iC2mpi: A Platform for Parallel Execution of Graph-Structured Iterative Computations

Harnish Botadra, Qiong Cheng, Sushil K. Prasad
Georgia State University
Atlanta, Georgia, U.S.A.
{hbotadra1, qcheng1}@student.gsu.edu,
sprasad@cs.gsu.edu

Eric Aubanel and Virendra Bhavsar
University of New Brunswick
Fredericton, New Brunswick, Canada
{aubanel, bhavsar}@unb.ca

Abstract

Parallelization of sequential programs is often daunting because of the substantial development cost involved. Various solutions have been proposed to address this concern, including directive-based approaches and parallelization platforms. These solutions have not always been successful, partly because many try to address all types of applications. We propose a platform for parallelization of a class of applications that have similar computational structure, namely graph-structured iterative applications. iC2mpi is a unique proof-of-concept prototype platform that provides relatively easy parallelization of existing sequential programs and facilitates experimentation with static partitioning and dynamic load balancing schemes. We demonstrate with various generic application graph topologies that our platform can produce good performance with very little effort. The iC2mpi platform has a good potential for further performance improvements and for extensions to related classes of application domains. Because of the relative ease of implementation, this platform can also serve as a model for developing other domain-specific platforms.

Keywords: Parallelizing platform, Graph-based iterative computations, Message Passing Interface (MPI), Modular Architecture, Third-party Plug-ins for Partitioning.

1. Introduction

Many application domains, wherein distributed computation has been successfully employed, fall into the class of iterative computations – this class can be characterized by underlying mesh or graph structured application programs and iterative local computations over nodes, dependent only on the neighboring nodes. Examples of such application domains include many time-stepped simulations, such as battlefield management [5], weather forecasting [10], or fluid dynamics [8], and mesh-structured computations, such as difference equations [16], finite element methods [17], and cellular automata [2]. Despite sharing essentially similar program structures, there is no generic framework to help the programmers of these applications to easily transition from their sequential implementations to distributed machines or to grid platforms (see Section 5 for relevant literature). Our iC2mpi platform addresses this and related issues.

Current Limitations: Typically, scientists and engineers have proven sequential C/C++ codes and converting these to distributed versions in MPI entails challenges of explicit parallel programming, debugging, and revalidating. Most applications programmers, therefore, are limited to automatic loop-based parallelization, or to inserting compiler directives (as in OpenMP [14] and HPF [22]). If distributed versions are manually developed (usually employing MPI), quite often these programs hardcode a specific pattern of domain decomposition to statically partition the application program graph/mesh among the processors of a target machine topology, aiming to best balance computational load and minimize communication. Therefore, the resultant MPI code that programmers develop typically requires code changes to study various static partitioning techniques, as these are not developed in a framework-like fashion to easily enable such performance studies. When load can be unpredictable, for example in a battlefield simulation where combat zones form dynamically, they also need to employ dynamic load balancing, which requires even more versatile design of the distributed programs, further challenging an application programmer.

On the other hand, the algorithm designers of graph partitioning packages, such as Metis [11], Jostle [21], and PaGrid [20, 6], are also limited as they can only estimate the efficiency of their techniques analytically. There is currently no general-purpose test-bed available that allows them to easily plug-in their algorithms, and execute and verify the performances on various program graphs and processor architectures.

Goals for the Platform: This paper describes our project to develop a suitable platform with the following goals.

1. Design an MPI-based platform with an open architecture for the class of iterative graph-structured application programs, possibly with dynamically varying computational loads, into which application programmers can *plug-in* the code and the data structures for their computational nodes, the graphs

1-4244-0910-1/07/$20.00 c2007 IEEE.

for their application programs and for the processor network, and the third-party algorithms for partitioning and load balancing.

2. Enable application programmers to
   a. easily execute their sequential code for iterative computations on distributed architectures without any code change in their node computations or in the basic node data structures, and without any MPI coding, and
   b. to compare the performance of different static graph partitioners, and the impact of various dynamic load balancing and repartitioning techniques, without any additional coding.
3. Enable designers of algorithms for graph partitioning and for dynamic load balancing to validate the efficiency of their techniques by actual execution over a variety of graph-structured iterative computations and load characteristics on different parallel and distributed architectures and heterogeneous grids instead of typical analytical estimation.
4. Enable carrying out of refinements and performance tuning for efficient computation and communication on the platform itself to impact the entire class of relevant iterative computations.
5. Demonstrate that a domain-specific platform can be readily built, using existing components, and that this provides an attractive alternative to other approaches, from more general platforms to hand-coded MPI.

iC2mpi Solution: Our resultant platform, namely iC2mpi (for enabling transition from an *i*terative *c*omputation in *C* to an *MPI*-enabled execution), is our first prototype demonstrating a proof of concept. Its initial goal has been to architect a generic platform, fulfilling Goals 1, 2, and 3. Although efficiency enhancements of the platform have not been taken up yet (Goal 4), we do obtain reasonably good speedups over a variety of example codes (speedups of 12-14 on 16 processors). One important conclusion of this work is the relative ease with which this platform was built, a result of its being restricted to iterative graph-based applications, and the use of tools and ideas from the literature. This suggests that this approach be more widely used (Goal 5).

