LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-501974

Applying graph partitioning methods in measurement-based dynamic load balancing

A. Bhatele, S. Fourestier, H. Menon, L. V. Kale, F. Pellegrini

September 29, 2011

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Applying graph partitioning methods in measurement-based dynamic load balancing

Abhinav Bhatele[†], Sébastien Fourestier[⋆], Harshitha Menon[‡], Laxmikant V. Kale[‡] and François Pellegrini[⋆]

[†]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, USA
[⋆]Laboratoire Bordelais de Recherche en Informatique & INRIA Bordeaux – Sud-Ouest, 33405 Talence, France
[‡]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
E-mail: bhatele@llnl.gov, sebastien.fourestier@labri.fr, gplkrsh2@illinois.edu, kale@illinois.edu, francois.pellegrini@labri.fr

***Abstract*— Load imbalance leads to an increasing waste of resources as an application is scaled to more and more processors. Achieving the best parallel efficiency for a program requires optimal load balancing which is a NP-hard problem. However, finding near-optimal solutions to this problem for complex computational science and engineering applications is becoming increasingly important. Charm++, a migratable objects based programming model, provides a measurement-based dynamic load balancing framework. This framework instruments and then migrates over-decomposed objects to balance computational load and communication at runtime. 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 developed in a graph partitioning toolbox called SCOTCH that consider the previous mapping to minimize migration costs. We also discuss a new imbalance reduction algorithm for graphs with irregular load distributions. We compare several load balancing algorithms using micro-benchmarks on Intrepid and Ranger and evaluate the effect of communication, number of cores and number of objects on the benefit achieved from load balancing. New algorithms developed in SCOTCH lead to better performance compared to the METIS partitioners for several cases, both in terms of the application execution time and fewer number of objects migrated.**

***Keywords*-load balancing; graph partitioning; instrumentation; communication-aware; performance**

I. Introduction

Long-running scientific simulations can be performed in a reasonable time only by means of parallel processing because of the amount of computation and data that they involve. To do so, data and their associated computations are distributed across several interconnected processing elements (processors) which work in parallel. Accessing data on remote processors requires inter-processor communication. Consequently, the efficient use of such distributed memory parallel machines requires spreading the computation load evenly across different processors and minimizing the communication overhead.

When the tasks/processes 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 unoriented graph. The vertices of this *process graph* represent the computation to be performed and its edges represent inter-process communication. In case of MPI applications, the number of processes is the same as the number of processors, p and the problem of mapping processes to processors is reduced to a communication-minimizing mapping problem. In the case of other programming models, such as Charm++ [1], ParalleX [2], FG-MPI [3], Adaptive MPI [4] and others, computation is over-decomposed into fine-grained tasks or objects, where the number of such tasks, n is much greater than p. For applications written in these programming models, the problem of mapping objects to processors can be viewed as the partitioning and mapping of a graph of n vertices to p physical processors. The aim is to assign the same load to all processors and to minimize the edge cut of the graph, that is, the sum of the weights of edges whose ends are on different physical processors.

Although the problem of partitioning communicating objects to processors appears similar to that of partitioning large unstructured meshes to processors, the differences are significant and lead to major algorithmic changes. The most significant difference is that the number of objects per processor is of the order of ten in load balancing, whereas for meshes, the number is closer to a million.

In this paper, we evaluate the deployment of static mapping and graph repartitioning, traditionally used for partitioning physical domains/meshes, for balancing load dynamically in over-decomposed parallel applications. We have chosen a specific programming model, Charm++ and a graph partitioning library, Scotch [5] for implementing the new algorithms and heuristics. However, the techniques described here are generally applicable to the programming models mentioned above and use of other graph partitioning libraries [6], [7].

Charm++ includes a mature load balancing framework which provides a friendly user-interface for incorporating external load balancing algorithms. For applications in which computational loads tend to persist over time, the framework supports measurement-based load balancing. It records object 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. Based on different partitioning and repartitioning algorithms in Scotch, we have developed ScotchLB and ScotchRefineLB for comprehensive (fresh assignment of all objects to processors) and refinement-based load balancing respectively.

We discuss modifications to existing algorithms in Scotch

to make them more suitable for scenarios encountered in load balancing. This presents a distinct set of challenges compared 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 object migration cost into account and tries to minimize the time spent in migrations, and 2. a new 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 ScotchLB and ScotchRefineLB with other existing (greedy, refinement and communication-aware) load balancing algorithms in CHARM++ and with a METIS-based load balancer. We evaluate the algorithms based on various metrics for success: 1. reduction in execution time of the application, 2. time spent in load balancing, 3. number of objects migrated, and 4. reduction in inter-processor communication. We use three micro-benchmarks with different computation-communication characteristics and present results for runs on Intrepid (Blue Gene/P) and Ranger (an Opteron-Infiniband cluster). New algorithms developed in SCOTCH lead to better performance compared to the METIS partitioners for several cases, both in terms of the application execution time and fewer number of objects migrated.

The rest of the paper is organized as follows: Section II introduces measurement-based load balancing in CHARM++ and describes the existing load balancers in the framework. Section III presents existing and new partitioning algorithms developed in SCOTCH that are well suited for load balancing. A comparative evaluation of SCOTCH-based load balancers is presented in Section IV and Section VI summarizes the work.

II. DYNAMIC COMMUNICATION-AWARE LOAD BALANCING

An intelligent load balancing algorithm must take into account, both the characteristics of the parallel application and the topology of the target architecture. 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. A mapping is called static if 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 [8]–[10]. However, with increasingly complex multi-physics simulations and heterogeneous architectures, there is a growing need for dynamic load balancing during program execution. This requires input from the application about the changing computational loads and communication patterns. Zoltan [11]–[13] is one such framework for dynamic load balancing of parallel applications that uses hypergraph partitioners to balance entities indicated by the application. CHARM++ also includes an automatic dynamic load balancing framework. It assists the application by instrumenting the work units or objects in the application at runtime and making this information available for any load balancing strategy to use.

