Multilevel Mesh Partitioning for Heterogeneous Communication Networks

C. Walshaw and M. Cross

Computing and Mathematical Sciences, University of Greenwich, Park Row, Greenwich, London, SE10 9LS, UK.

email: C.Walshaw@gre.ac.uk

Mathematics Research Report 00/IM/57

March 29, 2000

Abstract

Multilevel algorithms are a successful class of optimisation techniques which address the mesh partitioning problem for distributing unstructured meshes onto parallel computers. They usually combine a graph contraction algorithm together with a local optimisation method which refines the partition at each graph level. To date these algorithms have been used almost exclusively to minimise the cut-edge weight in the graph with the aim of minimising the parallel communication overhead, but recently there has been a perceived need to take into account the communications network of the parallel machine. For example the increasing use of SMP clusters (systems of multiprocessor compute nodes with very fast intra-node communications but relatively slow inter-node networks) suggest the use of hierarchical network models. Indeed this requirement is exacerbated in the early experiments with meta-computers (multiple supercomputers combined together, in extreme cases over inter-continental networks). In this paper therefore, we modify a multilevel algorithm in order to minimise a cost function based on a model of the communications network. Several network models & variants of the algorithm are tested and we establish that it is possible to successfully guide the optimisation to reflect the chosen architecture.

Keywords: graph partitioning, mesh partitioning, multilevel optimisation, mapping, meta-computing, SMP clusters.

1 Introduction

The need for mesh partitioning arises naturally in many finite element (FE) and finite volume (FV) applications. Meshes composed of elements such as triangles or tetrahedra are often better suited than regularly structured grids for representing completely general geometries and resolving wide variations in behaviour via variable mesh densities. Meanwhile, the modelling of complex behaviour patterns means that the problems are often too large to fit onto serial computers, either because of memory limitations or computational demands, or both. Distributing the mesh across a parallel computer so that the computational load is evenly balanced and the data locality maximised is known as mesh partitioning. It is well known that this problem is NP-complete, so in recent years much attention has been focused on developing suitable heuristics, and some powerful methods, many based on a graph corresponding to the communication requirements of the mesh, have been devised, e.g. [13].

A particularly popular and successful class of algorithms which address this mesh partitioning problem are known as multilevel algorithms. They usually combine a graph contraction algorithm which creates a series of progressively smaller and coarser graphs together with a local optimisation method which, starting with the coarsest graph, refines the partition at each graph level. To date these algorithms have been used almost exclusively to minimise the cut-edge weight, a cost which approximates the total communications volume in the underlying solver. This is an important goal in any parallel application, in order to minimise the communications overhead, however, this edge cut model, in itself somewhat inadequate, [12],

assumes a flat or homogeneous communications network. In fact the trend for connecting together multiprocessor machines results in architectures which exhibit significant network heterogeneities. For example the increasing use of SMP clusters (systems of multiprocessor compute nodes with very fast intra-node communications but relatively slow inter-node networks) suggest the use of hierarchical network models. Indeed this requirement is exacerbated in the early experiments with meta-computers (multiple supercomputers combined together, in extreme cases over inter-continental networks). In this paper therefore, we modify the multilevel algorithms in order to minimise a cost function based on a model of the communications network supplied by the user at run-time. We aim to make the optimisation as generic as possible so that, if and when different architectures appear, the algorithms still apply and can be used simply by changing the network model.

1.1 Overview

The paper is organised as follows. First we define both the partitioning & mapping problems and discuss some of the architectures for which we wish to optimise mappings of unstructured meshes. Then in Section 2 we discuss the multilevel paradigm and outline a multilevel partitioning algorithm which optimises for cut-weight. In Section 3 we describe how different components of this algorithm, in particular the initial partition (§3.2) and the gain & preference functions (§3.3 & §3.4), can be modified to take account of network costs. A large proportion of the paper, Section 4, is given over to experimental results and, having presented several metrics (§4.1), we discuss different ways of modelling the network (§4.2), present the results of the mapping algorithm (§4.3), test different versions of the preference function (§4.4) and in §4.5 give a comparison with a combined partitioning & processor assignment algorithm. Finally we summarise the findings in Section 5 and suggest some future research.

The main contribution of this paper is to describe a multilevel optimisation algorithm which can be influenced to take account of a user supplied model of the communications network. As part of that, the principal innovations are:

- in Section 3.1 we motivate why the multilevel paradigm is so good at this task and why we believe it to provide a powerful solution to the mapping problem.
- in Section 3.4 we suggest a simplification of the preference function without the requirement for $O(P^2)$ operations (where P is the number of processors).
- also in Section 3.4 we describe an algorithm for determining which processors are adjacent in an arbitrarily processor graph.
- in Section 4.2 we discuss how to construct network models which can achieve certain mappings.

1.2 Notation and definitions

Let $G = G(V, E)$ be an undirected graph of vertices V, with edges E which represent the data dependencies in the mesh. The graph vertices can either represent mesh nodes (the nodal graph), mesh elements (the dual graph), a combination of both (the full or combined graph) or some other special purpose representation. We assume that both vertices and edges can be weighted (with positive integer values) and that $|v|$ denotes the weight of a vertex v and similarly for edges and sets of vertices and edges. Given that the mesh needs to be distributed to P processors, let $\mathcal{P}$ be the set of processors and define a partition $\pi : V \rightarrow \mathcal{P}$ to be a mapping of V into P disjoint subdomains S_p such that $\bigcup_P S_p = V$. The weight of a subdomain is just the sum of the weights of the vertices in the subdomain, $|S_p| = \sum_{v \in S_p} |v|$ and we denote the set of inter-subdomain or cut edges (i.e. edges cut by the partition) by E_c (note that $|E_c| = |L|$). Vertices which have an edge in E_c (i.e. those which are adjacent to vertices in another subdomain) are referred to as *border* vertices. Finally, note that we use the words subdomain and processor more or less interchangeably: the mesh is partitioned into P subdomains; each subdomain S_p is assigned to a processor p and each processor p owns a subdomain S_p.

The definition of the graph partitioning problem is to find a partition which evenly balances the load or vertex weight in each subdomain whilst minimising the communications cost. To evenly balance the load,

the optimal subdomain weight is given by $\overline{S} := \lceil |V|/P \rceil$[1] and the *imbalance* is then defined as the maximum subdomain weight divided by the optimal (since the computational speed of the underlying application is determined by the most heavily weighted processor). It is normal practice in graph partitioning to approximate the communications cost by $|E_c|$, the weight of cut edges or cut-weight and, if we define the cost function $\Phi = \Phi(\pi, G) := |E_c|$ the usual (although not universal) definition of the graph partitioning problem is therefore to find π such that $|S_p| \leq \overline{S}$ and such that Φ is minimised.

1.3 The mapping problem

As stated, the usual practice in graph partitioning is to approximate the communications cost by Φ, the cut-weight and then attempt to minimise this quantity. However, for the purposes of this paper we are interested in parallel machines or networks in which the communications cost (both latency and bandwidth) is not uniform across the interprocessor network and in this case the cut-weight is certainly an inadequate measure. For instance, a cut edge between two processors which are 'neighbouring' in some sense will contribute far less to the overall cost than an edge between two processors which are 'far apart'. Unfortunately however, modelling the true communication cost in detail is close to impossible as it depends not only on the latency and bandwidth of point-to-point communications (a cost which can be instrumented) but also on the network loading and congestion at any given time, a factor which is at best highly complex and indeed which can easily be affected by entirely independent applications competing for the same resources. For this reason, and to give us a cost function for which optimisation is tractable, we assign a weight to the link between every pair of processors giving us a network N represented by a weighted graph $N(\mathcal{P}, \mathcal{L})$ where $\mathcal{P}$ is the set of P processors and $\mathcal{L}$ the set of interprocessor edges which is complete (i.e. there is an edge for every pair of processors) and weighted. We can then define the contribution to the cost function from every cut edge (v, w) with $v \in S_p$ & $w \in S_q$ to be $|(v, w)|.|(p, q)|$, the weight of the cut edge multiplied by the weight of the link over which it passes. Thus given a partition $\pi : V \to P$, the cost function is given by

$$\Gamma = \sum_{(v,w) \in E_c} |(v, w)|.|(\pi(v), \pi(w))| \tag{1}$$

Note that for an interconnect with uniform links we have $|(p, q)| = C$, a constant for all $p, q \in \mathcal{P}$ and then this cost just reduces to the cut-weight (modulo C).

Using this new cost function we can then define the *mapping* problem similarly to the partitioning problem as: given a graph $G(V, E)$ and a processor network $N(\mathcal{P}), \mathcal{L}$, find $\pi : V \to \mathcal{P}$, a mapping of vertices to processors, such that $|S_p| \leq \overline{S}$ for all subdomains S_p and such that Γ is minimised.

Note that we distinguish the mapping problem, which is an extension of the partitioning problem, from the *processor assignment problem* (sometimes also called the mapping problem) which, given a partition of a graph, deals with assigning the P subdomains to the P processors again to minimise a cost function such as Γ but typically without changing the assignment of vertices to subdomains. We discuss this processor assignment problem further in §3.2 and compare mapping to partitioning combined with processor assignment in §4.5.

1.4 The network cost matrix: modelling the communications overhead

In order to address the mapping problem we first consider how to represent the communications network in terms of N the weighted complete graph. Firstly it is useful to motivate some of the ideas with the terminology *compute node* which we use to refer to a group of tightly coupled processors. Typically this might be a shared memory multi-processor – sometimes known as a symmetric multi-processor (SMP) – or in the case of meta-computing, any form of supercomputer. We shall generally assume that communications between compute nodes, or *inter-node* communications are relatively slow, whilst those within a compute node, or *intra-node* communications, are relatively fast and uniform. However, the model we use is general and non-uniformity could even be built into intra-node links.