Section 2 describes the overall architecture of iC2mpi, and Section 3 gives the details of its internal algorithms and data structures. Section 4 shows the results of various experiments with different iterative applications on the iC2mpi platform. We employ Metis and PaGrid as examples of third-party plug-ins to study various static partitioning schemes. We have also developed our own simple repartitioning heuristic plug-in to experiment with dynamic task migrations. We also demonstrate that the overhead of the platform is acceptably low and scales well. Section 5 briefly reviews related work and compares them with the current approach, and Section 6 concludes by discussing ongoing and future work in extending the iC2mpi platform and improving its performance.

2. Overview of Architecture and Design Issues

Platform Architecture: Due to the goals 1-3, the architecture had to be a layered architecture with the application program interfacing only with the two third party components, namely, the graph partitioner and the load balancer, in addition to the platform itself, and not with the underlying MPI communication infrastructure. Figure 1 shows our layered architecture view of the platform. The application program provides the application graph, node data structures and the node computation function as user plug-ins to the platform. The platform uses a static graph partitioner for the initial partitioning. A dynamic load balancer is incorporated in the platform for load balancing of dynamic domain applications. The platform itself uses an MPI approach for parallelization, one of the most widely used methods to achieve parallelism on today's clusters and multiprocessor supercomputers [19]. The need to flexibly incorporate any third party partitioner also dictated a standard format for plug-ins. We employed Chaco format [11] for the application program graph as input to the partitioners employed, namely Metis and PaGrid. Figure 2 shows the detailed platform architecture, and explicitly indicates user plug-in points (application program graph, node data structures, and node computation function) and the interaction of the platform components with the data structures they use. Solid templates are the platform components.

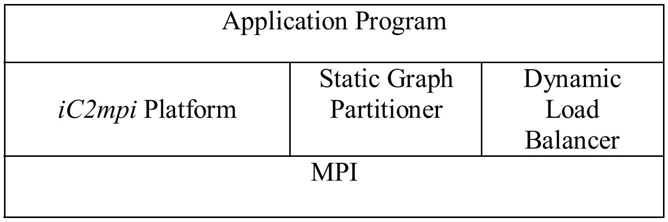

Figure 1: Layered Architecture View of the iC2mpi Platform

Data Structures: Selection of data structures is central to the efficient solving of irregular concurrent problems on distributed memory architectures [15]. Two issues dictated the choice of data structures: flexibility and fast access. The processors themselves would need to maintain an arbitrary and variable number of program nodes and their connectivity with their neighboring nodes – this precluded array based structures. However, the processor would need fast access to a node for computing over them or to communicate shadow nodes to neighboring processors. A hash table providing quick link to a node based on its global id was a natural choice [15]. In addition to the data of the nodes owned by the processor (some are internal nodes and others are peripheral nodes, the ones which have at least one neighboring node on another processor), it also needs to maintain shadow node information locally (the non-local nodes which are neighboring to its peripheral nodes). The data structures set up at each processor, thus, are (i) a list of data nodes for maintaining the physical node data allocated to the processor, (ii) two lists of pointers, one to the internal nodes and other to the peripheral nodes, and (iii) a hash table containing pointers to data nodes. Note that the data node list not only includes data for the nodes owned by the processor but also the shadow node data. By keeping the peripheral nodes separate, communication buffers to neighboring processors can be setup while computing over these nodes. A modulo hash function suffices on the node global ID to obtain the location for node data. The buckets (sorted linked lists) maintain pointers to the node data in the data node list.