A. The CHARM++ load balancing framework

Applications written in CHARM++ over-decompose their computation into virtual processors or objects called "chares" which are then mapped on to physical processors by the runtime system. This initial static mapping can be changed as the execution progresses by migrating objects to other processors if the simulation leads to a load imbalance. This is facilitated by a load balancing framework that instruments the application to obtain the computational loads and the communication graph of the objects and uses them to make informed decisions for migrating objects [14]. Measurement-based load balancing is effective when the load and communication pattern of the application either change slowly, or change abruptly but infrequently. In these situations, data from the recent past is a good predictor of the near future. For other situations, the application can provide performance estimates for the objects to supplant the measurements.

There are several in-built load balancing strategies in CHARM++ that can be used by application developers, some of which are described here for completeness (and for the benefit of the reader to understand the results better):

- **GreedyLB**: A comprehensive load balancer based on the greedy heuristic that maps the heaviest objects on to the least loaded processors until the load of all processors is close to the average load.
- **RefineLB**: A refinement load balancer that migrates objects from processors with greater than average load (starting with the most overloaded processor) to those with less than average load. The aim of this strategy is to reduce the number of objects migrated.
- **RefineCommLB:** A refinement strategy similar to RefineLB that also considers the communication between different objects when trying to choose the best underloaded processor to place an object on.
- **MetisLB:** A strategy that passes the load information and the communication graph to METIS, a graph partitioning library, and uses the recursive graph partitioning algorithm in it for load balancing.

The CHARM++ runtime also encourages application developers to write application-specific load balancing strategies or use external libraries for the task. For this, it provides an easy interface to write new load balancers. Figure 1 presents the data structures that provide information useful for a load balancing strategy when making migration decisions. The runtime instruments a few time steps of the application before load balancing and this information is available in the form of

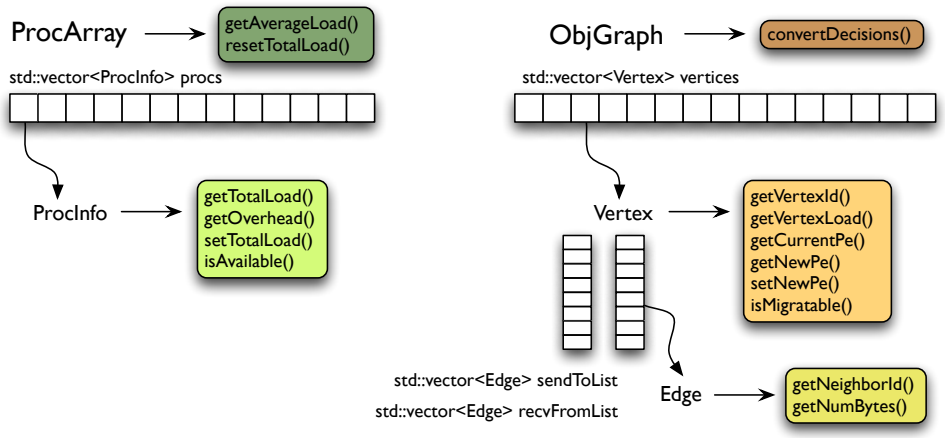

Fig. 1. A user friendly API for plugging in new load balancing strategies in CHARM++

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 has information about the number of messages exchanged between a pair of vertices and also the total number of bytes exchanged.

Using the above mentioned information provided by the CHARM++ runtime, 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 to migrate objects for the subsequent time steps. This setup facilitates the use of external load balancing algorithms/libraries for measurement-based dynamic load balancing of parallel applications, since they do not have to deal with the mechanics of instrumentation and object migration.

III. SCOTCH: A SOFTWARE FOR GRAPH PARTITIONING AND LOAD BALANCING

SCOTCH [5] is a software project developed at the *Laboratoire Bordelais de Recherche en Informatique* (LaBRI) of the Université Bordeaux 1, and now within the Bacchus project-team at INRIA Bordeaux Sud-Ouest. Its goal is to provide efficient graph partitioning heuristics for scientific computing, making them available to the community as a software toolbox. This toolbox is called SCOTCH in the general case, while its parallel subset is called PT-SCOTCH (for "Parallel and Threaded SCOTCH"). In this paper, we will only evaluate the sequential version of SCOTCH because the graph repartitioning algorithms that we discuss here are only available in the sequential version to date. This is feasible because for load balancing, the size of the graph where each vertex is a coarse-grained object is small — typically, 10-20 times larger than the number of processors.

A. *Static mapping methods in* SCOTCH

Although SCOTCH also deals with combinatorial problems such as sparse matrix ordering, its first purpose was to study static mapping by means of graph theory, using a "divide and conquer" approach. Its core static mapping algorithm, the Dual Recursive Bipartitioning (DRB) algorithm [15], recursively allocates subsets of processes to subsets of processors. It starts by considering a set of processors, also called the domain, containing all the processors of the target machine, and with which all the processes to be mapped are associated. At each step, the algorithm bipartitions an unprocessed domain into two disjoint subdomains, and calls a graph bipartitioning algorithm to split the subset of processes associated with the domain across the two subdomains. When the processor graph is a complete graph, this process degenerates into simple recursive graph bipartitioning, as all processors in the domain are at the same distance from each other.

Bipartitions are computed using a multilevel framework. The graph to be bipartitioned is repeatedly coarsened by matching neighboring vertices, down to a coarse graph of hundred vertices. Then, an initial partition is computed on the coarse graph obtained, and this bipartition is refined back, from coarser to finer graphs, to yield a bipartition of the original graph [16], [17]. In order to preserve locality, initial bipartitions are computed by way of greedy "graph growing" heuristics. First an appropriate seed vertex is chosen in the graph to bipartition. Then these algorithms perform breadth-first traversal of the graph, accumulating all the traversed vertices in the first partition, and stop when enough load is collected. The remaining vertices are then assigned to the second partition [18].

