like problems that compute electric fields in a parallel plate capacitor, elements in problems that are solved by Finite Element Methods (FEM) like in structural analysis for simulating car crashes. FEM were originally developed for structural analysis applications which are often characterized by an irregular geometry. The finite difference methods did not cope well with such geometry. The applications required changes in mesh spacing and the accommodation of irregular boundaries. To meet these requirements FEM were introduced that divided the data domain into elements, for example triangular or quadrilateral elements, that are described by a number of connecting node points. Instead of performing the calculations on the grid points directly as in finite difference methods, the calculations are performed on the elements based on the locations of the grid points constituting an element, the properties of materials, and such other geometric properties as the thickness of plate elements. The FEM that were developed were general enough to be also applicable to other fields such as in heat transfer problems, in electro-magnetics problems and in fluid dynamics.

Without loss of generality, we will concentrate in the rest of this module on data sets arising from FEM applications. In Figure 3 an example of a data set from a FEM application is shown with triangular elements.

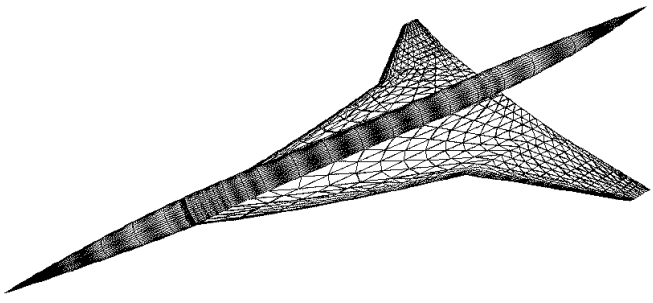

Figure 3: Example of a data set from a FEM application; model of a High Speed Transport Carrier aircraft.

Two important characteristics of data sets that are decisive for the choice of a decomposition method are *homogeneity* and *regularity*. If the data set is homogeneous all items in the data domain are equal in the sense that the amount of work associated with each item is the same. When the amount of work per item is not the same for all items, the domain is called inhomogeneous or heterogeneous. Heterogeneity of a domain can be caused by data items having different sizes, different types, different materials etc.

A data set is said to be regular, when all data items in the domain are placed in some nice pattern, for example grid points in a rectangular grid.

Although the nature of the problems generating data sets can be very different, they generally share a common property: the data sets are very large. Problems with $O(10^4)$ data items are regarded as small-medium sized problems. Large sized problems contain $O(10^5)$ or $O(10^6)$ data items and the need to handle even larger data domains is already there, but cannot be handled due to hardware limitations.

A.3. Domain Decomposition

When the number of data items increases any computer will be faced with two problems: the increase in time to process the complete data domain and the increase in storage requirements. For a sequential computer with sufficient storage capacity an excessive processing time will in general imply a constraint to the size of the data domain. This constraint can be overcome by using a multiprocessor computer instead of a sequential one.

An important class of parallel machines is the distributed memory computer. For this type of computers each processor has its own local memory that cannot be accessed by other processors. In practice, the size of the memory of each processor of a distributed memory machine will be (much) smaller than the size of the memory of a shared memory machine. However, in a distributed memory machine due to the scalability of the hardware the complete memory size can be much larger than for a shared memory machine. By adding more processors to a distributed memory machine the total amount of memory can be enlarged to extents that cannot be handled by shared memory machines. For distributed memory machines the utilization of the total memory can only be fully exploited through efficient domain decomposition.

Also, the processing speed of single processing nodes will always be limited by physical constraints (speed of light). The only way to circumvent this restriction is by using several processing nodes, as is the case for distributed memory machines.

Another argument was already shown. As soon as memory capacity is no longer a limiting factor, the need to reduce processing time can be met by splitting the work over several processors. This is in particular true when the Single Program Multiple Data (SPMD) model is being used for parallelization.

Most problems in scientific and engineering areas are solved by discretizing the continuous domain and by computing the behaviour of a large number of discrete grid points or elements. For such problems the SPMD model is a good way to obtain parallelism, without requiring a large rewrite of already existing code. In the SPMD model each processor executes the same code, but for a different part of the discrete data domain. On Multiple Instruction Multiple Data (MIMD) machines all processors can execute their programs asynchronously. Many large-scale problems that are solved according to the SPMD model on a MIMD platform fall into the class of *loosely synchronous* problems: each processor independently performs a cycle of computations followed by a communication step. The communication step acts as a synchronization point for the processors. A situation that should be avoided is that during the computational phase of a cycle one or more processors are idle, while others are still making their computations. Nor is it a good case when the communication step of the cycle is very time consuming.

For SPMD programs on a MIMD machine a good domain decomposition is characterized by two things:

- all processors should have the same amount of work (load balancing)
- the communication between the processors is minimized

Unfortunately, these are conflicting goals. Communication is minimized when all work is performed by a single processor, but then there is no load balance. On the other hand, a well-balanced decomposition will always incur a communication overhead. In most cases a trade-off must be made as to which of these goals has priority.

The problem of decomposing a data domain is a very hard optimization problem, a so-called *NP*-complete problem. Formally a problem is *NP*-complete if it can only be solved in polynomial time by non-deterministic methods. This implies that deterministic methods are not able to solve the problem in polynomial time, but require exponential time instead. In practice *NP*-complete problems are hard to solve. They are solved by a variety of heuristic methods that only give approximate answers near the optimum solution. For a domain of N data items and P requested subdomains, there are N^P possible solutions for the partitioning of the data set and we cannot afford to examine every one of them. Therefore, in general we will have to be satisfied with a good near-optimal solution. For this purpose both deterministic and non-deterministic methods exist.

A.4. Difference between Domain Decomposition and Load Balancing

In literature one often finds the terms domain decomposition and load balancing being used interchangeably. This is, however, not correct. Consider the following example: a problem with a homogeneous and regular data domain is decomposed for a parallel machine consisting of two connected workstations. Both machines execute the same program, but with different sets of data. Even when there is a perfect domain decomposition, which assigns exactly half of the domain to each processor and minimizes communication, there is not necessarily load balance. The workload on one or both machines can change dynamically, for example by the execution of other tasks that have nothing to do with our problem or due to people logging in. It is not possible or at least very difficult for a decomposition method to take into account such external factors that influence the execution pattern of individual processors.

From the example it appears that load balance is a purely dynamics phenomenon, whereas domain decomposition is not. Decomposition techniques are based on both problem and machine dependent properties. In all decomposition methods that will be discussed later, it is assumed that a parallel machine is dedicated to the problem that is to be solved. Dynamic factors that influence the execution pattern of the machine are discarded.

The formulation of domain decomposition can now be refined to: for a given problem with associated data set, an interconnectivity schema for the data items and a given parallel machine, find a partitioning of the data domain into a number of subdomains such that the workload is evenly distributed over all processors in case these are fully dedicated to the problem in question the communication between the processors is minimized. Table 1 shows the priorities of the requirements 1 and 2 for (in)homogeneous domains and (ir)regular execution patterns.

Domain	**Execution Pattern**	
	Regular	Irregular
Homogeneous	*Minimize communcation*	?
Heterogeneous	*Minimize communication + load balance*	?

Table 1: Minimization requirements depending on data domain and execution pattern.

A.5. Classes of load balancing strategies

Several combinations can be distinguished for different application types and machine architectures as shown in Figure 4. On the application side we can have a situation in which the locality of the data is preserved throughout the program (static) or a situation in which the locality of data changes (dynamic). In the former case there is no need to rebalance the system from the point of view of the application, whereas there is such a need for the latter case.

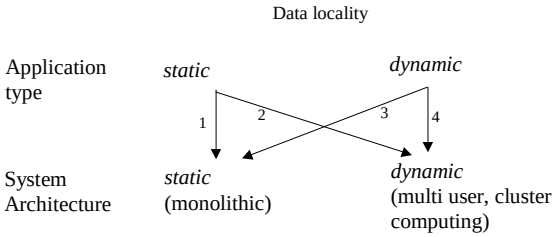

Figure 4: Combinations of application types and machine architectures.

The same distinction between static and dynamic can be made for the system upon which the application is run. It is important to note that in this system several items are included: hardware, operating system and runtime support system. In case a user has full and exclusive access to a machine's resources a system can be considered as being static. On the other hand, when a user shares these resources with other users or when the (availability of) resources change in time a system can be regarded as being dynamic.

The easiest case for domain decomposition is the situation of an application with static data locality on a static system. Examples of this case are decompositions of images from image processing applications and decompositions of database queries on monoliths. This situation is shown in the figure by the arrow 1. The load balance of the execution is only dependent on the data decomposition.

When the application's data locality is static, but the resources of the machine change (arrow 2), there is not much that can be done from the application side to prevent load imbalance. Example of this case are all applications with static data locality on a cluster of computers, where participating computers may have differently sized resources or more than one user has access to the system. Now, only the runtime support system can take action to keep the execution well balanced. An example of such a runtime support system is Dynamite.

For applications with static data locality there are two types of load balancing strategies

1. By inspection

   In this case the load balancing strategy may be determined by inspection, such as with a rectangular lattice of grid points split into smaller rectangles, so that the load balancing problem is solved before the actual program is written.
2. Static

   In this case the optimization is non-trivial, but may be done by a sequential machine before starting the parallel program, so that the load balancing problem is solved before the parallel program begins.

As soon as the data locality is no longer static it is necessary to rebalance the execution from time to time. From the application's point of view (quasi-)dynamic load balance strategies should be applied in order to prevent load imbalance. Moreover, to avoid a sequential bottleneck while making a new decomposition of the data it is also required that the decomposition is done in parallel. Examples of applications with dynamic data locality on a static system (arrow 3) are car crash simulations (changing number of contact points), Computational Fluid Dynamics (CFD) and many particle problems on a monolith.

The last case, in which both the application and the system are dynamic, requires both parallel, (quasi-)dynamic load balance strategies in combination and assistance from the runtime support system (arrow 4). Examples are car crash simulations, CFD and many particle problems on a cluster.

For applications with dynamic data locality there are also two types of load balancing strategies

1. Quasi-Dynamic

   In this case the circumstances determining the optimal balance change during

program execution, but discretely and infrequently. Because the change is discrete, the load balance problem, and hence its solution, remain the same until the next change. If these changes are infrequent enough, any savings made in the subsequent computation make up for the time spent solving the load balance problem. The difference between this and the static case is that the load balancing must be carried out in parallel in order to prevent a sequential bottleneck.

2. Dynamic
   In this case the circumstances determining the optimal balance change frequently or continuously during execution, so that the cost of the load balancing calculation after each state change should be minimized in addition to optimizing the splitting of the actual calculation. This implies that there must be a decision made every so often to decide if load balancing is necessary, and how much time to spend on it.

