Applying Graph Partitioning Methods in Measurement-based Dynamic Load Balancing

HARSHITHA MENON, University of Illinois at Urbana-Champaign
ABHINAV BHATELE, Lawrence Livermore National Laboratory
SÉBASTIEN FOURESTIER, INRIA Bordeaux Sud-Ouest
LAXMIKANT V. KALE, University of Illinois at Urbana-Champaign
FRANÇOIS PELLEGRINI, INRIA Bordeaux Sud-Ouest

Load imbalance in an application can lead to degradation of performance and a significant drop in system utilization. Achieving the best parallel efficiency for a program requires optimal load balancing which is an NP-hard problem. This paper explores the use of graph partitioning algorithms, traditionally used for partitioning physical domains/meshes, for measurement-based dynamic load balancing of parallel applications. In particular, we present repartitioning methods that consider the previous mapping to minimize dynamic migration costs. We also discuss the use of a greedy algorithm in conjunction with iterative graph partitioning algorithms to reduce the load imbalance for graphs with heavily skewed load distributions. These algorithms are implemented in a graph partitioning toolbox called SCOTCH and we use CHARM++, a migratable objects based programming model, to experiment with various load balancing scenarios. To compare with different load balancing strategies based on graph partitioners, we have implemented METIS and ZOLTAN-based load balancers in CHARM++. We demonstrate the effectiveness of the new algorithms developed in SCOTCH in the context of the NAS BT solver and two micro-benchmarks. We show that SCOTCH based strategies lead to better performance compared to other existing partitioners, both in terms of the application execution time and fewer number of objects migrated.

1. INTRODUCTION

The efficient use of large parallel machines requires spreading the computational load evenly across all processors and minimizing the communication overhead. When the processes/tasks that perform the computation co-exist for the entire duration of the parallel program, the load balance problem can be modeled as a constrained graph partitioning problem on an undirected graph. The vertices of this *process graph* repre-

sent the computation to be performed and its edges represent inter-process communi-
cation. The problem of mapping these processes/tasks to processors can be viewed as
the partitioning and mapping of a graph of n tasks to that of p processors. The aim is
to assign the same load to all processors and to minimize the edge cut of the graph,
which translates to reducing communication among processors.

Although the problem of partitioning communicating tasks to processors appears
similar to that of partitioning large unstructured meshes to processes, the differences
are significant and lead to major algorithmic changes. The most significant difference
is that the number of tasks per processor is on the order of ten in load balancing,
whereas for meshes, the number of mesh elements per process is closer to a million.
The other difference is that the distribution of computational load associated with each
task can be highly uneven or skewed, which is not encountered often in more common
graph partitioning settings.

In this paper, we evaluate the deployment of static mapping and graph repartition-
ing algorithms, traditionally used for partitioning physical domains/meshes, for bal-
ancing load dynamically in over-decomposed parallel applications. We have chosen a
specific programming model, CHARM++ [Kalé and Krishnan 1993] and a graph parti-
tioning library, SCOTCH [scotch] for implementing the new algorithms and heuristics.
However, the techniques described here are generally applicable to other programming
models and other graph partitioning libraries [Hendrickson and Leland 1995; Karypis
and Kumar 1996a].

The CHARM++ runtime system includes a mature load balancing framework. The
runtime system records task loads for previous iterations to influence load balancing
decisions for the future and hence can adapt to slow or abrupt but infrequent changes
in load. There are existing load balancing strategies in CHARM++ and the framework
also facilitates adding new strategies. Measurement-based load balancing schemes in
this framework work well for applications in which computational loads tend to persist
over time [Bhatelé et al. 2009; Jetley et al. 2008]. We have developed two strategies
called ScotchLB and ScotchRefineLB based on different partitioning and repartition-
ing algorithms in SCOTCH. These are for comprehensive (fresh assignment of all tasks
to processors) and refinement load balancing, respectively. For comparison with other
graph partioners, we have implemented METIS and ZOLTAN based load balancers in
CHARM++.

In this paper, we discuss modifications to existing algorithms in SCOTCH that make
them more suitable for scenarios encountered in balancing the load in computational
science and engineering applications. This presents a distinct set of challenges com-
pared with mesh partitioning, which is what graph partitioners are usually designed
for. In addition to evaluating the classical recursive bipartitioning method in SCOTCH,
we discuss two new algorithms in this paper: 1. a k-way multilevel framework for
repartitioning graphs that takes the task migration cost into account and tries to min-
imize the time spent in migrations, and 2. a greedy algorithm for balancing graphs
with irregular load distributions and localized concentration of vertices with heavy
loads, a scenario which is not handled well by the classical recursive bipartitioning
method.

We present a comprehensive comparative evaluation of SCOTCH-based load bal-
ancers with those based on METIS and ZOLTAN and load balancing strategies (greedy
and refinement) in CHARM++. We evaluate the algorithms based on various metrics
for success: 1. execution time of the application, 2. time spent in load balancing, and
3. number of tasks migrated. We use two micro-benchmarks and the multi-zone ver-
sion of the NAS Block Tri-diagonal (BT) solver for comparisons and present results
from runs on Vulcan (IBM Blue Gene/Q), Intrepid (IBM Blue Gene/P) and Steele (an
Intel Infiniband cluster). New algorithms developed in SCOTCH lead to better perfor-

mance compared to other graph partitioners, both in terms of the application execution time and fewer number of tasks migrated. We also present capabilities in SCOTCH to handle different kinds of applications and the freedom given to the user/runtime system to decide if load balance or reducing communication among processors is more important. We also discuss the impact of application characteristics such as the ratio of computation to communication on the success of the load balancer.

The rest of the paper is organized as follows: Section 2 introduces measurement-based load balancing in CHARM++ and describes the existing load balancers in the framework. Section 3 presents existing and new partitioning algorithms developed in SCOTCH that are well suited for load balancing. A comparative evaluation of SCOTCH-based load balancers with other state-of-the-art algorithms is presented in Section 4. Section 5 discusses related work and Section 6 provides a summary of the paper.

2. DYNAMIC COMMUNICATION-AWARE LOAD BALANCING

An intelligent algorithm must consider both the characteristics of the parallel application as well as the target architecture when balancing load. The application information includes task processing costs (computational loads) and the amount of communication between tasks. The architecture information includes the processing speeds of the cores and the costs of communication between different cores and nodes. When the loads and communication patterns do not change during program execution, load balancing can be done statically at program startup. This is referred to as static or initial mapping because it is computed prior to the execution of the program and is never modified at run-time. However, if the load and/or communication patterns change dynamically, the mapping must be done at runtime (often called graph repartitioning or dynamic load balancing).

Graph partitioning has been used in the past to statically partition computational tasks to processors [Attaway et al. 1997; Shadid et al. 1997]. However, complex multi-physics simulations and heterogeneous architectures present a need for dynamic load balancing during program execution. This requires input from the application about the changing computational loads and communication patterns. The CHARM++ runtime system enables automatic dynamic load balancing through runtime instrumentation of the user code and by providing load balancing strategies. ZOLTAN [Çatalyürek et al. 2009] is a library for dynamic load balancing of parallel applications that uses hypergraph partitioners to balance entities indicated by the application.

2.1. The CHARM++ load balancing framework

Applications written in CHARM++ over-decompose their computation (in comparison to MPI) into virtual processors or objects called "chares" which are then mapped onto physical processors by the runtime system (shown in Figure 1). This initial static mapping can be changed as the execution progresses if the application suffers from load imbalance by migrating objects to other processors. This is facilitated by the load balancing framework in CHARM++ [Brunner and Kalé 2000]. Load balancing in CHARM++ is based on instrumenting the load from the recent past as a guideline for the near future, a heuristic known as the *principle of persistence* [Kalé 2002]. It posits that empirically, the computational loads and communication patterns of the tasks or objects *tend* to persist over time, even in dynamically evolving computations. Therefore, the load balancer can use instrumented load information to make load balancing decisions. The key advantage of this approach is that it is application independent and it has been shown to be effective for a large class of applications such as NAMD [Bhatele et al. 2008], ChaNGa [Jetley et al. 2008] and Fractography3D [Mangala et al. 2007].
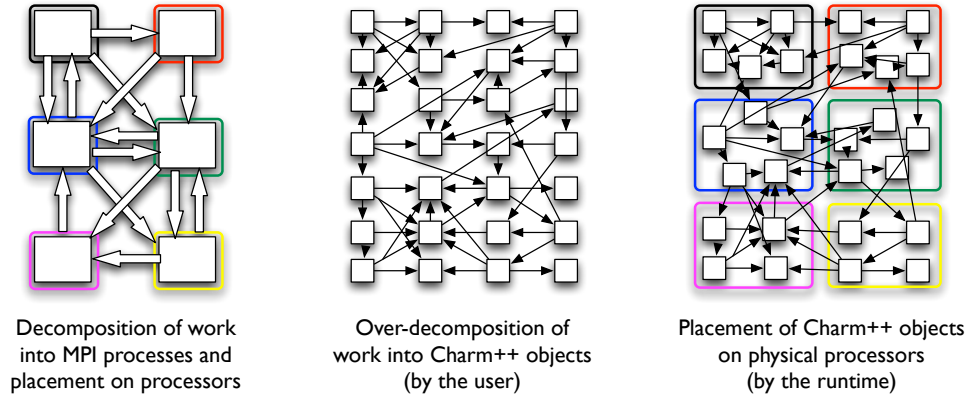
| Decomposition of work into MPI processes and placement on processors | Over-decomposition of work into Charm++ objects (by the user) | Placement of Charm++ objects on physical processors (by the runtime) |

Fig. 1: Charm++ system view with over-decomposition

2.2. Task and target graph

Charm++ runtime system does not depend on the application to provide the task graph and the cost models. It automatically instruments the time taken to execute a task, which is used as the vertex weight in the task graph. It also collects the communication pattern and volume, which can be used as edge weights by graph partitioning based load balancers. The measurement-based load balancing has the advantage of being accurate and requires little effort from the programmer.