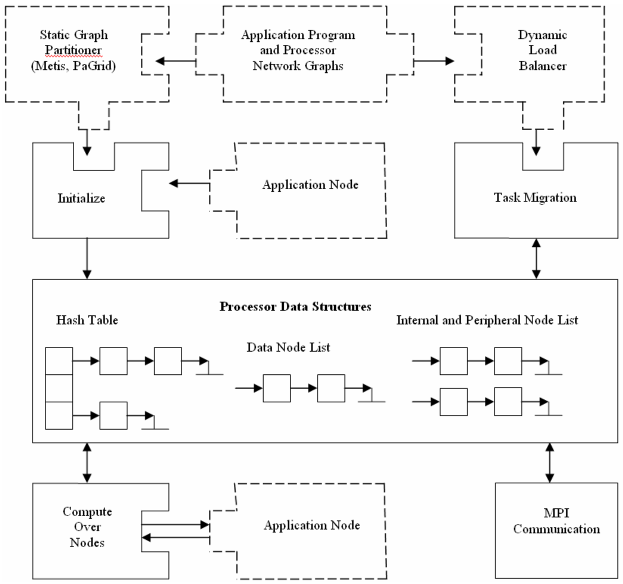
Static Graph Partitioner (Metis, PaGrid)
Application Program and Processor Network Graphs
Dynamic Load Balancer
Initialize
Application Node
Task Migration
Processor Data Structures
Hash Table
Data Node List
Internal and Peripheral Node List
Compute Over Nodes
Application Node
MPI Communication

Figure 2: Modular Architecture of iC2mpi Platform

Overheads: A generic platform needs to contend with overheads which most MPI programs can do away with or optimize. One key issue is how to invoke a generic function to compute over an application node from the platform? For complete flexibility, for each node, currently we pass a linked list of that node as the header node followed by its neighbors to the user supplied node computation function. Of course, arrays/vectors are alternatives for cases when partitions remain static throughout or change only periodically. Another overhead, within each iteration, is the need to explicitly prepare a communication buffer containing the shadow node data for each neighboring processor. Again, these buffers can be initialized once and updated directly in place, if no dynamic load balancing is needed. In the latter case, such initialization can occur after each task migration. We retain the basic approach incurring this overhead for maximum flexibility, with an eye toward extending the platform to adaptive meshes. As evidenced in Section 4, these overheads are tolerated to a good extent.

3. Algorithmic and Implementation Issues

There are three major phases involved in the platform execution: initialization, computation & communication, and load balancing & task migration.

Initialization Phase: During this phase, all the processors initialize data structures in their local memories to maintain graph connectivity of the iterative problem, node information and node data. The initial input graph and the initial node-to-processor mapping yielded by graph partitioner is stored into appropriate data structures, for easy access and retrieval. Besides these, hash tables are also set up in the local memory of each processor.

Computation & Communication Phase: During this phase, the processor performs computations for each of its nodes using data of the neighboring nodes. For each node it owns, it forms a list comprising of the current node's data as the head followed by the data of the neighbors to be passed to the application node computation function. The platform maintains a pointer to the application node function supplied by the user. This allows for a clean and robust decoupling between the iC2mpi platform and the application program code. Internal nodes are updated followed by the peripheral nodes. As the peripheral node data is updated, this updated data is packed into communication buffers even as the next peripheral node is ready to be updated.

For physical communication of these buffers, the structure type used to set up the buffers is committed to an MPI data type (using MPI_Type_commit), since it is not a standard MPI data type but a derived data type. All the processors send these buffers at one go. MPI_Isend(), a non-blocking call, is used for sending these buffers to appropriate processors. MPI_Recv() receives these buffers which are then used to update the locally maintained shadow node information.

A version of iC2mpi employs MPI_Irecv() to overlap the computations with communications, by processing the peripheral nodes first and dispatching the shadow nodes to neighboring processors, and while the communication takes place, proceeds with the processing of the internal nodes. These and other performance enhancements are currently underway – the results reported in Section 4 employs the basic prototype of iC2mpi platform.

Load Balancing & Task Migration Phase: Dynamic applications require periodic load balancing, since the computational workload of individual nodes may change throughout the course of computation. Repartitioning should not only balance the workload among the processors but also keep the edge-cut to a minimum, so as to minimize the inter-processor communication [3]. We now briefly describe a centralized heuristic algorithm used in the platform in order to attain dynamic load balancing. A third party plug-in is possible here.

1) A weighted processor network graph is set up. The execution time of the processors for a specific number of iterations represents the weight on the nodes and the weight of the edge connecting two processors is the amount of communication between the two estimated by the length of the communication buffers.

2) A designated processor examines the processor graph so as to measure the relative workload on the processors spread across the computational domain. The processor doing 25% more work (obtained from the node weights) than all its neighbors is considered to be the 'busy' processor. The one among its neighbors doing the least amount of work is labeled as an 'idle' one. The central processor obtains all such 'busy-idle' pairs from the processor graph.