A.6. Cost functions

A logical question that arises is how to measure the quality of a domain decomposition. The answer to this question is given by introducing a real function H, the *cost function*, that is minimized when the total running time of the code is minimized and is reasonably simple and independent of the details of the code. For most applications the cost function is constructed as the sum of a part that minimizes load imbalance and one that minimizes communication:

$$H = H_{calc} + \mu H_{comm}, \qquad [1]$$

where H_{calc} is the part of the cost function which is minimized when each processor has equal work, H_{comm} is minimal when communication time is minimized and μ is a parameter expressing the balance between the two, where μ is proportional to f_c, the fractional communication overhead. For applications with a great deal of calculation compared to communication, like in many FEM applications, μ should be small, and vice versa. If μ is increased, the number of processors in use must be decreased until eventually the communication is so costly that the entire calculation is done on a single processor.

For FEM applications a convenient choice for H_{calc} is

$$H_{calc} = \alpha \sum_{i=1}^{p} W_i^2 \qquad [2]$$

where α is a scaling factor andW_i denotes the load of the processor with subdomain i. By this choice H_{calc} is minimized when each processor has equal work. Besides, the function H_{calc} is nicely smooth. The exact form of W_i is problem dependent. For a homogeneous domain and identical processors, we can simply take the number of elements in the subdomain. For non-homogeneous elements or different types of processors, however, we need more information. In this case, we need some abstract model of both the application and the processors in order to determine the workload of each processor for a decomposition.

In addition to this choice for H_{calc} one often sees the following choice for H_{comm} [3, 4]

$$H_{comm} = \beta \sum_{i,j} b_{i,j} \qquad [3]$$

where β is a scaling factor and $b_{i,j}$ is a boundary of both element i and element j such that i and j lie in different subdomains.

If we take the load balance and communication term as described above, we obtain the following expression for the cost function

$$H = \alpha \sum_{i=1}^{p} W_i^2 + \mu\beta \sum_{i,j} b_{i,j} \qquad [4]$$

If all the elements in the mesh are equal than we can simply take W_i to be the number of elements in subdomain *i*. The scaling factors α and β are chosen such that the optimal H_{calc} and H_{comm} have contributions of about unit size from each processor. A good choice for α and β is

$$\alpha = \frac{p^2}{n^2}, \quad \beta = \left(\frac{p}{n}\right)^{\frac{d-1}{d}}, \qquad [5]$$

where *n* is the number of elements in the mesh and *d* the dimensionality of the problem. The form for β is chosen such that we take into account that the surface area of a compact shape in *d* dimensions varies as the *d*-1 power of the size, while volume varies as the *d* power. The final form of the cost function is then

$$H = \frac{p^2}{n^2} \sum_{i=1}^{P} W_i^2 + \mu \left(\frac{p}{n}\right)^{\frac{d-1}{d}} \sum_{i,j} b_{i,j} \ . \qquad [6]$$

A.7. Other views on Domain Decomposition

In the previous section data decomposition was regarded as a combinatorial optimization problem. A cost function was introduced of which the global minimum represented the configuration that resulted in the best data decomposition in terms of load balance and smallest communication overhead. Domain decomposition can also be given a physical interpretation. The elements in the domain to be distributed can be thought of as particles moving around in the discrete space formed by the processors (subdomains). This physical system is controlled by the energy function or *Hamiltonian* given in equation [6]. The two terms in the Hamiltonian have simple physical meanings illustrated in Figure 5.

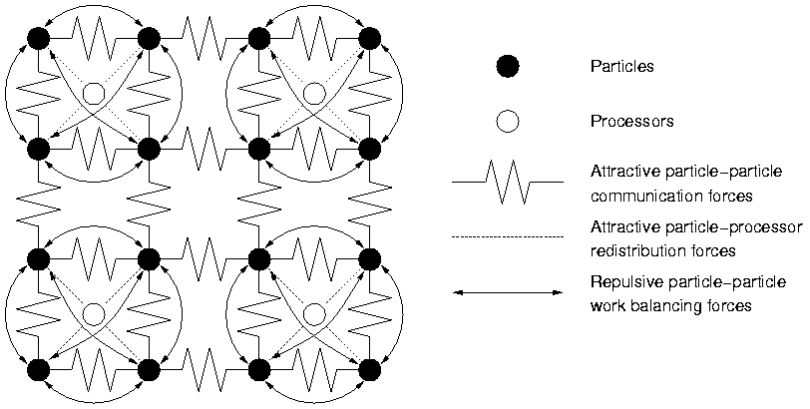

Figure 5: Sixteen data points optimally distributed on four processors illustrating the physical analogy of domain decomposition.

The decomposition problem now becomes one of finding the equilibrium state of a system of particles with a "conflict" between short-range repulsive and long-range attractive forces.

Yet another way to view the domain decomposition problem is to consider it as a generalization of the graph bisection problem, which is defined as follows. Given an

undirected graph G, with the set of vertices V and the set of edges E, $G = G(V, E)$, partition $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, such that

$$|E| = |\{e \mid e \in E; e = (v_1, v_2); v_1 \in V_1; v_2 \in V_2\}|$$

is minimized ($|\ .\ |$ being the number of elements in the set), subject to some constraint on the decomposition. For example, if all vertices have equal weights (same amount of computational work) the constraint can be $|V_1| = |V_2|$, if $n = |V|$ is even and $|V_1| = |V_2| - 1$, if n is odd.

The assumption that the underlying problem can be expressed as an undirected graph is in no way restrictive. For example, the unstructured mesh on the left in Figure 6 can easily be "changed" into its *dual graph*. The elements of the original mesh are the vertices of the dual graph and two vertices are considered to be adjacent vertices of the dual graph if and only if they share an edge in the original mesh. The dual graph of the unstructured mesh is shown on the right in Figure 6. A graph partitioning of the dual graph will thus yield an assignment of elements to processors (subdomains).

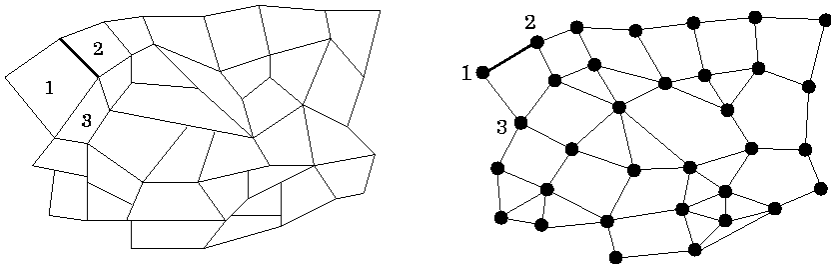

Figure 6: An unstructured mesh consisting of quadrilateral elements (left) and its dual representation (right)

In a similar way, most general decomposition problems can be transformed to a graph partitioning problem. Thus, a decomposition method that partitions the data domain in an efficient way will be quite generally applicable.

B. Decomposition Methods

B.1. Scattered Decomposition

The scattered decomposition is an effective solution strategy for the decomposition problem posed by a large class of irregular problems [5]. The scattered decomposition has also been successfully applied to the parallel solution of irregular finite element problems.

The concept of scattered decomposition is illustrated in Figure 7. As in the usual rectangular decomposition, we map the parallel machine onto a two-dimensional mesh. Instead of simply superimposing this mesh on the data domain, however, we subdivide the data domain into many rectangular regions and subsequently distribute these subregions or templates among the processors of the parallel machine. Each processing node ends up being responsible for many small mutually disjoint regions, which we will refer to as *granules*. Figure 7 shows, for example the granules for which processor 0 would be responsible in a typical scattered decomposition.

The scattered decomposition accomplishes load balancing for most domain configurations. Each processor has a piece of action across the entire space of the data domain and will end up with roughly the same work load. This spatial averaging approach to load balancing works best as the granule size decreases, although the communication overhead increases in this limit.

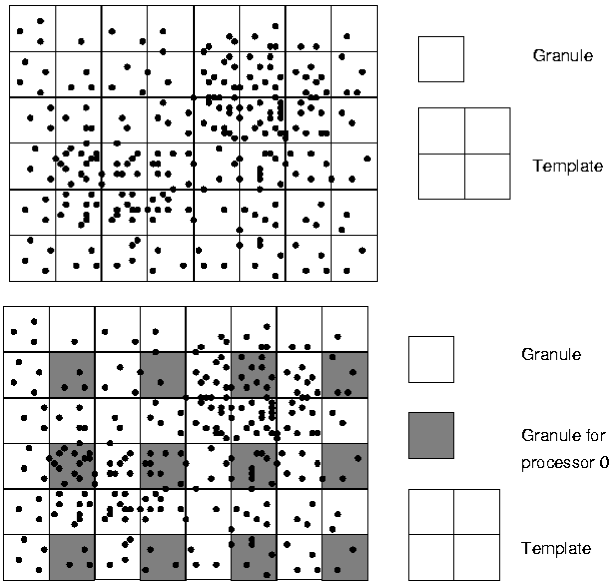

Figure 7: Scattered decomposition for a 2 ×2 array of processors laid over the domain twelve time (top) and .the granules for which processor 0 is responsible (bottom).

Another appealing feature of the scattered decomposition is that the required communication pattern between the nodes retains a rectangular topology. Since this regular topology is often directly reflected in the machine hardware and the associated low-level message-passing system communications will generally be fast in this method. On the other hand, as mentioned above, small granule sizes yield a larger communication perimeter relative to the contained area of useful work (i.e. the surface-to- volume ratio is increased). As a result, it is not immediately clear whether scattered decomposition leads to higher or lower absolute communication overhead costs. The answer to this question is dependent upon the choices of both machine and algorithm.

For any implementation of a scattered decomposition care must be taken not to impose excessive synchronization conditions during update of a template. If synchronization-imposed communication occurs after the update of each granule of the template (for instance for implicit updating schemes such as SOR), the program is self-defeating, in that at any given time different processors may be working on granules with very different loads. In fact, the result will be a worse load imbalance than without the scattered decomposition! In order to correctly achieve load balancing, we need to allow processors to move immediately to the next granule without waiting, so that the load balance is averaged over the whole problem. The processors run free without communication until reaching the last template. As a result, the communications between all of the granules throughout the data domain must be buffered and stored as the processors sweep through the data domain, and only at the very end of an update cycle the communication actually is performed. We should be aware that the above remarks are only valid for explicit updating schemes.

In the scattered decomposition algorithm contention may arise. To avoid this, a possible solution is to have processors skip over some of the templates. The update row now proceeds as a sequence of "mini-cycles". As an example, in the first mini-cycle, processor 0 would update its granule in template 0, 8, etc., processor 1 its granule in template 1, 9, and so on. The pattern is chosen so as to load balance and keep the processors far apart during any mini- cycle. As we proceed through the mini-cycles, the patterns change and are chosen so as to eventually update all the granules.

It seems that scattered decomposition is the ideal method for obtaining a well balanced domain decomposition. In any practical application, however, the problem-dependent and machine-dependent factors influencing the solution of irregular problems may tilt the balance in favour of other approaches.

B.2. Simulated Annealing

Simulated annealing is a powerful and general algorithm for solving optimization problems in which the problem to be optimized can be represented as a function that has many variables and many local minima. The results obtained from this algorithm are very good but the time required to find the optimal result can be very long. Hence it is used to approximate the global minimum as closely as computational resources permit. For problems that are not well understood, it may not be possible to find an algorithm that can take advantage of problem-specific properties. For such problems, simulated annealing has been very successful, even with long computation time. Simulated annealing has been widely used to solve problems like VLSI placement and load balancing.