Figure 1 shows some typical processor graphs which model machine interconnection networks. For example, Figure 1(a) is a 1d array, a configuration which may not actually occur in practice as a physical machine interconnect, but which nonetheless can be a useful concept, particularly for machines with

[1] where the ceiling function $\lceil x \rceil$ returns the smallest integer greater than x

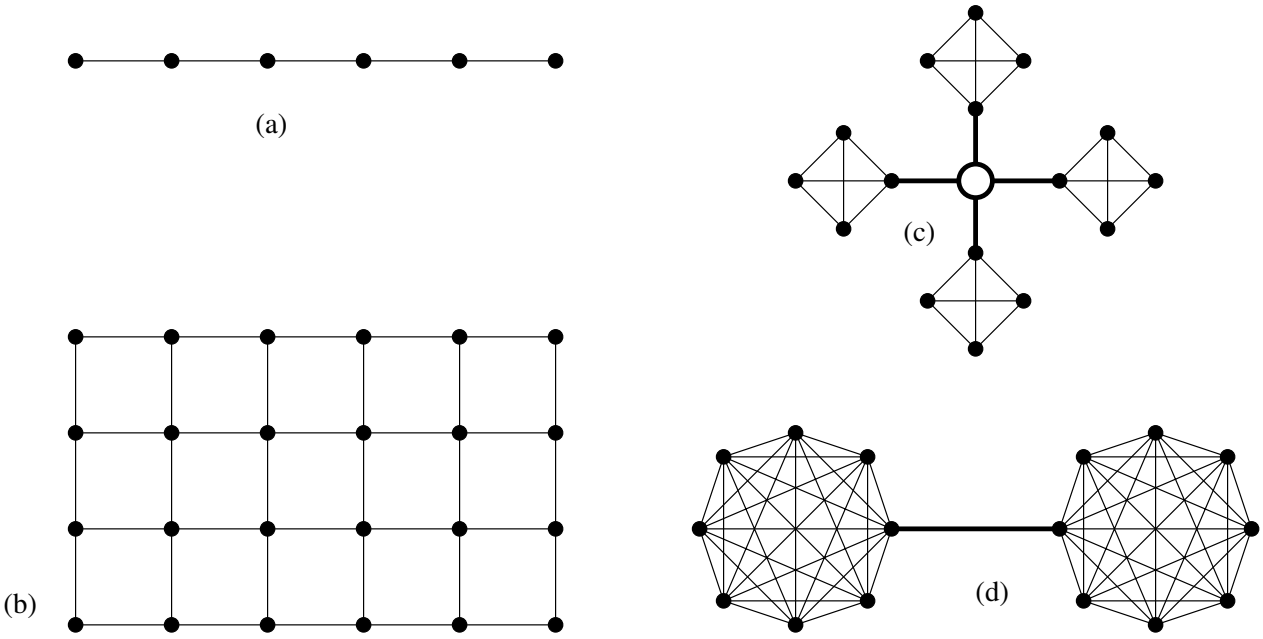

Figure 1: Example processor graphs: (a) 1d array, (b) 2d array, (c) cluster hub and (d) meta-computer.

very high communication latencies, since, if the mesh can be successfully mapped onto this topology, each subdomain will have at most 2 neighbours. Figure 1(b) is a 2d array, a topology which has in fact been realised in the past for the Intel Paragon (and similarly in 3d for the Cray T3D). More recently however, and of particular interest for the purposes of this paper, machines have appeared which have a hierarchical network. For example, Figure 1(c) shows an SMP cluster of 4 compute nodes (each of 4 processors) with all inter-node communications passing through a hub. Meanwhile a meta-computer is illustrated in Figure 1(d). Such machines are not physically assembled as such but consist of two or more compute nodes (typically supercomputers) connected together. For example in [10] experimentation was carried out on a meta-computer consisting of a Cray T3E in Stuttgart, Germany connected to a Cray T3E in Pittsburgh, USA. In this respect they are more extreme examples of network heterogeneities.

Even given relatively simple processor graphs such as those shown in Figure 1, choosing the weighting of links to model the machine is by no means straightforward, e.g. [6]. However, for the example processor graphs shown here we might start by weighting all normal width edges by 1 and the thicker edges by 2. To weight a link between two processors without an explicit edge between them, we can then just sum the weights of the shortest path between them. For example, the weight of the link between the two processors at either end of the 1d array, Figure 1(a), is then 5, whilst the link weight between the two processors at the extreme right and left hand ends of the meta-computer, Figure 1(c), is 4 ($= 1 + 2 + 1$). It turns out (see Section 4.2) that this *linear path length* (LPL) weighting is not sufficiently distinct and so another possibility which we use is to square the path lengths to give the *quadratic path length* (QPL).

This then gives us the complete weighted graph N described above. In practice, however, we can describe the network sufficiently by simply providing a *network cost matrix*, a $P \times P$ matrix of weights modelling the cost of communication between every pair of processors. In fact the matrix is symmetric (since we assume that communication in either direction across the link is equally expensive) and has zeroes down the diagonal (since no communication is required from a processor to itself) and so we actually only need specify the upper diagonal part of the matrix to the partitioning code. Some example network matrices are shown in Figure 2 using the QPL model described above. Figure 2(a) shows the matrix for a 1d array of 8 processors, whilst Figure 2(b) shows that of a cluster of 2 compute nodes each with 4 processors.

$$\begin{bmatrix} 0 & 1 & 4 & 9 & 16 & 25 & 36 & 49 \\ 1 & 0 & 1 & 4 & 9 & 16 & 25 & 36 \\ 4 & 1 & 0 & 1 & 4 & 9 & 16 & 25 \\ 9 & 4 & 1 & 0 & 1 & 4 & 9 & 16 \\ 16 & 9 & 4 & 1 & 0 & 1 & 4 & 9 \\ 25 & 16 & 9 & 4 & 1 & 0 & 1 & 4 \\ 36 & 25 & 16 & 9 & 4 & 1 & 0 & 1 \\ 49 & 36 & 25 & 16 & 9 & 4 & 1 & 0 \end{bmatrix}$$

(a)

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 9 & 16 & 16 & 16 \\ 1 & 0 & 1 & 1 & 9 & 16 & 16 & 16 \\ 1 & 1 & 0 & 1 & 9 & 16 & 16 & 16 \\ 1 & 1 & 1 & 0 & 4 & 9 & 9 & 9 \\ 9 & 9 & 9 & 4 & 0 & 1 & 1 & 1 \\ 16 & 16 & 16 & 9 & 1 & 0 & 1 & 1 \\ 16 & 16 & 16 & 9 & 1 & 1 & 0 & 1 \\ 16 & 16 & 16 & 9 & 1 & 1 & 1 & 0 \end{bmatrix}$$

(b)

Figure 2: Example network matrices: (a) 1d array, (b) cluster of 2 compute nodes each with 4 processors.

1.5 Related work

Despite the fact that the partitioning problem has received a lot of attention in recent years, the mapping problem has been relatively little studied. Even in those papers which have considered it the additional complexity of the problem have led to approaches which are either very limited in application or which focus on particular architectures such as the hypercube. For example, in [8] Dormanns & Heiss describe an approach to map onto grid like networks (e.g. the 1d & 2d arrays that we consider) which uses self-organising maps to geometrically 'fit' the processor grid onto the graph; vertices are then assigned to their nearest processor. Unfortunately, however it is a slow process and it is is difficult to see how it could be adapted for more irregular networks such as the SMP cluster and meta-computer.

In an earlier attempt to address this problem, Walshaw *et al.* used graph-based distance function to calculate the 'width' of a subdomain (in graph terms) and then migrated vertices furthest away from the centre of the subdomain to an adjacent processor, [26]. This technique worked reasonably well for mapping onto 1d & 2d arrays but again it is difficult to see how it could be extended to SMP clusters or meta-computers. Perhaps more interestingly, in tests with a solver using the resulting mappings on parallel machine with 2d array type architecture McManus *et al.*, [19], showed that despite an increase in cut-weight the application scalability & efficiency was much increased using a 2d array mapping as compared to a partitioning/processor assignment approach (see §4.5). Indeed the same was true even for a 1d array mapping with a far greater cut-weight and for certain experiments the efficiency of the 1d mapping even exceeded that of the 2d, [18].

Another more general approach to the mapping problem was developed by Pellegrini, [20, 21], and by Hendrickson *et al.*, [14, 15]. The technique uses recursive bisection of both the mesh (or source graph) and the processor graph (or target graph). This means that the partition of the mesh somehow reflects the natural partition of the network. However additionally within each bisection, apart from the cut-weight, vertices are also assigned to processors based on which portion of the parallel machine their neighbours have already been assigned to. Pellegrini tests the algorithms on a number of architecture models and provides some interesting results, whilst Hendrickson *et al.* incorporate the technique within a multilevel framework (although unlike here as recursive bisection based method) and generalise the idea which (they call *skewed graph partitioning*) to address other partitioning problems.

More recently Teresco *et al.* have discussed a hierarchical model of network performance within a dynamic load-balancing framework, although they do not describe how the load-balancing is able to incorporate the model in order to optimise the mapping, [23].

Perhaps most interesting for the methods described here is the work of Chen & Taylor. They have investigated partitioning for distributed systems and, in [6], provide experimental and theoretical analysis which suggests that for architectures such as the cluster and meta-computer, it can be most beneficial to the efficiency if all inter-node communications to/from a given compute node is done by just one processor of the node. They have also constructed a parallel mapping tool called ParaPART which takes into account the network costs, [7]. This uses a 3 stage process; firstly the mesh is partitioned into n parts (where n is the number of compute nodes) and then as a second step the portion of the mesh assigned to each compute node is partitioned amongst its processors. In a final step the partition for each compute node is retrofitted using simulated annealing to ensure that only one processor carries out the communication to/from each

compute node and that this processor has a correspondingly smaller portion of the mesh because of its additional communication load.

1.6 Related issues

We do not address here the issues of inhomogeneous CPU performance. In fact this is a somewhat simpler problem to solve and the software, JOSTLE, [24], in which we have implemented and tested the schemes presented here is able to take this into account using its integral load-balancing capabilities. For example, given a graph of say 75 vertices and two processors, with processor 1 twice as fast as processor 2, the user may impose a penalty weight (based on the relative speeds and the total vertex weight; in this case 25) on processor 2 to simulate its slower performance. The load-balancer within JOSTLE then balances the total graph weight plus any penalty weights (in this example $75 + 25 = 100$) and gives an equal share (50) to each processor. Because processor 2 has a penalty weight of 25, its share of the vertices is 25 as compared with the 50 of processor 1 and so the partition is balanced to reflect the relative performance of the processors. We have not yet tested this functionality in combination with the network optimisation ideas described here but see no reason why it should not work successfully.

A related but more complex issue is addressed by Chen & Taylor, [6], who examine the balancing of computation and communication. For example, if one processor within a compute node is having to do all the communication to external compute nodes then it should be given less computational load. Once again we have not explored this problem but believe that with a judicious choice of penalty weight, it could be handled using the functionality described in the previous paragraph. This is essentially the method that Chen & Taylor use to address the problem although it does make the assumption that the extra communications load can be estimated prior to partitioning. In fact the extra communications load is a function of the resulting partition and so this assumption may not be valid. However, building a true representation of this function into the partitioning cost model may be intractable.

2 Multilevel mesh partitioning

In this section we discuss the multilevel paradigm in the context of the mesh partitioning problem and outline our multilevel algorithm, described in [24], for addressing it. The modifications to the algorithm for optimising a network based cost function are deferred to Section 3.