3) As a final step, we just need to decide upon the task that should be migrated from the 'busy' processor to the 'idle' processor. The task which keeps the edge-cut to a minimum becomes the candidate for 'task migration.' For example, in Figure 3, out of the two nodes A and B which could be migrated from processor 0 to processor 1, we choose the migration of node B. Migrating node A would cause three edges to cross the boundary of processor 0, which would increase the overall edge cut by 2, and so we choose B in this case.

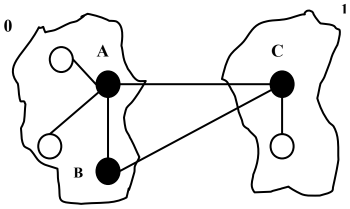

Figure 3: Choosing B over A for task migration, keeping edge-cut to a minimum

Task Migration: A single task migration involves the 'busy' processor which sends the task, an 'idle' processor which receives the task, and a set of processors which hold shadows for the migrating node. Changes are made to the data structures of the sending and receiving processors, as well as the processors that hold neighbors of the migrating node. In iC2mpi platform, many such migrations occur in parallel across the computational domain. Our distributed algorithm for migration module ensures that all such simultaneous migrations can be accomplished without any conflict. For dynamic applications, the data redistribution cost for load balancing can be comparable to the actual time for computation [18]. Therefore, it is only apt to leverage the maximum out of the overhead incurred by the load balancing routine to gather the load statistics across the computational domain.

4. Experimental Results

The experimental results were conducted on a Silicon Graphics Origin-2000 computer with 24 CPUs. It has CRAY link interconnects with hypercube NUMA architecture. Two generic applications with hexagonal grid and random graph topologies are parallelized using the iC2mpi platform. In each case, the application node function and data structure was prepared and plugged into the iC2mpi platform with relative ease. A dummy 'for loop' is used to inject an appropriate fine (3 ms) or coarse (30 ms) grain size on the nodes. Most data corresponds to average over five different fine grained graphs using Metis static graph partitioner, unless specified otherwise.

4.1. Hexagonal Grids

We first tested the performance of the platform on hexagonal grids, which is typical for such simulations as battlefield management [5]. Each node computes the average of the data maintained by all its six neighbors. Figure 4 shows the speedup plots for 20 iterations on 32, 96 and 160-node fine grained hexagonal grids. For smaller graphs, the increase in speedup reduces as we increase the number of processors, due to the increase in the communication overhead with the number of processors. But, for larger graphs, we obtain considerable speedup with increasing number of processors. For instance, for the 160-node hexagonal grid using 16 processors we obtain a speedup of 12.8 (self-relative). For large graphs, the communication overhead remains more or less the same with the increase in the number of processors, while the computation workload gets divided among the processors resulting in large speedups. We further explore the overheads later in the section.

Next, we compare the performance of the platform with the dynamic load balancing utility. We create inertial and incremental imbalance which is hard to capture for static partition: For the first 33% of the iterations, the first 50% nodes use coarse grain size and the rest of the nodes use fine grain size. Similarly, for the next 33% of the iterations, nodes ranging for 25% to 75% use coarse grain size while the rest use fine grain size. Finally for the final 33% of the iterations, nodes ranging from 50% to 100% use coarse grain size and rest of the nodes use fine grain size. Speedups are measured for 25 iterations and load balancing routine is invoked every 10 time steps. Figure 5 shows the comparison plots for the static and dynamic load balancing utilities of the platform. Dynamic partitioning slightly outperforms the static partitioning as we increase the number of processors. We expect this performance difference to widen further for systems where load changes are more dynamic and excessive.