Simulated annealing is the mathematical analogue of the physical process of annealing. The energy of the system at any state S_i is $E(S_i)$ and is determined by a cost function used to assign a value to that state. Temperature is used as a control parameter to guide the system to a low cost (low energy) state. The value of temperature determines whether a perturbation that increases the energy is to be accepted. The simulated annealing is started with temperature Tequal to the initial temperatureT_0 and some initial state S_0. The system is perturbed to get a new state S_n. The change in energy ΔE is calculated. If the energy is decreased, the perturbation is accepted. Otherwise the perturbation is accepted with a probability $\exp(-\Delta E/T)$. At higher temperatures, this probability is large and most of the moves which increase the energy are accepted. As temperature falls, only a small number of perturbations are allowed. At each temperature, the algorithm reaches equilibrium and then the temperature is reduced. The system is frozen when the system will not improve despite further reductions in temperature. The algorithm for simulated annealing is shown below.

```
Choose an initial configuration
Choose an initial temperature T > 0
while T > T_min do
   while(no equilibrium) do
      perturbate system to obtain a new state
      ΔE = cost(new state) - cost(old state)
      if ΔE < 0 then
         accept new state
      else
         accept new state with probability exp(-ΔE/T)
      end if
   end while
   decrease temperature T
end while
```

In order to implement simulated annealing for a specific purpose, the following items have to be defined:

1. a state
   The precise definition of a state is problem-dependent. For example, if we want to partition an unstructured mesh into p subdomains, a state can be defined as a function that assigns one of the p subdomains to each of the elements in the mesh.
2. a perturbation strategy
   An efficient strategy should be provided that transforms a state into another state. For the unstructured mesh example, one can think of moving a boundary element (an element that has at least one neighbour that lies in a different subdomain) to a neighbouring subdomain.
3. cost (energy) function
   A function should be defined that gives the ‘‘quality” of a state. For example, if we want to minimize communication overhead for the unstructured mesh example, the cost function can be defined as the total number of boundary edges for a distribution of the elements over the subdomains.

4. a cooling schedule
   This includes
   - initial temperature T_0
     The initial temperature should be chosen such that all possible states can be reached (ergodicity). In general, this implies that the initial temperature should be sufficient high.
   - an equilibrium state
     For a given temperature, it should be defined when there is no further decrease of the cost function when perturbating the system.
   - a cooling rate.
     A schedule should be provided that gradually lowers the temperature parameter T to zero asymptotically. Two common cooling schedules are the logarithmic schedule, defined as

     $$T_k = \frac{T_0 \log(k_0)}{\log(k + k_0)}$$

     where T_k is the temperature at iteration k and $k_0 > 1$ is some fixed integer, and the geometric schedule, defined as

     $$T_k = \alpha^k T_0$$

     where $0 < \alpha < 1$ is a user-defined parameter.
   - a stopping criterion.
     The SA algorithm stops when no further significant improvement has been observed over a number of iterations. The algorithm often terminates as soon as the cost of the final state is ‘‘satisfactory”.

An Example

In this section we will demonstrate how to apply SA to domain decomposition [1]. The task is to partition an unstructured mesh of N elements over P processors. We assume that the cost of sending a message from a processor to another processor is dependent on the physical distance between the processors. Further, we assume that the computational work associated with each element is the same for all elements. This must be taken into account when the cost function is being constructed. The items described above, necessary to implement the SA algorithm can be as follows

1. state definition
   A state is a distribution of the N elements over the P processors or, more formally, a function that assigns to each of the elements in the mesh one of the P subdomains.
2. a perturbation strategy
   A simple perturbation technique consists of moving an element from one processor to another processor.
3. a cost (energy)
   The cost function can be defined as

   $$H = \sum_{i=1}^{P} W_i^2 + \mu \sum_{i,j} C_{ij} D_{ij}$$

   with W_i the computational load on processor i, C_{ij} the amount of communication between processor i and processor j, D_{ij} the distance between processors i and j and μ the weight factor of the computational part of the cost function with respect to the communication part of the cost function. For the applications chosen, all communications were of equal length and the factor C_{ij} was set to one. The weight factor μ was also set to one.
4. the cooling schedule
   - The initial temperature was experimentally chosen to be 4 (50\% of the steps are accepted).

- An equilibrium state has been reached when either the number of moves accepted at a specific temperature exceeds 10% of the number of elements *N*, or the number of attempts to move elements equals *N*.
- The stopping criterion was simply set to $T \leq 0.1$.
- The cooling rate should be chosen small. As the initial and the stopping criterion temperatures do not differ to a great extent, the decrease of the temperature should be small to reach the global minimum of the cost function.

The SA algorithm now becomes

```
Randomly distribute the elements over the processors
T = 4
while (T > 0.1) do
   for (all elements) do
      if (number of accepted moves  > 0.1×N) then
         break inner loop
      Determine (randomly) a destination for the element
      ΔE = cost(new state) - cost(old state)
      if (ΔE < 0) then
         accept the move of the element
      else
         accept the move with probability exp(-ΔE/T)
      end if
   end do
   temperature T
end while
```

A typical behavior of the SA algorithm is shown in Figure 8. For three different temperatures --- the start, end and an intermediate temperature --- the cost function is depicted as a function of the number of iterations. In all cases the system searches for an equilibrium state. It can be seen that for the start temperature there is a lot of fluctuation. As the temperature is high, many moves are accepted that do not decrease the cost function.

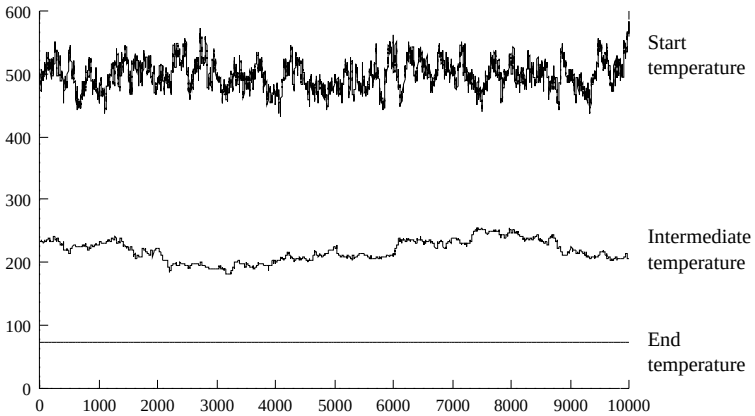

Figure 8: Typical behavior of the SA algorithm for different temperatures.

For the intermediate temperature the fluctuation is far less, since “bad moves” are accepted. The end temperature does not or hardly allow for moves that increase the cost function. When the optimal decomposition has not been found but a local one instead, it is very difficult to escape for very low temperatures.

In Figure 9 the result is shown for a mesh of 735 elements distributed over 16 processors. The average time to complete the domain decomposition was about 63 seconds. All moves of the elements were determined at random.

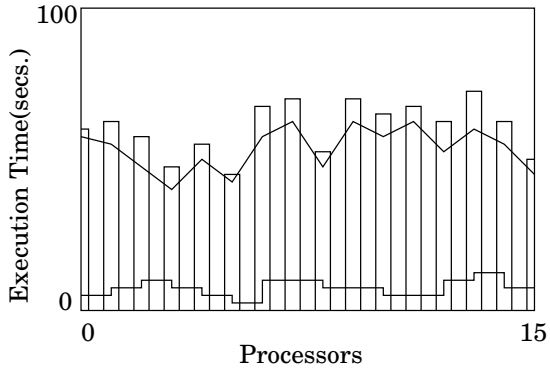

Figure 9: Computation (line), communication (steps) and total loads distribution (bars) vs processors for random moves.

B.3. Genetic Algorithms

The previous section showed how to use SA to solve the domain decomposition problem. There is also another class of natural solvers that can be employed for domain decomposition: genetic algorithms [6]. For a given mesh (or arbitrary interconnectivity scheme), these algorithms can be used as follows. First we have to define how the chromosomes look. A chromosome can be encoded as a string of integers, where each integer refers to a subdomain (or processor) and the *n*-th position in the string corresponds to element *n* in the mesh. For example, for a mesh of 16 elements that has to be divided into 4 subdomains a chromosome has the form

(2 0 3 1 0 2 2 3 0 1 2 3 1 2 1 3).

For this chromosome element 0 is assigned to subdomain 2, element 1 to subdomain 0, element 2 to subdomain 3, etc.

The fitness of a chromosome is determined by a cost function, to be more precise by the reciprocal of the cost function. The cost function given in the example in the section on SA can also be used here. It should be noted that a good cost function takes into account both the connectivities in the mesh and the processor topology. However, as the populations in a genetic algorithm will be large in general, the evaluation of the cost function for all the chromosomes can be very expensive. Therefore, one is often satisfied with a cost function that is less accurate with respect to the connectivity and/or processor topology, but of which the evaluation is less expensive. An example of such a simpler cost function is the one used in the SA example.

An initial population of chromosomes can be generated randomly. Work by Mansour and Fox [7] has shown that this is a reasonable heuristic to initiate the genetic optimization for the type of problems that we are addressing.

Generally a genetic algorithm (GA) consists of several operations that transform a population of chromosomes into a new one. The reproduction operation basically is the operation that is responsible for the tendency that consecutive populations that are generated evolve in the direction of higher average fitness. The other operations are introduced to create populations with high diversity.

GA is a blind search technique. Very often the implementation of an efficient GA is confronted with the problem of premature convergence into local optima. On the other hand an implementation that is not specifically efficient may take very long times to reach a good suboptimal solution.

Incorporating problem specific knowledge in the blind GA search to direct it to the fruitful regions in the search space is a possible technique to get more efficient GA's. So-called hybrid techniques can take into account domain specific knowledge in their search. The hybrid genetic algorithm for our purposes has to satisfy the following conditions: The likelihood of premature convergence has to be minimized, the search efficiency must be

high while computational costs are kept as low as possible, and problem specific knowledge has to be utilized.

Several operators can be distinguished that are typical for the genetic algorithm kernel:

- Reproduction
- Crossover
- Mutation and Inversion

Reproduction of a population of size of *N* individuals produces a new population of size *N*. The probability that an individual that appeared in the old population will be reproduced and put in the new one is higher when the fitness of the individual is higher. This can be done e.g. according to the so-called biased "wheel of fortune" trick [8] or "ranking" [7]. For our purpose, ranking is a good choice for a reproduction scheme, as it has shown to be able to control premature convergence and to be computationally relatively cheap compared to other methods.

The crossover operation manipulates the population generated in the reproduction phase to obtain a new population. Randomly two chromosomes are picked out of the population. Then two new chromosomes are generated using a crossover operation.

Mutation allows a small fraction of a chromosome to change spontaneously. These operations allow the genetic search to spontaneously direct the search to a region in the space of solutions that otherwise would not be reached. In the domain decomposition problem formation of partitions of adjacent elements into the same subdomain is highly desirable. Mutation therefore is restricted to boundary elements, i.e. elements that have at least one of its neighboring elements residing in another subdomain.