2.1 The multilevel paradigm

In recent years it has been recognised that an effective way of both speeding up mesh partitioning algorithms and/or, perhaps more importantly, giving them a global perspective is to use multilevel techniques. The idea is to match pairs of vertices to form *clusters*, use the clusters to define a new graph and recursively iterate this procedure until the graph size falls below some threshold. The coarsest graph is then partitioned (possibly with a crude algorithm) and the partition is successively optimised on all the graphs starting with the coarsest and ending with the original. This sequence of contraction followed by repeated expansion/optimisation loops is known as the multilevel paradigm and has been successfully developed as a strategy for overcoming the localised nature of the Kernighan-Lin (KL), [17], and other optimisation algorithms. The multilevel idea was first proposed by Barnard & Simon, [1], as a method of speeding up spectral bisection and improved by both Hendrickson & Leland, [13] and Bui & Jones, [2], who generalised it to encompass local refinement algorithms. Several algorithms for carrying out the matching have been devised by Karypis & Kumar, [16], while Walshaw & Cross describe a method for utilising imbalance in the coarsest graphs to enhance the final partition quality, [24].

Graph contraction. To create a coarser graph $G_{l+1}(V_{l+1}, E_{l+1})$ from $G_l(V_l, E_l)$ we use a variant of the edge contraction algorithm proposed by Hendrickson & Leland, [13]. The idea is to find a maximal independent subset of graph edges, or a *matching* of vertices, and then collapse them. The set is independent if no two edges in the set are incident on the same vertex (so no two edges in the set are adjacent), and maximal if no more edges can be added to the set without breaking the independence criterion. Having found such a set, each selected edge is collapsed and the vertices, $u_1, u_2 \in V_l$ say, at either end of it are merged to form a new vertex $v \in V_{l+1}$ with weight $|v| = |u_1| + |u_2|$.

A simple way to construct a maximal independent subset of edges is to create a randomly ordered list of the vertices and visit them in turn, matching each unmatched vertex with an unmatched neighbouring vertex (or with itself if no unmatched neighbours exist). Matched vertices are removed from the list. If there are several unmatched neighbours the choice of which to match with can be random, but it has been shown by Karypis & Kumar, [16], that it can be beneficial to the optimisation to collapse the most heavily weighted edges and our matching algorithm uses this heuristic.

The initial partition. Having constructed the series of graphs until the number of vertices in the coarsest graph is smaller than some threshold, the normal practice of the multilevel strategy is to carry out an initial partition. Here, following the idea of Gupta, [11], we contract until the number of vertices in the coarsest graph is the same as the number of subdomains, P, and then simply assign vertex i to subdomain S_i. Unlike Gupta, however, we do not carry out repeated expansion/contraction cycles of the coarsest graphs to find a well balanced initial partition but instead, since our optimisation algorithm incorporates balancing, we commence on the expansion/optimisation sequence immediately.

Partition expansion. Having optimised the partition on a graph G_l, the partition must be interpolated onto its parent G_{l-1}. The interpolation itself is a trivial matter; if a vertex $v \in V_l$ is in subdomain S_p then the matched pair of vertices that it represents, $v_1, v_2 \in V_{l-1}$, will be in S_p.

2.2 The iterative optimisation algorithm

The iterative optimisation algorithm that we use at each graph level is a variant of the Kernighan-Lin (KL) bisection optimisation algorithm which includes a hill-climbing mechanism to enable it to escape from local minima. Our implementation uses bucket sorting, the linear time complexity improvement of Fiduccia & Mattheyses, [9], and the buckets are accessed via a tree structure, which we refer to as a bucket tree. The algorithm is a partition optimisation formulation; in other words it optimises a partition of P subdomains rather than a bisection (this functionality is sometimes referred to as multiway, P-way or k-way optimisation). The algorithm is fully described and tested in [24].

As is typical for KL type algorithms, the optimisation has inner and outer iterative loops with the outer loop terminating when no migration takes place during an inner loop. It uses two bucket sorting structures or bucket trees and is initialised by calculating the gain – the potential improvement in the cost function (the cut-weight in the classical graph partitioning context) – for all border vertices and inserting them into one of the bucket trees. These vertices are referred to as *candidate* vertices and the tree containing them as the *candidate tree*.

The inner loop proceeds by examining candidate vertices, highest gain first (by always picking vertices from the highest ranked bucket), testing whether the vertex is acceptable for migration and then transferring it to the other bucket tree (the tree of *examined* vertices). If the candidate vertex is found to be acceptable (i.e. it does not overly upset the load-balance), it is migrated, its neighbours have their gains updated and those which are not already in the examined tree are relocated in the candidate tree according to this updated gain. This inner loop terminates when the candidate tree is empty although it may terminate early if the partition cost rises too far above the cost of the best partition found so far. Once the inner loop has terminated any vertices remaining in the candidate tree are transferred to the examined tree and finally pointers to the two trees are swapped ready for the next pass through the inner loop.

The algorithm also uses a KL type hill-climbing strategy; in other words vertex migration from subdomain to subdomain can be *accepted* even if it degrades the partition quality and later, based on the subsequent evolution of the partition, either rejected or *confirmed*. During each pass through the inner loop, a record of the optimal partition achieved by migration within that loop is maintained together with a list of vertices which have migrated since that value was attained. If subsequent migration finds a 'better' partition then the migration is *confirmed* and the list is reset. Note that it is possible to find better partitions despite selecting some vertices with negative gain because, as the optimiser runs, the gains of adjacent vertices will change and so the migration of a group of vertices some or all of which start with negative gain can in fact decrease the overall cost (i.e. produce a net positive gain). Once the inner loop is terminated, any vertices remaining in the list (vertices whose migration has not been confirmed) are migrated back to the subdomains they came from when the optimal cost was attained.

The algorithm, together with conditions for vertex migration acceptance and confirmation is fully described in [24].

3 Modifying the method for mapping

3.1 Motivation

In this section we describe the modifications required to allow the multilevel algorithm to optimise a cost function based on network costs. In fact the coarsening algorithm is left unchanged and the cost function is first taken into account when the P vertices of the coarsest graph are assigned to the P processors (§3.2). The cost is subsequently optimised on each of the multilevel graphs in succession by relatively simple changes to the gain and preference functions (§3.3 & §3.4). A successful mapping is then one in which subdomains are constructed such that adjacent subdomains generally lie on adjacent processors. The power of the process to compute such a mapping stems from the global properties of the multilevel algorithm. Edges which cross expensive links are penalised heavily within the cost function and so vertices at either end of such an edge tend to migrate to more adjacent processors (more adjacent to the processor owning the vertex at the other end of the edge) and create a sort of buffer zone. However, because this occurs high up in the multilevel process, where each vertex v represents many vertices in the original graph, the buffer zone which may start off only one vertex wide, can actually represent reasonably broad regions in the mesh. In this way the partition is given a good global quality on the coarse graphs which is refined on the finer graphs.

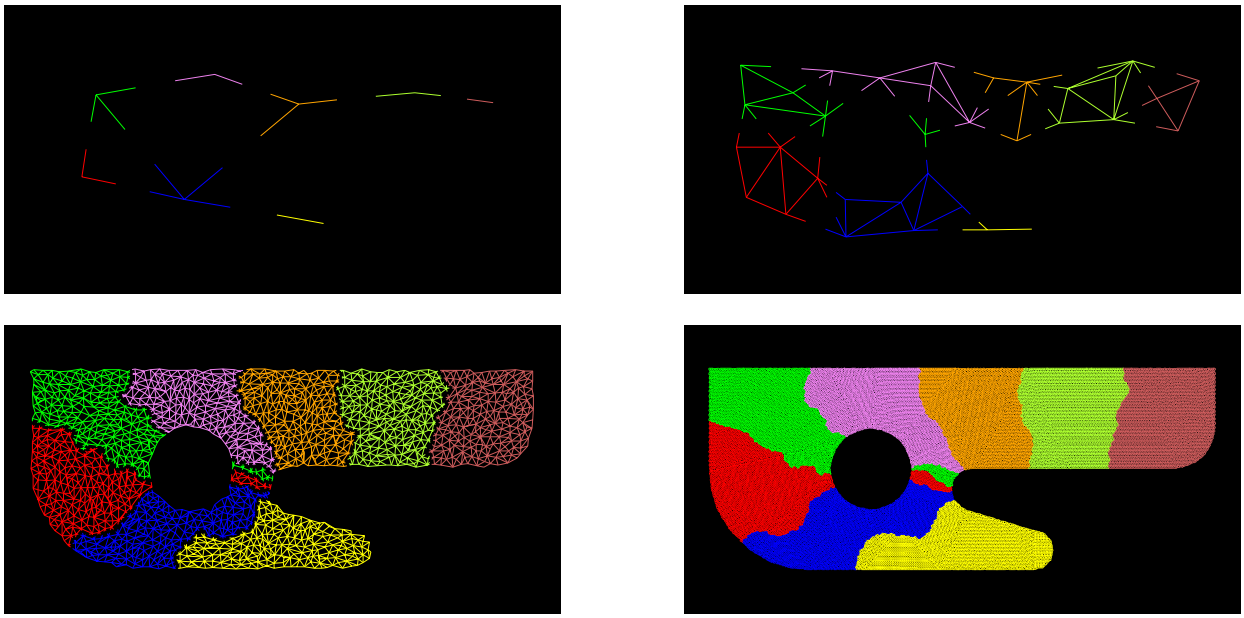

Figure 3: The multilevel mapping illustrated for the mesh t60k on a 1d array with 8 processors

Figure 3 illustrates this process for the t60k mesh (as used in the testing, Section 4) on a 1d array of 8 processors. It can be seen (top left) that the coarsest graph, G_{15}, with 8 vertices is fairly linear and so the initial partition is reasonably good, although there is a suboptimal cut edge between processor 2 (blue) and processor 6 (beige). After a couple of optimisations the mapping algorithm has already started to buffer these two processors away from each other as shown by the partition on graph G_{12} with 34 vertices (top right). By the time the multilevel process reaches graph G_6 with 1488 vertices (bottom left) the mapping has succeeded in separating all non-neighbouring processors although the buffer region of processor 4 (green) is only 1 vertex wide in some parts. However in the final partition (bottom right), on graph G_0 with 60005 vertices, this has been redressed.

3.2 The initial partition

As in the standard multilevel algorithm we contract the graph until the number of vertices in the coarsest graph, $G_L(V_L, E_L)$, is the same as the number of processors, P, and assign each coarse vertex to a processor.

However, it makes sense at this point to try and map the vertices so as to minimise the cost function Γ from (1). In other words we wish to minimise

$$\sum_{(v_i,v_j)\in E_L} |(v_i,v_j)|.|(\pi(v_i),\pi(v_j))| \tag{2}$$

Suppose now that we write the coarsest graph G_L as a matrix with the ij^{th} entry equal to the weight of the edge between vertices v_i & v_j so that

$$g_{ij} = \begin{cases} |(v_i,v_j)| & \text{if } v_i \text{ is adjacent to } v_j \\ 0 & \text{if } v_i \text{ is not adjacent to } v_j \\ 0 & \text{on the diagonal, i.e. if } i=j. \end{cases}$$

Then, since there are P vertices in G_L and P processors, every edge in E_L must be cut and so (2) is equivalent to minimising

$$\sum_{i=1}^{P}\sum_{j=1}^{P} g_{ij} n_{\pi(i)\pi(j)}$$

where $\pi(i)$ is shorthand for $\pi(v_i)$ and n_{kl} is the kl^{th} entry in the network cost matrix (see §1.4).