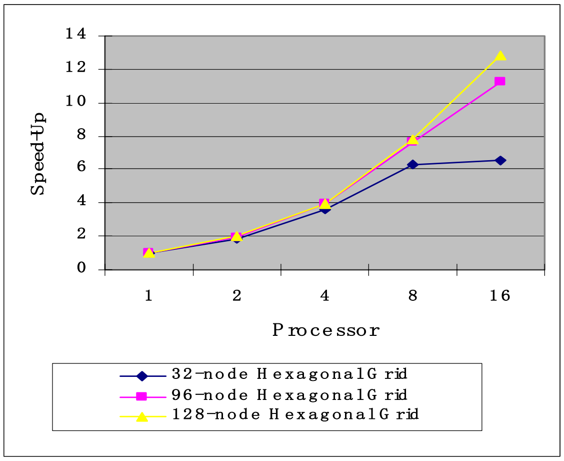

Figure 4: Speedup for Hexagonal Grids

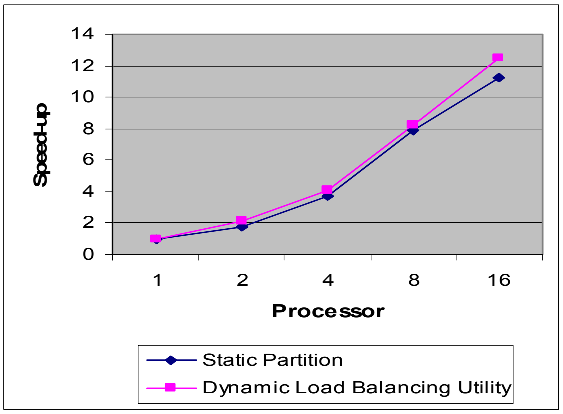

Figure 5: Static vs. Dynamic Partitioning on 128-node Hexagonal Grids

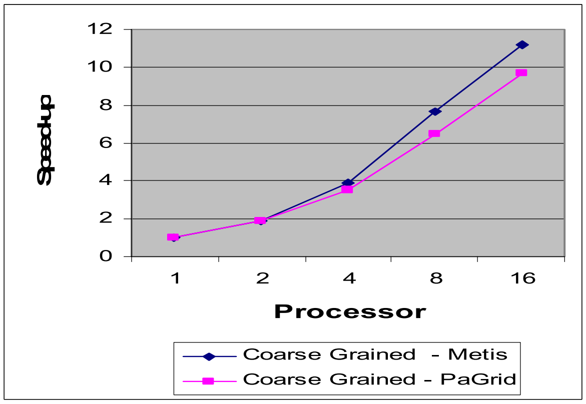

Figure 6: Metis vs PaGrid for Coarse Grained 160-node Hexagonal Grids

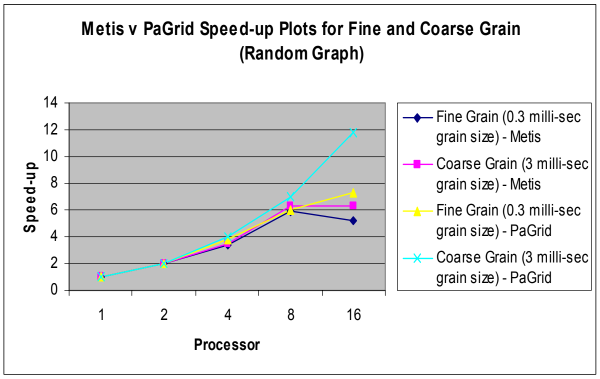

Figure 7: Metis vs. PaGrid on Fine & Coarse Grained 64-node Random Graphs

The platform allows users to choose the appropriate static partitioner which provides optimum performance for the iterative application on hand. In Figure 6, we compare the performance of Metis and PaGrid. This figure shows the speedup plots for 20 iterations for 160-node coarse-grained hexagonal grid using Metis and PaGrid as the static graph partitioners. The processor network graph used for PaGrid (hypercube) used the grid format specified in [20, 6]. Of particular interest is the “Rref” parameter for PaGrid, defined as the ratio of communication time to the computation time per node in the application graph. For the graph topologies discussed here, we used Rref = 0.45. For Metis, we set the parameter fmt = 0, indicating uniform weighted program graph; Metis does not use processor network graph [11]. From the plot, we can see that Metis provides better speed-ups as compared to PaGrid as expected for large regular hexagonal graphs – the latter is known to perform better for heterogeneous networks. This is demonstrated for the random graph with 64 nodes, wherein PaGrid does better than Metis, as shown in Figure 7.

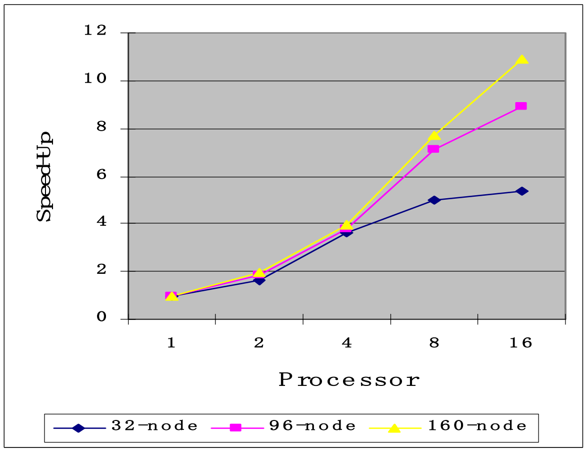

Figure 8: Speedup for Random Graphs with Static Partition (Metis)