Inversion within a chromosome is also allowed with some inversion probability. The chromosome is considered as a ring. A contiguous section of this ring gets inverted. A simple hill climbing technique to direct the blind GA search into the more profitable regions of the search space can be useful to add to a genetic algorithm. In the hill climbing boundary elements are considered one at a time. A boundary element is transferred from the subdomain in which it currently resides to a subdomain on which a neighboring element resides if the fitness of the chromosome increases or stays the same. In this way locality within the task graph can be utilized to approach optimal solutions faster.

Figure 10 shows the fitness function (the reciprocal of the cost as a function of the number of iterations for a domain decomposition problem for a mesh consisting of 640 quadrilateral elements that is divided into eight subdomains. The population contained 100 chromosomes. From the figure it can easily be seen that the fitness evolves in the direction of higher fitness due to the reproduction procedure.

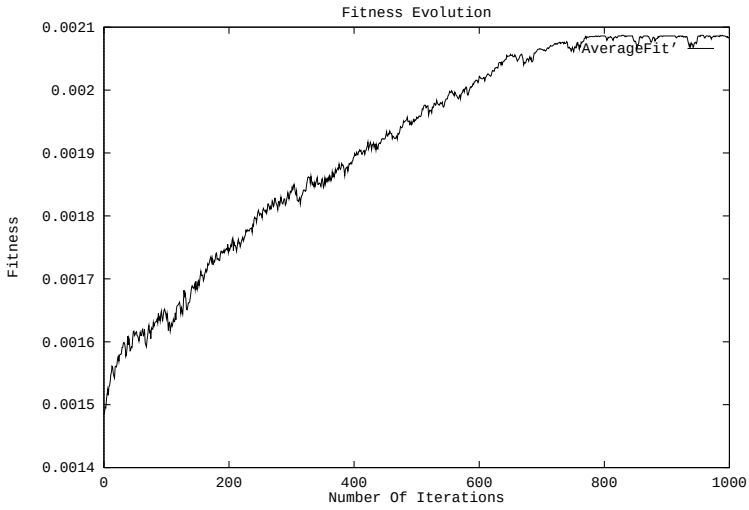

Figure 10: Fitness evolution for a domain decomposition of a mesh of 640 elements into eight subdomains by a genetic algorithm.

C. Recursive Bisection Methods

C.1. Introduction

Bisection methods constitute an important class of decomposition techniques. The bisection methods that will be discussed here are all recursive methods, i.e. the computational domain is subdivided by some strategy into two subdomains, and then the same strategy is applied to the subdomains recursively [9]. In this way a partition into $p = 2^k$ subdomains is obtained after carrying out k of these recursive partitioning steps. The algorithms that are considered here thus only differ by the partition strategy of a single domain into two subdomains.

The general idea behind the partitioning algorithms is to use an optimal strategy to divide a domain into two subdomains. Then, by recursively applying the optimal splitting, the final result should be also optimal or near optimal. This is not true in general, but for well shaped meshes (as in most cases), recursive bisection methods will almost always come close to the optimal solution. [Simon, 1997 #14]

The bisection methods that will be discussed here are:

- Recursive Coordinate Bisection (RCB)
- Orthogonal Recursive Bisection (ORB)
- Recursive Graph Bisection (RGB)
- Recursive Spectral Bisection (RSB)

C.2. Recursive Coordinate Bisection

RCB is conceptually the easiest algorithm of the bisection methods that will be discussed. It is based on the assumption that along with the set of vertices $V = (v_1,v_2,\ldots,v_n)$, there are also two- or three-dimensional coordinates available for the vertices. For each $v_i \in V$ we thus have an associated tuple $v_i = (x_i, y_i)$ or triple $v_i = (x_i, y_i, z_i)$ depending on whether we have a two- or three-dimensional model. A simple bisection strategy for the domain is then to determine the coordinate direction of longest expansion of the domain. Without loss of generality, assume that this is the x-direction. Then all vertices are sorted according to their x-coordinate. Half of the vertices with small x-coordinates are assigned to one domain, the other half with the large x-coordinates are assigned to the second subdomain. Figure 11 illustrates a single coordinate bisection step with the x-coordinate for a simple domain. The triangles in the figure denote the set of vertices V and the coordinates of a vertex are determined by the position of the black dot in a triangle, e.g.

the center of mass of a triangle. The next step in the bisection process would be the bisection of each of the two subdomains, again with respect to the *x*-coordinate.

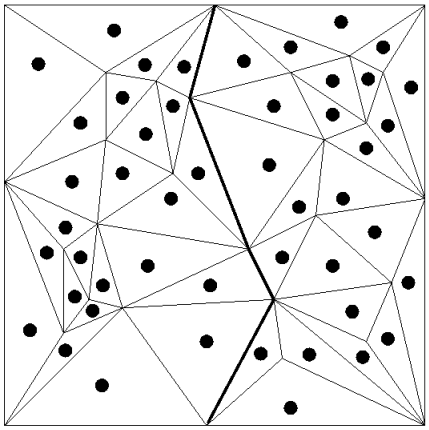

Figure 11: A single step of coordinate bisection for a simple mesh.

C.3. Orthogonal Recursive Bisection

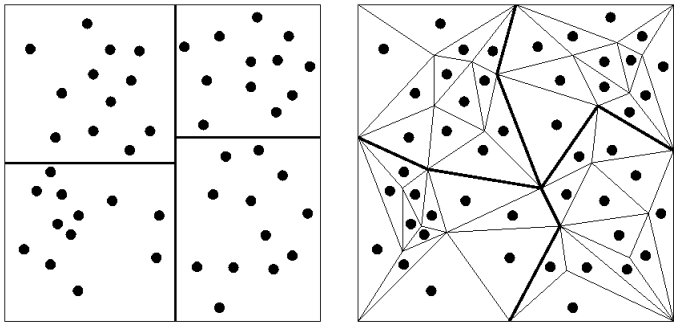

Figure 12: Two steps of ORB for a simple mesh. The domain is split vertically and then each half is split horizontally (left). The resulting subdomains are shown in the right figure.

A method that is very similar to RCB is Orthogonal Recursive Bisection (ORB). Like RCB, it is assumed that for each vertex a coordinate tuple or triple is available. Again, vertices are sorted and assigned to one of the subdomains based on the value of one of the coordinates. The difference between the two methods is that for the ORB method the bisection direction is alternating the *x*- and *y*-coordinate (or *x*-, *y*- and *z*-coordinate for the three-dimensional case). Thus, first the domain is split into two subdomains by a vertical cut (using the *x*-coordinate); in the next step the two subdomains are split into two parts by a horizontal cut (using the *y*-coordinate); then the *x*-coordinate is used again for the splitting of the "quarte" subdomains etc. Figure 12 shows two steps of the ORB method for the same domain as in the example for the RCB method.

C.4. Recursive Graph Bisection

The weakness of RCB and ORB is that both algorithms do not take advantage of the connectivity information given by the graph. After all, the main goal for most unstructured meshes is to minimize the number of graph edges, which are connecting different subdomains. Thus, instead of using the Euclidean distance between vertex coordinates, one should rather consider the graph distance between vertices given by $d(v_i,v_j)$ = |shortest path connecting v_i and v_j|. The left figure of Figure 13 shows the dual graph of the simple mesh that we have seen before in the RCB and ORB method. Each triangular element has been replaced by a vertex and two vertices are connected by an edge if and only if the corresponding elements in the mesh share a boundary. From this

figure it can easily be seen that the graph distance between the vertices *A* and *B* is four. The vertices *A* and *C* have a maximum graph distance in this mesh, which is eleven. Note that there may be several vertices that have a maximum graph distance. With this change in metric one can define a new partitioning algorithm, which is called Recursive Graph Bisection.

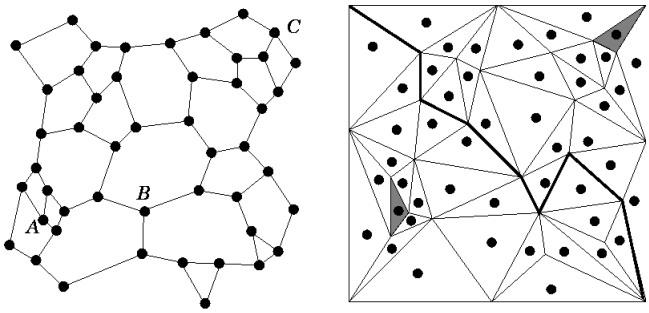

Figure 13: A single step of RGB for a simple mesh. The left figure shows the dual graph of the mesh with the vertices A *and* C *that are farthest apart. In the right figure, the elements closer to* C *than to* A *are assigned to one subdomain and the other elements to the other one.*

In the RGB algorithm first two vertices of maximal or near maximal distance in the graph are determined. Then all other vertices are sorted in order of increasing distance from one of the extremal vertices. Finally vertices are assigned to two subdomains according to the graph distance. The only difficulty is the determination of the diameter (or at least of a pseudo- diameter) of the graph. For the simple mesh above it is still possible to determine the vertices that have maximum graph distance. For large problems on the other hand, the time required to find these vertices will be too time consuming.

In practice, some very good heuristic algorithms exist for finding the vertices that have (near) maximal distance. These algorithms are also quite well known in the structures community. since they can also be used for reducing the storage requirements of sparse matrices in envelope or skyline storage format. Here we will discuss briefly one of those heuristic algorithms, the reverse Cuthill-McKee (RCM) algorithm. The RCM algorithm first finds two pseudo-peripheral vertices in the graph (i.e. vertices which have a very large distance, but which are not necessarily the pair of vertices with maximum distance). Then, starting from one of the vertices, the root vertex, a so-called level structure is constructed. The level structure is a convenient way of organizing the vertices in the graph in sets of increasing distances from the root. Hence the level structure delivered by the RCM algorithm forms the basis for the recursive graph bisection algorithm. Half of the vertices, the ones that lie closer to the root are assigned to one subdomain, the remaining vertices to the other subdomain. If we start out with a connected graph then by construction it is guaranteed that at least one of the two subdomains (the one including the root) is connected.

C.5. Recursive Spectral Bisection

The last algorithm that is discussed here is of quite a different nature and on first glance considerably less intuitive. The RSB algorithm is derived from a graph bisection strategy developed by *Pothen et al.* [10], which is based on the computation of a specific eigenvector of the Laplacian matrix of the graph $G = G(V, E)$. For unstructured meshes G is the dual graph of a mesh, as described previously in A.7. The Laplacian matrix $L(G)$ is defined by

$$L_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ -\deg(v_i) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} . \qquad [7]$$

In this definition the degree of a vertex, deg(v_i), is the number of edges in the graph that have vertex v_i as an endpoint. It is easily seen that $L(G)=-D + A$, where A is the adjacency matrix of the graph, and D the diagonal matrix of vertex degrees. Traditionally, spectral properties of the adjacency matrix have been investigated in graph theory. In 1988, however, Mohar has gathered convincing evidence that the Laplacian matrix is the more natural object for the study of spectral properties of graphs. One of the reasons is that $L(G)$ is closely related to the Laplacian operator. Specifically, if we consider the standard discrete five point Laplacian on a rectangular grid, then the discrete Laplacian and the Laplacian matrix coincide, if Neumann boundary conditions are imposed. Thus one can consider the Laplacian matrix as a generalization of the discrete Laplacian operator for general graphs.