Target graph represents the machine topology. The vertices are the processors and the edges are the network links. In SCOTCH, the distance between two process graph vertices mapped onto two distinct target graph vertices is the weighted shortest-path distance in the target graph between the two target vertices. For all the experiments in this paper, we consider the target graph to be a complete graph. Also note that SCOTCH handles only undirected graphs.

2.3. Load balancing strategies

There are several load balancing strategies built into CHARM++, some of which are used in this paper for comparison with SCOTCH-based load balancers. These load balancers are described below:

— **GreedyLB**: A "comprehensive" load balancer that does a fresh assignment of all tasks to processors. It is based on a greedy heuristic that maps the heaviest objects onto the least loaded processors iteratively until the load of all processors is close to the average load.
— **RefineLB**: A "refinement" load balancer that tries to minimize the number of migrations by considering the previous load balancing decisions. It migrates objects from processors with greater than average load (starting with the most overloaded processor) to those with less than average load.
— **MetisLB:** A strategy that passes the load information and the communication graph to METIS, a graph partitioning library, and uses the recursive graph bipartitioning algorithm in it for load balancing.
— **ZoltanLB:** A hypergraph partitioning based load balancer which uses ZOLTAN.

2.4. Load balancing datastructure

The CHARM++ runtime system encourages application developers to write application-specific load balancing strategies or use external libraries for the task. For this, it
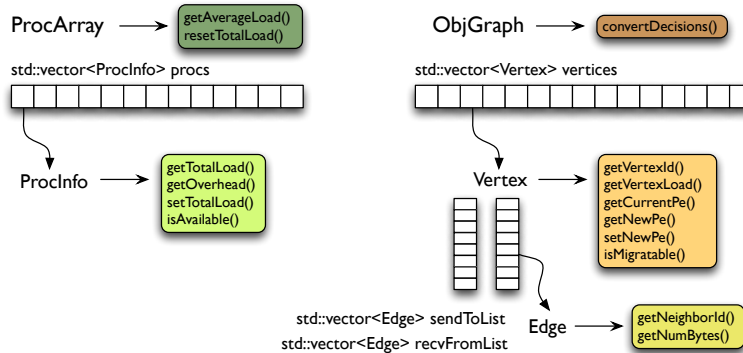
Fig. 2: A user friendly API for plugging in new load balancing strategies in CHARM++

provides an easy interface to write new load balancers. Figure 2 presents the data structures that provide useful information to a load balancing strategy for making migration decisions. The runtime system instruments a few time steps of the application before load balancing and this information is available in the form of two data structures. The ProcArray (on the left) provides the load on each processor for the previously instrumented time steps to identify the overloaded and underloaded processors in the application. The ObjGraph (on the right) is an adjacency list representation of the directed communication graph. The vector of vertices contains the load of each vertex in the graph. Each vertex also has pointers to two edge lists, one for the vertices it sends messages to and the other for those that it receives messages from. Each edge contains information about the number of messages and the total number of bytes exchanged between the vertex and one of its neighbors.

Using the information mentioned above, a load balancing strategy can be implemented that returns a new assignment for the vertices in the ObjGraph. This information is then used by the runtime system to migrate objects for the subsequent time steps. This setup facilitates the use of external load balancing strategies/libraries for measurement-based dynamic load balancing of parallel applications by hiding the mechanics and complexities of instrumentation and object migration.

3. SCOTCH FOR GRAPH PARTITIONING AND LOAD BALANCING

SCOTCH [scotch] is a software project developed jointly at the *Laboratoire Bordelais de Recherche en Informatique* of Université Bordeaux 1 and INRIA Bordeaux Sud-Ouest. Its goal is to provide efficient graph partitioning heuristics for scientific computing, and it is available to the community as a software toolbox. In this section, we will describe the existing mapping algorithms available in SCOTCH, the new repartitioning method implemented in SCOTCH as well as the new algorithm to reduce load imbalance.

3.1. Static mapping methods in SCOTCH

The two main classes of algorithms used to compute static mappings are direct k-way methods and recursive bipartitioning methods. Both k-way and bipartitioning implementations can take advantage of the multilevel graph partitioning paradigm, which helps reduce the problem complexity and execution time. The multilevel paradigm consists of three phases: graph coarsening, initial mapping, and uncoarsening. In the graph coarsening phase, the graph to partition is repeatedly coarsened into a series of smaller graphs. The graph at each step is derived from the previous graph by collapsing adjacent pairs of vertices. In the initial mapping phase, mapping is performed

on the coarsest graph. Finally, in the uncoarsening phase, the mapping of the coarsest graph is prolonged back to the original input graph [Barnard and Simon 1994; Hendrickson and Leland 1995]. After each uncoarsening step, the mapping is refined to improve the quality, using algorithms such as the Kernighan-Lin (KL) [Kernighan and Lin 1970b] and Fiduccia-Mattheyses (FM) [Fiduccia and Mattheyses 1982a] methods. These algorithms incrementally move vertices between parts, but cannot perform major changes to the projected partition. The KL algorithm minimizes the edge cut by performing swaps between pairs of vertices. Because it selects pairs of vertices, its time complexity is quadratic in the number of vertices. The FM algorithm is a modification of the KL algorithm that improves the time complexity without significantly decreasing the quality for most graphs. Instead of performing swaps, it moves vertices one at a time, from one part to another, so as to reduce edge cut while preserving load balance.

Since SCOTCH was initially designed to compute process-processor static mappings, it implements a modified recursive bipartitioning method called the Dual Recursive Bipartitioning (DRB) method [Pellegrini 1994]. This method uses a *divide and conquer* approach to recursively allocate processes to processors. In order to compute a mapping, an undirected process graph, that models the computation, is constructed. In the process graph, a vertex represents a process, vertex weight represents the computational load and the edges represent the communication between the processes. The set of processors onto which the processes are mapped is also modeled as an undirected graph, called the target graph. The DRB algorithm starts by considering a graph of processors, also called the domain, containing all the processors of the target graph. At each step, the algorithm bipartitions the domain into two disjoint subdomains, and calls a graph bipartitioning algorithm to partition the process graph onto the two subdomains. Initial bipartitions are computed using locality-preserving greedy *graph growing* heuristics, that grow parts from randomly selected seed vertices [Karypis and Kumar 1996a]. Initial partitions are refined using the FM algorithm, possibly using the multilevel framework when graphs to bipartition are big enough. In this case, the FM algorithm is applied to *band graphs*, that is, restrictions of the graphs to a small number of layers of vertices around the prolonged partition [Pellegrini 2007], so as to reduce problem space.

The objective of the mapping process is to obtain the load balance within the specified imbalance threshold while minimizing the communication cost. The imbalance metric for bipartioning is given by:

$$|w_{P_0} - w_{P_1}|$$

where w_{P_i} is the sum of the load of the vertices in part P_i. We chose this metric because it can be incrementally computed when moving a vertex from P_0 to P_1. Similarly, the imbalance metric for k-way partitioning is:

$$\frac{\sum_{i=1}^{n} |w_{avg} - w_{P_i}|}{w_V}$$

where w_V is the sum of the load of all the vertices and w_{avg} is the average load per partition.

In its most recent version, SCOTCH performs k-way multilevel coarsening down to a size of 20 vertices per part, after which the DRB algorithm is called to compute the initial partition on the coarsest k-way graph. Then, the k-way partition is refined using a k-way version of the FM algorithm. Although these algorithms are fast, we will see in Section 3.3 that FM may result in high imbalance for graphs with irregular load
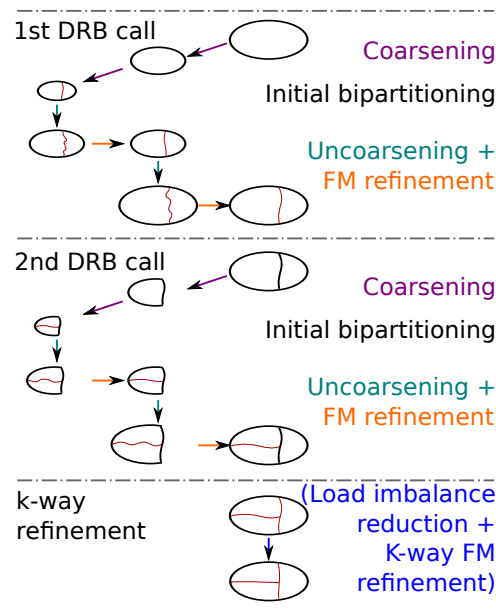
Fig. 3: Three phase partitioning in SCOTCH

distributions. Such graphs require a new algorithm for reducing load imbalance (see Section 3.3).