4.2. Random Graphs

We experimented with random graphs on the iC2mpi platform to ascertain that the platform copes with communication irregularity gracefully. Figure 8 shows the speedup plots for 20 iterations for 32 and 96 and 160-node fine grained random. Even with the random nature of the graphs, we obtain good speedups for larger graphs as we increase the number of processors. Figure 9 shows the performance comparison of the random graphs with dynamic load balancing. Speedups are calculated for 25 iterations, and load balancing routine is invoked every 10 time steps for the dynamic load balancing approach. Dynamic partitioning captures the random nature of the graph and the load imbalance across the computational domain to balance the work load among the processors periodically resulting in better speedups while the static partitioning falls short.

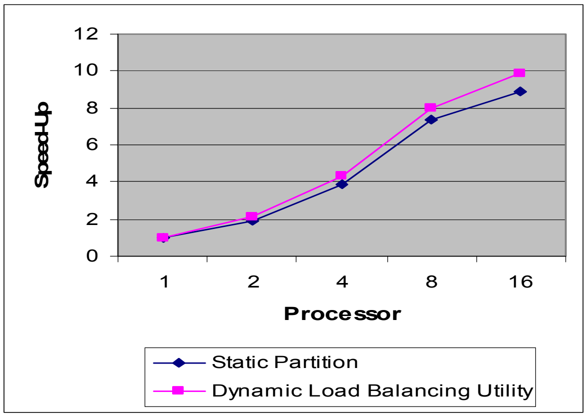

Figure 9: Performance of Dynamic Partitioning on 128-node Random Graphs

4.3. Measurement of Overheads

We measured the current overheads of the iC2mpi platform's various phases. We plot the overheads for fine-grained 64-node hexagonal grids (with dynamic load balancing) and for 256-node random graphs (without dynamic load balancing) for 35 iterations each in Figure 11 and 12, respectively. Here the phases and overheads are defined as follow: (i) Initialization: Includes setting up the data structures for graph connectivity, node-to-processor mapping, internal and peripheral node lists, data node lists and hash tables; (ii) Computation Overhead: Setting up the list of the current node and its neighbors to be passed on to the application node function for node computation, and updating of the data node lists after computation; (iii) Compute: Actual node computation; (iv) Communication Overhead: Packing buffers for communication, unpacking the received buffers and updating the data node lists; (v) Communicate: Communication of the shadow node information to appropriate processors and receiving the required information from neighbors; and (vi) Load Balancing & Task Migration: Gathering information about load imbalance across the computational domain and balancing the work load among the processors. A key observation from the bar graphs is that both the communication overhead and the actual communication between the processors, more or less remains the same as we increase the number of processors. Thus, the platform does a good job of keeping the communication overhead in check and prevents it from limiting the scalability. Meanwhile the time for computation reduces with the increase in the number of processors due to the division of work load among the processors. This is the key to large speedups obtained as we increase the number of processors with increasing size of the graphs as evidenced from Figure 12.

