LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-532851

Applying graph partitioning methods in measurement-based dynamic load balancing

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

February 27, 2012

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[1,⋆], Sébastien Fourestier[2], Harshitha Menon[3], Laxmikant V. Kale[3] and François Pellegrini[2]

[1] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
[2] Laboratoire Bordelais de Recherche en Informatique & INRIA Bordeaux
[3] Department of Computer Science, University of Illinois at Urbana-Champaign
⋆ bhatele@llnl.gov

Abstract. Load imbalance can lead to wasted CPU hours especially when running on a large number of processors. Achieving the best parallel efficiency for a program requires optimal load balancing which is a NP-hard problem. CHARM++, a migratable objects based programming model, provides a measurement-based dynamic load balancing framework. 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 a micro-benchmark and the NAS BT multi-zone benchmark. New algorithms developed in SCOTCH lead to better performance compared to other existing partitioners, both in terms of the application execution time and fewer number of objects migrated.

Keywords: load balancing, graph partitioning, dynamic, performance

1 Introduction

The efficient use of large 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. The problem of mapping these vertices 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 tasks 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 tasks per processor is on 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++ [10] and a graph partitioning library, Scotch [14] for implementing the new algorithms and heuristics. However, the techniques described here are generally applicable to other programming models and use of other graph partitioning libraries [9, 11].

The Charm++ runtime includes a mature load balancing framework. For applications in which computational loads tend to persist over time, the framework supports measurement-based load balancing. It 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. Based on different partitioning and repartitioning algorithms in Scotch, we have developed ScotchLB and ScotchRefineLB for comprehensive (fresh assignment of all tasks 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 Scotch-based load balancers with other existing (greedy and refinement) load balancing algorithms in Charm++ and with MeTiS and Zoltan-based load balancers. We use a micro-benchmark and the NAS BT multi-zone benchmark for comparisons and present results for runs on Intrepid (Blue Gene/P) and Steele (a Dell-Intel cluster). New algorithms developed in Scotch lead to better performance compared to other graph partitioners, both in terms of the application execution time and fewer number of tasks migrated.

2 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 [1,15]. 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. Zoltan [5] 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.

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 [4]. 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 load balancing strategies built in to Charm++, two of which are used in this paper for comparison with Scotch-based load balancers. *GreedyLB* is a comprehensive load balancer based on the greedy heuristic that maps the heaviest objects on to the least loaded processors iteratively until the load of all processors is close to the average load. *RefineLB* is 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. We also compare with *MetisLB* and *ZoltanLB*, strategies that use MeTiS (recursive graph partitioning) and Zoltan (hypergraph partitioning) respectively.

3 Scotch for graph partitioning and load balancing

Scotch [14] is a software project developed 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, making them available to the community as a software toolbox.

3.1 Static mapping methods in Scotch

Although Scotch also deals with combinatorial problems such as sparse matrix ordering, its first purpose was to compute static mappings by means of

graph algorithms. Two main classes of algorithms are used to compute static mappings: direct k-way methods and recursive bipartitioning methods. Both k-way and bipartitioning methods take advantage of a multilevel framework to reduce problem complexity and compute time. In this framework, the graph to bipartition or k-map is repeatedly coarsened by matching neighboring vertices. Then, an initial mapping is computed on the coarsest graph, and this mapping is prolonged back, from coarser to finer graphs [2, 9].

In order to ensure that the granularity of the multilevel solution is that of the original graph and not that of the coarsest graph, the prolonged mappings are refined at every level using local optimization algorithms. The most commonly used algorithms in literature are the Kernighan-Lin [12] (KL) and Fiduccia-Mattheyses [7] (FM) algorithms. The 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 its vertex pairing routine. The FM algorithm considers single vertex movements only, is therefore quasi-linear in time, and hence used often. SCOTCH uses two kinds of FM-based algorithms: one for optimizing bipartitions, and one for optimizing k-way mappings. These algorithms operate in two modes: when the current partition is heavily imbalanced, they first seek to restore the prescribed balance, even it if means increasing the edge cut metric. When the current partition is balanced, they seek to minimize the edge cut metric without getting out of the user-prescribed imbalance threshold. Although these algorithms are fast, we will see in Section 3.3 how using FM rather than KL may lead to problems for graphs with irregular load distributions.