Depending on the load balancing needs, two flavors of the SCOTCH static mapping method can be used:

(1) STRAT_QUALITY - It gives preference to obtaining the best possible edge cut over minimizing load imbalance. In the DRB algorithm, for each bipartition, the complete multilevel bipartitioning V-cycle is performed thrice, and the best result is used.

(2) STRAT_BALANCE - It gives preference to minimizing load imbalance even if it increases the edge cut. Here, after each bipartitioning V-cycle, an extra FM pass is performed on the whole graph (and not only on the band graph around the cut). Once the k-way partition has been computed on the finer graph, a specific k-way load imbalance reduction algorithm is called (described in Section 3.3). Then, a final k-way FM algorithm is called in order to reduce communication as much as possible.

Please note that for all the applications used in this paper, the size of the process graphs is typically less than 20 vertices per part. Hence, the DRB algorithm is used instead of the k-way multilevel framework for computing the initial k-way partition. However, for STRAT_BALANCE, a k-way refinement algorithm is invoked on the finest graph. Figure 3 shows how these algorithms are called in sequence on such graphs.

3.2. Repartitioning methods in SCOTCH

A new feature added to SCOTCH 6.0 is the ability to compute a remapping of a graph, based on an existing mapping. This scheme is referred to as refinement load balancing in CHARM++, which aims at reducing the number of migrations resulting from load balancing. As in [Çatalyürek et al. 2009], the main idea is to add a fictitious edge to every vertex of the graph that connects it to a fictitious vertex representing the old part. An example of repartitioning with fictitious edges is shown in Figure 4. All the

(a) Old partition.

(b) New optimal partition computed from scratch.

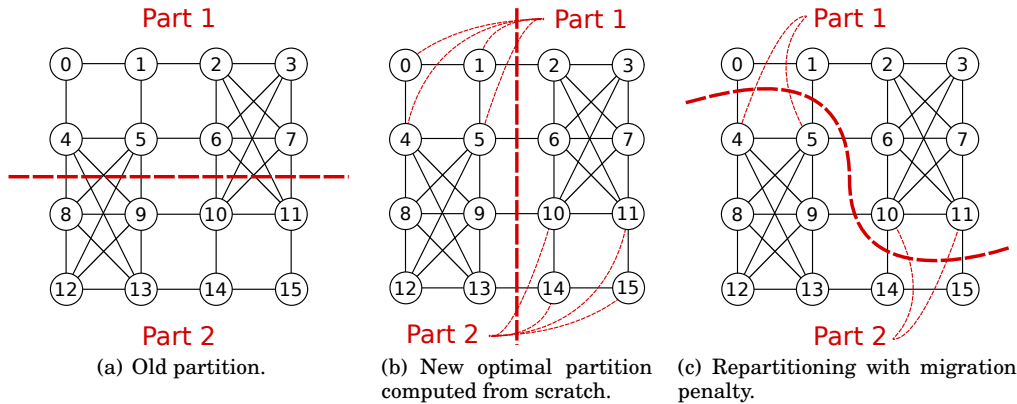(c) Repartitioning with migration penalty.

Fig. 4: Example of repartitioning with fictitious edges. Considering that all edge loads and migration costs are equal to 1, subfigure (a) shows an old partition with a cut of 12. Subfigure (b) shows a new partition computed from scratch with a cut equal to 4. By considering additional migration penalty, the cost of partition (b) increases up to 12 (4 from edge loads plus 8 from vertex migrations). The partition shown in subfigure (c) is optimal with a cost of 10 (6 from edge loads plus 4 from vertex migrations).

fictitious edges are assigned weights corresponding to the cost of migrating the vertex to another part. This method allows one to integrate, without much modification, the migration cost into existing edge cut minimization software. However, it can be expensive, because copying and processing graph data to add extra edges can be costly.

In order to reduce the overhead of adding fictitious edges, we take advantage of the structure of SCOTCH and reduce the need for such edges. Firstly, in the k-way coarsening phase, we match together only vertices that have the same characteristics, e.g. that belong to the same old part (for repartitioning). This requires us to add only one data per vertex, instead of one per fictitious edge. When the current graph to partition does not differ too much topologically from the old graph that gave the old partition, the multilevel k-way coarsening process is not hindered too much. Secondly, since SCOTCH was initially designed to compute process-processor static mappings using recursive bisection, its recursive bisection algorithms allow for biases in the cut cost function, so that vertices can have a preferred part. To compute the initial partition, biases are added to each vertex according to the old partition. Hence, the initial partition accounts for all of these constraints, without having to add any extra edges. Note that, unlike the k-way coarsening algorithm, the coarsening algorithm of the DRB framework can mate with any neighbor vertex, by summing its bias. Hence, even if k-way coarsening could not operate properly due to the lack of mates with same characteristics, the multilevel framework can still be used at the DRB level. Finally, in the FM algorithm on the band graphs of the k-way uncoarsening phase, the extra migration cost is taken into account in the cost function that is considered when we are choosing which vertex to move. This global, minimal use of fictitious edge is novel as a whole, and allows us to handle fixed vertices as well.

3.3. A greedy algorithm for reducing load imbalance

All of the algorithms mentioned above were designed for graphs whose vertex load distribution is regular and not severely imbalanced. In particular, it is assumed that it is always possible to achieve load balance by a sequence of moves involving the ver-

---

ALGORITHM 1: Pseudo-code of the load imbalance reduction algorithm

---

Inputs : V, the set of vertices
$\qquad$ P, the collection of parts
$\qquad$ weight, an array containing the weight of each vertex
$\qquad$ partition, an array containing the part of each vertex
$\qquad$ load, an array containing the load assigned to each part
$\qquad$ average_load, $\sum_v$ weight$[v]$/size(P) where $v \in V$
Output: partition, the updated part array
Initialize load array of dimension size(P) to all zeroes

for $v \in V$ *by decreasing order of* weight$[v]$ **do**
$\quad$ best_imbalance $\leftarrow +\infty$
$\quad$ **for** $p \in P$ *in DRB order* **do**
$\quad\quad$ **if** load$[p]$ + weight$[v]$ $<=$ average_load **then**
$\quad\quad\quad$ best_partition $\leftarrow p$
$\quad\quad\quad$ break
$\quad\quad$ **else**
$\quad\quad\quad$ imbalance $\leftarrow$ |load$[p]$ + weight$[v]$ − average_load|
$\quad\quad\quad$ **if** imbalance $<$ best_imbalance **then**
$\quad\quad\quad\quad$ best_imbalance $\leftarrow$ imbalance
$\quad\quad\quad\quad$ best_partition $\leftarrow p$
$\quad\quad\quad$ **end**
$\quad\quad$ **end**
$\quad$ **end**
$\quad$ partition$[v]$ $\leftarrow$ best_partition
$\quad$ load[best_partition] $\leftarrow$ load[best_partition] + weight$[v]$
end

---

tices. However, when the load distribution is very irregular, such algorithms may fail to provide adequate load balance. Load distribution artifacts may not be compensated if, for instance, some vertices with very high loads are localized in a small, strongly coupled, portion of the graph. These vertices will most likely be kept together by the first levels of the recursive bipartitioning algorithm. Due to the high granularity of the vertex load, the bipartitioning algorithm creates partitions that are highly imbalanced. Moreover, since the FM algorithm uses vertex movements instead of vertex swaps as in the KL algorithm, movements of the heaviest vertex will never be considered. This is because moving a heavy vertex out of its slightly overloaded part may result in overloading the destination part as well as underloading the original part.

To address this problem, a greedy load imbalance reduction algorithm, presented in Algorithm 1, has been implemented in SCOTCH. This algorithm is not novel but we use it along with the graph partitioning algorithm to specifically handle heavy load imbalance. It is activated when the load imbalance ratio of the k-way partitioning is above a specified threshold. The main loop of the algorithm considers all vertices in descending order of their weights. If adding the vertex to its current part does not overload the part, then the vertex is not moved. But if the vertex causes its original part to be overloaded, possible alternate destination parts are tried out in the target domain in recursive bipartitioning tree order. The neighboring domain of the last bipartition level is tried first followed by the two children of the neighboring domain in the second-last level, and so on. Therefore, closest domains in the target architecture partition are considered first, before farther ones, taking into account the topology of the target architecture. Once a balanced partition is achieved, k-way FM is used to minimize the communication cost.

In summary, the algorithms that have been experimented with in this paper are: (i) the classical dual recursive bipartitioning (or static mapping) method of SCOTCH adapted for repartitioning, (ii) a new k-way multilevel framework for partitioning and repartitioning graphs. Due to the small size of the application graphs only the k-way FM algorithm is used, and (iii) a greedy algorithm along with graph partitioning algorithm for handling graphs with irregular load distributions and localized concentration of vertices with heavy loads.