In fact this expression is a simplification (in the more general case there is also a linear term) of a well known optimisation problem the *quadratic assignment problem* (QAP), [3]. This has been extensively studied since 1957 and is NP-complete, [4]. There are many heuristic algorithms which address the problem, some of which are available in a software library, QAPLIB[2]. For the results in this paper we use one such algorithm based on simulated annealing and described in [5].

3.3 The gain function

Once the initial partition has been computed, the multilevel approach uses a modification of the optimisation algorithm (outlined in §2.2) successively on each of the coarsened graphs and finally on the original. As is usual for such KL like optimisation algorithms, a key concept in the method is the idea of *gain*. The gain $g(v,q)$ of a vertex v in subdomain S_p can be calculated for every other subdomain, S_q, $q \neq p$, and expresses how much the cost of a given partition would be improved were v to migrate to S_q. Thus, if π denotes the current partition and π' the partition if v migrates to S_q then for a cost function Ψ, the gain $g(v,q) = \Psi(\pi) - \Psi(\pi')$. Normally in mesh partitioning the cost function is simply the total weight of cut or inter-subdomain edges, $\Psi(\pi) = \Phi(\pi) = |\{(v,w) \in E : v \in S_p \,\&\, w \in S_q, p \neq q\}|$ and in this case the gain is calculated as follows: Given a vertex $v \in S_p$, let $e_q(v)$ denote the set of edges from v to vertices in S_q, $e_q(v) = \{(v,w) \in E : w \in S_q\}$. Then the part of the cost function $\Phi(\pi)$ associated with v is

$$\sum_{r\neq p} |e_r(v)| = \sum_{r\in\mathcal{P}} |e_r(v)| - |e_p(v)|.$$

If v migrates to S_q then the part of the new cost, $\Phi(\pi')$, associated with v becomes

$$\sum_{r\neq q} |e_r(v)| = \sum_{r\in\mathcal{P}} |e_r(v)| - |e_q(v)|. \tag{3}$$

No other part of the cost function is affected so the gain is simply

$$\begin{aligned} \texttt{gain}(v,q) &= \Phi(\pi) - \Phi(\pi') && (4)\\ &= \left[\sum_{r\in\mathcal{P}} |e_r(v)| - |e_p(v)| + \Phi_0\right] - \left[\sum_{r\in\mathcal{P}} |e_r(v)| - |e_q(v)| + \Phi_0\right] && (5)\\ &= |e_q(v)| - |e_p(v)| && (6) \end{aligned}$$

[2]available from `http://www.imm.dtu.dk/~sk/qaplib/`

(where Φ_0 simply represents the part of the cost function unaffected by v).

Recall however, that in this paper we are interested in the mapping cost in which edges between different subdomains are weighted differently depending on the cost of communication between the processors owning these subdomains. Thus the cost associated with $v \in S_p$ is

$$\sum_{r \neq p} |e_r(v)|.|(p,r)| \tag{7}$$

where $|(p,r)|$ represents the weight of an edge between processor p & r. Equation (7) can more conveniently be written

$$\sum_{r \in \mathcal{P}} |e_r(v)|.|(p,r)|$$

since we can take $|(p,p)| = 0$. Similarly to (3) the new contribution to the cost if v migrates to S_q is

$$\sum_{r \in \mathcal{P}} |e_r(v)|.|(q,r)|$$

and so the gain is

$$\begin{aligned}
\texttt{gain}(v,q) &= \Gamma(\pi) - \Gamma(\pi') && (8)\\
&= \left[\sum_{r \in \mathcal{P}} |e_r(v)|.|(p,r)| + \Gamma_0\right] - \left[\sum_{r \in \mathcal{P}} |e_r(v)|.|(q,r)| + \Gamma_0\right] && (9)\\
&= \sum_{r \in \mathcal{P}} |e_r(v)|.\left(|(p,r)| - |(q,r)|\right). && (10)
\end{aligned}$$

Note that whilst this expression is not in itself difficult to evaluate, it has complexity $O(P)$ and is thus considerably more costly than that for the cut-weight gain function (6). This additional complexity will have a bearing on the evaluation of the preference below.

3.4 Setting the preference

The preference of a vertex $v \in S_p$ expresses the migration that maximises the gain. Thus if $\texttt{gain}(v,q) = \max_{r \neq p} \texttt{gain}(v,r)$ or in other words the gain of migrating v to subdomain S_q produces the maximum gain in the cost function over all possible migrations of v, then the preference of v is set to q, $\texttt{pref}(v) = q$. In this section we describe 3 possible ways of setting the preference:

Adjacent subdomain preference. For the cut-weight cost function, it is impossible to achieve a positive gain by migrating a vertex to a subdomain to which it is not adjacent and it is thus quite usual to make a simplification and only maximise the gain over subdomains adjacent to the vertex. Indeed most border vertices will only be adjacent to one subdomain S_q and then the preference is simply set to q without the need to find a maximum. For those adjacent to more than one subdomain, it is still inexpensive to find the maximum as the number of neighbouring subdomains is bounded by the degree of the vertex (which is usually low for graphs arising from finite element and finite volume applications).

Full processor preference. For optimising a mapped partition, however, it is no longer true that migrating a vertex to a non-adjacent subdomain cannot accrue a positive gain. Consider the graph in Figure 4(a) being mapped to the processor graph in Figure 4(b) with the edges weighted as shown. Note that this processor graph is simply a 1d array (see §1.4) of three processors p, q & r with the non-local edge given a weight of 4. Vertex $v \in S_p$ is only adjacent to one other subdomain S_r and this cut edge adds a cost of 4 to the cost function (because the edge (p,r) is weighted 4 in the processor graph). However migration of v to S_r does not improve the cost and so $\texttt{gain}(v,r) = 0$. On the other hand, if, as in Figure 4(c), v migrates to S_q (to which v is not adjacent) the cost is improved by 2 (since S_p and S_r are no longer adjacent) and so $\texttt{gain}(v,q) = 2$. Thus, although one more edge is cut, because of the migration of v to a non-adjacent subdomain, the cut edges map better onto the least costly edges in the processor graph. It is also interesting to note that this is the optimal mapping for this graph, despite the fact that subdomain S_q is disconnected.

An obvious conclusion is that the simplification of limiting the preference to adjacent subdomains is not appropriate in the mapping case and to set the preference for $v \in S_p$ by maximising the gain over all

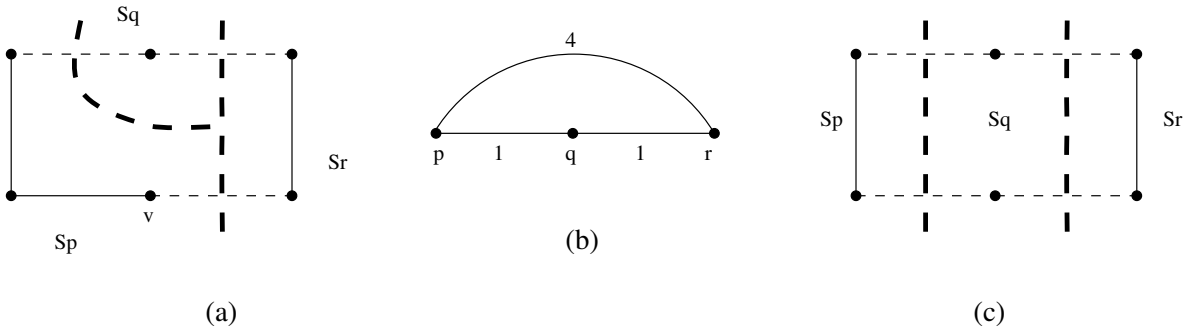

Figure 4: An example mapping: (a) the partitioned graph, (b) the target processor graph and (c) the optimal mapping.

$r \neq p$. This ensures that the method finds the maximum gain, however it does mean that, since the gain calculation has complexity $O(P)$, the preference calculation is $O(P^2)$ because we must calculate the gain for all $P-1$ possible migrations.

Adjacent subdomain/processor preference. For small numbers of processors, P, this is not a serious overhead, however, calculating gains & preferences is a fairly fundamental operation in the optimisation algorithm and for large numbers of processors it can be prohibitive to compute so many gains, many of which may be far from optimal. For this reason we suggest an alternative simplification which retains the spirit of the previous one and for a vertex $v \in S_P$ seek the maximum gain over a **union** of subdomains adjacent to v together with processors adjacent to p in the processor graph.

Looking at the example processor graphs in Figure 1, it is easy to see which processors are adjacent to each other, however, for an arbitrary processor graph which is simply specified as $P(P-1)/2$ edge weights (see §1.4) the processor adjacencies must be determined. To simply say that processors p & q are adjacent if the edge weight $|(p,q)|$ is 1 is sufficient for the 1d & 2d arrays and renders the visual representation shown in Figures 1(a) & (b). However it gives disconnected graphs in the cases of the cluster hub and meta computer architectures and this can be deleterious to the optimisation which ideally should migrate vertices to near neighbours; if we restrict the preference to only the most closely coupled processors, the algorithm is given no information on how to minimise the inter-node links.

We therefore determine the processor adjacencies by sorting the $P(P-1)/2$ edges into sets distinguished by weight (i.e. set 1 contains edges of weight 1, set 4 contains edges of weight 4, etc.) and including all those with minimum weight in the processor adjacency graph. We then test if this graph is disconnected with a breadth first search from any one of the processors. If it fails to span the graph (visit all the processors), then the graph is disconnected and we add in the next set of edges. This process is repeated until a connected graph, the processor adjacency graph, is recovered.

Note the distinction between the two representations of the network – the processor graph, $N(\mathcal{P},\mathcal{L})$ or network cost matrix has P vertices and $P(P-1)/2$ edges and is a complete graph (i.e. for every $p,q \in \mathcal{P}$ there is an edge $(p,q) \in \mathcal{L}$), whilst the processor adjacency graph $N'(\mathcal{P},\mathcal{L}')$ has P vertices but a subset of edges $\mathcal{L}' \subset \mathcal{L}$ (the minimally weighted subset of edges sufficient to connect N').

Summary. Returning to the preference, this then gives us three possible functions to choose from. In order of computational complexity:

- the adjacent subdomain preference function (or f_s for short) where a vertex's preference for migration is selected from subdomains to which it is adjacent
- the adjacent subdomain/processor preference function (or f_{sp}) where, additionally, the preference can be selected from processors adjacent in the processor graph to the processor which owns the vertex
- the full processor preference function (or f_P) where the preference is selected from among every processor

Of these three f_P is always of complexity $O(P^2)$ to calculate but is the true representation for calculating the maximum gain. The other two, f_s & f_{sp} cannot guarantee to find the maximum gain, but if P is not too small are unlikely to be $O(P^2)$. In §4.4 we determine test the three possibilities and determine the best function to use in terms of complexity and results.