In order to ensure that the granularity of the solution is that of the original graph and not that of the coarsest graph, the refined partitions are smoothened at every level. The partitions are refined at each level by using local optimization algorithms such as Kernighan-Lin [19] (KL) or Fiduccia-Mattheyses [20] (FM). Because these algorithms perform vertex movements across the current boundary only, they can only help smoothen

the frontier, but cannot perform large-scale changes of the provided partition. The original KL algorithm reduces the cut by performing swaps of vertices, and is consequently quadratic in time with respect to the number of vertices because of vertex pairing routines. The FM algorithm considers single vertex movements only, and is therefore quasi-linear in time, which is why it is mostly used. However, this may lead to problems for graphs with irregular load distributions, as we will see in Section III-C.

B. Repartitioning methods in SCOTCH

A new feature of SCOTCH that is experimented with in this paper, is the ability to compute re-partitions of a graph, based on an existing partition. This scheme is referred to as refinement load balancing in CHARM++. For repartitioning, every vertex of the graph to be remapped is associated with a fictitious edge that connects it to a fictitious vertex that represents the old partition, like in [21]. Hence, this edge is cut when the vertex changes partitions and remains intact if the vertex remains in its old partition. Doing so allows us to integrate the migration cost within the existing edge cut minimization process. All of the fictitious edges are weighted, with a weight that represents the cost of migrating the vertex to another partition. Since edge dilation is accounted for in the edge cut cost function, moving a vertex to a distant processor costs more than moving it to a nearby one.

These repartitioning algorithms have been embedded in a new k-way multilevel framework. Instead of doing coarsening and uncoarsening for computing every bipartition, the original graph is coarsened once, down to a size equal to a few tens of vertices times the number of expected parts. The aforementioned recursive bipartitioning process is called on this coarsest graph, after which the obtained k-way repartition is refined back to the original graph. This is done using k-way local optimization algorithms such as a k-way variant of the FM algorithm. This variant stores every possible move of boundary vertices to their neighboring partitions, sorting the potential gains in a Fibonacci heap structure.

All of the above algorithms have been adapted so as to account for these fictitious edges, most of the time without having to create and store them individually, thus reducing the memory footprint of the repartitioning process [22]. For instance, fictitious edges are not necessary in the k-way coarsening process, but are created when computing the k-way band graph that is used by local optimization algorithms (see [23] for a description of band graphs).

C. A new algorithm for reducing load imbalance

All of the aforementioned algorithms were designed for graphs whose vertex load distribution is regular and not severely imbalanced. In particular, it was assumed that it is always possible to achieve load balance by sequences of moves involving only those vertices that are located on the current boundaries of the partitions, as in the case of domain decomposition problems. Such assumptions result in algorithms that privilege locality by design.

However, when 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 until, when trying to bipartition the cluster, the algorithm can only compute bipartitions that are imbalanced because of the very high granularity of the vertex loads. Moreover, since the FM algorithm uses vertex movements and not vertex swaps as in the KL algorithm, movements of the heaviest vertices can never be considered. Moving a heavy vertex out of its slightly overloaded partition may result in heavy overload of the destination partition, as well as leaving its original partition drastically underloaded.

To address this problem, a new load imbalance reduction algorithm has been implemented in SCOTCH. It is activated when the load imbalance ratio of the current k-way partition at some uncoarsening level is above the prescribed threshold. Based on the discussion above, this current partition is assumed to preserve locality. The main loop of the algorithm considers all vertices in descending weight order. If the considered vertex fits in its current destination partition, it remains there. If the vertex causes its destination partition to be overloaded, possible alternate destination partitions are tried out in target domain recursive bipartition tree order. The neighboring domain of the last bipartition level is tried first, then the two sons of the neighbor domain in the second-last level, and so on. Therefore, closest domains in the target architecture partitions are tried out first, before farther ones. This algorithm may increase the communication cut, but only locally, as far as mapping is concerned. Once a balanced partition is achieved, communication cost minimization can be applied, by using k-way FM, so that vertices that have been placed alone in a distant partition can try to pull neighboring vertices in their partition so as to reduce the cut locally.

In summary, the algorithms available in the new release of SCOTCH, and which have been experimented with in this paper, comprise of: (i) the classical recursive bipartitioning (or static mapping) method of SCOTCH 5.1 (referred to as **ScotchLB**), (ii) a new k-way multilevel framework for partitioning and repartitioning graphs (referred to as **ScotchRefineLB**), and (iii) a specific algorithm for handling graphs with irregular load distributions and localized concentration of vertices with heavy loads.

IV. CASE STUDIES

We compare the performance of different load balancing strategies using three micro-benchmarks:

kNeighbor: kNeighbor is a communication intensive benchmark in which each object exchanges a message of size 8 KB with fourteen other objects in every iteration. The IDs of these objects differ from that of the given object by $\{-35, \cdots, -5, 5, \cdots, 35\}$. Each object is assigned a random computational load with the difference between maximum and minimum load across all objects being 8 times.

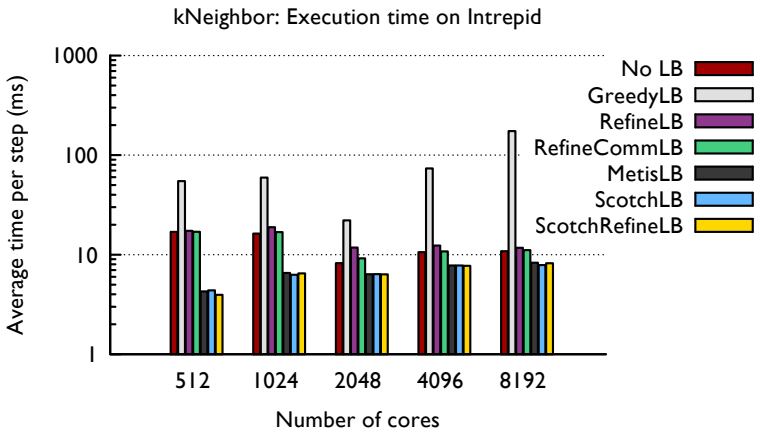

Fig. 2. Comparison of the average execution time per step for different load balancers (kNeighbor benchmark on Intrepid)

lb_test: lb_test is another communication-bound benchmark which involves communication with randomly selected objects. On an average, an object exchanges 32 KB of data with ten objects which are selected uniformly at random. In each iteration, time spent in computation per object varies from 0.1 to 1 millisecond. This benchmark differs from kNeighbor in two aspects: 1. The number of neighbors each object communicates with is not fixed, and 2. The set of objects cannot be subdivided into groups that do not communicate with one another.

stencil4d: stencil4d is representative of the communication pattern in MILC [24], 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 direction). 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 introduced by having each object do the stencil computation a random number of times within each iteration.

