

Synthetic Simulation of Mesh-based Parallel Applications Driven by Fine-Grained Profiling

Qingyuan Liu, Amol S Deshmukh, Karen A. Tomko

Department of Electrical and Computer Engineering and Computer Science

University of Cincinnati, Cincinnati, OH 45221

{liuqu, ktomko}@ececs.uc.edu

Abstract

We are interested in discovering the intrinsic dynamics of parallel applications, which are independent of runtime environment, to aid in the development of appropriate tuning policies, especially dynamic load balancing policies. Based on the novel idea of profiling mesh-based applications at a fine granularity of each mesh element, this paper proposes a synthetic application simulator which is driven by a series of application signatures mapping to the mesh structure. By integrating the ZOLTAN library into the system, our simulator provides a convenient test bed for developing and evaluating load balancing policies.

1. Introduction

The complex, dynamic nature of current parallel applications has driven many efforts on performance tuning [1, 2]. For this purpose, the profiling technique is popularly adopted to generate application signatures, discover performance bottlenecks and suggest optimization strategies.

Recent profiling efforts have focused on online performance analysis techniques in an attempt to use the performance analysis for runtime optimization [3, 4]. For example, many runtime tools (Falcon [5], SvPablo

[6], Autopilot [7], Harmony [8] etc) have implemented profiling to serve their tuning objectives. Generally, Profiling for online optimization is constrained in that it should be lightweight so that it does not add any significant overhead in terms of execution time and the performance data must be available to the decision making component without significant delay. It is also evident that online profiling is not a substitute for offline profiling. When used to drive simulations based on profiled data, offline profiling allows the user to replay the execution profile of the program any number of times, even allowing replaying in ‘slow motion’ and replaying ‘backwards’ in time [9]. This affords the possibility of performing different analyses on trace data obtained from a single profiling run. Offline performance analysis tools include AIMS [3], Carnival [10], TAU [11], etc.

For either online or offline analysis, application profiling is absolutely necessary to aid in understanding the dynamic behavior of parallel programs. The PMaC lab at San Diego Supercomputer Center (SDSC) has developed two profiling tools, MAPS and MetaSim [12, 13] to generate Machine Signatures and Application Signatures respectively, which are both fed into a network simulator DIMEMAS[14] for performance characterization. Similarly, C. Lu and D. Reed et. al. at UIUC proposed that Compact Application Signatures[15] be generated to capture the time-varying resource needs

of scientific codes, for the purpose of real-time performance adaptation.

The work presented in this paper is based on the belief that fine grained profiling of mesh-based parallel programs at the level of each mesh element could provide a more complete view of the application performance. This idea is novel, as the profiling information can be mapped to the geometric domain of the application and the results of profiling information can be easily associated with each element in the geometric domain. Thus, unlike traditional profiling approaches which target the tracing of events or determining performance parameters for subroutines, the approach described in this paper gathers profiling data that maps to the data structure of the mesh-based application.

It is necessary to point out that the scope of our research is the large category of computation-intensive parallel programs with a mesh structure, in which parallelism is achieved by splitting the mesh and distributing it among the processors and performing per-element computation concurrently. Our initial efforts are focused on those applications with structured meshes, e.g. rectangular meshes.

We have developed a portable, generic profiling tool Chiffon [16], which can profile mesh-based applications by measuring the value of the performance parameters at the level of each data element. This tool enables the construction of a synthetic application that will mimic the execution of the target program at the level of individual mesh elements. For this purpose, we propose the design of a synthetic application simulator driven by the trace data generated from Chiffon.

The primary significance of the proposed simulator is that it can provide an environment for researchers to develop and test optimization strategies on the ‘synthetic application’ instead of the real code. The advantages of such a capacity are obvious: firstly, various strategies can be explored without the need to modify application source code; secondly, it is not necessary to run the simulation on the target platform; thirdly, as an offline

tool, optimal solutions may be explored. As the traces characterize the application behavior at a very fine granularity, a design developed in the simulator is expected to be equally appropriate when applied to the real application as it would be if it had been originally developed within the application.

The rest of the paper is organized as follows: section 2 presents an overview of the profiling tool Chiffon, section 3 introduces the synthetic simulator, section 4 gives a case study and its experimental results, section 5 describes limitations and future work, finally section 6 presents a conclusion.

2. Chiffon - a fine-grained profiling tool

Generally speaking, fine grained profiling conveys detailed information but results in large trace volumes. Using traditional tracing for the problem of fine-grained profiling of mesh-based parallel programs would result in tens of thousands or more traces depending on mesh size. Recent work in [15] describes a viable approach to characterizing the behavior of parallel programs by compaction of the traces as they are generated. The approach is effective as described for characterizing the execution of the application process per node. Chiffon employs a similar polyline fitting strategy for trace compaction, which is based on a least square linear fitting [15] algorithm. Although both systems consider trace compaction, Chiffon is quite different in three aspects: 1) Chiffon generates trace mapping to each mesh cell instead of each processor node. 2) Traces in Chiffon are compacted in space rather than time, which results in individual traces per sampling time-step. 3) A separate process is executed on each node for trace compaction.

Figure 1 illustrates the runtime system architecture of Chiffon. The system consists of a daemon process executing on each node, the LocalAgent, which compacts and communicates the profiled trace information to a central node running another server process, the GlobalAgent. The application program to be

profiled is hand-instrumented using an easy to use C API (with Fortran 90 wrapper). The communication of profiling information from the application to the LocalAgent process is done via shared memory data exchange.