4 Results

We have implemented the algorithms described here within the framework of JOSTLE, a mesh partitioning software tool developed at the University of Greenwich and freely available for academic and research purposes under a licensing agreement[3]. The experiments were carried out on a DEC Alpha with a 466 MHz CPU and 1 Gbyte of memory.

The test graphs have been chosen to be a representative sample of medium to large scale real-life problems and include both 2D and 3D examples of nodal graphs (where the mesh nodes are partitioned) and dual graphs (where the mesh elements are partitioned). None of the graphs are weighted; such graphs are not widely available since most applications do not accurately instrument costs and it is difficult to draw meaningful conclusions from the few examples that we have access to.

	size		degree			
mesh	V	E	max	min	avg	type
crack	10240	30380	9	3	5.93	2D nodal graph
4elt	15606	45878	10	3	5.88	2D nodal graph
t60k	60005	89440	3	2	2.98	2D dual graph
dime20	224843	336024	3	2	2.99	2D dual graph
144	144649	1074393	26	4	14.86	3D nodal graph
m14b	214765	1679018	40	4	15.64	3D nodal graph
cyl3	232362	457853	4	2	3.94	3D dual graph
mesh1m	1119663	2212012	4	2	3.95	3D dual graph

Table 1: A summary of the test meshes.

Table 1 gives a list of the graphs, their sizes, the maximum, minimum & average degree of the vertices and a short description. The degree information (the degree of a vertex is the number of vertices adjacent to it) gives some idea of the character of the graphs. These range from the relatively homogeneous dual graphs, where every vertex represents a mesh element, in these cases a triangle (2D) or tetrahedron (3D) and so every vertex has at most 3 or 4 neighbours respectively, to the more complex nodal graphs with their more irregular interconnections. As the graphs are not weighted, the number of vertices in V is the same as the total vertex weight $|V|$ and similarly for the edges E.

4.1 Metrics

Unfortunately there is no clear metric to measure the quality of a partition and so we use a variety, as follows:

- **Cut Weight** (Φ). As discussed in §1.2 the classical, although disputed, measure for partition quality is the total weight of cut edges or cut-weight. Thus if E_c denotes the set of cut edges then the cut-weight Φ is given by

$$\Phi = |E_c| = \sum_{(v,w)\in E_c} |(v,w)|.$$

  This metric approximates the total communication volume for the sort of homogeneous graphs which represent meshes (although see [12, 22] for further discussion). However it is not appropriate for heterogeneous networks since a cut edge between vertices on 'neighbouring' processors does not have the same impact on the runtime of the underlying solver as a cut edge between 'non-neighbouring' processors.

- **Network Cost** (Γ). A better metric is the (network) cost function which we are trying to minimise within the optimisation and which, from (1), is given by

$$\Gamma = \sum_{(v,w)\in E_c} |(v,w)|.|(\pi(v),\pi(w))|.$$

[3]available from `http://www.gre.ac.uk/jostle`

Recall from §1.3 that π is the mapping of vertices to processors, so that $\pi(v)$ represents the processor which owns v and hence the network edge weight $|(\pi(v), \pi(w))| = |(p, q)|$ for two processors p, q is just the entry n_{pq} of the network cost matrix. However, as we will see below, §4.2, the network cost matrix is constructed so as to guide the mapping appropriately rather than measured from machine response times, and in this sense the network cost, Γ, is a somewhat abstract measure.

- **Average Dilation** (Δ). Perhaps a better measure for comparing different partitions, particularly to quantify the success in following the guidance given by the network cost matrix, is the average dilation. This measure describes the average link weight (averaged over all of the cut edges). In other words, we can define the average dilation as

$$\Delta = \left[\sum_{(v,w) \in E_c} |(\pi(v), \pi(w))| \right] / O(E_c)$$

where $O(E_c)$ is just the number of edges in E_c. In fact, because all the meshes have unit edges weights (i.e. $|(v, w)| = 1$ for all $(v, w) \in E$ and hence all $(v, w) \in E_c$), the size of E_c is the same as the cut-weight, $O(E_c) = |E_c| = \Phi$ and hence the average dilation is just the network cost divided by the cut-weight

$$\Delta = \left[\sum_{(v,w) \in E_c} |(\pi(v), \pi(w))| \right] / O(E_c) = \left[\sum_{(v,w) \in E_c} |(v, w)|.|(\pi(v), \pi(w))| \right] / O(E_c) = \Gamma / \Phi.$$

However this is not true for weighted graphs.

- **Average Path Length/Unweighted Dilation** (δ). Related to the average dilation is the average path length

$$\delta = \left[\sum_{(v,w) \in E_c} l(\pi(v), \pi(w)) \right] / O(E_c)$$

where $l(p, q)$ is just the minimum path length (number of edges) in the processor graph between processors p & q. In fact if we define the weight of a link between two processors to be the path length then Δ and δ are the same. However, we believe that in order to heavily penalise cut-edges across non-local links, it is better for the link weights not to be a linear function of path length.

Note that in terms of interpretation of the results, what we typically look for is low average dilation (i.e. most of the communication occurs across the fastest links). For architectures such as the cluster and meta-computer a certain amount of communication inevitably crosses the slower inter-node links but, if we can keep it to a small proportion then it should not slow the application too much. Indeed even for architectures such as the 1d & 2d arrays, it can be the case that single message interprocessor latencies & bandwidths are almost identical for both 'nearby' and 'distant' processors. However in this case we still wish to achieve a good mapping of the mesh to the network to avoid the congestion which occurs when most of the processors have to communicate across the machine rather than locally. Once again success is indicated by low *average* dilation (since a small amount of non local communication should not overly impede the application.

For a final comparison metric we also use the partitioning run-time, τ, for some of the tests. We also measure the imbalance (as defined in §1.2) but we do not record it here since in all the experiments it did not exceed the specified tolerance (set at run-time) of 1.03, or 3% imbalance.

4.2 Network modelling

The ultimate aim of this work would be to derive a generic optimisation technique which can map a mesh onto a parallel interconnection network given an appropriate network cost matrix (NCM). In an ideal world one could imagine that it should be possible for the optimisation software to run some quick tests on the parallel network in question, measure the response times and derive an NCM automatically. However, we do not believe that such a technique would necessarily provide a good NCM. As mentioned in §1.5, Chen

& Taylor, [6], suggest that it can be most beneficial to efficiency if all inter-node communications to/from a given compute node is done by just one processor of the node (as we have suggested in the way that we have drawn Figures 1(c) & 1(d)). The NCM should therefore be weighted such that one processor per node has easier access to remote processors. However any instrumentation of such architectures (e.g. ping-pong style tests where messages are passed back and forth and response times measured) would be likely to show roughly similar rapidity for **all** intra-node communications and roughly similar slowness for any inter-node messages. The NCM would therefore be a good representation of the network but not a good enough blueprint for guiding the mapping.

The next question that arises, given that instrumentation of the network may be insufficient for the mapping task, is what value to give the weights. It is helpful here to consider one of the simplest architectures, the 1d array. In fact a successful mapping for a 1d array usually corresponds to a slicing of the domain and tends to result in long thin subdomains However, it is not an entirely unreasonable architecture to map onto and, for example, on systems which are very heavily latency dominated (e.g. networks of workstations), the minimum possible number of communication startups per processor is 2 (except for those at the ends of the array) and it can be worth putting up with longer subdomain boundaries and hence longer messages in order to achieve this, [18]. The 1d array thus gives us a very simple but not unrealistic architecture on which to do some initial tests. We can then define the network cost matrix N, to be $n_{pq} = |p - q|^\lambda$, i.e. the path length between any two processors to some power λ.

In Tables 2 & 3 we test three different network models, the linear, quadratic & cubic path length models (or in other words $\lambda = 1, 2, 3$), of a 1d array. In order to compare them fairly (i.e. in the same metric) we have run the mapping algorithm for all the meshes using the three values of λ and then for each result measured the average path length (which is the same as the average dilation for the linear path length model).

	$P = 8$			$P = 16$			$P = 32$		
mesh	δ^2	δ^1/δ^2	δ^3/δ^2	δ^2	δ^1/δ^2	δ^3/δ^2	δ^2	δ^1/δ^2	δ^3/δ^2
crack	1.00	1.04	1.00	1.00	1.33	1.00	1.00	1.42	1.00
4elt	1.00	1.24	1.00	1.00	1.61	1.00	1.01	1.66	1.00
t60k	1.00	1.16	1.00	1.00	1.20	1.00	1.00	1.41	1.00
dime20	1.00	1.36	1.00	1.00	1.55	1.00	1.00	1.94	1.00
144	1.00	1.04	1.00	1.02	1.30	0.98	1.13	1.35	0.91
m14b	1.00	1.06	1.00	1.00	1.07	1.00	1.03	1.23	0.99
cyl3	1.00	1.36	1.00	1.00	1.97	1.00	1.01	2.40	1.00
mesh1m	1.00	1.15	1.00	1.00	1.35	1.00	1.00	1.33	1.00
Average		1.18	1.00		1.42	1.00		1.59	0.99

Table 2: Comparison of average path length for a 1d array for different network models: δ^2 is the quadratic path length model, δ^1 is linear path length and δ^3 cubic path length.

Table 2 shows a comparison of these three values of λ as follows. For each value of P, the first column shows δ^2, the average path length for the quadratic model ($\lambda = 2$), whilst the second & third columns show the average path length for the linear ($\lambda = 1$) & cubic ($\lambda = 3$) models, respectively, scaled by δ^2. Thus for the crack mesh and $P = 8$, the value $\delta^1/\delta^2 = 1.04$ indicates that the average path length for the linear model is 4% worse than that for the quadratic model. Since a figure of $\delta^\lambda = 1.00$ indicates complete success in the mapping task (since no message has to pass between non-neighbouring processors) we can see that the quadratic model is very successful with most values of $\delta^2 = 1.00$ or close to it. In fact the cubic model is marginally better (on average about 1% better for $P = 32$ as indicated by the average value of $\delta^3/\delta^2 = 0.99$). However, the linear model is considerably worse (on average 18% worse for $P = 8$ to 59% worse for $P = 32$). We believe that this is because cut-edges between non-neighbouring processors are not sufficiently penalised in the cost function.