The Laplacian matrix has a number of intriguing properties, which are just listed here. For details and proofs we refer to the list of references. First note that the bilinear form associated with the Laplacian matrix can be written as

$$x^t Lx = - \sum_{(v,w)\in E} (x_v - x_w)^2 . \qquad [8]$$

In this form all edges are counted once. From this it follows that $L(G)$ is negative semi-definite. From the definition of L it also follows that the largest eigenvector λ_1 is zero, and that the associated eigenvector is **e**, the vector of all ones. This is simply a consequence of the particular choice of diagonal elements in $L(G)$. If G is connected then λ_2, the second largest eigenvalue, is negative. The magnitude of λ_2 is a measure of connectivity of the graph or its expansion.

What is of interest here is the eigenvector $\mathbf{x}_2$ associated with λ_2. It turns out that this eigenvector gives some directional information on the graph. If the components of $\mathbf{x}_2$ are associated with the corresponding vertices of the graph, they yield a weighting for the vertices. Differences in this weight give distance information about the vertices of the graph. Sorting the vertices according to this weight provides then another way of partitioning the graph.

The special properties of $\mathbf{x}_2$ have been investigated by Fiedler. His work gives most of the theoretical justification of the uses of the second eigenvector of the Laplacian matrix for the partitioning algorithm. Hence this vector is called the *Fiedler vector*.

The actual challenge of the RSB algorithm is the effective computation of the Fiedler vector. In many scientific and engineering problems, the number of mesh elements is very large. As a consequence, an eigenvalue and eigenvector of a very large linear system have to be calculated, which is by no means trivial. A method that can be used for this purpose is an algorithm that is akin to the Conjugate Gradient method and is called the Lanczos algorithm, an example of a so called *iterative relaxation* method. The algorithm is depicted below for a symmetric $n \times n$ matrix A.

```
Compute a sequence of Lanczos vectors v1,v2,… as follows:
Choose an arbitrary v1, |v1| = 1
u1 = Av1
for j = 1,2, … do
    αφ = uj.vj
    rj = uj - αjvj
    βφ = |rj|
    vj+1 = rj/βj
    uj+1 = Avj+1 - βjvj
done
```

The equations in the Lanczos algorithm can also be written down in condensed matrix form:

$$AV_j - V_j T_j = \beta_j v_{j+1} e_j^T, \qquad [9]$$

where $V_j = (v_1, v_2, v_3, \ldots, v_j)$, $(e_j)^T = (0, 0, 0, \ldots, 1)$ and

$$T_j = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \ddots & \ddots & \\ & & \ddots & \alpha_{j-1} & \beta_{j-1} \\ & & & \beta_{j-1} & \alpha_j \end{pmatrix}.$$

The algorithm terminates if $\beta_j = 0$, which will happen for some $j \leq n$ in exact arithmetic. The eigenvalues of the symmetric tridiagonal matrix T_j, also called the *Ritz values*, are approximations to eigenvalues of matrix A. For increasing j these values will become better and better approximations of the eigenvalues of A. In practice, one often needs only $O(\sqrt{n})$ iterations to obtain an approximation of the required eigenvalue of that is accurate enough.

The advantage of the Lanczos algorithm is that it enables us to extract just a single eigenvalue from the matrix. Further, it reduces the problem of finding an eigenvalue of a general symmetric matrix A to the problem of finding an eigenvalue of a symmetric, tridiagonal matrix, which is numerically not very difficult. By using rational function approximations it is possible to compute the largest eigenvalue of T_j, and thus of$A$, in an elegant and fast way [11]. Finally, by using the Lanczos algorithm it is not necessary to manipulate the Laplacian matrix $L(G)$. All that is needed are matrix vector multiplications with $L(G)$.

C.6. A comparison

In the previous sections we have described four recursive bisection methods: RCB, ORB, RGB and RSB. Figure 14-Figure 16 show the results of decomposing an unstructured mesh for a four component airfoil by RCB, RGB and RSB respectively. The problem is to model the turbulence of air around the wing of an aircraft. The unstructured grid that is used for the model consists of triangular elements and has 11451 elements, 6019 vertices, 17473 edges and four bodies. The three above mentioned decomposition methods do not act directly on this grid, but on the dual graph instead. This graph has 11451 vertices, 16880 edges (interior edges in the original grid) and also four bodies. In all figures the dual graph is shown for a decomposition into eight subdomains.

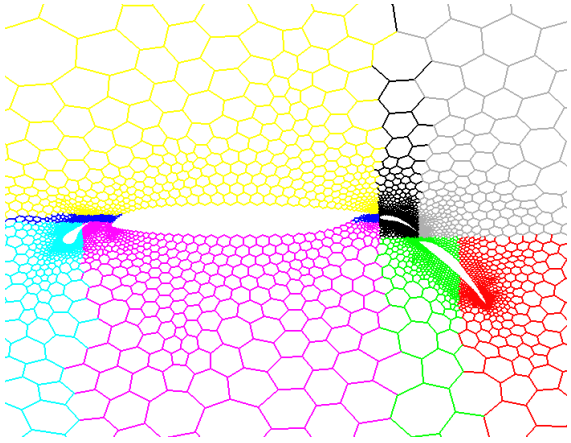

Figure 14: Four element airfoil partitioned into eight subdomains by RCB.

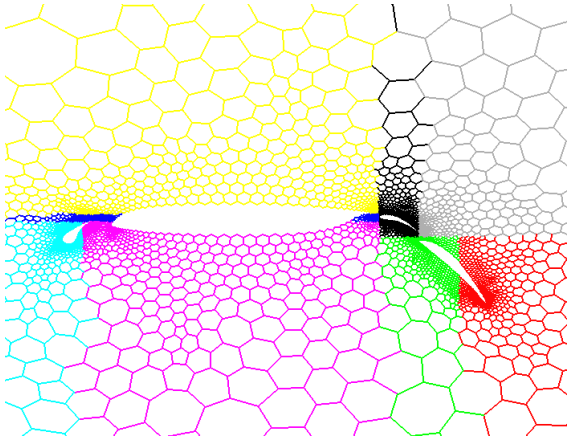

Figure 15: Four element airfoil partitioned into eight subdomains by RGB.

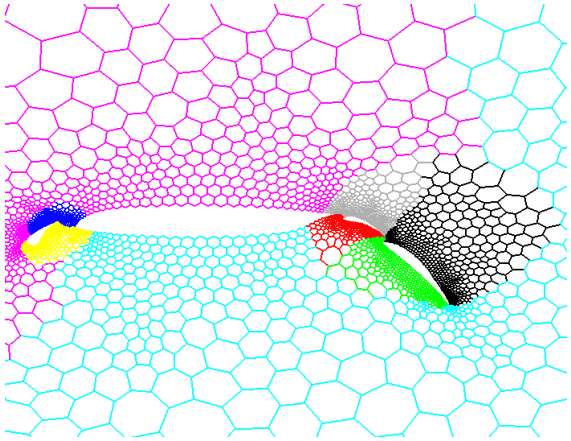

Figure 16: Four element airfoil partitioned into eight subdomains by RSB

The decomposition with RCB, as shown in Figure 14, gives long and skinny subdomains as a result of the very fine grid near to the airfoil. The domains are also disconnected and have a few isolated outlying vertices. This very undesirable property of the partition is a result of the very uneven spacing of the grid points. Also, connectivity information does not enter the RCB algorithm at all. Hence, one should not expect (in general) that the algorithm creates connected subdomains.

When RGB is used for the decomposition, the maximum distance in the graph is apparently from a point in the very fine mesh near the front of the large airfoil (red domain) to somewhere in the mesh between the third and fourth airfoil (white domain). Compared to the RCB the domains are better connected and somewhat more compact. However, the dark blue domain has a very long boundary, winding around the back of the airfoil configuration, as well as a small disconnected component at the trailing edge of the first airfoil. Both phenomena are not desirable. From a qualitative point of view RGB appears to be better than RCB, since the domains maintain better connectivities.

In Figure 16 the results for the RSB method are displayed. It can be seen that the subdomains obtained from RSB are both connected, although there is no theoretical guarantee for it, and nicely compact.

A quantitative comparison of the three methods is shown in Table 2. In this table the number of edges in the *cut-set* is listed of the dual graph of the airfoil problem. The cut-set consists of all edges in the graph of which the vertices lie in different subdomains. In the original grid this relates to triangles that share a boundary, but lie in different subdomains. From this table it can be seen that RSB minimizes the number of edges in the cut-set while RGB gives the worst results.

Partitions	RCB	RGB	RSB
2	118	175	91
4	296	436	208
8	529	681	299
16	863	950	462
32	1193	1334	743
64	1653	1878	1154
128	2218	2529	1763

Table 2: Number of edges in the cut-set for the airfoil problem (total number of edges is 16880).

D. Measuring Results

In section A.5 it was suggested that the quality of a decomposition can be measured by a cost function. For a decomposition method like Simulated Annealing this is evident as the cost function is in this case the function that has to be minimized. Other decomposition methods, on the other hand, do not use a cost function explicitly. The quality of such methods should be measured with respect to other criteria. We shall discuss a number of these criteria on the basis of an example. This example was taken from Reference [12], chapter 11.

We consider a two-dimensional Laplace equation with Dirichlet boundary conditions on the domain shown in Figure 17. The boundary conditions are defined as follows. The square outer boundary has voltage linearly increasing vertically from -1.2 to +1.2, the lightly shaded S-shaped internal boundary has voltage +1 and the dark shaded hook-shaped internal boundary has voltage -1. Contour lines of the solution are also shown in the figure, with contour interval 0.08.

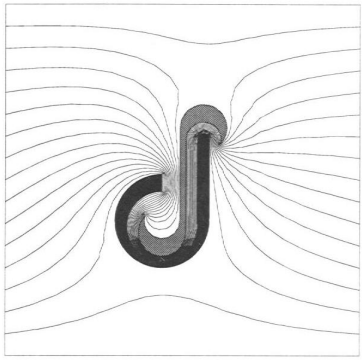

Figure 17: Solution of the Laplace equation used to test decomposition methods.

The Laplace equation is solved by *solution-adaptive* refinement. This means that we start with an initially coarse mesh as shown on the left in Figure 18. The equation is then solved for this mesh by Jacobi iteration and the mesh is refined based on the solution obtained so far. Next the refined mesh is decomposed by a decomposition method and the whole sequence — solve, refine, decompose — is repeated. In our test example we will repeat this sequence seven times. The mesh has then been refined from 280 elements initially to 5772 elements in the last step. The decision of which part of the mesh is to be refined is not arbitrary. The refinement criterion that will be used here is based on the magnitude of the gradient of the solution, so that the most heavily refined part of the domain is that between the S-shaped and hook-shaped boundaries where the contour lines are closest together. At each refinement, the criterion is calculated for each element of the mesh, and a value is found such that a given proportion of the elements are to be refined,

and those with higher values than this are refined loosely synchronously. In our case, we refined 54% of the elements of the mesh at each stage.