In the context of this paper, the target topologies are assumed to be homogeneous, and remapping reduces to repartitioning.

3.4. Comparison with other partitioning tools

Two third-party tools are used for comparison with the methods developed in this paper: METIS and ZOLTAN. We are going to present their most salient features.

The METIS library performs sequential graph partitioning and PARMETIS performs both parallel graph partitioning and repartitioning. Graph repartitioning feature is not available in the sequential version. Also, METIS does not perform static mapping. Its sequential version is based on a k-way multilevel framework, in which graphs are coarsened by means of heavy-edge matching. Following this, an initial k-way partition is computed by means of recursive bipartitioning. Finally, this partition is projected back to finer graphs and refined using a variant of gradient k-way Fiduccia-Mattheyses algorithm. In this algorithm, boundary vertices are sorted according to external degree. Vertices that improve the cut or communication volume and that do not create imbalance are moved to their most beneficial neighboring part. The major difference between the METIS library and SCOTCH is that unlike METIS, SCOTCH includes static mapping capabilities and uses global algorithms such as diffusion-based methods to refine the partition.

ZOLTAN is a parallel partitioning tool for graph and hypergraph partitioning and for sparse matrix ordering. In addition to providing its own algorithms, it encapsulates many third-party tools such as PATOH [Catalyürek and Aykanat 1999], PARMETIS and even SCOTCH. Its most interesting feature in the context of our paper is that, it provides a hypergraph repartitioning feature using fictitious vertices and edges [Çatalyürek et al. 2009]. ZOLTAN is also based on a multilevel framework. Hypergraphs are coarsened using a variant of the heavy edge matching heuristic, called *inner-product matching*. Initial partitions are computed using recursive bisection. Local refinement is performed using a localized version of the Fiduccia-Mattheyses algorithm. Since graphs are degenerate hypergraphs with hyperedges comprising of only two vertices, it is quite straightforward to use ZOLTAN in our experimental framework. For our experiments, we use it in degenerate mode. Since hypergraph partitioning is known to be more expensive than plain graph partitioning, we do not expect good performance in terms of speed, but its quality is interesting to analyze.

Parallel versions of the partitioners are MPI-based and require interoperability between Charm++ and MPI. There are various challenges involved with this interoperation which has been studied in detail recently [Jain et al. 2015]. Therefore, in the context of this paper, we consider only the sequential versions of the partitioning tools.

4. CASE STUDIES

We compare the performance of different load balancing strategies using microbenchmarks and the NAS BT solver on multiple machines. Table I presents an overview of the benchmarks in terms the maximum number of neighbors, maximum and minimum communication volume between tasks and load imbalance as described by equation 1. For all the experiments, the target graph is a complete graph and the task graph is provided by Charm++ runtime system. The vertex weights are obtained

	kNeighbor	BT_MZ	stencil4d
Max Neighbors	8	4	8
Max Edge Weight (Bytes)	32768	687744	32768
Min Edge Weight (Bytes)	32768	76224	32768
Load Imbalance (max/avg)	5	20	8

Table I: Characteristics of the benchmarks

by instrumenting the time taken to execute the task and the edge weights are obtained by collecting the communication pattern and volume.

The experiments were run on Vulcan, Intrepid and Steele. Vulcan is a $24,576$ node 5 Petaflop BG/Q production system at Lawrence Livermore National Laboratory with 5D torus chip-to-chip network. Each node consists of 16 1600 MHz PowerPC A2 cores each with 4 hardware threads. Intrepid is a $40,960$ node Blue Gene/P installation at the Argonne National Laboratory. Each node on Intrepid consists of four 850 MHz PowerPC cores. The primary interconnect for point-to-point communication in this system is a 3D torus with a bi-directional link bandwidth of 850 MB/s. The experiments were run in VN mode using all four cores per node. Steele is a Dell cluster at Purdue University, operated by the Rosen Center for Advanced Computing. Each node on Steele has two quad-core 2.33 GHz Intel E5410 chips or two quad-core 3.00 GHz Intel E5450 chips. The interconnect used is Gigabit Ethernet or InfiniBand for different nodes.

In section 4.1, we discuss the different load balancers used for performance evaluation. Section 4.2 and 4.3 present comparisons of the load balancers using kNeighbor micro-benchmark and NAS BT_MZ benchmark respectively. We evaluate the advanced features available in SCOTCH based load balancers in section 4.4. In section 4.5, we compare the performance of various load balancers by varying the ratio of computation to communication.

4.1. Evaluation of SCOTCH-based load balancers

We implemented two load balancing strategies in CHARM++ that use graph partitioning methods available in SCOTCH. The first one, ScotchLB, does a fresh partitioning and assignment of objects to processors ignoring the previous mapping. The second one, ScotchRefineLB, uses repartitioning methods (section 3.2) to refine the initial partitioning and mapping created by ScotchLB. For ScotchRefineLB results, ScotchLB is invoked once, when program execution begins, followed by several calls to ScotchRefineLB. We compare the performance of SCOTCH-based load balancers with those in CHARM++, GreedyLB and RefineLB. For comparison with other graph partitioners, we also implemented METIS and ZOLTAN-based load balancers. For MetisLB, both recursive bipartitioning and k-way multilevel partitioning were used. For ZoltanLB, the *hypergraph* partitioner is used with the *partition* and *re-partition* method for kNeighbor. Finally, for SCOTCH-based load balancers, two flavors of the mapping methods were tried, namely STRAT_QUALITY and STRAT_BALANCE (section 3.1).

The following metrics are used to compare the performance of the load balancing algorithms:

(1) Speedup obtained in the execution time per step of the application, which is the best indication of the success of a load balancer.
(2) Time spent in the load balancing strategy and migration. This, along with the frequency of load balancing, determines whether load balancing is beneficial.
(3) Number of objects migrated, signifying the amount of data movement resulting from load balancing and hence the associated communication costs.

kNeighbor on Vulcan: Ratio of remote to local comm.   kNeighbor on Vulcan: Number of migrations

(a) Ratio of remote to local communication   (b) Number of migrations per load balancing step
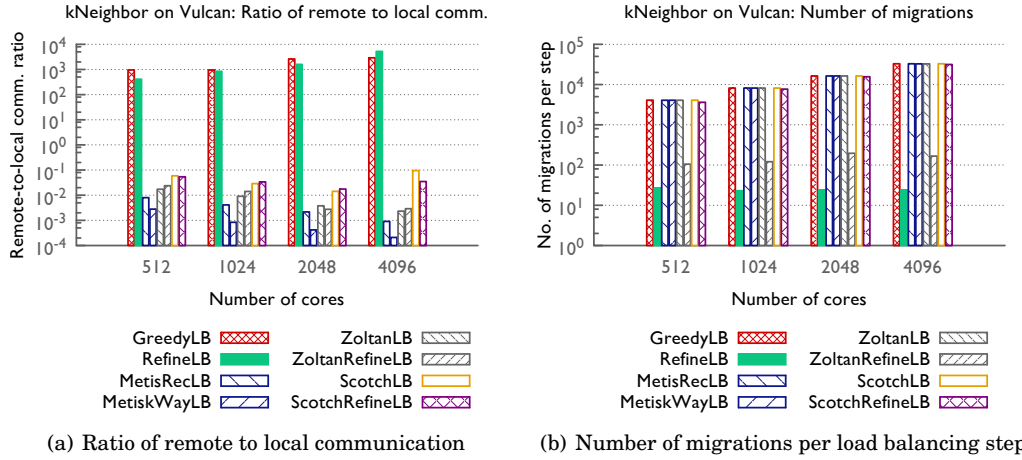
Fig. 5: Evaluation of kNeighbor benchmark

(4) Speedup obtained in the total application time, which includes the time for the iterations, load balancing strategy and migration.

Here we consider speedup with respect to the baseline where no load balancing is done (*NoLB*), and it is defined as the ratio of the time for *NoLB* to that for a specific strategy.

4.2. Comparisons using kNeighbor

kNeighbor is a micro-benchmark with a near-neighbor communication pattern. In this benchmark, each object exchanges $32KB$ sized messages with a fixed set of objects in every iteration. The communication pattern of the objects can be a disconnected graph. Each object is assigned computational load such that less than 3% processors are heavily loaded. The load of heavy objects can be up to 5 times the average load of the objects in the system.

In this section, we present results for the kNeighbor micro-benchmark running on Vulcan. For these experiments, the number of objects is eight times the number of processors. The baseline experiment is referred to as *NoLB*, where, no load balancing is performed and the runtime system does a static mapping of all objects to processors, attempting to assign equal number of objects to each processor.

Figure 5(a) demonstrates the capability of graph partitioning based load balancers in mapping communicating objects to the same processor. Communication between objects on the same processor is called local, whereas, between objects on different processors is considered remote. This figure presents the ratio of remote to local communication for different load balancers. We can see that graph partitioners succeed in maintaining this ratio less than one i.e. restricting communication to within a processor as much as possible. In contrast, for other load balancers, this ratio is at the least five orders of magnitude higher denoting excess of remote communication.