Looking at the results in Table 3 (which are presented in the same format as Table 2 with the results of the linear and cubic models scaled by those of the quadratic) we see the consequences of the choice of model on the cut-weight. It is almost inevitable that the mapping task will be detrimental in some way to the cut-weight (this is particularly true for the 1d array architecture) and so we see conversely that the linear model which is not so good for mapping is considerably better for optimising the cut-weight (on average 10% better for $P = 8$ to 37% better for $P = 32$). On the other hand we can also see that the cubic model appears to enforce the mapping too rigidly and, as a result, ends up with a worse cut-weight than

mesh	$P=8$			$P=16$			$P=32$		
	Φ_2	Φ_1/Φ_2	Φ_3/Φ_2	Φ_2	Φ_1/Φ_2	Φ_3/Φ_2	Φ_2	Φ_1/Φ_2	Φ_3/Φ_2
crack	1179	1.00	1.03	2376	0.78	1.00	5167	0.68	1.01
4elt	1279	0.71	0.92	2430	0.64	1.15	5045	0.57	1.00
t60k	647	0.85	1.24	1353	0.81	1.14	3062	0.68	0.99
dime20	1023	0.94	1.13	2547	0.69	0.84	4555	0.66	1.14
144	47572	0.88	0.92	117826	0.64	0.90	205841	0.62	1.00
m14b	37792	1.29	1.14	76136	0.90	1.39	174113	0.75	0.92
cyl3	14905	0.75	1.04	29819	0.55	1.01	63577	0.41	1.05
mesh1m	22075	0.80	1.02	44841	0.73	1.18	86239	0.70	1.04
Average		0.90	1.05		0.72	1.08		0.63	1.02

Table 3: Comparison of cut-weight for a 1d array for different network models: Φ_2 is the quadratic path length model, Φ_1 is linear path length and Φ_3 cubic path length.

the quadratic model (on average between 8% worse for $P = 16$ to 2% worse for $P = 32$).

From these two tables (and other experimentation not reported here) we conclude that the model of network costs must have weights sufficiently large to heavily penalise communication across undesirable links but that enforcing this too rigidly can actually be detrimental to the partitioning without significantly enhancing the mapping. On this basis we use the quadratic path length model for the remainder of the experiments.

In the following sections we test the algorithms on 4 different classes of architecture (as illustrated in Figure 1) and on each architecture for 3 values of P, the number of processors. We informally notate each architecture as follows:

	$P=8$	$P=16$	$P=32$
1d array	8×1	16×1	32×1
2d array	4×2	4×4	8×4
cluster	2[4]	4[4]	8[4]
meta-computer	2[4]	2[8]	2[16]

In this notation the networks in Figure 1 can be described as (a) 6×1, (b) 6×4, (c) 4[4] and (d) 2[8].

In the corresponding NCMs for the 1d & 2d arrays, given as length × height, processor connections to the immediate left or right and up or down neighbours have a weight of 1. Any other connections between processors p & q say have a weight of l^2 where l is the minimum path length along edges of unit weight. For the cluster and meta-computer architectures, the notation $n[c]$ refers to n compute nodes each of c processors. Each compute node is a completely connected subgraph and any intra-node edges have a path length of 1. For each compute node one of the processors is nominated as being responsible for remote communications and all inter-node edges between two of these nominated processors have a path length of 2. The value of the corresponding entry in the NCM, n_{pq}, is once again defined as l^2 where l is the minimum path length between p and q.

4.3 Mapping results

In Tables 4-7 we show the mapping results for the 4 different architecture classes, 1d & 2d arrays, cluster and meta-computer. For each value of P we give the cut-weight, Φ, the average dilation, Δ, both as described in §4.1, and the partitioning time in seconds, τ. Although the cut-weight figures are fairly meaningless in isolation, we will see in §4.5 (by comparing them to those from the standard partitioning for cut-weight) that partitioning for network mapping does not impose to great a penalty on the cut-weight (e.g. about 25% for a 16 processor cluster and only around 12% for a meta-computer).

More interesting are the average dilation figures. For example, for a 1d array, an average dilation of $\Delta = 1.00$ would indicate complete success in the mapping task as no message would have to pass between non-neighbouring processors. In that respect we can see that the figures in Table 4 are very good indeed – in all cases for $P = 8$ and most for $P = 16$ the value for Δ is indeed 1.0. Obviously the mapping is more difficult as P increases (since the subdomains must become longer and thinner) but in only two cases for $P = 32$ (for the meshes 144 & m14b) does the value of Δ exceed 1.1.

mesh	$P=8$			$P=16$			$P=32$		
	Φ	Δ	τ	Φ	Δ	τ	Φ	Δ	τ
crack	1179	1.00	0.27	2376	1.00	0.48	5167	1.01	1.08
4elt	1279	1.00	0.25	2430	1.01	0.53	5045	1.02	1.35
t60k	647	1.00	0.57	1353	1.00	0.68	3062	1.00	1.53
dime20	1023	1.00	2.53	2547	1.00	2.88	4555	1.06	3.55
144	47572	1.00	5.17	117826	1.06	10.32	205841	1.49	18.03
m14b	37792	1.00	7.48	76136	1.01	8.95	174113	1.12	19.48
cyl3	14905	1.00	7.92	29819	1.00	14.13	63577	1.03	36.85
mesh1m	22075	1.00	24.03	44841	1.00	50.42	86239	1.00	65.90

Table 4: The results of the mapping algorithm for a 1d array architecture showing the cut-weight Φ, average dilation Δ and CPU time in seconds τ.

mesh	$P=8$			$P=16$			$P=32$		
	Φ	Δ	τ	Φ	Δ	τ	Φ	Δ	τ
crack	882	1.01	0.20	1527	1.01	0.27	2272	1.03	0.43
4elt	765	1.00	0.22	1481	1.02	0.35	2391	1.02	0.57
t60k	527	1.00	0.55	1240	1.00	0.68	1992	1.00	1.08
dime20	817	1.00	2.55	1457	1.00	2.72	3014	1.00	3.30
144	40159	1.02	5.07	61461	1.03	6.07	93511	1.06	9.43
m14b	37772	1.02	6.83	56920	1.02	8.05	102969	1.04	11.85
cyl3	9107	1.00	5.08	14697	1.00	7.52	22409	1.00	10.60
mesh1m	14021	1.00	20.95	29333	1.00	26.03	48926	1.00	36.73

Table 5: The results of the mapping algorithm for a 2d array architecture showing the cut-weight Φ, average dilation Δ and CPU time in seconds τ.

The 2d results in Table 5 are perhaps even better. Once again an average dilation of $\Delta = 1.00$ indicates complete success, however this is not easy to achieve for an unstructured mesh (as compared to a structured mesh with a simple stencil) as a result of the diagonal processor links. For the NCMs we have chosen, these diagonal links (i.e. those which run between neighbouring processors one link to the left or right of each other *and* one link up or down) have weight of 4 (the path length squared). Since it appears to be almost impossible to partition certain of these meshes so that no diagonal neighbours are adjacent across a subdomain interface, diagonal communication is bound to arise. However it can be seen from the dilation figures that this is kept relatively very low since the values for Δ never rise above 1.06.

mesh	$P=8$			$P=16$			$P=32$		
	Φ	Δ	τ	Φ	Δ	τ	Φ	Δ	τ
crack	940	1.67	0.20	1597	1.74	0.30	2781	1.87	0.47
4elt	811	1.51	0.22	1478	1.82	0.37	2531	1.84	0.57
t60k	581	1.46	0.50	1197	1.66	0.75	2161	1.78	1.08
dime20	922	1.79	2.47	1854	1.83	2.78	3190	1.78	3.33
144	35068	1.62	4.88	67452	1.92	7.38	106394	2.03	9.88
m14b	29580	1.48	6.45	65217	1.76	8.38	129350	1.94	13.70
cyl3	10157	1.73	5.02	16743	1.85	8.55	25842	1.87	12.57
mesh1m	19017	1.79	22.28	30860	1.74	28.88	59443	1.88	49.78

Table 6: The results of the mapping algorithm for a cluster architecture showing the cut-weight Φ, average dilation Δ and CPU time in seconds τ.

The cluster and meta-computer architectures (Tables 6 & 7) are a somewhat different prospect since there is a certain amount of traffic which must travel across the slower inter-node links (which are weighted with a value of at least 4 in the NCMs). However, in all cases but 1 (cluster architecture, $P = 32$, mesh 144) the average dilation is below 2.00 indicating the success of the mapping (contrast this with the unmapped figures in §4.5).

With regard to timings, as can be seen the mapping algorithm is very fast. Ever for mesh1m (with over a million vertices) it normally takes less than a minute and for the smaller meshes it can be just fractions of a second. The timings also give a hint as to the additional complexity of the problem. For example, the 1d

mesh	$P=8$			$P=16$			$P=32$		
	Φ	Δ	τ	Φ	Δ	τ	Φ	Δ	τ
crack	940	1.67	0.20	1506	1.45	0.30	2143	1.32	0.42
4elt	811	1.51	0.20	1203	1.38	0.35	1817	1.23	0.47
t60k	581	1.46	0.50	1045	1.26	0.65	1702	1.19	0.88
dime20	922	1.79	2.48	1594	1.46	2.58	2658	1.31	3.17
144	35068	1.62	4.85	53289	1.44	5.62	77251	1.39	7.83
m14b	29580	1.48	6.43	49699	1.24	7.42	75928	1.18	10.63
cyl3	10157	1.73	4.98	14893	1.54	7.20	19224	1.37	11.33
mesh1m	19017	1.79	22.35	29880	1.41	26.70	42338	1.28	33.32

Table 7: The results of the mapping algorithm for a meta-computer architecture showing the cut-weight Φ, average dilation Δ and CPU time in seconds τ.

array mapping is probably the most challenging and takes the longest to partition, whilst the 2d and meta-computer architecture, where the relative connectivity of the processor graph is much greater, are fastest to compute. Once again we will contrast this with a standard cut-weight partitioner in §4.5.

4.4 Preference tests

In Tables 8 & 9 we compare the different versions of the preference function as described in §3.4. Here we just consider their effect on the optimisation cost function, Γ, and partitioning time, τ, and, as an example, we focus on the cluster architecture. Table 10, meanwhile, summarises results for all the four architectures under scrutiny. Recall that the preference function selects which subdomain a vertex would prefer to migrate to and, if computed so as to take every possibility into consideration, results in an $O(P^2)$ operation which must be carried out many many times throughout the course of an optimisation. We denote the cost function & partitioning time for this full preference evaluation, f_P, by Γ_P & τ_P. However we have also (in §3.4) suggested two variants (approximations) with lower complexity, the adjacent subdomain preference, f_s, (as used in cut-weight optimisation) with metrics denoted by Γ_s & τ_s and the adjacent subdomain/processor preference, f_{sp}, a search over adjacent subdomains and processors adjacent in the processor graph (with metrics denoted by Γ_{sp} & τ_{sp}).

mesh	$P=8$			$P=16$			$P=32$		
	Γ_{sp}	Γ_s/Γ_{sp}	Γ_P/Γ_{sp}	Γ_{sp}	Γ_s/Γ_{sp}	Γ_P/Γ_{sp}	Γ_{sp}	Γ_s/Γ_{sp}	Γ_P/Γ_{sp}
crack	1567	1.04	0.99	2782	1.03	1.00	5208	1.14	0.98
4elt	1228	1.00	1.00	2690	1.21	1.06	4655	1.17	1.08
t60k	851	1.11	1.03	1986	1.15	0.97	3841	1.35	1.00
dime20	1648	1.37	1.02	3396	1.52	0.90	5674	1.31	0.97
144	56839	1.00	1.00	129839	1.02	1.03	216042	0.97	0.99
m14b	43875	0.98	0.99	114975	0.99	0.97	250616	0.99	0.92
cyl3	17531	1.09	1.08	31050	1.07	0.99	48372	1.19	1.06
mesh1m	34131	0.99	0.85	53741	1.14	1.04	111661	1.10	1.00
Average		1.07	1.00		1.14	1.00		1.15	1.00

Table 8: Comparison of mapping costs, Γ, for a cluster architecture for different preference functions: Γ_{sp} is where the preference is chosen from neighbouring subdomains & processors, Γ_s from neighbouring subdomains and Γ_P from all processors.