The experiments were run on Intrepid and Ranger. Intrepid is a $40,960$-node Blue Gene/P installation at the Argonne National Laboratory. Each node in Intrepid consists of four 850 MHz PowerPC cores. The principal interconnect for point-to-point communication in this system is a 3D torus. Each link on the torus offers a bi-directional bandwidth of 850 MB/s. The experiments were run in VN mode using all four cores per node. Ranger is a SUN constellation cluster at the Texas Advanced Computing Center consisting of $3,936$ nodes. Each node in Ranger is a 16-way SMP running at 2.3 GHz and has 32 GB of memory. The system is connected via a full-CLOS Infiniband interconnect providing 1 GB/s of peer-to-peer bandwidth.

A. *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. Two flavors of the SCOTCH static mapping method (Section III-A) can be used for this:

1) `STRAT_QUALITY` - Preference is given to obtaining the best edge cut.
2) `STRAT_BALANCE` - Preference is given to maintaining computational load balance as described in Section III-C.

For both of these, we also vary the imbalance ratio (described in Section IV-B) from 1% to 15% and use data from the experiment with the minimum application execution time for the various plots. The second load balancer, ScotchRefineLB, uses repartitioning methods (Section III-B) to refine the initial partitioning and mapping created by ScotchLB. To this end, ScotchLB is invoked once, when program execution begins, followed by several calls to ScotchRefineLB. As in the case of ScotchLB, we use both `STRAT_QUALITY` and `STRAT_BALANCE` with different values for the imbalance ratio and use the experiment with the minimum application execution time for reporting the results.

The performance of different load balancing strategies is evaluated on the basis of the following metrics:

1) Execution time per step for the application, which is the best indication of the success of a load balancer.
2) Time spent in the load balancing strategy. This, along with the frequency of load balancing, determines whether load balancing is beneficial for performance.
3) Number of objects migrated, which signifies the amount of data movement resulting from load balancing and hence the communication costs associated with it.
4) Reduction in inter-processor communication (or the ratio of remote to local communication)

Figures 2 and 3 present a comparison of the execution time per iteration for kNeighbor using different load balancers on Intrepid and Ranger respectively. Unless otherwise specified, the number of objects is eight times the number of processors

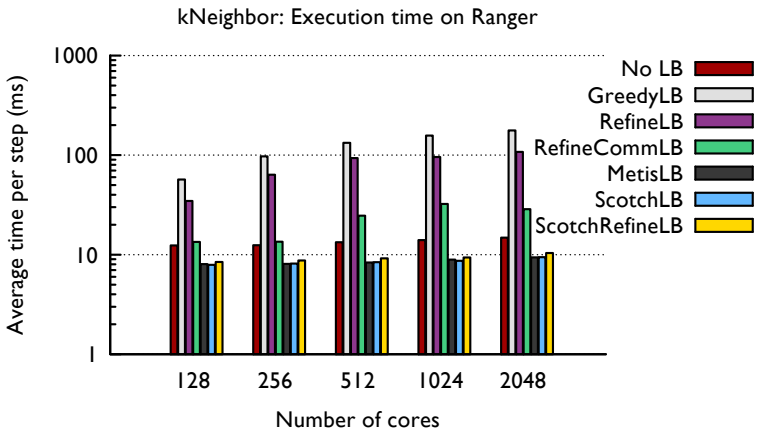

Fig. 3. Comparison of the average execution time per step for different load balancers (kNeighbor benchmark on Ranger)

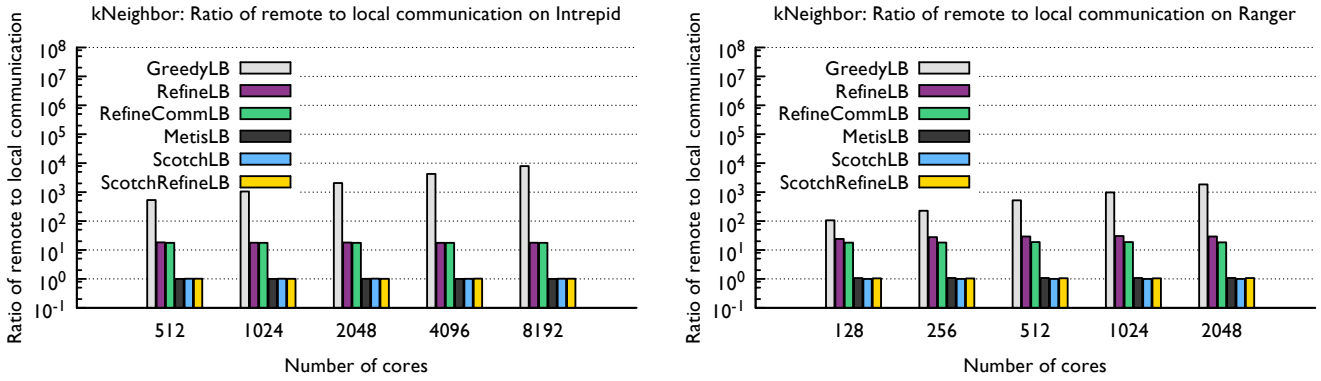

Fig. 4. Comparison of the ratio of remote to local communication after load balancing (kNeighbor benchmark on Intrepid and Ranger)

for all the experiments. **No LB** refers to no load balancing being performed and the runtime does a static mapping of all objects to processors, attempting to assign equal **number** of objects to each processor. For experiments involving MetisLB, both recursive bipartitioning and k-way multilevel partitioning were tried and the better of the two (lower application execution time) was used for reporting results.