Figure 5(b) presents the number of migrations as a result of performing load balancing. RefineLB, which considers the existing mapping, successfully reduces the number of migrations. However, there exists a tradeoff between reducing the number of migrations and improving application performance. RefineLB, which performs the least number of migrations, suffers from performance issues as it migrates without taking the communication of the objects into account. ZoltanRefineLB is also able to considerably reduce the migration but we will see in Figure 7(b) that its performance is lesser

(a) Time taken by migration
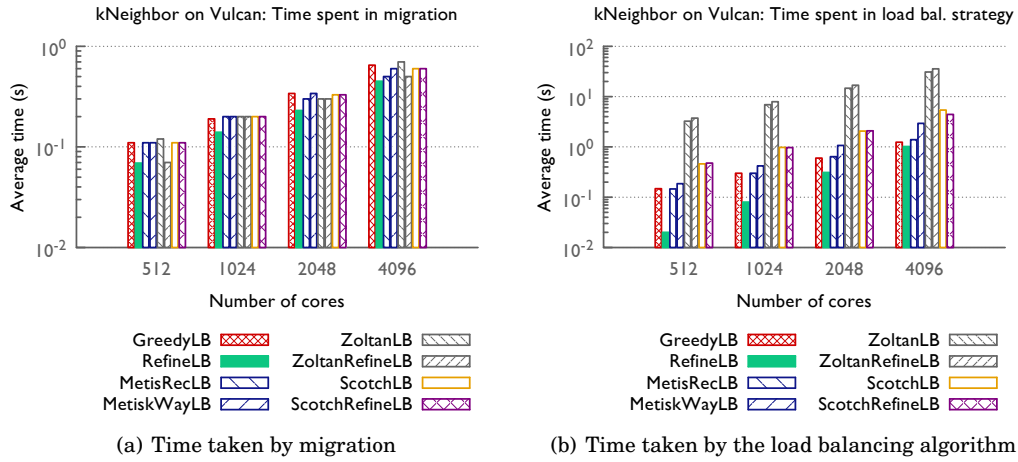
(b) Time taken by the load balancing algorithm

Fig. 6: Load balancing cost per load balancing step for kNeighbor

in comparison ZoltanLB. For this benchmark, ScotchRefineLB is not able to reduce the number of migrations because repartitioning of disconnected graphs is more challenging.

Next we compare the time spent in the load balancing strategy, which includes the time for migration and the load balancing strategy (see Figure 6). We find that, as the number of cores increases, the strategy time as well as the migration time for all the load balancers increases. In some cases RefineLB and ZoltanRefineLB has the least migration time because it is successful in reducing the number of migrations in comparison to other strategies. RefineLB also has the smallest strategy time among all but does not improve performance. Graph partitioning based load balancers, in general, incur a higher cost. This overhead, however, is not significant and is offset by the better performance that the application achieves when using them.

An indicator of load imbalance in the system is the ratio of maximum load to average load. Therefore, we define the load imbalance metric I to be:

$$I = \frac{L_{max}}{L_{avg}} \tag{1}$$

Figure 7(a) shows the value of the load imbalance metric, I, for different load balancing strategies. We see that GreedyLB and RefineLB, which consider only the computation load, have a value of I around 1.05. Metis and Zoltan based load balancers have higher I value of 1.5 and above. SCOTCH based load balancers achieve an imbalance metric value of 1.05 similar to that of GreedyLB and RefineLB while also taking the communication into account. As a result, in Figure 7(b) we can see that the load balancers based on SCOTCH consistently outperform all the other load balancers. Since the process graph for the kNeighbor micro-benchmark is a disconnected graph with high imbalance of vertices, the traditional graph partitioning algorithms suffer from load imbalance. The high speedup of SCOTCH based load balancers, in comparison to others, is due to the addition of the greedy algorithm for reducing load imbalance. We found that the STRAT_BALANCE strategy of SCOTCH with an imbalance tolerance of 5% gave the best performance.

(a) Load imbalance

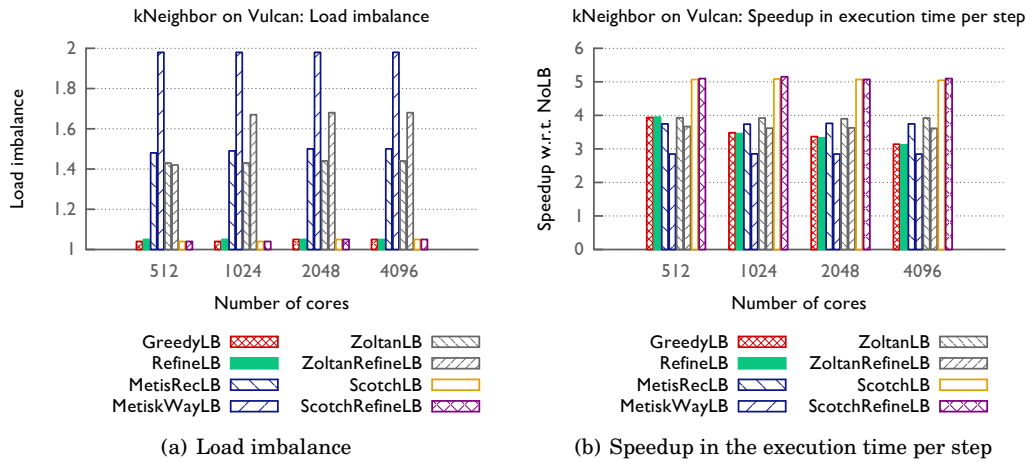(b) Speedup in the execution time per step

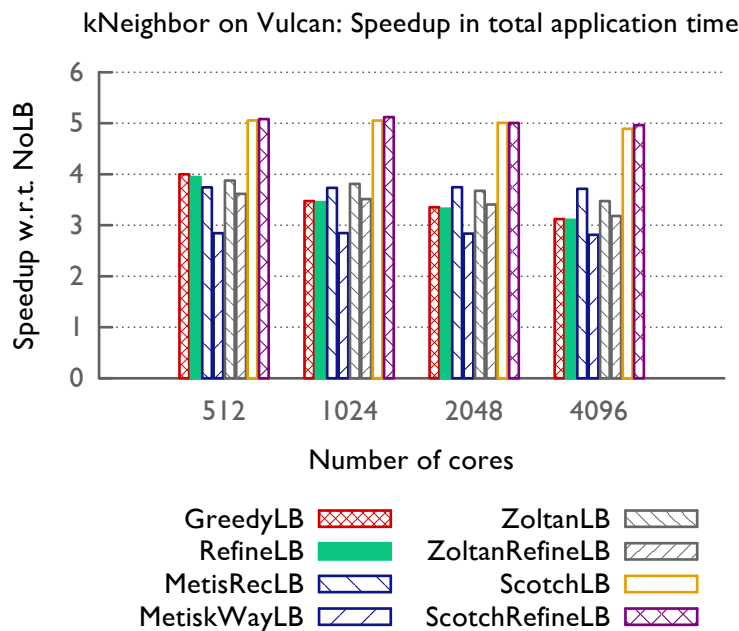Fig. 7: Load imbalance and Speedup for kNeighbor on Vulcan

Fig. 8: Speedup in the total execution time

Finally, Figure 8 presents results for an end-to-end execution of the kNeighbor micro-benchmark. For these runs, we perform load balancing once every 500 iterations. The application time is the sum of the times for all the iterations and the load balancing time, which includes the strategy time and the time for migration of objects. We can see in Figure 8 that the ScotchLB and ScotchRefineLB consistently obtain the best performance on all system sizes. ScotchLB and ScotchRefineLB gives a reduction

(a) Ratio of remote to local communication

(b) Number of migrations per load balancing step
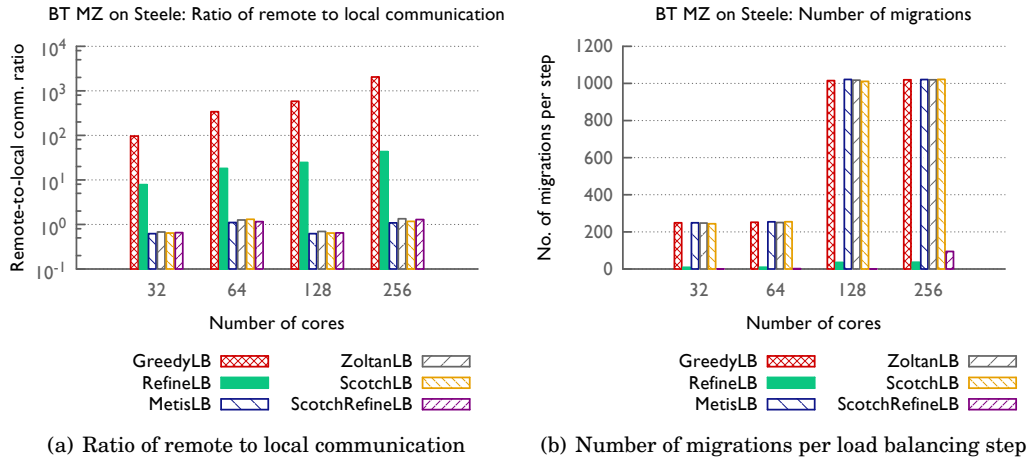
Fig. 9: Evaluation of BT_MZ on Steele

in total execution time of up to 25% in comparison to the best among other load balancers. When compared to the baseline, NoLB, ScotchLB and ScotchRefineLB give 80% reduction in execution time (overall speedup of up to 5).