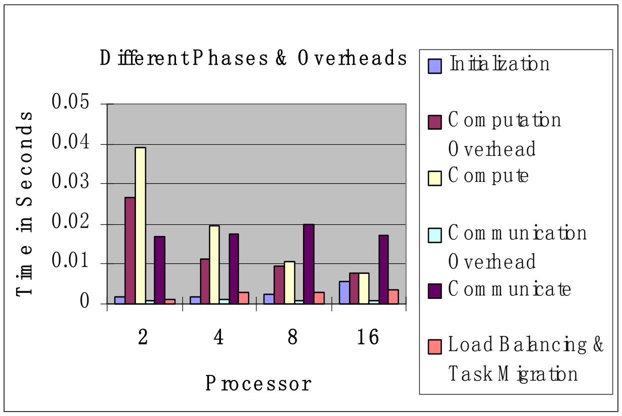

Fig. 11: Overheads for 64-node Hexagonal Grids with dynamic load

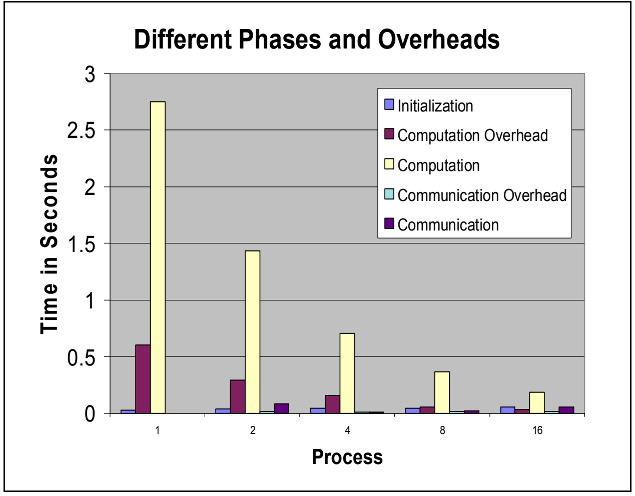

Fig. 12: Overheads for 256-node fine grained Random Graphs

5. Related Work

Existing alternatives to time-consuming manual parallelization with MPI for application programmers range from the use of compiler directives to assist the compiler in parallelizing sections of a sequential program to the use of parallel utility libraries and high-level frameworks. The first approach has mainly been implemented in High Performance Fortran [22] for multicomputers and OpenMP [14] for multiprocessors. The former has had limited success for applications with unstructured communication patterns, where the parallelization effort is comparable to using MPI [19], although additional directives have been proposed to address this issue [15]. Even if this limitation is addressed by future HPF implementations, a high-level framework such as iC2mpi can be an attractive alternative as the former still requires significant code modification and exposes data distribution to the user, while the latter hides the technicalities associated with parallelism. OpenMP is mainly limited to expensive multiprocessor platforms, can produce non-deterministic programs that are hard to debug, and requires significant work (comparable to MPI) to achieve highly scalable programs [9].

The use of parallel libraries can significantly reduce the time required to parallelize an application. Zoltan [4] is the library that comes closest to this work. Zoltan is a library of data management services for parallel, unstructured and dynamic applications. Zoltan basically simplifies load balancing, data movement, unstructured communication and memory usage difficulties that arise in dynamic applications such as adaptive finite-element methods. Zoltan provides utilities to assist in the development of a parallel application, while our platform does not require the user to do any parallel programming. Zoltan provides utilities like distributed data directories for locating off-processor data and a communication library to incorporate changing communication patterns for dynamic applications. In our platform, these utilities are built-in, so the user need not worry about them.

Paramesh is a Fortran 90 toolkit for parallel adaptive mesh refinement applications [12]. It provides a package of subroutines that ease the parallelization of sequential mesh-based application, and also ease the adoption of adaptive mesh refinement. It is restricted to regular Cartesian meshes, however, whereas our platform applies to arbitrary unstructured meshes.

High-level frameworks hide the complexities associated with parallel programming, allowing for much more rapid development. KeLP [1] is a framework that assists in the implementation of parallel applications involving block decompositions of structured data. It is intended for applications that adapt to data-dependent or hardware dependent conditions at run time. Our framework does not share KeLP’s restriction to structured data and is designed for iterative applications employing unstructured data. CO_2P_3S is a parallel programming system that combines three abstraction techniques, namely, object-oriented programming, design patterns, and frameworks, using a layered programming model which supports fast development of parallel programs and fine tuning of the resulting programs for performance. CO_2P_3S uses design patterns to ease the effort required to write parallel programs [13]. While CO_2P_3S emphasizes easy parallelization of a range of applications based on user selection of pattern templates and fine-tuning of the parallel programs for performance, there are no dynamic load balancing capabilities incorporated in the mesh framework. While CO_2P_3S does well to separate the application-independent framework structure from application-dependent code, it does not ensure that the mesh framework employed by it is a black box from the point of view of the user, whereas in our iC2mpi platform, the user needs to model the iterative application as per the platform specification without worrying about its architecture.

6. Conclusions and Future Work

We have presented a unique proof-of-concept prototype platform for parallelization of iterative graph-structured applications. It provides a relatively easy transition from sequential programs to their distributed executions, and facilitates experimentation with static partitioning and dynamic load balancing schemes. We demonstrated with two generic iterative applications with underlying hexagonal and random graph structures that our platform can produce good performance with very little effort. The iC2mpi platform has good potential for further improvements as indicated and extensions. It also can serve as a model for other domain-specific platforms. Future work will include applying our platform to real applications, to assess the balance between the performance impact of optimizations produced by the compiler or manually and the reduction in development time resulting from the use of the platform. The performance of the iC2mpi platform can still be improved, and future work will address this by reducing its overheads, and extending it to adaptive mesh-based applications. We will also explore extending it to applications that use the BSP model [7], as this model essentially divides the computation from communication phases as iC2mpi does. Finally, we will employ the platform to perform comprehensive evaluation of static and dynamic partitioners.

Acknowledgements: We thank Yan Chen for helping run some of the experiments to collect data and draw plots. VB and EA acknowledge support from NSERC Canada.