Table 8 shows the cost function results for the cluster architecture broadly presented in the same format as Table 2. For each value of P, the first column gives the results for f_{sp} whilst the second & third columns show those for f_s & f_P scaled by f_{sp}. Thus for the crack mesh & $P=8$, Γ is 4% worse for f_s than it is for f_{sp} ($\Gamma_s/\Gamma_{sp}=1.04$). However, it is the averages (bottom row) which show the overall trend of the results. For all values of P, both f_{sp} & f_P give the same cost function results on average (although there is up to 15% variation in the individual figures) indicating that the simplification of f_P to f_{sp} does not significantly affect the overall quality. However, simplifying further to f_s does impact on the quality (on average rendering it 7% worse for $P=8$ to 15% worse for $P=32$).

mesh	$P=8$			$P=16$			$P=32$		
	τ_{sp}	τ_s/τ_{sp}	τ_P/τ_{sp}	τ_{sp}	τ_s/τ_{sp}	τ_P/τ_{sp}	τ_{sp}	τ_s/τ_{sp}	τ_P/τ_{sp}
crack	0.20	1.10	1.85	0.30	1.10	4.17	0.47	0.96	12.66
4elt	0.22	1.00	1.45	0.37	1.00	3.84	0.57	0.96	14.60
t60k	0.50	1.04	1.40	0.75	0.89	2.56	1.08	0.97	11.00
dime20	2.47	1.04	1.13	2.78	0.97	1.45	3.33	0.94	4.90
144	4.88	1.01	1.91	7.38	1.01	2.54	9.88	1.05	7.39
m14b	6.45	0.99	1.43	8.38	0.97	2.69	13.70	1.01	5.85
cyl3	5.02	1.09	1.93	8.55	0.93	3.31	12.57	0.85	13.78
mesh1m	22.28	1.04	1.55	28.88	0.92	2.88	49.78	0.99	14.49
Average		1.04	1.58		0.97	2.93		0.97	10.58

Table 9: Comparison of mapping times, τ, for a cluster architecture for different preference functions: τ_{sp} is where the preference is chosen from neighbouring subdomains & processors, τ_s from neighbouring subdomains and τ_P from all processors.

Considering the timings shown in Table 9, however, we see that, as we would expect, f_{sp} with complexity $O(P)$ is considerably faster than f_P with its $O(P^2)$ complexity and that this difference is greatly exaggerated as P increases (e.g. the mapping is over 10 times faster on average for $P=32$). Interestingly, f_s, which is simpler again than f_{sp}, although generally faster can sometimes be slower (up to 15% for $P=32$ with the cyl3 mesh). We believe that this is because the algorithm is unable to produce such high quality partitions and thus the outer loop of the optimisation (see §2.2) takes longer to converge.

architecture	$P=8$		$P=16$		$P=32$	
	f_s/f_{sp}	f_P/f_{sp}	f_s/f_{sp}	f_P/f_{sp}	f_s/f_{sp}	f_P/f_{sp}
	partition costs, Γ					
1d array	1.23	1.02	1.65	0.96	1.71	0.99
2d array	1.07	1.00	1.05	1.00	1.10	1.01
cluster	1.07	1.00	1.14	1.00	1.15	1.00
meta-computer	1.07	1.00	1.11	0.98	1.11	0.98
	partitioning times, τ					
1d array	1.00	1.91	0.89	3.59	0.89	13.63
2d array	0.98	1.48	0.95	3.56	0.98	10.67
cluster	1.04	1.58	0.97	2.93	0.97	10.58
meta-computer	1.06	1.59	0.98	2.86	0.92	7.78

Table 10: Averages over all meshes of partition cost ratios and partitioning time ratios for different preference functions: f_{sp} chooses the preference from neighbouring subdomains & processors, f_s from neighbouring subdomains and f_P from all processors.

To demonstrate that these figures hold for all four of the architectures under consideration, Table 10 summarises each by presenting the averages (i.e. the final row of Tables 8 & 9 are duplicated). As before, the Γ_{sp} results are on average almost identical to the Γ_P results and are sometimes even better than them (1d array, $P=8$ & 2d array, $P=32$). However, the Γ_s results can be considerably worse, particularly for the 1d array (up to 71% worse on average for $P=32$). The timings also confirm those in Table 9; f_s is marginally faster overall than f_{sp} but f_P is much slower (over 13 times slower for the 1d array & $P=32$).

In summary then, these results demonstrate that not only is f_{sp}, the adjacent subdomain/processor preference function, a valid simplification of f_P, the full preference function, but also that f_P can be prohibitively expensive to use. Meanwhile f_s, the simplification that a cut-weight partitioner could use without performance degradation, does not produce mappings of the same quality and does not even appear to offer much of an advantage in terms of faster partitioning times. For these reasons, all the other results in this paper have been computed using f_{sp}.

4.5 Comparison with processor assignment

In this final section of results we compare the mapping algorithm with the two stage approach of partitioning for cut-weight followed by the mapping of the subdomains to processors (often known as processor

assignment). The partitioning algorithm is just the multilevel algorithm outlined in Section 2 (and fully described in [24]) whilst the processor assignment which seeks to map the subdomains graph onto the processor graph whilst minimising the cost is once again the Quadratic Assignment Problem described in §3.2 and uses the algorithm outlined there.

This type of two stage approach has been suggested previously (e.g. [26]) but, since the network costs are not taken into account during the partitioning stage, the subdomains are not 'shaped' so as to take account of the processor topology and the overall combination may be far from optimal.

The tests also give a good comparison of the mapping algorithm against standard cut-weight partitioning (with no consideration of network cost) since the assignment stage is very rapid (an $O(P)$ algorithm where the number of processors $P \ll V$, the number of graph vertices) and does not increase the partitioning time very much. In addition, the processor assignment algorithm does not change the cut-weight achieved by the partitioner (since it merely reassigns subdomains to processors). The cut-weight partitioner alone would therefore produce the same cut-weight results and marginally faster timings, but mapping costs & average dilation figures that are never better and may be considerably worse than if the processor assignment algorithm is used in addition.

	$P=8$		$P=16$		$P=32$	
mesh	Δ_p	$\Delta_p/Delta_m$	Δ_p	$\Delta_p/Delta_m$	Δ_p	$\Delta_p/Delta_m$
crack	3.86	2.31	4.15	2.39	5.76	3.08
4elt	3.44	2.28	4.85	2.66	4.76	2.59
t60k	2.40	1.64	3.95	2.38	4.84	2.72
dime20	3.58	2.00	3.84	2.10	4.08	2.29
144	4.23	2.61	5.79	3.02	6.38	3.14
m14b	3.79	2.56	4.03	2.29	5.85	3.02
cyl3	5.26	3.04	6.18	3.34	6.69	3.58
mesh1m	3.84	2.15	4.24	2.44	5.75	3.06
Average		2.32		2.58		2.93

Table 11: The average dilation for processor assignment, Δ_p, compared with that for mapping, Δ_m, on a cluster architecture.

Tables 11 & 12 focus on a comparison of results for the cluster architecture and show average dilation, Δ, & cut-weight, Φ. As previously the first column for each value of P shows the average dilation, Δ_p & cut-weight, Φ_p, respectively, for the combined partitioning/processor assignment algorithm, whilst the second column shows these figures scaled by the respective figures from Table 6, for the mapping algorithm (denoted Δ_m & Φ_m). Thus for the crack mesh and $P = 8$, the partitioning/processor assignment algorithm has average dilation 2.31 times worse than the mapping algorithm. In fact, on average the situation is worse and the average dilation is between 2.32 times worse for $P = 8$ and 2.93 times worse for $P = 32$ indicating that the mapping algorithm certainly makes a considerable difference.

	$P=8$		$P=16$		$P=32$	
mesh	Φ_p	Φ_p/Φ_m	Φ_p	Φ_p/Φ_m	Φ_p	Φ_p/Φ_m
crack	751	0.80	1191	0.75	1804	0.65
4elt	656	0.81	1012	0.68	1687	0.67
t60k	530	0.91	984	0.82	1588	0.73
dime20	636	0.69	1274	0.69	2282	0.72
144	28150	0.80	41842	0.62	60467	0.57
m14b	30663	1.04	45988	0.71	72997	0.56
cyl3	6798	0.67	10188	0.61	15179	0.59
mesh1m	13798	0.73	24522	0.79	35178	0.59
Average		0.81		0.71		0.63

Table 12: The cut-weight for processor assignment, Φ_p, compared with that for mapping, Φ_m, on a cluster architecture.

As stated previously mapping does have a negative impact on the cut-weight but, as we can see from Table 12, only fairly minimally for $P = 8$, i.e. just 19% on average, rising to around 37% for $P = 32$.

Tables 13 & 14 present the same information for the meta-computer architecture. In fact, with the relatively richer structure in the processor graph (recall that this features only two fully connected compute

mesh	$P=8$		$P=16$		$P=32$	
	Δ_p	$\Delta_p/Delta_m$	Δ_p	$\Delta_p/Delta_m$	Δ_p	$\Delta_p/Delta_m$
crack	3.86	2.31	2.83	1.95	2.48	1.88
4elt	3.44	2.28	2.94	2.13	2.20	1.79
t60k	2.40	1.64	2.02	1.60	1.75	1.47
dime20	3.58	2.00	2.51	1.72	1.99	1.52
144	4.23	2.61	3.62	2.51	2.65	1.91
m14b	3.79	2.56	2.05	1.65	1.90	1.61
cyl3	5.26	3.04	4.23	2.75	3.40	2.48
mesh1m	3.84	2.15	3.26	2.31	2.86	2.23
Average		2.32		2.08		1.86

Table 13: The average dilation for processor assignment, Δ_p, compared with that for mapping, Δ_m, on a meta-computer architecture.

mesh	$P=8$		$P=16$		$P=32$	
	Φ_p	Φ_p/Φ_m	Φ_p	Φ_p/Φ_m	Φ_p	Φ_p/Φ_m
crack	751	0.80	1191	0.79	1804	0.84
4elt	656	0.81	1012	0.84	1687	0.93
t60k	530	0.91	984	0.94	1588	0.93
dime20	636	0.69	1274	0.80	2282	0.86
144	28150	0.80	41842	0.79	60467	0.78
m14b	30663	1.04	45988	0.93	72997	0.96
cyl3	6798	0.67	10188	0.68	15179	0.79
mesh1m	13798	0.73	24522	0.82	35178	0.83
Average		0.81		0.82		0.87

Table 14: The cut-weight for processor assignment, Φ_p, compared with that for mapping, Φ_m, on a meta-computer architecture.

nodes rather than the multiple compute nodes of the cluster architecture), the differences between partitioning/assignment compared to mapping are less marked. Even so the average dilation figures are still around 2 times worse on average for partitioning/assignment, whilst the cut-weight figures only show approximately 13-19% degradation for the mapping algorithm.