Looking at Figure 2, we observe that the load balancers based on graph partitioners, METIS and SCOTCH, perform considerably better than the other load balancers and when no load balancing is done (No LB). This is in agreement with our expectations because kNeighbor is a communication intensive benchmark with nicely discernible and sparsely connected partitions. In particular, at 512 cores of Intrepid, ScotchRefineLB, which has the best performance among the graph partitioners, decreases the execution time by 77% as compared to RefineCommLB. It is to be noted that the time per step, when using other sub-optimal load balancers, reduces at 2048 cores (512 nodes, one midplane) because torus links become available and communication is automatically optimized. Hence, the execution time reduction is smaller (26%) for ScotchRefineLB versus RefineCommLB at 8192 cores. The results are different on Ranger (Figure 3) where GreedyLB, RefineLB and RefineCommLB give worse execution times compared to when no load balancing is performed. The benefit (reduction in execution time per step compared to No LB) from using ScotchLB and ScotchRefineLB remains constant around 36% across runs on different number of cores.

Figure 4 demonstrates the capability of graph partitioning based load balancers in mapping communicating processes onto the same processor. This figure compares the ratio of remote to local communication for different load balancers on Intrepid and Ranger. We note that irrespective of the system size, METIS and SCOTCH based load balancers succeed in maintaining this ratio close to one i.e. restricting half of total communication to the local processor. In contrast, for other load balancers, this ratio is at least an order of magnitude higher resulting in excess of remote communication, adversely affecting the performance. Ideally all communication should be local to the processors and hence this ratio should be close to 0. However, the communication graph is chordal and the distribution of outgoing links is such that, with respect to the size of the partitions, half of the outgoing links have to be remote, and hence these results. The locality preserving

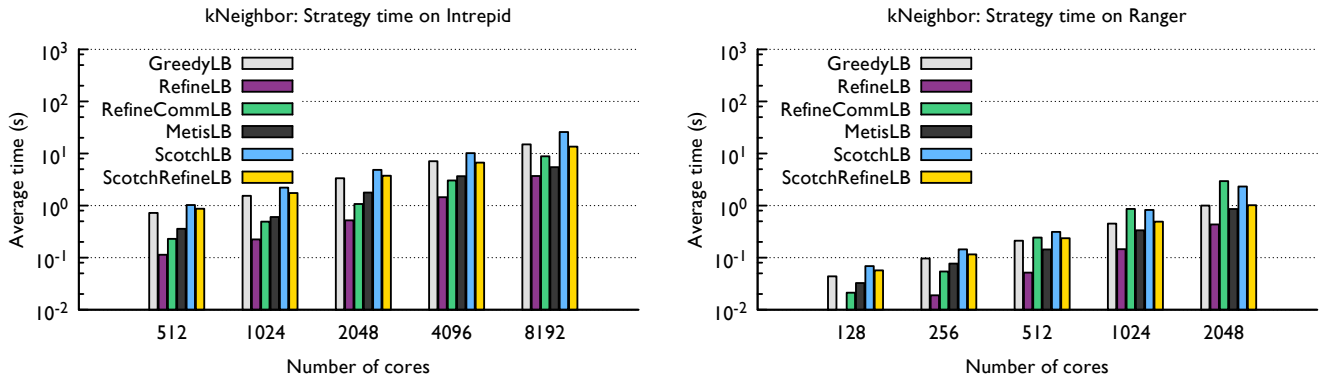

Fig. 5. Comparison of time spent in different load balancing strategies (kNeighbor benchmark on Intrepid and Ranger)

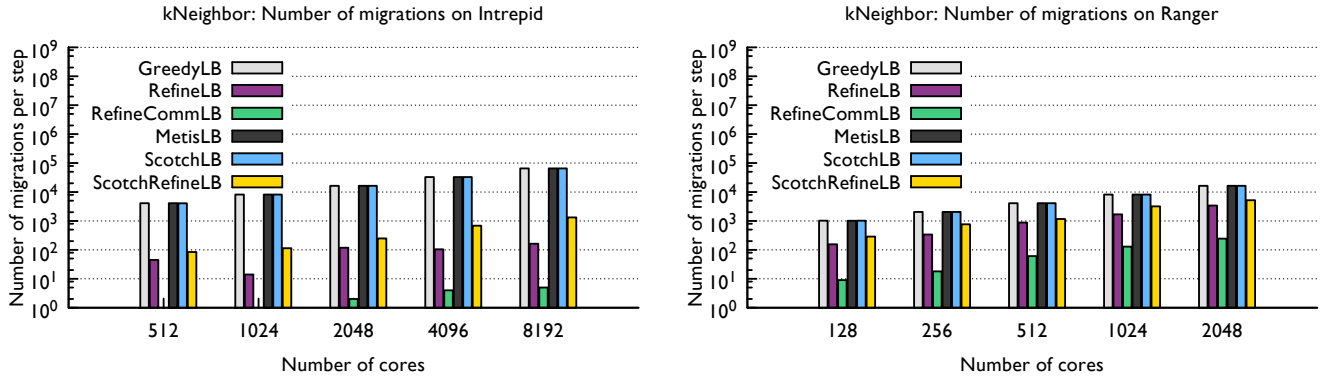

Fig. 6. Comparison of the number of objects migrated by different load balancers (kNeighbor benchmark on Intrepid and Ranger)

features of graph partitioning algorithms help, as it is most likely that, the partitions will be evenly spread as "donut slices" on the chordal graph.

Figure 5 compares the overhead incurred in load balancing, specifically, the time spent on load balancing strategies. We find that, as the number of cores increases, the strategy time for all the load balancers increases. RefineLB is the fastest among all and SCOTCH-based load balancers incur a higher cost over some of the other load balancers. This is, however, offset by the better performance the application achieves when using them and with an intelligent choice of the load balancing frequency, graph partitioning based load balancers can be used to our advantage. Note that the time spent in computing decisions is smaller for ScotchRefineLB as compared to ScotchLB. This is because, in spite of the additional work, the direct k-way multilevel framework is more efficient than a set of multilevel recursive bisections. However, the difference is small for smaller graphs as the k-way process quickly stops computing the initial k-way partition by means of the classical multilevel DRB scheme.