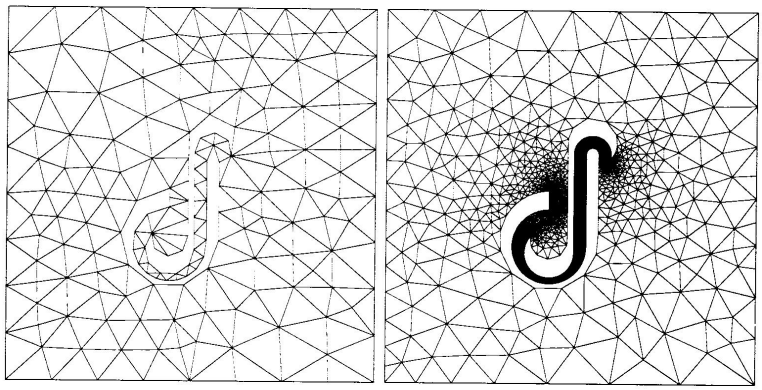

Figure 18: Initial and final meshes for the decomposition tests. The initial mesh has 280 elements roughly uniformly distributed over the square. The final mesh has 5772 elements and is dominated by the highly refined S-shaped region.

It should be noted that the choice of refinement criterion is not particularly to improve the accuracy of the solution, but to test the decomposition methods as the mesh distribution changes. The initial mesh is essentially a square covered in mesh of roughly uniform density, and the final mesh is dominated by the long, thin S-shaped region between the internal boundaries, so the mesh changes from two-dimensional to almost one-dimensional.

For the decomposition three different decomposition methods are used: SA, ORB and a variant of the RSB algorithm called Eigenvalue Recursive Bisection (ERB). In fact, two versions of SA were used, the difference being a factor of ten in cooling rate and a different starting temperature. The runs were performed on 16 nodes of an nCUBE/10 parallel system.

Another difference with the SA as discussed in a previous section is that for the tests a parallel version of the algorithm is used which is called *collisional* simulated annealing. In this version each processor has its own SA running. At certain points each processor makes a proposal for changes to the state of the system. Then, all processors evaluate the Metropolis criterion simultaneously and make those changes that are accepted. A possible consequence of this parallel variant is that small improvements are made per processor, but a worsening is obtained for the total system.

The parameters for the two versions of the collisional SA were as follows:

- the starting temperature for the run labeled SA1 was 0.2 and for SA2 1.0. In the former case, no large fluctuations of the decomposition is allowed, whereas for SA2 it is allowed and the system is heated to randomness possibly losing all information of previous configurations.
- the interface importance, μ (see section A.5), was set to 0.1, which is large enough to make communication important in the cost function, but small enough that all processors will get their share of elements.
- the curves labeled SA1 correspond to cooling to zero temperature in 500 stages, those labeled SA2 to cooling in 5000 stages.
- each stage consisted of finding either one successful change or 200 unsuccessful changes before communicating, and thus getting the correct global picture.

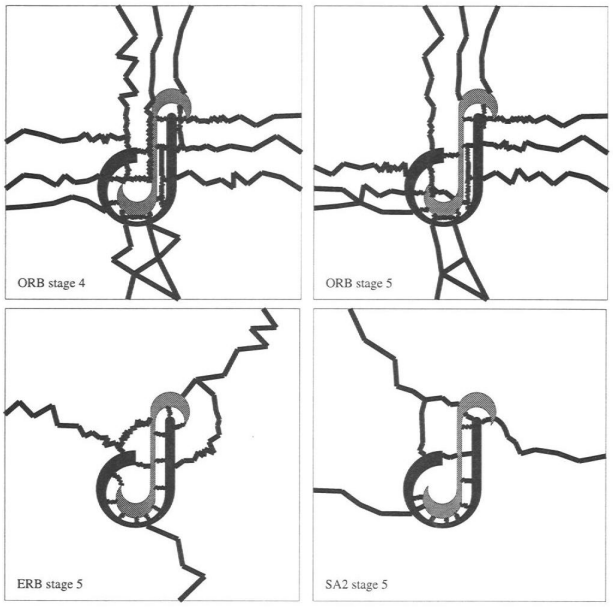

Figure 19: Subdomain boundaries for the decomposition methods. Top, ORB at the fourth and fifth stages; lower left, ERB at the fifth stage; lower right, SA2 at the fifth stage.

In Figure 19 the divisions are shown between processor domains for three methods at the fifth stage of the refinement, with 2393 elements in the mesh. The figure also shows the divisions for the ORB method at the fourth stage. Note the unfortunate processor division to the left of the S-shaped boundary which is absent at the fifth stage. The measurements of the code that solve the Laplace equation of our example can be divided into three categories:

- machine-independent measurements,
- machine dependent measurements,
- measurements for dynamic load balancing.

D.1. Machine-independent Measurements

These are measurements of the quality of the solution of the decomposition problem which are independent of the particular machine on which the code is run.

Let us define *load imbalance* to be the difference between the maximum and minimum numbers of elements per processor compared to the average number of elements per processor. In fact, we have to be more precise and use numbers of equations (i.e. work) per processor, as with Dirichlet boundary conditions the finite element boundary nodes are inactive and generate no equations.

The two criteria for measuring communication overhead are the *total traffic size*, which is the sum over processors of the number of floating-point numbers sent to other processors per iteration of the Laplace solver, and the *number of messages*, which is the sum over processors of the number of messages used to accomplish this communication.

The results for the different decomposition methods are shown in Figure 20. For both the SA runs the load balance is significantly poorer then for ORB and ERB. This is caused by the fact that SA does not have the exact balance built in as do the recursive bisection methods, but instead exchanges load imbalance for reducing the communication part of the cost function. The imbalance for the recursive bisection methods comes about from splitting an odd number of elements. From Figure 20 it also appears that there is a sudden reduction in total traffic size for the ORB between the fourth and fifth stage of refinement. This is caused by the geometry of the mesh as shown at the top of Figure 19. At the fourth stage the first vertical bisection is just to the left of the light S-shaped region creating a

large amount of unnecessary communication, and for the fifth and subsequent stages the cut fortuitously misses the highly refined part of the mesh.

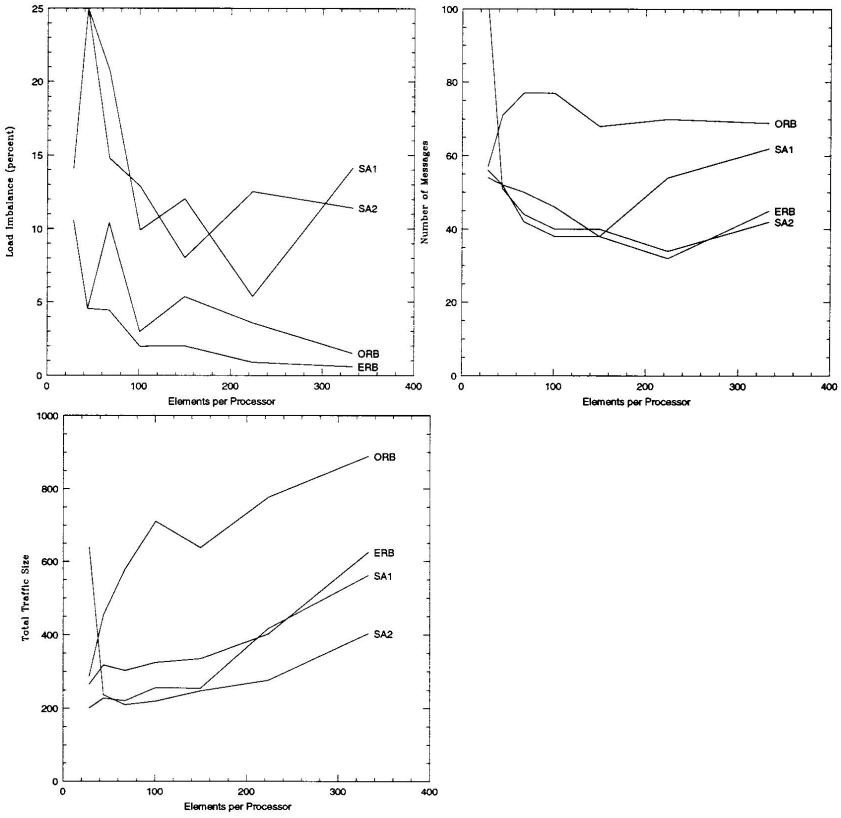

Figure 20: Machine-independent measures of load balancing performance. Upper left, percentage load imbalance; lower left, total amount of communication; right, the total number of messages.

D.2. Machine-dependent Measurements

These are measurements which depend on the particular hardware and message-passing software on which the code is run. The primary measurement is the time it takes the code to run to completion; this is the sum of startup time, domain decomposition time and the product of the number of iterations of the inner loop times the time per iteration. For quasi-static load balancing, we are assuming that the time spent on the basic problem computation is much longer than the load balance time, so parallel computation time is our primary measurement of load balancing performance. Rather than use an arbitrary time unit such as seconds for this measurement, we count this time per iteration as an equivalent number of floating point operations (*flops*). For the nCUBE, this time unit is 15μs for a 64-bit multiply. Thus, we measure flops per iteration of the Jacobi solver. The secondary measurement is the communication time per iteration, also measured in flops. This is just the local communication in the domain and does not include the time for the global combine which is necessary to decide if the Laplace solver has reached convergence.

Figure 21 shows the machine-dependent timings for the various decomposition methods. For the largest mesh, the difference in running time is about 25% between the cheapest load balancing method (ORB) and the most expensive (SA2). The ORB method spends up to twice as much time communicating as the others, which is not surprising, since ORB pays little attention to the structure of the domain it is partitioning, concentrating only on getting exactly half of the elements on each side of an arbitrary line.

The curves on the right of Figure 21show the time spent in local communication at each stage of the test run. Note the similarity with the lower left panel of Figure 20, showing that the time spent communicating is roughly proportional to the total traffic size.

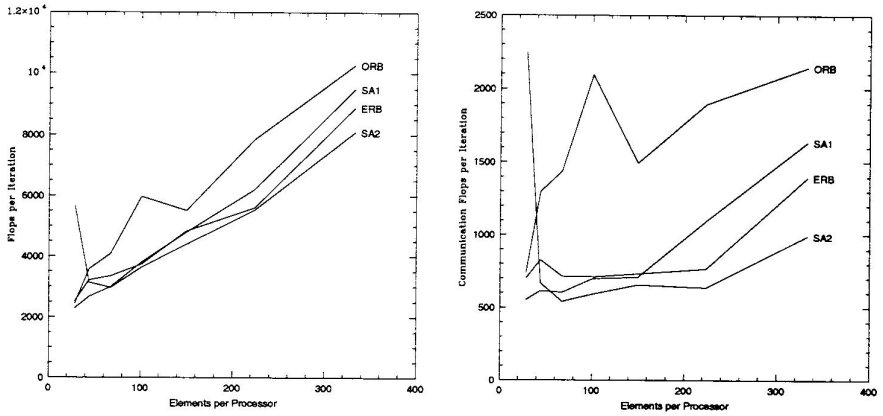

Figure 21: Machine-dependent measures of load balancing performance. Left, running time per Jacobi iteration in unit of the time for a floating point operation (flop); right, time spent doing local communication in flops.