Table 15 summarises these four previous tables and includes similar comparisons for the 1d & 2d arrays and for partitioning times. Here we can clearly see that the mapping algorithm has the greatest effect for the 1d array with its very sparse processor graph, particularly as P increases. For example the average dilation is over 15 times worse for $P = 32$. This, however, impacts on the cut-weight figures and mapping to a 1d array inevitably involves a greater increase in cut-weight than for other architectures (although this may not affect run-time or scalability of the underlying application, [18]). Similarly, since the mapping task is more complicated, the mapping algorithm takes on average about twice as long to run for $P = 32$. However, none of the other architectures with richer structure in the processor graph, exhibit such extreme results. Typically then we see that for architectures other than the 1d array the mapping algorithm can halve the average dilation compared to partitioning/assignment whilst adding only around 13-37% more cut edges. The partitioning times are comparable and indeed for $P = 8$ the mapping algorithm is faster on average although we have not yet been able to explain this properly.

5 Summary and future research

In this paper we have modified a multilevel algorithm to minimise a cost function based on a model of the (heterogeneous) communications network. This has been motivated by the increasing use of SMP clusters (systems of multiprocessor compute nodes with very fast intra-node communications but relatively slow inter-node networks) and the development of meta-computers (multiple supercomputers combined together, in extreme cases over inter-continental networks). The model of the communications network is supplied by the user at run-time and in this sense the technique is fairly generic since, if and when different architectures appear, the mapping algorithm should still apply and can be used simply by changing the

architecture	$P = 8$ assignment/ mapping	$P = 16$ assignment/ mapping	$P = 32$ assignment/ mapping
	average dilation		
1d array	4.28	8.01	15.52
2d array	1.96	2.57	3.43
cluster	2.32	2.58	2.93
meta-computer	2.32	2.08	1.86
	cut-weight		
1d array	0.63	0.50	0.38
2d array	0.84	0.77	0.73
cluster	0.81	0.71	0.63
meta-computer	0.81	0.82	0.87
	partitioning times		
1d array	0.95	0.71	0.53
2d array	1.06	0.97	0.85
cluster	1.08	0.88	0.75
meta-computer	1.11	1.00	0.91

Table 15: Averages of processor assignment compared with mapping for average dilation ratios, cut-weight and partitioning times.

network cost matrix.

The mapping algorithm is an adaptation of a standard multilevel partitioner (outlined in Section 3) with modifications to the initial partition (§3.2) and, in particular, the gain & preference functions (§3.3 & §3.4), to take account of network costs. The power of the process to compute such a mapping stems from the global properties of the multilevel algorithm. Edges which cross expensive links are penalised heavily within the cost function and so vertices at either end of such an edge tend to migrate to more adjacent processors (more adjacent to the processor owning the vertex at the other end of the edge) and create a sort of buffer zone. However, because this occurs high up in the multilevel process, where each vertex v represents many vertices in the original graph, the buffer zone which may start off only one vertex wide, can actually represent reasonably broad regions in the mesh. In this way the partition is given a good global quality on the coarse graphs which is refined on the finer graphs.

The algorithm was tested in a number of ways and initially we demonstrated (§4.2) that the network cost matrix requires weights sufficiently large to heavily penalise communication across undesirable links but that enforcing this too rigidly can actually be detrimental to the partitioning without significantly enhancing the mapping. On this basis we used the quadratic path length model.

Results for the mapping algorithm on 4 different classes of architecture were given (§4.3) and shown to provide low average dilation figures and fast run-times. Inevitably the mapping requirement tends to increase the cut-weight, but in fact, in comparison with a standard cut-weight version of the multilevel algorithm in combination with a processor assignment algorithm, the mapping algorithm gave average dilation results 2 times better with only an approximately 10-30% increase in cut-weight and similar run-times. We noted at several points that these differences tend to be more exacerbated for a sparse processor adjacency graph such as the 1d array and less so for networks with a richer communications structure.

Internally within the mapping algorithm we also tested three different preference functions (§4.4) and showed that the adjacent subdomain/processor preference function is a valid simplification and avoids the need for an $O(P^2)$ operation deeply embedded within the code.

The network cost matrices that we have used here to model the parallel architectures have been based on previous research which suggest that, for example, with certain latency dominated architectures a 1d mapping with its relatively high cut-weight can still provide the best application run-times and scalability, [18]. Meanwhile the hierarchical models are based on work which shows that parallel efficiency can be increased if all inter-node communications are handled by one processor per compute node, [6]. Nonetheless, having established that our algorithm can respond well to a given network cost matrix, the next step is to test the mappings with a genuine application on various networks to establish exactly how to weight the network cost matrices. However, for the reasons stated in §3.1 we do not believe that this process can be automated.

A further extension to this work would be to parallelise the mapping algorithm. In fact we do not believe

that this to be a very difficult task given that we have described three parallel optimisation algorithms (for use in the context of a multilevel partitioner) in [25]. Although slightly different in nature to the serial multilevel algorithm outlined in Section 2, all three also rely on the same gain & preference functions and we believe that it should be easy enough to simply insert the versions derived here (in §3.3 & §3.4).

References

[1] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice & Experience*, 6(2):101–117, 1994.

[2] T. N. Bui and C. Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In R. F. Sincovec *et al.*, editor, *Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.

[3] R. E. Burkard, E. Cela, P. M. Pardalos, and L. S. Pitsoulis. The Quadratic Assignment Problem. SFB-Report 126, Inst. Mathematik B, Tech. Univ. Graz, Austria, 1998.

[4] R. E. Burkard, S. E. Karisch, and F. Rendl. QAPLIB – A Quadratic Assignment Problem Library. *J. Global Optim.*, 10(4):391–403, 1997.

[5] R. E. Burkard and F. Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European J. Operations Research*, 17(2):169–174, 1984.

[6] J. Chen and V. Taylor. Mesh Partitioning for Distributed Systems: Exploring Optimal Number of Partitions with Local and Remote Communication. (submitted to 1999 SIAM Conf. Par. Processing Sci. Comp.), 1999.

[7] J. Chen and V. Taylor. ParaPART: Parallel Mesh Partitioning for Distributed Systems. In *Proc. Irregular '98: Solving Irregularly Structured Problems in Parallel, San Juan, Puerto Rico*, volume 1586 of *LNCS*. Springer, 1999.

[8] M. Dormanns and H.-U. Heiss. Mapping Large-Scale FEM-Graphs to Highly Parallel Computers with Grid-Like Topology by Self-Organization. Technical report, Dept. Informatics, Univ. Karlsruhe, D-76128 Karlsruhe, Germany, Feb 1994.

[9] C. M. Fiduccia and R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, IEEE, Piscataway, NJ, 1982.

[10] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed computing in a heterogeneous computing environment. In *Proc. Euro PVM/MPI '98, Liverpool*, 1998.

[11] A. Gupta. Fast and effective algorithms for graph partitioning and sparse matrix reordering. *IBM Journal of Research and Development*, 41(1/2):171–183, 1996.

[12] B. Hendrickson and T. G. Kolda. Graph Partitioning Models for Parallel Computing. To appear in *Parallel Comput.*

[13] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In S. Karin, editor, *Proc. Supercomputing '95*, New York, NY 10036, 1995. ACM Press.

[14] B. Hendrickson, R. Leland, and R. Van Driessche. Enhancing Data Locality by Using Terminal Propagation. In *Proc. 29th Hawaii Int. Conf. System Science*, 1996.

[15] B. Hendrickson, R. Leland, and R. Van Driessche. Skewed Graph Partitioning. In M. Heath *et al.*, editor, *Parallel Processing for Scientific Computing*. SIAM, Philadelphia, 1997.

[16] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[17] B. W. Kernighan and S. Lin. An Efficient Heuristic for Partitioning Graphs. *Bell Systems Tech. J.*, 49:291–308, February 1970.

[18] K. McManus, M. Cross, C. Walshaw, S. Johnson, and P. Leggett. A Scalable Strategy for the Parallelization of Multiphysics Unstructured Mesh-Iterative Codes on Distributed-Memory Systems. *Int. J. High Performance Comput. Appl.*, 14(2):137–175, 2000.

[19] K. McManus, C. Walshaw, M. Cross, P. Leggett, and S. Johnson. Evaluation of the JOSTLE mesh partitioning code for practical multiphysics applications. In A. Ecer *et al.*, editor, *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers*, pages 673–680. Elsevier, Amsterdam, 1996. (Proc. Parallel CFD'95, Pasadena, 1995).

[20] F. Pellegrini and J. Roman. Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping. TR 1038-96, LaBRI, URA CNRS 1304, Univ. Bordeaux I, 351, cours de la Libération, 33405 TALENCE, France, 1996.

[21] F. Pellegrini and J. Roman. SCOTCH : A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In H. Liddell *et al.*, editor, *High-Performance Computing and Networking, Proc. HPCN'96, Brussels*, volume 1067 of *LNCS*, pages 493–498. Springer, 1996.

[22] K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. In J. Dongarra *et al.*, editor, *CRPC Parallel Computing Handbook*. Morgan Kaufmann, 2000. (in press).

[23] J. D. Teresco, M. W. Beall, J. E. Flaherty, and M. S. Shephard. A hierarchical partition model for adaptive finite element computation. 1998.

[24] C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. To appear in *SIAM J. Sci. Comput.* (originally published as Univ. Greenwich Tech. Rep. 98/IM/35), 1998.

[25] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. To appear in *Parallel Comput.* (originally published as Univ. Greenwich Tech. Rep. 99/IM/44), 1999.

[26] C. Walshaw, M. Cross, M. Everett, S. Johnson, and K. McManus. Partitioning & Mapping of Unstructured Meshes to Parallel Machine Topologies. In A. Ferreira and J. Rolim, editors, *Proc. Irregular '95: Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *LNCS*, pages 121–126. Springer, 1995.