Figure 6 shows the number of objects migrated for different load balancers. Fewer migrations lead to less communication and hence this is a desired outcome of load balancing. The refinement load balancers, which includes RefineLB, RefineCommLB and ScotchRefineLB, are successful in reducing the number of migrations considerably when compared to strategies that map the objects to processors from scratch. For runs conducted on Intrepid, the refinement load balancers reduce the number of migrations by orders of magnitude when compared to other strategies. GreedyLB, MetisLB and ScotchLB end up migrating almost all objects in the application. RefineCommLB moves next to nothing but does not give the best load balance. ScotchRefineLB, however, moves 50 to 70 **times** fewer objects than MetisLB and ScotchLB and still gives performance very similar to both (see Figure 2). For a given number of cores, the number of migrations for refinement load balancers are much higher on Ranger than on Intrepid. This might be, in part, because communication is better optimized on the Blue Gene/P torus than on Ranger. Even so, ScotchRefineLB migrates 2-3 times fewer objects than MetisLB and ScotchLB (which migrate almost all of them) and also gives good performance.

We present only the application execution times for the other two benchmarks, lb_test and stencil4d, using different load balancers in Figures 7 and 8 respectively. For lb_test, we observe that the load balancers based on graph partitioning reduce the execution time by 13-19% compared to the default case (No LB). ScotchLB gives the best performance among

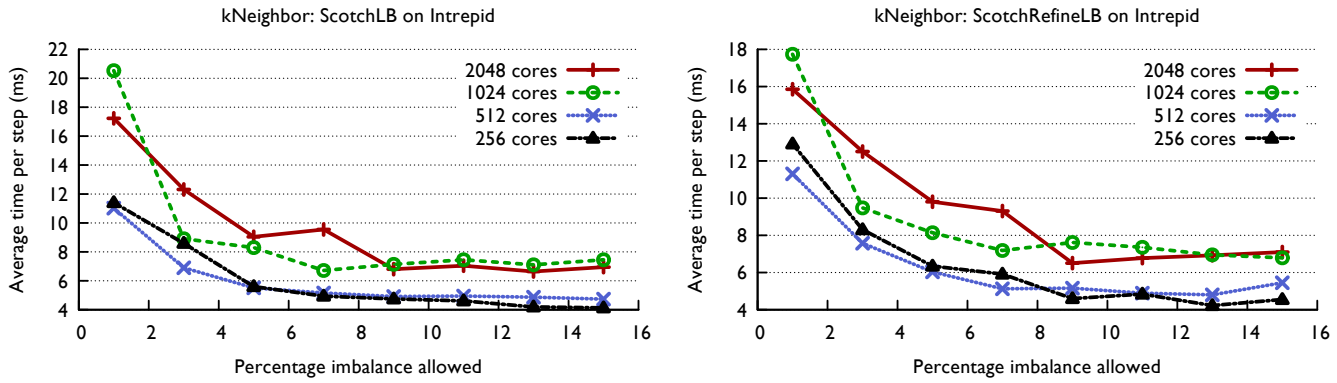

Fig. 9. Impact of imbalance allowance on the quality of load balancing for ScotchLB and ScotchRefineLB (kNeighbor on Intrepid)

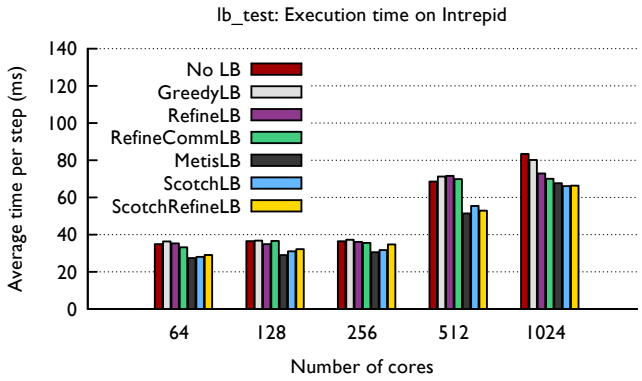

Fig. 7. Comparison of the average execution time per step for different load balancers (lb_test on Intrepid)

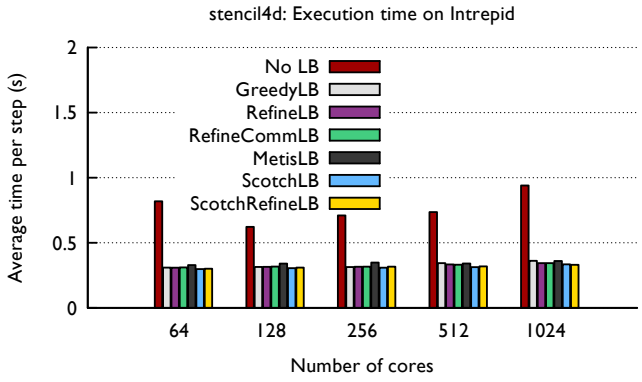

Fig. 8. Comparison of the average execution time per step for different load balancers (stencil4d on Intrepid)

all the load balancers at 1024 cores. It is to be noted that the gains obtained for lb_test are smaller in comparison to those for kNeighbor. We attribute this to the randomness of the communication graph in lb_test which makes it harder to partition into load balanced subsets while minimizing the cut.

Figure 8 presents execution times for stencil4d comparing the different load balancing strategies. The performance results for this benchmark are considerably different from kNeighbor and lb_test as stencil4d is a computationally intensive benchmark. As shown in Figure 8, most of the load balancers behave similarly and reduce the execution time by 50 to 65% compared to No LB. SCOTCH-based load balancers, which allow for an input parameter for load imbalance ratio, outperform the other load balancers for this benchmark. ScotchLB gives 7-11% better performance compared to MetisLB. These results demonstrate the robust nature of SCOTCH-based load balancers, which irrespective of the nature of the application, can be tuned to obtain good results. It also 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.