The core static mapping algorithm that is used to compute mappings from scratch is the Dual Recursive Bipartitioning (DRB) algorithm [13]. It 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 a 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. Initial bipartitions are computed by way of locality-preserving greedy *graph growing* heuristics, that grow parts from randomly selected seed vertices [11].

Combining all of the above, the standard strategy for computing static mappings from scratch in SCOTCH 6.0 is to use k-way multilevel partitioning down to a size of 20 vertices per partition, after which the DRB algorithm is called on the coarsest k-way graph. Bipartitions are computed in the DRB algorithm using a multilevel method of greedy graph growing for computing the initial bipartition on the coarsest graph.

In this paper, the size of the test graphs and the desired number of partitions limits the use of k-way methods to a few levels only. Using a k-way framework around the DRB algorithm is however still beneficial, because even a single k-way FM on the finest graph can compensate for the load imbalance accumulated in the course of multiple levels of recursive bipartitioning.

3.2 Repartitioning methods in Scotch

Another new feature of Scotch 6.0 is the ability to compute remappings of a graph, based on an existing mapping. In the context of this paper, target topologies are assumed to be homogeneous, and remapping degenerates into repartitioning. This scheme is referred to as refinement load balancing in Charm++. 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 [5]). 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.

All of the aforementioned mapping algorithms have been adapted to take into account the information borne by the fictitious edges. In most cases, these edges do not have to be added to the graphs, thus reducing the cost of the method. For instance, in the DRB algorithm, bipartitioning algorithms already account for a bias. The estimated distance of graph vertices to ends of edges that have been cut at previous stages of the recursive bipartitioning process is modeled, and it is straightforward to add the migration penalty to this bias. The coarsening phase does not require the presence of fictitious edges either. It is only during the local refinement phases that they are necessary, yet on smaller, band graphs.

3.3 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 highly 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 children 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 that have been experimented with in this paper comprise of: (i) the classical dual recursive bipartitioning (or static mapping) method of Scotch (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.

4 Case Studies

We compare the performance of different load balancing strategies using a micro-benchmark and the NAS BT multi-zone benchmark on multiple machines.

kNeighbor is a micro-benchmark with a near-neighbor communication pattern. In this benchmark, each object exchanges messages with a fixed set of objects in every iteration. Each object is assigned some amount of computational load.

BT_MZ is one of the multi-zone applications in the NAS Parallel Benchmark suite (NPB) [16]. It is a parallel implementation for solving a synthetic system of non-linear PDEs using block tridiagonal matrices. It consists of uneven size zones within a problem class and hence is useful for testing the effectiveness of a load balancer. 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.

The experiments were run on Intrepid and Steele. 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 principal 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 Steel 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.

We compare performance of Scotch-based load balancers with load balancers in Charm++, GreedyLB and RefineLB, and also with MeTiS and Zoltan-based load balancers. For MetisLB, both recursive bipartitioning and

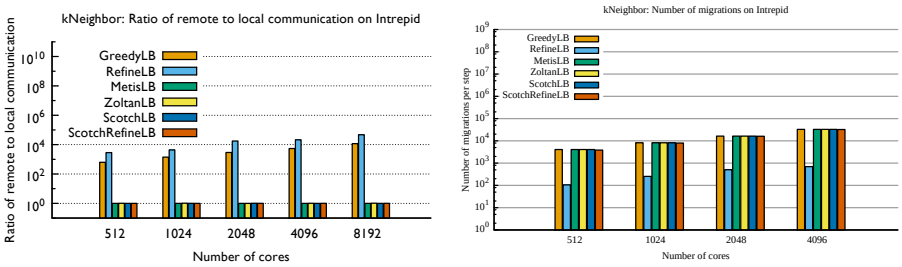

Fig. 1. Ratio of local to remote communication and number of migrations for kNeighbor