4.3. Comparisons using BT_MZ

BT_MZ is the multi-zone version of the Block Tri-diagonal (BT) solver in the NAS Parallel Benchmark suite (NPB) [der Wijngaart and Jin 2003]. It is a parallel implementation for solving a synthetic system of non-linear PDEs using block tri-diagonal matrices. It consists of uneven sized zones within a problem class and hence is useful for testing the effectiveness of a load balancer. The ratio of largest to smallest zone is approximately 20. In this paper, we compare the performance of various load balancers for class C and class D of BT_MZ in NPB 3.3. For class C, the benchmark creates a total of 256 zones, and for class D, it creates 1024 zones, with an aggregated grid size of $480 \times 320 \times 28$ and $1632 \times 1216 \times 34$, respectively. Boundary values between zones are exchanged after each iteration.

In this section, we present results for the BT_MZ performance on Steele. For these experiments, the number of objects per processor varies with the class of the benchmark used and the system size. As an example, a class D run on 256 processors will have, on average, four objects per processor. We ran class C on 32 and 64 cores and class D on 128 and 256 cores. For these experiments, we compare the performance of SCOTCH based loadbalancers with GreedyLB, RefineLB, MetisLB and ZoltanLB. For MetisLB, we use both k-way and recursive-bisection and report the result of the better performing strategy.

Figure 9(a) presents the ratio of remote to local communication for different load balancers for BT_MZ. As in the case of kNeighbor, the graph partitioners obtain a ratio close to one. This reduces the out-of-processor communication in comparison to other load balancers. Figure 9(b) shows the number of migrations. The amount of data to be transferred, when an object is migrated, is substantially large in case of BT_MZ. Refinement-based load balancers, RefineLB and ScotchRefineLB which take the migration cost into account, migrate very few objects. Hence, they spend a significantly smaller time in migration, as seen in Figure 10(a), nearly an order of magnitude smaller than other balancers, in some cases.
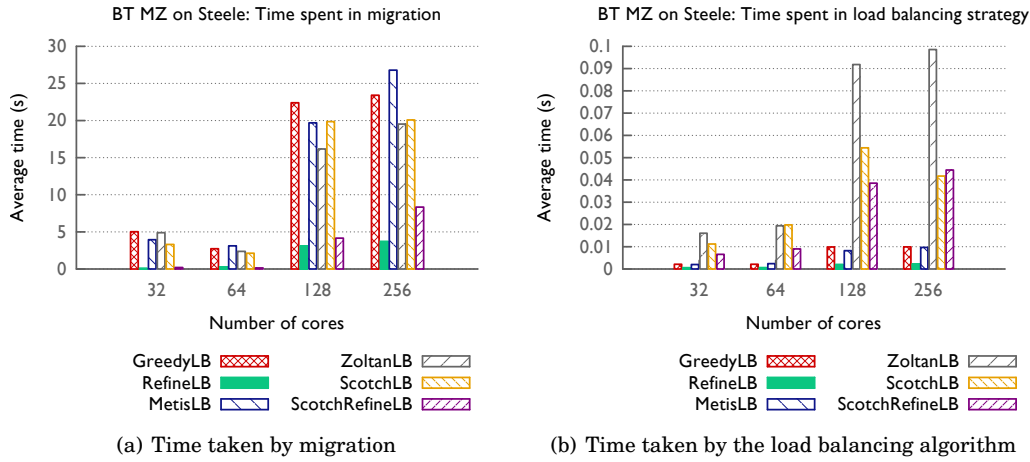
(a) Time taken by migration

(b) Time taken by the load balancing algorithm

Fig. 10: Load balancing cost per load balancing step for BT_MZ on Steele

(a) Speedup in the execution time per step
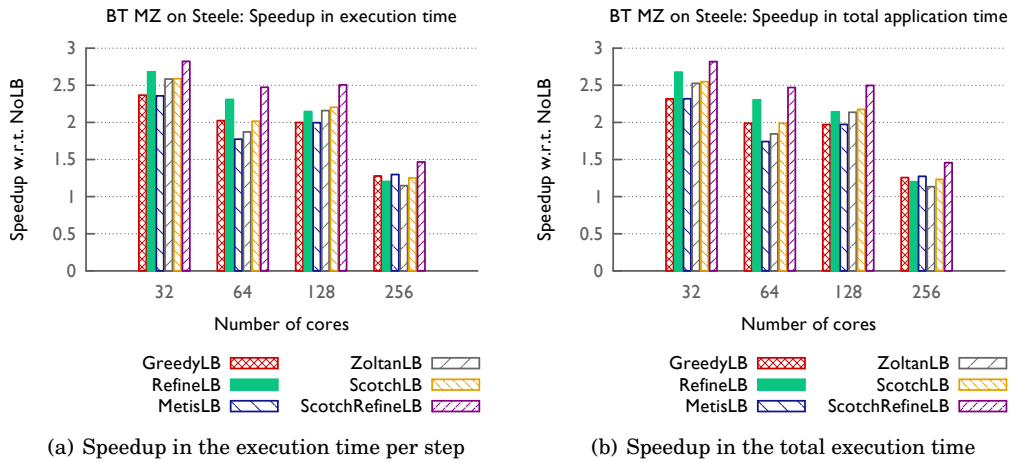
(b) Speedup in the total execution time

Fig. 11: Speedup for BT_MZ on Steele

Figure 10(b) compares the time spent in the load balancing strategies. We observe that, as the system and problem size increase, the strategy time for all the load balancers increases. However, the strategy time is insignificant in comparison to time per step for BT_MZ. Figure 11(a) presents the speedups for the execution time per step for all the load balancers. ScotchRefineLB performs best among all the load balancers and shows a speedup of 2.5 to 3 times in comparison to NoLB. In comparison to other load balancers, ScotchRefineLB reduces the execution time by 11%.

The results for a complete run of BT_MZ in which load balancing is performed once every 1000 iterations are presented in Figure 11(b). Since the strategy time is negligible, the application time primarily consists of the time per step and the migration time. The trends are similar to the speedups observed in Figure 11(a). ScotchRefineLB performs better than all other load balancers consistently and reduces the total execution
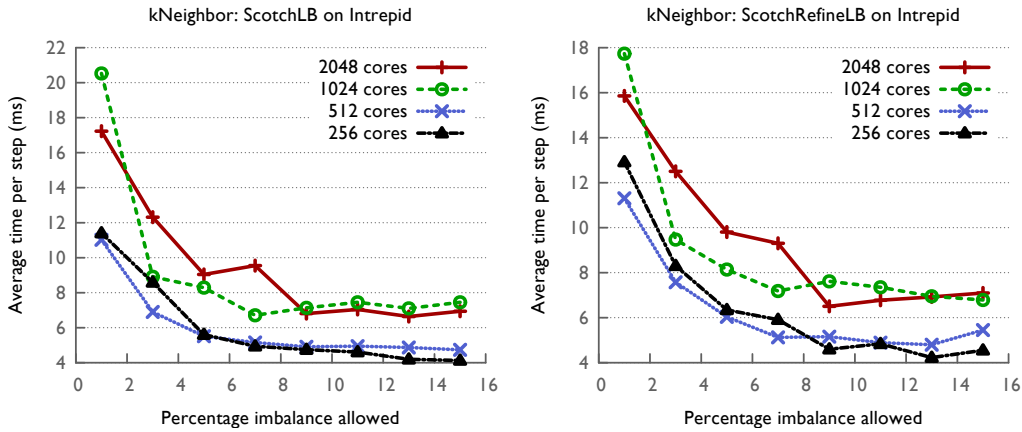
Fig. 12: Impact of imbalance allowance on execution time for ScotchLB and ScotchRe-fineLB (kNeighbor on Intrepid)

time by up to 12%. In comparison to NoLB, ScotchRefineLB obtains speedups ranging from 1.5 to 2.8 and ScotchLB obtains speedups from 1.2 to 2.5. Among the strategies in SCOTCH based load balancers, STRAT_BALANCE with 15% imbalance tolerance gave the best performance.

4.4. Strategies to handle different classes of applications

The end user can assist the partitioning strategies in SCOTCH in making good load balancing decisions by indicating whether computational load balance or minimizing the communication cut is more important for an application. The user can pass a parameter to SCOTCH which indicates the percentage of load imbalance permissible for an application. If the application is computation-bound, this value should be set to a low number; on the other hand, if it is communication-bound and can tolerate some degree of computational load imbalance, then this parameter can be set to a higher value.