B. 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.

Figure 9 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 10, 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 III-C presented a new algorithm for balancing applications that have irregular load imbalance and localized concentrations of vertices with heavy loads, a scenario which is not handled efficiently by recursive bipartitioning. This is yet another technique to give more importance

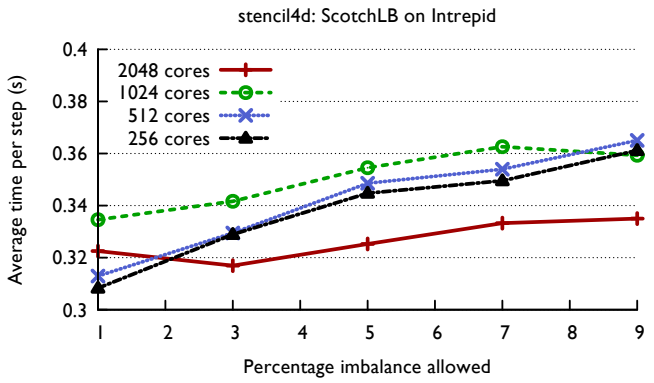

Fig. 10. Impact of imbalance allowance on the quality of load balancing for ScotchLB (stencil4d on Intrepid)

to balancing computational load than trying to achieve a minimal cut. Figure 11 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 11 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.

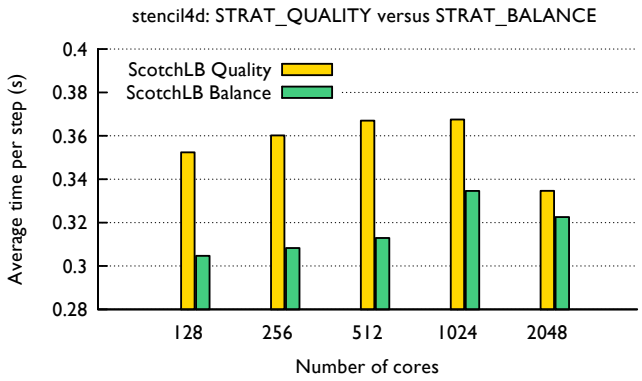

Fig. 11. Comparison of application performance for the `STRAT_QUALITY` versus `STRAT_BALANCE` strategy within ScotchLB (stencil4d on Intrepid)

As noted earlier, all the results in Section IV-A were obtained by running both flavors (`STRAT_QUALITY` and `STRAT_BALANCE`) and different values of the percent imbalance allowed and the run with the minimum execution time for a given number of cores was chosen.

C. Effect of application features on performance benefits

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 12 plots the average execution time per step of kNeighbor 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%. Hence, graph partitioning based load balancers should definitely be used with applications that are communication-intensive.

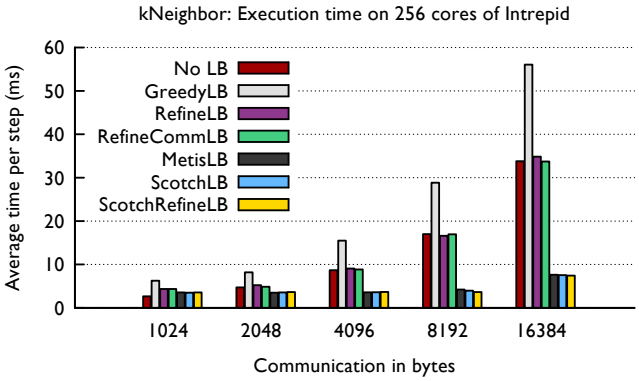

Fig. 12. Impact of increasing communication on the quality of load balance (kNeighbor running on 256 cores of Intrepid)

The ratio of number of objects in the application graph to the number of processors can also affect the quality of load balance. When work in an application is divided into more number of fine-grained objects, it gives additional flexibility and control to the load balancing strategy to migrate objects around. This should help all load balancers, though, by different amounts depending on how well they can exploit this degree of freedom. Figure 13 presents the performance of stencil4d when using different load balancing algorithms for different values of the average number of objects per core (the total amount of work across all processors was kept constant).

Fig. 13. Impact of number of objects per core on the quality of load balance (stencil4d running on 2048 cores of Intrepid)

When no load balancing is done, as the number of objects is increased, heavier objects are broken down into smaller ones and the default mapping by the runtime improves because of more numbers and finer granularity. Different load balancing strategies begin with a worse solution than is possible at two objects per core but all of them converge towards similar better performance at 16 objects per core. The only exception is ScotchLB that gives its best results to begin with and hence, does not shown any improvement as the number of objects is increased. It is advisable to choose a suitable ratio of objects to processors which creates a balance between good load balancing and avoiding overheads of excessive fine-graining.

V. Related Work

The problem of load balancing (also known as multiprocessor scheduling) of computational tasks is known to be NP-hard [25]–[27]. The load balancing of n jobs on to p processors is strongly NP-hard [25], [28]. 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, space filling curves to work stealing and graph partitioning.

Graph partitioning has been used for static load balancing of parallel applications for some time now [8]–[10]. METIS [7], Chaco [6] and SCOTCH [5] 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 rapidly growing sizes of parallel applications and machines. Parallel algorithms help reduce the time and memory requirements for partitioning large meshes/graphs.

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 one of the few general frameworks that supports dynamic load balancing of applications [12], [13]. It provides a suite of load balancing algorithms including parallel graph partitioning and also allows use of external libraries such as ParMETIS. However, it depends on the application to provide a cost model for the loads and the communication graph. Other frameworks such as DRAMA [29] and Chombo [30] provide load balancing capabilities for specific classes of parallel applications: finite element methods and finite difference methods respectively.

CHARM++ provides a framework similar to the Zoltan toolkit with inbuilt load balancing strategies and the option to deploy external libraries that provide load balancing algorithms [14]. The CHARM++ runtime does not depend on the application to provide the object graph, the cost models for which might be inaccurate. The runtime 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 [31] and hierarchical load balancing [32] schemes implemented in CHARM++ can also be exploited in conjunction with graph partitioning and will be discussed in future work.