k-way multilevel partitioning were used. For ZoltanLB, *hypergraph* partitioner is used with the *partition* and the *re-partition* method. Finally, for SCOTCH-based load balancers, two flavors of the mapping methods, `STRAT_QUALITY` and `STRAT_BALANCE`, were tried. In all of the above cases, we report the result of the better performing strategy. As mentioned earlier, 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. Hence for ScotchRefineLB results, ScotchLB is invoked once, when program execution begins, followed by several calls to ScotchRefineLB.

The following metrics are used to compare the performance of the load balancing algorithms: (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 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. (4) Total application time, which includes the time for the iterations, load balancing strategy and migration.

4.1 Comparisons using kNeighbor

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

Figure 1 (left) 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 close to one i.e. restricting

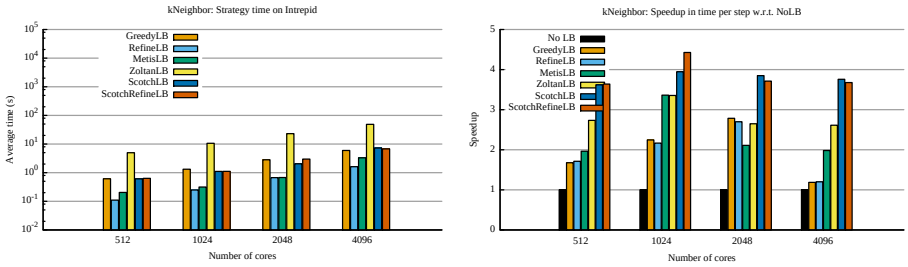

Fig. 2. Time spent in the load balancing algorithm and the speedup in the execution time per step for the application (kNeighbor on Intrepid)

half of the total communication to within a processor. In contrast, for other load balancers, this ratio is at least an order of magnitude higher denoting excess of remote communication.

Figure 1 (right) presents the number of migrations as a result of performing load balancing. Refinement load balancers, RefineLB and ScotchRefineLB, which take the migration cost into account, successfully reduce the number of migrations. However, there exists a tradeoff between reducing the number of migrations and improving application performance. ScotchRefineLB results in moderate number of migrations but obtains best performance whereas RefineLB, which performs the least number of migrations, suffers from performance issues.

Next we compare the time spent in the load balancing strategy (see Figure 2, left). We find that, as the number of cores increases, the strategy time for all the load balancers increases. RefineLB is the fastest 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. As seen in Figure 2 (right), the load balancers based on Scotch consistently outperform all the other load balancers. On 8192 cores, ScotchRefineLB gives 42% better performance than MetisLB. When using ScotchRefineLB, ScotchLB is called once in the beginning which improves the performance by 15% compared to NoLB. The speedup from using ScotchRefineLB at 8.192 cores is 1.8.

Finally, Figure 3 presents results for a complete run of the kNeighbor benchmark. During this run, we perform load balancing once every 10000 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 seen in Figure 3, that ScotchRefineLB consistently gives the best performance on all system sizes. It gives a performance benefit of up to 26% in comparison to other load balancers. When compared to the baseline, NoLB, ScotchRefineLB and ScotchLB give 31% and 18% improvement respectively (overall speedups of up to 1.4 and 1.2).