We use two micro-benchmarks, kNeighbor(section 4.2) and stencil4d, to evaluate this. stencil4d is representative of the communication pattern in MILC [Bernard et al. 2000], a Lattice QCD code. In this computationally intensive benchmark, each object is assigned a block of $16 \times 16 \times 16 \times 16$ doubles. In each iteration, every object exchanges boundary data with its eight neighbors (two in each of four directions). This results in exchange of multiple messages of size 32 KB each. Once the data exchange is done, each process computes a 9-point stencil on its data. Load imbalance is artificially introduced by having each object do the stencil computation a random number of times within each iteration.

Figure 12 shows the execution time per step of kNeighbor for different values of this parameter when used within ScotchLB and ScotchRefineLB. kNeighbor is communication-intensive and hence, a value that permits between 8 to 12% imbalance gives the best results. However, if we look at Figure 13, which presents execution times for stencil4d, a benchmark affected more by computational load imbalance than inefficient communication, best performance is obtained when strict load balance is ensured (1% imbalance permitted).

Section 3.3 presented a greedy algorithm for balancing applications that have irregular load distributions and localized concentrations of vertices with heavy loads, a
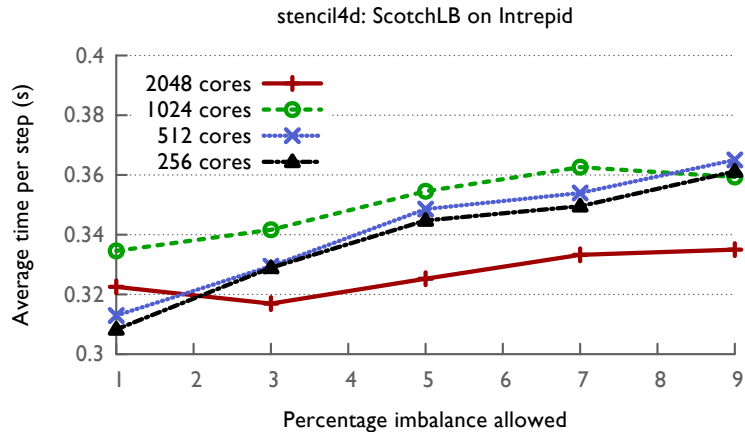
Fig. 13: Impact of imbalance allowance on execution time for ScotchLB (stencil4d on Intrepid)
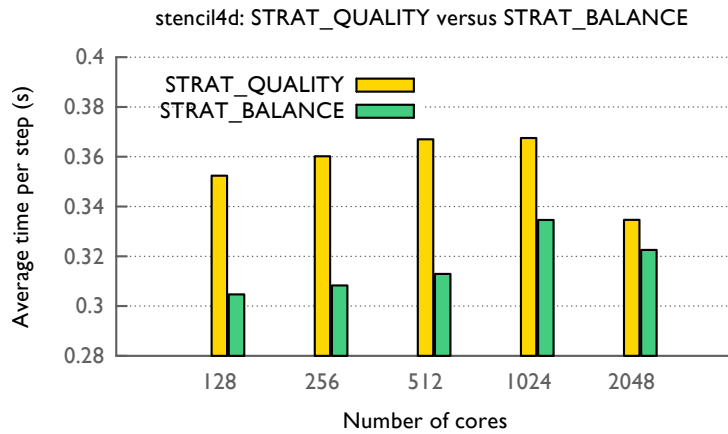

Fig. 14: Comparison of application performance for the STRAT_QUALITY versus STRAT_BALANCE strategy within ScotchLB (stencil4d on Intrepid)

scenario which is not handled efficiently by recursive bipartitioning. This is yet another technique to give more importance to balancing computational load than trying to achieve a minimal cut. Figure 14 presents a comparison of the default scheme (STRAT_QUALITY) versus this new scheme (STRAT_BALANCE) that attempts to achieve better load balance by considering all vertices and not only the ones in the neighborhood. Figure 14 shows that STRAT_BALANCE consistently outperforms STRAT_QUALITY for stencil4d. The performance gains are in the range of 10-15%. These results are in accordance with our expectation for a computationally intensive benchmark, such as stencil4d, for which balancing of load should be preferred over optimizing communication.

4.5. Effect of application features on performance

The previous section described advanced features in SCOTCH and their ability to handle different classes of parallel applications. In this section, we discuss the impact of application characteristics on the performance benefits obtained from load balancing. One relatively straightforward conclusion is that as the amount of communication in an application is increased, load balancing strategies, such as graph partitioning, that take the communication into account, give increasingly larger benefits. Figure 15 shows the speedup obtained in execution time per step with respect to NoLB for different message sizes. As we increase the message size from 2 KB to 16 KB, the improvement in the time per step using the graph partitioning based load balancers over RefineLB increases from 30% to 79%. When compared to the baseline performance, the graph partitioning based load balancers give an overall speedup of 1.2 to 4.6 when varying the message size from 2 KB to 16 KB. Hence, graph partitioning based load balancers should definitely be used with applications that are communication-intensive.
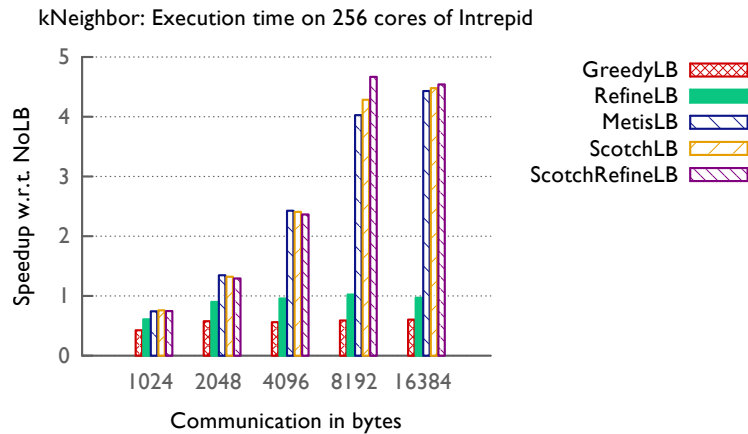

Fig. 15: Impact of increasing communication on the quality of load balance (kNeighbor running on 256 cores of Intrepid)

5. RELATED WORK

The problem of load balancing (also known as multiprocessor scheduling) of n computational tasks on p processors is known to be NP-hard [Garey and Johnson 1979; Leung and Whitehead 1982; Applegate and Cook 1991]. However, solutions that can bring the load imbalance (ratio of maximum to average load) within 5-10% of the optimal are still desirable. Load balancing is a much studied problem and algorithms and heuristics from various fields have been applied to it, ranging from prefix sum, recursive bisection [Berger and Bokhari 1987; Simon 1991], space filling curves [Sagan 1994] to work stealing [Blumofe and Leiserson 1999] and graph partitioning.

Graph partitioning has been used for static load balancing of many parallel applications for some time now [Attaway et al. 1997; Shadid et al. 1997; Buluç and Madduri 2013]. [Bichot and Siarry 2013] studies graph partitioning in the area of numerical analysis which includes techniques for graph partitioning, hypergraph partitioning and parallel methods. METIS [Karypis and Kumar 1996a], CHACO [Hendrickson and Leland 1995] and SCOTCH [Pellegrini and Roman 1996] are some popular graph partitioning libraries. PARMETIS and PT-SCOTCH are the parallel versions of METIS and

SCOTCH, respectively, that were developed to handle the increasing sizes of parallel applications and machines. Parallel algorithms reduce the time and memory requirements for partitioning large graphs.

Most graph partitioners iteratively improve the initial solution. Local search is a widely used heuristics for optimization. The running time of KL [Kernighan and Lin 1970a] local search method was improved by Fiduccia and Mattheyses [Fiduccia and Mattheyses 1982b] resulting in KL/FM local search algorithm. [Karypis and Kumar 1996b] improves the KL/FM algorithm by only allowing boundary nodes to move and presents a k-way partitioning of the KL/FM algorithm. [Osipov and Sanders 2010] propose a highly localized version of KL/FM. To improve the edge cut, [Sanders and Schulz 2012] gives a max-flow min-cut based technique. SCOTCH uses KL/FM and diffusion for bipartitioning [Pellegrini 2007].

With the emergence of large-scale heterogeneous architectures and development of complex multi-physics applications, the challenge has shifted towards developing algorithms and techniques for topology-aware, scalable and dynamic load balancing. ZOLTAN is a library, containing a collection of geometric and hypergraph-based partitioners that can be called from standard applications [Çatalyürek et al. 2009]. It provides a suite of load balancing algorithms including parallel graph partitioning and also allows the use of external libraries such as PARMETIS. Jostle [Walshaw and Cross 2007] is a software package designed to partition unstructured meshes, which can be also used for repartition and load balancing. Other frameworks such as DRAMA [Basermann et al. 2000] and Chombo [Colella et al. 2003] provide load balancing capabilities for specific classes of parallel applications: finite element methods and finite difference methods, respectively. UMPa [Catalyiirek 2013] is a multi-level partitioner using a directed hypergraph model and k-way refinement strategy.