References

[1] S. Baden, The KeLP Programming System, http://www-cse.ucsd.edu/groups/hpcl/scg/kelp.html

[2] R. Cappuccio , G. Cattaneo , G. Erbacci , U. Jocher, A parallel implementation of a cellular automata based model for coffee percolation, *Parallel Computing,* v.27 n.5, p.685-717, April 2001

[3] Jose G. Castanos and John E. Savage, *"The Dynamic Adaptation of Parallel Mesh-Based Computation",* Technical Report: CS-96-31, 1996.

[4] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson and Courtenay Vaughan, "Zoltan Data Management Services for Parallel Dynamic Applications", *Computing in Science & Engineering*, Vol. 4, No. 2, March/April 2002, pp. 90-97.

[5] Narsingh Deo, Muralidhar Medidi and Sushil K Prasad, "Load balancing in parallel battlefield management simulation on local- and shared-memory architectures", *J. Computer Systems: Science & Engineering,* Special Issue on 'Simulation in parallel and Distributed Computing Environments', Guest Editor: A. Zomaya, Vol. 13, No. 1, pp. 55-65, 1998.

[6] S. Huang, E. Aubanel, and V.C. Bhavsar, "PaGrid: A Mesh Partitioner for Computational Grids", *Journal of Grid Computing,* Vol. 4 No. 1, pp. 71 - 88, 2006.

[7] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. *BSPlib: The BSP Programming Library*. Parallel Computing, 1998, pp. 1947-80.

[8] R. Henderson, D. Meiron, M. Parashar, R, Samtaney, "Parallel Computing in Computational Fluid Dynamics". In J. Dongarra et al., editors, *"Sourcebook of Parallel Computing",* Chapter 5, Morgan Kaufmann, 2003.

[9] G. Krawezik AND F. Cappello," Performance comparison of MPI and OpenMP on shared memory multiprocessors", Concurrency Computat.: Pract. Exper. 2006; 18:29–61

[10] Kauranne, T., 1990: An introduction to parallel processing in meteorology. *The Dawn of Massively Parallel Processing in Meteorology,* G. R. Hoffman and D. K. Maretis, Eds., Springer-Verlag, 3–20.

[11] George Karypis and Vipin Kumar, "Multilevel k-way Partitioning Scheme for Irregular Graphs", *J. Parallel & Distributed Computing*, 48(1): 96-129, 1998.

[12] Peter MacNeice, KevinM.Olsonb, Clark Mobarry, Rosalinda de Fainchtein, Charles Packer, "PARAMESH: A parallel adaptive mesh refinement community toolkit", *Computer Physics Communications* 126 (2000) 330–354

[13] Steve MacDonald, Duane Szafron, Jonathan Schaeffer and Steven Bromling, "Generating Parallel Program Frameworks from Parallel Design Patterns", in proceedings of *6th International Euro-Par Conference* (Euro-Par 2000), Munich, Germany, August 2000, Lecture Notes in Computer Science 1900, Springer-Verlag, pages 95-104.

[14] www.openmp.org

[15] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang and Geoffrey Fox, "Runtime Support & Compilation Methods for User-Specified Irregular Data Distributions", *IEEE Transactions on Parallel and Distributed Systems*, 1995.

[16] Michael J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2004.

[17] I.M. Smith and D.V. Griffiths : *Programming the finite element method,* (John Wiley & Sons, 2004, 4th edn.)

[18] Kirk Schloegel, George Karypis and Vipin Kumar, "A Unified Algorithm for Load-balancing Adaptive Scientific Simulations", *Supercomputing, 2000.*

[19] Dale Shires and Ram Mohan, "An Evaluation of HPF & MPI Approaches and Performance in Unstructured Finite Element Simulations", *Journal of Mathematical Modeling and Algorithms* 1:153-167, 2002.

[20] R. Wanschoor and E. Aubanel, "Partitioning and Mapping of Mesh-Based Applications onto Computational Grids", *5th IEEE/ACM International Workshop on Grid Computing (Grid 2004),* Nov. 8, 2004, Pittsburgh, PA, USA, IEEE Computer Society, pp. 156-162.

[21] C. Walshaw and M. Cross, "Multilevel Mesh Partitioning for Heterogeneous Communication Networks", *Future Generation Computer Systems,*17(5):601-623, 2001

[22] H. P. Zima, "High Performance Fortran - History, Status and Future", *Proceedings of the 4th international Symposium on High Performance Computing* (May 15 - 17, 2002), Lecture Notes In Computer Science, vol. 2327. Springer-Verlag, London, 490, 2002.