VI. 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 different benchmarks by 20-70% over the existing load balancers in CHARM++. They also reduce the number of migrations, by several orders of magnitude in some cases, which should reduce the associated communication costs. ScotchRefineLB migrates 50 to 70 times fewer objects than MetisLB on Intrepid. For applications with irregular load distributions, SCOTCH-based load balancers outperform METIS by 7-11%. 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.

Future work involves developing an intelligent load balancing framework that can choose the best strategy (comprehensive versus refinement, favoring load balance versus minimizing the edge cut and choosing a good value for the allowed imbalance) 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.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-TR-501974). This research was supported in part by Le Centre national de la recherche scientifique (CNRS) and Région Aquitaine.

Neither the U.S. government nor Lawrence Livermore National Security, LLC (LLNS), nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, or process disclosed, or represents that its use would not infringe privately owned rights. The views and opinions of authors expressed herein do not necessarily state or reflect those of the U.S. government or LLNS, and shall not be used for advertising or product endorsement purposes.

This research used running time on Surveyor and Intrepid at the Argonne National Laboratory, which is supported by the U.S. Department of Energy under contract DE-AC02-06CH11357. Runs on Ranger were done under the TeraGrid [33] allocation grant ASC050040N supported by the National Science Foundation.

REFERENCES

[1] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[2] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu, "ParalleX: A study of a new parallel computation model," in *EEE International Parallel and Distributed Processing Symposium, 2007*, march 2007, pp. 1 –6.

[3] H. Kamal and A. Wagner, "FG-MPI: Fine-grain MPI for multicore and clusters," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010*, april 2010, pp. 1 –8.

[4] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, "Performance Evaluation of Adaptive MPI," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[5] "SCOTCH: Static mapping, graph partitioning, clustering and sparse matrix block ordering package," http://www.labri.fr/~pelegrin/scotch.

[6] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 1995, p. 28.

[7] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, 1996, p. 35.

[8] S. Attaway, T. Barragy, K. Brown, D. Gardner, B. Hendrickson, S. Plimpton, and C. Vaughan, "Transient solid dynamics simulations on the sandia/intel teraflop computer," 1997.

[9] J. Shadid, S. Hutchinson, G. Hennigan, H. Moffat, K. Devine, and A. Salinger, "Efficient parallel computation of unstructured finite element reacting flow solutions," *Parallel Computing*, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819197000550

[10] O. Sahni, M. Zhou, M. S. Shephard, and K. E. Jansen, "Scalable implicit finite element solver for massively parallel processing with demonstration to 160k cores," ser. SC '09. ACM, 2009.

[11] K. Devine, B. Hendrickson, E. Boman, M. S. John, and C. Vaughan, "Design of Dynamic Load-Balancing Tools for Parallel Applications," in *Proc. Intl. Conf. Supercomputing*, May 2000.

[12] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, "New challenges in dynamic load balancing," *Appl. Numer. Math.*, vol. 52, no. 2–3, pp. 133–152, 2005.

[13] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007, pp. 1–11, best Algorithms Paper Award.

[14] R. K. Brunner and L. V. Kalé, "Handling application-induced load imbalance using parallel objects," in *Parallel and Distributed Computing for Symbolic and Irregular Applications*. World Scientific Publishing, 2000, pp. 167–181.

[15] F. Pellegrini, "Static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proc. SHPCC'94, Knoxville*. IEEE, May 1994, pp. 486–493.

[16] S. T. Barnard and H. D. Simon, "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," *Concurrency: Practice and Experience*, vol. 6, no. 2, pp. 101–117, 1994.

[17] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proc. ACM/IEEE conference on Supercomputing (CDROM)*, Dec. 1995, 28 pages.

[18] G. Karypis and V. Kumar, "Multilevel graph partitioning schemes," in *Proc. 24th Intern. Conf. Par. Proc., III*. CRC Press, 1995, pp. 113–122.

[19] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitionning graphs," *BELL System Technical Journal*, pp. 291–307, Feb. 1970.

[20] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. 19th Design Automation Conference*. IEEE, 1982, pp. 175–181.

[21] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen, "A repartitioning hypergraph model for dynamic load balancing," *J. Parallel Distrib. Comput.*, vol. 69, pp. 711–724, August 2009.

[22] S. Fourestier and F. Pellegrini, "Adaptation au repartitionnement de graphes d'une mthode d'optimisation globale par diffusion," in *Proc. RenPar'20, Saint-Malo, France*, May 2011.

[23] C. Chevalier and F. Pellegrini, "Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework," in *Proc. Euro-Par'06, Dresden*, ser. LNCS, vol. 4128, Sep. 2006, pp. 243–252.

[24] C. Bernard, T. Burch, T. A. DeGrand, C. DeTar, S. Gottlieb, U. M. Heller, J. E. Hetrick, K. Orginos, B. Sugar, and D. Toussaint, "Scaling tests of the improved Kogut-Susskind quark action," *Physical Review D*, no. 61, 2000.

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

[26] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237 – 250, 1982.

[27] G. Malewicz, "Parallel scheduling of complex dags under uncertainty," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '05. New York, NY, USA: ACM, 2005, pp. 66–75.

[28] D. Applegate and B. Cook, "A computational study of the job-shop scheduling problem," *ORSA Journal of Computing*, vol. 3, no. 2, pp. 149 – 156, 1991.

[29] 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, "Dynamic load balancing of finite element applications with the DRAMA Library," in *Applied Math. Modeling*, vol. 25, 2000, pp. 83–98.

[30] "Chombo Software Package for AMR Applications," http://seesar.lbl.gov/anag/chombo/.

[31] A. Bhatelé, L. V. Kalé, and S. Kumar, "Dynamic topology aware load balancing algorithms for molecular dynamics applications," in *23rd ACM International Conference on Supercomputing*, 2009.

[32] G. Zheng, A. Bhatele, E. Meneses, and L. V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *International Journal of High Performance Computing Applications (IJHPCA)*, March 2011.

[33] C. Catlett *et al.*, "TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications," in *HPC and Grids in Action*, L. Grandinetti, Ed., vol. 16. Amsterdam: IOS Press, 2007, pp. 225–249.