CHARM++ provides a framework with inbuilt load balancing strategies and the option to deploy external libraries that provide load balancing algorithms [Brunner and Kalé 2000]. The CHARM++ runtime system does not depend on the application to provide the task graph, the cost models for which might be inaccurate. The runtime system uses automatic instrumentation to obtain the loads and the communication graph which is used by the load balancing framework. We believe that this paper presents one of the first analyses of using graph partitioning in a *measurement-based* dynamic load balancing framework. The added benefits of interconnect topology awareness and hierarchical load balancing schemes implemented in CHARM++ can also be exploited in conjunction with graph partitioning and will be discussed in future work.

6. SUMMARY AND NEXT STEPS

This paper represents an attempt at exploiting graph mapping and repartitioning methods for load balancing in parallel computing. Combined with measurement-based dynamic load balancing capabilities of an adaptive runtime system, a powerful technique for automatic balancing of applications is created. We present new algorithms, implemented in SCOTCH, such as k-way multilevel repartitioning and a load imbalance reduction algorithm that favors load balance over minimizing the edge cut. This is especially useful for computation-bound applications with irregular load distributions.

SCOTCH-based load balancers improve performance for the kNeighbor microbenchmark and the NAS BT solver by 12-42% over the existing load balancers in CHARM++ and METIS and ZOLTAN-based balancers. They also reduce the number of migrations, by several orders of magnitude in some cases, which reduces the associated communication costs. ScotchRefineLB migrates nearly 11 times fewer objects than MetisLB and ZoltanLB for BT_MZ. This shows that graph partitioning algorithms specifically designed for mapping objects to processors give better performance than

using generic graph partitioners, such as METIS, for this purpose. Compared to the baseline performance, ScotchRefineLB leads to overall speedups of 5 for kNeighbor and 1.5 to 2.8 for BT_MZ.

Future work involves developing an intelligent load balancing framework that can choose the best strategy automatically (comprehensive versus refinement, favoring load balance versus minimizing the edge cut, etc.) depending on the computation and communication characteristics of an application. Another area of exploration is the use of architecture-aware mapping strategies available in SCOTCH for interconnect topology-aware load balancing. An extension to the current capabilities in CHARM++ would be to enable the use of MPI-based PARMETIS in CHARM++ through interoperation of MPI and CHARM++.

REFERENCES

D. Applegate and B. Cook. 1991. A computational study of the job-shop scheduling problem. *ORSA Journal of Computing* 3, 2 (1991), 149 – 156.

S.A. Attaway, E.J. Barragy, K.H. Brown, D.R. Gardner, B.A. Hendrickson, S.J. Plimpton, and C.T. Vaughan. 1997. Transient Solid Dynamics Simulations on the Sandia/Intel Teraflop Computer. In *ACM/IEEE Supercomputing Conference*.

S. T. Barnard and H. D. Simon. 1994. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience* 6, 2 (1994), 101–117.

A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J.-M. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose, and C. Walshaw. 2000. Dynamic load balancing of finite element applications with the DRAMA Library. In *Applied Math. Modeling*, Vol. 25. 83–98.

Marsha J Berger and Shahid H Bokhari. 1987. A partitioning strategy for nonuniform problems on multiprocessors. *Computers, IEEE Transactions on* 100, 5 (1987), 570–580.

Claude Bernard, Tom Burch, Thomas A. DeGrand, Carleton DeTar, Steven Gottlieb, Urs M. Heller, James E. Hetrick, Kostas Orginos, Bob Sugar, and Doug Toussaint. 2000. Scaling tests of the improved Kogut-Susskind quark action. *Physical Review D* 61 (2000).

Abhinav Bhatelé, Laxmikant V. Kalé, and Sameer Kumar. 2009. Dynamic Topology Aware Load Balancing Algorithms for Molecular Dynamics Applications. In *23rd ACM International Conference on Supercomputing*.

Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. 2008. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*.

Charles-Edmond Bichot and Patrick Siarry. 2013. *Graph partitioning*. John Wiley & Sons.

Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

Robert K. Brunner and Laxmikant V. Kalé. 2000. Handling Application-Induced Load Imbalance using Parallel Objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*. World Scientific Publishing, 167–181.

Aydın Buluç and Kamesh Madduri. 2013. Graph partitioning for scalable distributed graph computations. *Contemp. Math.* 588 (2013).

Umit V Catalyiirek. 2013. UMPa: A multi-Objective, multi-level partitioner for communication minimization Umit V. Catalyiirek, Mehmet Deveci, Kamer Kaya, and Bora Ucar. *Graph Partitioning and Graph Clustering* 588 (2013), 53.

Umit V Catalyürek and Cevdat Aykanat. 1999. PaToH: a multilevel hypergraph partitioning tool, version 3.0. *Bilkent University, Department of Computer Engineering, Ankara* 6533 (1999).

Charlie Catlett and others. 2007. TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications, In HPC and Grids in Action, Lucio Grandinetti (Ed.). *Advances in Parallel Computing* 16 (2007), 225–249.

Umit V. Çatalyürek, Erik G. Boman, Karen D. Devine, Doruk Bozdağ, Robert T. Heaphy, and Lee Ann Riesen. 2009. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel Distrib. Comput.* 69 (August 2009), 711–724. Issue 8.

P. Colella, D.T. Graves, T.J. Ligocki, D.F. Martin, D. Modiano, D.B. Serafini, and B. Van Straalen. 2003. Chombo Software Package for AMR Applications Design Document. (2003). http://seesar.lbl.gov/anag/chombo/ChomboDesign-1.4.pdf.

Rob F. Van der Wijngaart and Haoqiang Jin. July 2003. *NAS Parallel Benchmarks, Multi-Zone Versions*. Technical Report NAS Technical Report NAS-03-010.

C. M. Fiduccia and R. M. Mattheyses. 1982a. A linear-time heuristic for improving network partitions. In *Proc. 19th Design Automation Conference*. 175–181.

Charles M Fiduccia and Robert M Mattheyses. 1982b. A linear-time heuristic for improving network partitions. In *Design Automation, 1982. 19th Conference on*. IEEE, 175–181.

Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Bruce Hendrickson and Robert Leland. 1995. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. ACM, New York, NY, USA, 28.

Nikhil Jain, Abhinav Bhatele, Jae-Seung Yeom, Mark F. Adams, Francesco Miniati, Chao Mei, and Laxmikant V. Kale. 2015. Charm++ & MPI: Combining the Best of Both Worlds. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (to appear) (IPDPS '15)*. IEEE Computer Society. LLNL-CONF-663041.

Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. 2008. Massively parallel cosmological simulations with ChaNGa. In *IPDPS*.

L.V. Kalé and S. Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93*, A. Paepcke (Ed.). ACM Press, 91–108.

Laxmikant V. Kalé. 2002. The Virtualization Model of Parallel Programming : Runtime Optimizations and the State of Art. In *LACSI 2002*. Albuquerque.

George Karypis and Vipin Kumar. 1996a. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. 35. DOI:http://dx.doi.org/10.1145/369028.369103

George Karypis and Vipin Kumar. 1996b. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. 35. DOI:http://dx.doi.org/10.1145/369028.369103

Brian W Kernighan and Shen Lin. 1970a. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal* 49, 2 (1970), 291–307.

B. W. Kernighan and S. Lin. 1970b. An efficient heuristic procedure for partitionning graphs. *BELL System Technical Journal* (Feb. 1970), 291–307.

Joseph Y.-T. Leung and Jennifer Whitehead. 1982. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation* 2, 4 (1982), 237 – 250.

Sandhya Mangala, Terry Wilmarth, Sayantan Chakravorty, Nilesh Choudhury, Laxmikant V. Kale, and Philippe H. Geubelle. 2007. Parallel Adaptive Simulations of Dynamic Fracture Events. *Engineering with Computers* 24 (December 2007), 341–358. Issue 4.

Vitaly Osipov and Peter Sanders. 2010. n-Level Graph Partitioning. In *Algorithms–ESA 2010*. Springer, 278–289.

F. Pellegrini. 1994. Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proc. SHPCC'94, Knoxville*. IEEE, 486–493.

F. Pellegrini. 2007. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In *Proc. EuroPar, Rennes (LNCS)*, Vol. 4641. 195–204.

François Pellegrini and Jean Roman. 1996. SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *HPCN Europe*. 493–498.

Hans Sagan. 1994. *Space-filling curves*. Vol. 18. Springer-Verlag New York.

Peter Sanders and Christian Schulz. 2012. High quality graph partitioning. *Graph Partitioning and Graph Clustering* 588 (2012), 1.

scotch. SCOTCH: Static mapping, graph partitioning, clustering and sparse matrix block ordering package. http://www.labri.fr/~pelegrin/scotch. (????).

John Shadid, Scott Hutchinson, Gary Hennigan, Harry Moffat, Karen Devine, and A.G. Salinger. 1997. Efficient parallel computation of unstructured finite element reacting flow solutions. *Parallel Comput.* 23, 9 (1997), 1307 – 1325.

Horst D Simon. 1991. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering* 2, 2 (1991), 135–148.

Chris Walshaw and Mark Cross. 2007. JOSTLE: parallel multilevel graph-partitioning software–an overview. *Mesh partitioning techniques and domain decomposition techniques* (2007), 27–58.