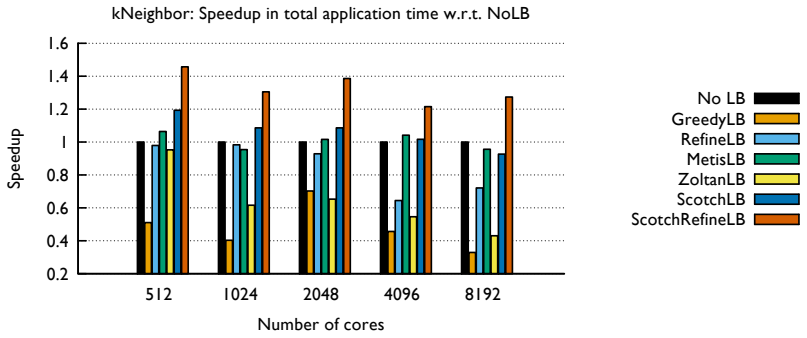

Fig. 3. Speedup in the total application time for kNeighbor on Intrepid

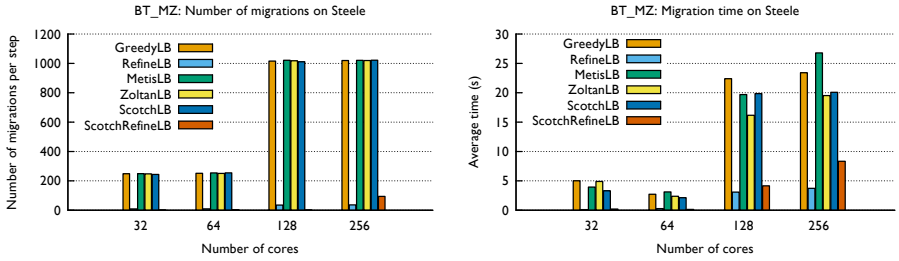

Fig. 4. Number of migrations and the migration time for BT_MZ on Steele

4.2 Comparisons using BT_MZ

In this section, we present the results for the NAS BT multi-zone benchmark runs on Steele. For these experiments, the number of objects per processor varies with the class of the benchmark used and the system size. For example, a run of class D on 256 processors will have, on average, four objects per processor. As in kNeighbor, the baseline run, in which no load balancing is performed, is called NoLB. We ran class C on 32 and 64 cores and class D on 128 and 256 cores.

Figure 4, presents the number of migrations and the migration time for different load balancing strategies. 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 4, nearly an order of magnitude smaller than other balancers, in some cases.

Figure 5 (left) 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 5 (right) 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

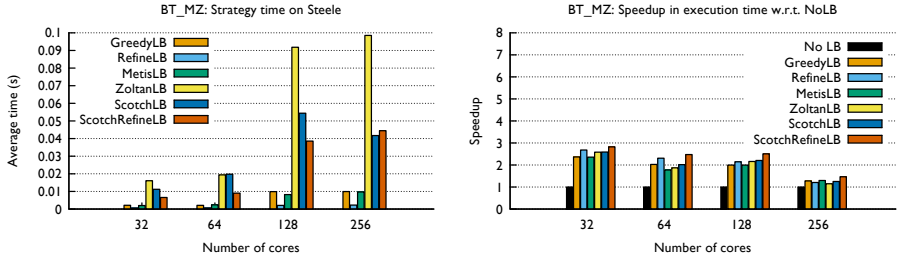

Fig. 5. Strategy time and speedup in the time per step for BT_MZ on Steele

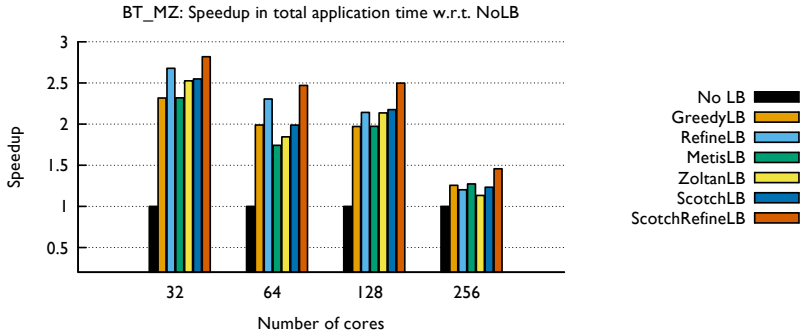

Fig. 6. Speedup in the total application time for BT_MZ on Steele

comparison to NoLB. In comparison to other load balancers, ScotchRefineLB performs better by 11%.