D.3. Measurements for Dynamic Load Balancing

After refinement of the mesh, one of the decomposition methods is run and decisions are made as to which of a processor's elements are to be sent away, and to which processor they are to be sent. A significant fraction of the time taken by the partitioner is taken in this migration of elements, since not only must the element and its data be communicated, but space must be allocated in the new processor and other processors must be informed of the new address of the elements, and so on. Thus, an important measure of the performance of an algorithm for dynamic (in contrast with quasi-dynamic) load balancing is the number of *elements migrated*, as a proportion of the total number of elements.

Figure 22 shows the percentage of the elements which migrated at each stage of the test run. The one which does best here is ORB, because refinement causes only slight movement of the vertical and horizontal median lines. The SA runs are different because of the different starting temperatures: SA1 started at a temperature low enough that the edges of the domains were just "warmed up", in contrast to SA2 which started at a temperature high enough to completely forget the initial configuration and, thus, essentially all the elements are moved.

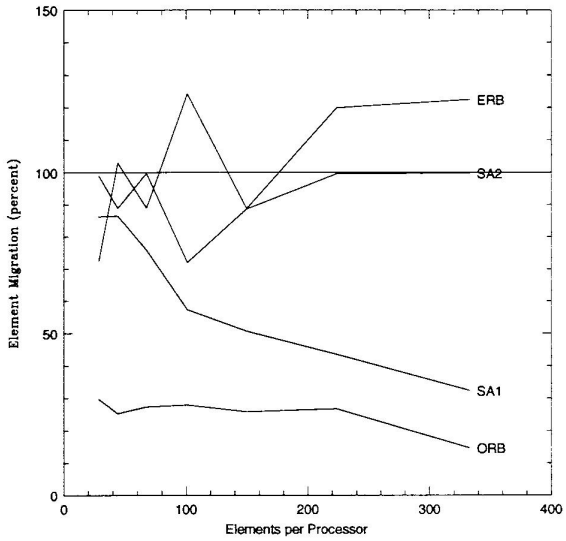

Figure 22: Percentage of elements migrated during each load balancing stage. The percentage may be greater than 100 because the recursive bisection methods may cause the same element to be migrated several times.

The ERB method causes thelargest amount of element migration, which is because of two reasons. The first is because some elements are migrated several times because the load

balancing is done in $\log_2 P$ stages for P processors on an nCube; this is not a fundamental problem, and arises from the particular implementation of the method used here. The second reason is that a small change in mesh refinement may lead to a large change in the second eigenvector. The migration time is only part of the time taken to do the load balancing, the other part being that taken to make the decisions about which element goes where. The total times for load balancing during the seven stages of the test run (solving the decomposition problem plus the migration time) are shown in Table 3.

Method	**Time (minutes)**
ORB	5
ERB	11
SA1	25
SA2	230

Table 3: Total times for load balancing of the complete test runs.

For the test run, the time per iteration was measured in fractions of a second, and it took few iterations to obtain full convergence of the Laplace equation, so that a high-quality load balance is obviously irrelevant for this simple case. The point is that the more sophisticated the algorithm for which the mesh is being used, the greater the time taken in using the distributed mesh compared to the time taken for the load balance. For a sufficiently complex application — for example, unsteady reactive flow simulation — the calculations associated with each element of the mesh may be enough that a few minutes spent load balancing is completely negligible, so that the quasi-dynamic assumption is justified.

E. More Graph Partitioning

E.1. Introduction

The idea to approach domain decomposition of meshes as a graph partitioning was introduced in section A.7 and some specific methods, based on recursive bisection, were shown in section C. This section will dwell some more on the topic of Graph Partitioning, introducing some more recent ideas for highly advanced *static graph partitioning*. Furthermore, we will shortly touch upon the issue of *adaptive* and *parallel graph partitioning* and finally present a number of high quality (public domain) *libraries* that contain a large collection of programs for graph partitioning.

E.2. Static Graph Partitioning

In static graph partitioning we can make a distinction between

1. *Geometric Techniques*
   Here, only coordinate information of mesh elements is used. They are typically very fast but produce worse results compared to schemes that take connectivity into account.
2. *Combinatorial Techniques*
   Attempt to group together highly connected vertices, regardless if their coordinates are close together or far apart. So, only based on adjacency information of the graph.
3. *Spectral Techniques*
   Spectral information that can be found in the Laplacian matrix describing the full connectivity of the graph is applied. This leads to very high quality partitioning, but the at the expense of a relative high computational cost.

Beside these basic graph partitioning methods, a very powerful idea that leads to highly efficient decomposition methods is that of *Multilevel Schemes*, were an initial partitioning is performed on a coarsened version of the graph, which is then refined and optimized in a number of steps. These multilevel schemes, in combination with an efficient static graph partitioning method and the recursive bisection method (see section C.1) lead to highly efficient decomposition methods that give very high quality partitioning.

Geometric techniques

The first geometric technique is *Recursive Coordinate Bisection* (RCB), introduced earlier in section C.2. RCB is also known as *Coordinate Nested Dissection* (CND). It is very fast, requires little memory, is easy to parallelize (how?). However, it also has low quality partitions with usually disconnected subdomains.

CND only bisects normal to coordinate axes. A variant, called *Recursive Inertial Bisection* (RIB) improves this by computing the inertial axes of the mesh, and then to project the mesh point onto this axes, and then sort. The difference between CND and RIB is shown in Figure 23.

Figure 23: The difference between CND (left) and RIB (right).

The Orthogonal Recursive Bisection, introduced in section C.3 improves on CND and RIB, but still has the problem of potential low quality partitions.

CND, RIB, and ORB all sort and partition in a single dimension (be it recursively in orthogonal directions, but each bisection is in principle in a single dimension). Using more dimensions may improve the results. This is the idea of *Space Filling Curves decomposition*. All elements in the mesh are connected by some space-filling curve (e.g. a Peano-Hilbert curve) and the immediately perform a k-ay partitioning on the list of elements sorted along the curve. An example is shown in Figure 24. Space filling curves techniques are also very fast, typically perform better than CND and RIB and works well for specific applications, e.g. hierarchical methods in *N*-body simulations.

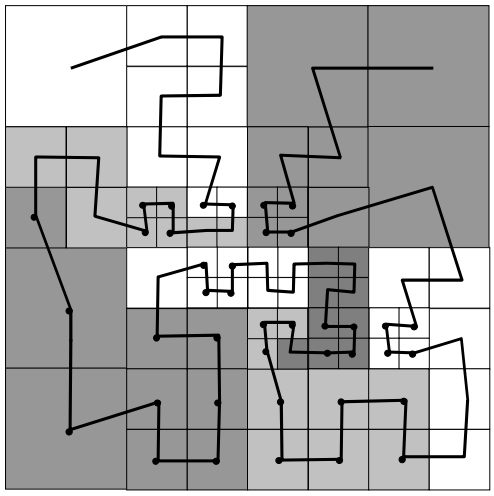

Figure 24: An example of using a space filling curve (in this case a Peano-Hilbert curve) to decompose a mesh into 8 partitions.

Combinatorial techniques

The idea behind combinatorial techniques is to attempt to group together highly connected vertices, regardless if their coordinates are close together or far apart. In other words, combinatorial techniques are only based on adjacency information of the graph.

The first technique to be discussed is *Levelized Nested Dissection* (LND). In LND one puts vertices together, by starting with a single vertex and then grow incrementally with other vertices that are connected to it. The algorithm for LND is as follows

1. Select a single vertex, assign number 0.
2. All vertices adjacent to 0 (i.e. connected to 0) are assigned number 1.
3. Repeat this now for number 1, numbering all non-numbered vertices with increasing numbers.
4. Terminate if half of the vertices have been assigned a number.
5. The numbered vertices go in one domain, the remaining in the other.

An example of LND for a very simple mesh in shown in Figure 25. LND tends to perform better if the initial vertex is one of a pair with largest distance on the graph. In that case LND is equal to Recursive Graph Bisection as introduced in section C.4.. In LND at least one of the domains is connected. Generally it produces comparable or better partitioning than geometric schemes. Usually, multiple LND runs are performed, using different initial vertices, and the best partitioning is chosen.

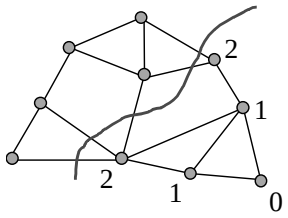

Figure 25: An example of LND.

A very important mesh partitioning algorithm, that is used a lot in practice, is the Kernigham-Lin / Fiduccia-Mattheyes Algorithm (KL/FM). The KL/FM algorithm starts with some initial partition, obtained using a fast technique, and then tries to improve the partition as good a possible, using some heuristic algorithm. The heuristic improvement rules are based on swapping nodes. Given a partition A, B find to equally sized subsets x $\in$ A and y $\in$ B such that swapping x to B and y to A gives the best possible reduction of the edge-cut. KL/FM are greedy algorithms to do this. KL is the original algorithm, FM is a refinement that gives the same results but is much faster. The KL algorithm is given below in Algorithm 1.

```
start with initial partition
in one pass do the following
   pick one vertex from one sub domain, and form pairs of vertices
     by picking vertices from other sub domains
   The pair that gives the best improvement to the edge-cut (or,
     the smallest increase in edge-cut, i.e. jump out of local
     minimum) is chosen and the vertices are swapped.
   This is repeated for the remaining vertices, where once moved
     vertices are not considered anymore in forming pairs.
   When all vertices have been moved, the pass ends.
   At this point, the state that, during the pass, had the
     smallest edge-cut is chosen and restored.
Several passes are performed, usually the algorithm takes just a
few passes to converge.
```

Algorithm 1: The KL Algorithm

Each pass of KL takes $O(|V|^2)$ operations, with |V| the number of vertices in the mesh, which is quite expensive if the number of vertices becomes large, which is the case in many realistic applications. The FM algorithm, a modification of the KL algorithm, improves run time without decreasing the effectiveness (at least for FE grids in scientific computing). FM only uses single vertices to move. It calculates for each vertex in a domain the gain (or loss) in quality if moved. Then it creates priority queues for both domains, and by handling these priority queues efficiently, FM is able to get good

refinement (comparable to KL) in O(|E|), with |E| the length of the edge-cut. This is a huge improvement over KL.

Spectral Techniques

With spectral techniques we refer to the Recursive Spectral Bisection, that was introduced in section C.5. It is all based on calculating the Fiedler vector of the Laplacian matrix. RSB gives very high quality partitions, but is very expensive because obtaining the Fiedler vector is an expensive operation.