The results for a complete run of BT_MZ in which load balancing is performed once in every 1000 iterations are presented in Figure 6. 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 5. ScotchRefineLB performs better than all other load balancers consistently 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.

5 Related Work

The problem of load balancing (also known as multiprocessor scheduling) of computational tasks is known to be NP-hard [8]. The load balancing of n jobs on to p processors is strongly NP-hard. 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 [1, 15]. METIS [11], CHACO [9] and SCOTCH [14]

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.

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 [5]. It provides a suite of load balancing algorithms including parallel graph partitioning and also allows use of external libraries such as ParMeTiS. Other frameworks such as DRAMA [3] and Chombo [6] 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 [4]. 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 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 kNeighbor and the NAS BT multi-zone benchmark 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 1.2 to 1.4 for kNeighbor and 1.5 to 2.5 for Class D 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.

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-532851). This research was supported in part by Le Centre national de la recherche scientifique (CNRS) and Région Aquitaine.

References

1. Attaway, S., Barragy, E., Brown, K., Gardner, D., Hendrickson, B., Plimpton, S., Vaughan, C.: Transient solid dynamics simulations on the sandia/intel teraflop computer. In: ACM/IEEE Supercomputing Conference (Nov 1997)
2. Barnard, S.T., Simon, H.D.: A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Concurrency: Practice and Experience 6(2), 101–117 (1994)
3. Basermann, A., Clinckemaillie, J., Coupez, T., Fingberg, J., Digonnet, H., Ducloux, R., Gratien, J.M., Hartmann, U., Lonsdale, G., Maerten, B., Roose, D., Walshaw, C.: Dynamic load balancing of finite element applications with the DRAMA Library. In: Applied Math. Modeling. vol. 25, pp. 83–98 (2000)
4. Brunner, R.K., Kalé, L.V.: Handling application-induced load imbalance using parallel objects. In: Parallel and Distributed Computing for Symbolic and Irregular Applications. pp. 167–181. World Scientific Publishing (2000)
5. Catalyurek, U.V., Boman, E.G., Devine, K.D., Bozdağ, D., Heaphy, R.T., Riesen, L.A.: A repartitioning hypergraph model for dynamic load balancing. J. Parallel Distrib. Comput. 69, 711–724 (August 2009)
6. Colella, P., Graves, D., Ligocki, T., Martin, D., Modiano, D., Serafini, D., Van Straalen, B.: Chombo Software Package for AMR Applications Design Document (2003), `http://seesar.lbl.gov/anag/chombo/ChomboDesign-1.4.pdf`
7. Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: Proc. 19th Design Automation Conference. pp. 175–181 (1982)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)
9. Hendrickson, B., Leland, R.: A multilevel algorithm for partitioning graphs. In: Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM). p. 28. ACM, New York, NY, USA (1995)
10. Kalé, L., Krishnan, S.: CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: Paepcke, A. (ed.) Proceedings of OOPSLA'93. pp. 91–108. ACM Press (September 1993)
11. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. In: Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM). p. 35 (1996)
12. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitionning graphs. BELL System Technical Journal pp. 291–307 (Feb 1970)

13. Pellegrini, F.: Static mapping by dual recursive bipartitioning of process and architecture graphs. In: Proc. SHPCC'94, Knoxville. pp. 486–493. IEEE (May 1994)
14. SCOTCH: Static mapping, graph partitioning, clustering and sparse matrix block ordering package. `http://www.labri.fr/~pelegrin/scotch`
15. Shadid, J., Hutchinson, S., Hennigan, G., Moffat, H., Devine, K., Salinger, A.: Efficient parallel computation of unstructured finite element reacting flow solutions. Parallel Computing 23(9), 1307 – 1325 (1997)
16. der Wijngaart, R.F.V., Jin, H.: NAS Parallel Benchmarks, Multi-Zone Versions. Tech. Rep. NAS Technical Report NAS-03-010 (July 2003)