Multi-Level Schemes

A very important development was the introduction of multi-level schemes for graph partitioning. In multi-level schemes a graph is first coarsened, then on the coarse graph an initial partition is obtained, and next the graph is refined again, using the KL/FM approach to refine the partitions. The multi-level approach is drawn schematically in Figure 26.

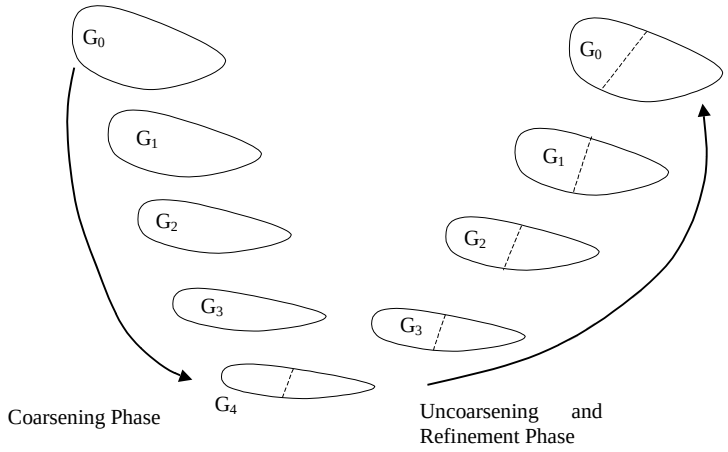

Figure 26: The three phases of the Multi-Level scheme. The original graph G_0 is coarsened in a number of steps. Next, on the coarsest level G4 an initial partitioning is obtained. Next, the original graph is recovered in a number of steps, and on each level the partitioning is refined using KL/FM.

A Multi-Level bisection scheme consists of three phases:

1. *Graph Coarsening*
   Construct series of graphs by collapsing together selected vertices. Repeat this a few times, until a sufficiently small graph is obtained.
2. *Initial Partitioning*
   On the coarsest graph, find an initial partitioning, any fast partitioning method will do.
3. *Multilevel refinement*
   On each level, a partitioning refinement using e.g. KL/FM type algorithms is applied.

This can now be used to build a Multi-Level recursive bisection scheme. Experience shows that this works very well. KL/FM become very powerful in the multilevel context. It is very robust, it outperforms RSB in quality of partitions, and is much faster. The choice of initial partitioning on the coarsest level does hardly have an effect on the final result. Instead of recursive bisection, one can also use k-way partitioning and refinement schemes (i.e. generalizations of KL/FM), thus ending up with Multilevel k-way Partitioning. :This gives comparable or even better partitioning than Multilevel Recursive Bisection. Moreover, it is very fast, typically 2 orders of magnitude faster than spectral methods!

Without any further comments a comparison between static graph partitioning methods is provided in Table 4.

	Needs coordinates	Quality	Local View	Global View	Run Time	Degree of Parallelism
Recursive Spectral Bisection	No	●●●●	o	●●●●	♦♦♦♦	■■
Multilevel Partitioning	No	●●●●●●	oo	●●●●	♦♦	■■
Levelized Nested Disection	No	●●	o	●●	♦♦	■■
Coordinate Nested Disection	Yes	●	o	●	♦	■■■
Recursive Inertial Bisection	Yes	●●	o	●●	♦	■■■

Table 4: A comparison of static graph partitioning methods

E.3. Adaptive Graph Partitioning

Go back to Figure 4. So far we have concentrated on the "easy" case, that of mapping a static application to a static resource (case number 1). Let us make thing a little bit more complicated and concentrate now on case 3, that of mapping a dynamic application to a static resource. You have already encountered an example of this case. The Diffusion Limited Aggregation in Chapter 2 is a dynamically changing application. While the aggregate is growing, the computational domain changes. The growing object is a sink, with the concentration being zero everywhere on the object. This means that while the initial load balancing may have been very good, during the simulation the load balancing will become worse and worse. This calls for a need for *rebalancing* of the parallel simulation.

The applications of type 3 are sometimes called *adaptive* computations. When executed on parallel computers, adaptive computations (e.g. through adaptive grid refinements or changing workload in elements during the simulation) cause load imbalance. Depending on the severity of the resulting fractional load imbalance overhead and the time needed to repair this situation it may pay off to rebalance the load. This question, to what extent it pays off to rebalance is not addressed here. By now you should be able to analyze a specific situation and be able to answer that question (so, how would you do that?). Here we will only introduce some methods that could be used for the actual rebalancing of the load.

The first decision to be taken is if rebalancing is *quasi-dynamic* or *dynamic*. The difference is, loosely speaking, that in the quasi-dynamic case the load is rebalanced not to often (e.g. only after many growth steps in the DLA). One could for instance introduce a parameter that actually measure the load imbalance, and decide to only rebalance if this parameter is larger than some preset value. The optimal frequency of rebalancing is determined by many details and must be set on a case by case basis. In dynamic rebalancing one constantly rebalances the load.

Adaptive computations lead to *adaptive graph partitioning* (a.k.a. *dynamic load balancing*). In adaptive graph partitioning we have the same demand as in static graph partitioning (i.e. good load balance plus minimal communication between sub domains) but added to that we get an extra demand: the amount of data that must be redistributed among processors should be minimized. So, not only the computational weight of a vertex is a parameter (as in the static case), but also it's data size.

Currently two basic approaches exist for adaptive graph partitioning.

1. Scratch-Remap Repartitioning

In scratch-remap repartitioning one forgets the previous partitioning and does it again from scratch. This is in fact what happened in the SA with high temperature in the

extended example of section D. This clearly leads to loss of information, and need for an enormous amount of data to be redistributed. Although relatively easy to realize, this method will, in general, be quite inefficient.

2. Diffusion Based Schemes

Here one performs incremental changes in the partitioning, by moving (diffusing) vertices to neighboring processors. This is a very promising method, but requires lots of fine tuning, depending on the specific application.

E.4. Parallel Graph Partitioning

Although graph partitioning is needed to prepare a large scale simulation for parallel execution, the process of graph partitioning itself should also, in some cases, be executed in parallel. Reasons to do partitioning in parallel are memory constrains due to very large meshes that just don't fit in memory of a single processor or the fact that in many cases the number of available processors (which dictates the number of partitions) may only be known at runtime (especially true in grid-computing environments). Moreover, for adaptive repartitioning, the application is already running on a parallel system. Here we will show no details with respect to parallel graph partitioning, but just limit ourselves by remarking that many parallel versions of partitioning schemes are available in partitioning libraries (see next section)!

E.5. Graph Partitioning Libraries

The Computational Science community has studied graph partitioning for domain decomposition in detail, and many of the resulting algorithms have been implemented in very efficient libraries that are available in the public domain. An overview of some of these libraries is shown in Table 5. At the time of writing, especially the (Par)MeTiS library is a very popular and widely used library for (parallel) graph partitioning.

For completeness, Table 6 provides an overview of the graph partitioning methods that are available in the libraries of Table 5. Note that at the time of writing many of these libraries are still under active development, and therefore it might be that these libraries contain actually more methods than listed in Table 6. We clearly advise you strongly to use one of these libraries if you need to perform domain decomposition on complicated meshes with local (stencil-like) operations!

Name	**URL**
Chaco	http://www.cs.sandia.gov/CRF/chac.html
JOSTLE	http://www.gre.ac.uk/~jjg01/
MeTiS ParMeTiS	http://www-users.cs.umn.edu/~karypis/metis/index.html http://www-users.cs.umn.edu/~karypis/metis/parmetis/index.html
PARTY	http://www.unipaderborn.de/fachbereich/AG/monien/RESEARCH/PART/party.html
SCOTCH	http://dept-info.labri.u-bordeaux.fr/~pelegrin/scotch/
S-HARP	http://www.cs.njit.edu/sohn/sharp/index.html

Table 5: An overview of some public domain graph partitioning libraries.

	Chaco	JOSTLE	MeTIS	ParMeTiS	PARTY	SCOTHC	S-HARP
Geometric Schemes	☺			☺	☺		☺
Coordinate Nested Dissection					●		
Recursive Inertial Dissection	●						●
Space-filling Curve Methods				●			
Spectral Methods	☺				☺		☺
Recursive Spectral Bisection	●				●		●
Multilevel Spectral Bisection	●						
Combinatorial Schemes	☺				☺	☺	
Levelized Nested Dissection					●	●	
KL/FM	●				●	●	
Multilevel Schemes	☺	☺	☺	☺	☺	☺	
Multilevel Recursive Bisection	●		●		●	●	
Multilevel k-way Partitioning		●	●	●			
Multilevel fill-reducing Ordering			●	●			
Dynamic Repartitioners	☺			☺	☺		☺
Diffusive Repartitioning	●			●	●		●
Scratch-remap Repartitioning				●			
Parallel Graph Partitioners		☺		☺			☺
Parallel Static Partitioning		●		●			●
Parallel Dynamic Partitioning		●		●			●
Other Methods			☺				
Multi-constraint Graph Partitioning		●	●	●			
Multi-objective Graph Partitioning			●				

Table 6: Overview of partitioning methods available in the libraries introduced in Table 5.

References

1. Singh, G.S.,Deshpande, K.R.: On fast load partitioning by simulated annealing and heuristic algorithms for general class of problems. Advances in Engineering Software (1993) 23-29
2. Hoekstra, A.G.: Syllabus APR Part 1 : Introduction to Parallel Computing. University of Amsterdam, (1999)
3. Fox, G.C.: Parallel Computers and Complex Systems. Complexity International (1994)
4. Mansour, N.,Fox, G.C.: Allocating data to multicomputer nodes by physical optimization algorithms for loosely synchronous computations. Concurrency: practice and experience 4 (1992) 557-574
5. Fox, G.C., Johnson, M., Lyzenga, G., Otto, S., Salmon, J.,Walker, D.: Solving Problems on Concurrent Processors. Pretince Hall, (1988)
6. de Ronde, J.: Mapping in High Performance Computing, a case on finite element simulation. University of Amsterdam (1997).
7. Mansour, N.,Fox, G.C.: A hybrid genetic algorithm for task allocation. In (Eds.): Proceedings of the 4th International Conference on Genetic Algorithms. (1991) 466-473.
8. Goldberg, D.E.: Genetic Algorithms in search, optimization and machine learning. Addison-Wesley, (1989)
9. Simon, H.D.: Partioning of unstructured problems for parallel processing. Computing Systems in Engeneering 2 (1991) 135-148

10. Pothen, A., Simon, H.,Liou, L.M.: Partitioning sparse matrices with eigenvectors of graphs. SIAM J. Math. Anal. Appl. 11 (1990) 430-452
11. Parlett, B.N., Simon, H.,Stringer, L.M.: On estimating the largest eigenvalue with the Lanczos algorithm. Mathematics of Computation 38 (1982) 153-165
12. Fox, G.C., Williams, R.D.,Messina, P.: Parallel Computing Works! (1994)
13. van de Velde, E.F.: Concurrent Scientific Computing. Springer-Verlag, (1994)