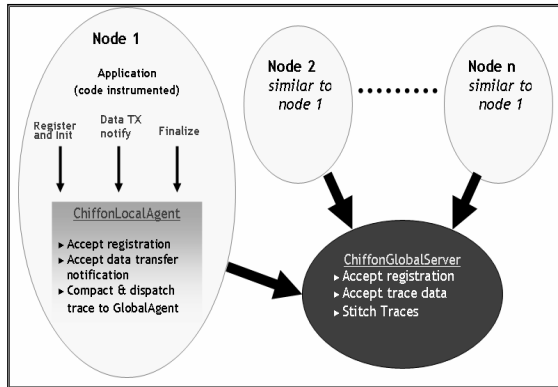


Figure 1. Chiffon runtime system [16]

The target application program is instrumented with calls to the library developed as an interface for the tool for profiling and communicating the profiled data to the ‘LocalAgent’ process running on that node.

Currently, probe functions available in the API are for measuring CPU cycles, CPU instruction count or execution time per mesh element. But the tool is not limited to probing a fixed set of performance metrics. Custom probe functions can easily be registered by calling the appropriate API functions.

3. Synthetic application simulator (SAS)

The idea behind the profiling approach is the expectation that it will capture the intrinsic dynamics of computation-intensive applications. By generating application signatures of given performance metrics mapping on a per element basis, this process results in a set of trace files which could individually characterize application behavior in the domain at a certain profiling time step. Therefore the simulator is designed to replay the profiling signatures, which characterize the program dynamics, so as to analyze the application’s behavior and develop feasible tuning policies.

3.1. Framework

Figure 2 is an illustration of the synthetic application simulator. Chiffon is included as a part of the framework for completeness. The main input to the simulator is the compact trace generated by Chiffon through an execution of the instrumented target code. Afterwards, runs of the real application are not needed. Another input is the configuration of parameters to adjust the simulation according to the user’s requirements.

In addition, the Zoltan [17] load balancing library is integrated as a module for the sake of developing appropriate load balancing policies. The mechanism of invoking the Zoltan library will be described in the following section.

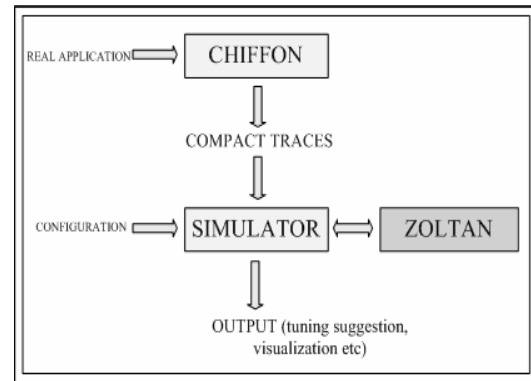


Figure 2. System framework

3.2. Modules in simulator

Currently, the following functional modules exist within the simulator:

I. Configuration Interface

This module is responsible for specification of parameters to steer the simulation according to the user’s need. The user provided configuration is used to coordinate the functional units in the simulator. A unique feature of the configuration interface is the ability to register custom functions to be invoked by the simulator kernel. A GUI is under construction to provide an easy-to-use means for users to set up the simulation configuration parameters to meet their needs.

II. Trace Processor

Each trace is compacted in space to avoid excessively large file sizes. The effect of profiling granularity and compaction error has been analyzed in [16]. Fortunately the adopted Polyline Fitting compaction algorithm permits fast ‘unzip’ of data points in a segment based on its start and end points. At the current stage of development we require a full decompression of traces, to the granularity of each mesh element, for simulation. Other efficient decompression choices, such as partial decompression, decompression in diverse granularities and dynamic decompression during simulation, are part of our future work. The details of this topic are not the emphasis of this paper.

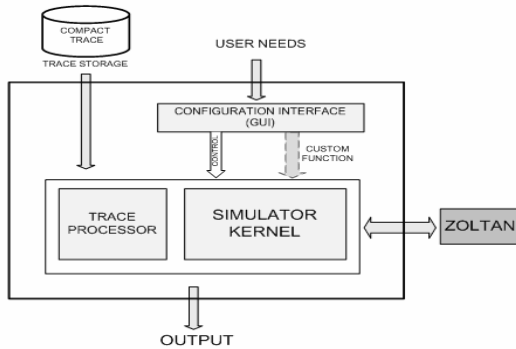


Figure 3. **Simulator structure**

III. Trace Storage

There are two alternatives for storing traces. Currently we use a simple file format, that is, one profiling file per sampling step. Another future choice is to organize the large volume of traces as a database. Thus traditional database techniques can be flexibly used for query and processing.

IV. Simulator Kernel

The basic function of the Kernel is to replay the application behavior through trace driven discrete event simulations, so as to explore the dynamic characteristics for tuning purposes. Users could choose to use the collection of analytical functions embedded in the kernel for modeling an event, which is under development, or register self-defined functions to be invoked by the kernel for specific experiments. Besides the traditional functions that a trace analyzer can perform, our

simulator is unique in providing a real parallel environment to call Zoltan.

As mentioned at the beginning, one of our big interests is the dynamic load balancing of computation-intensive applications. The Zoltan library is designed to perform adaptive repartitioning and load balancing of parallel applications. It provides implementations of many different load balancing algorithms, which can be selected during the initialization of the library. To realize a new partition, Zoltan also provides help for data movement and memory management.

We choose Zoltan because it provides convenient interface functions; users can easily choose/ change among different algorithms; partition information can be available for analysis. Correspondingly, workload metrics such as weight or computational time should be profiled to perform this category of experiments.

Since Zoltan is a parallel library involving MPI communication, the simulator kernel is designed to be able to invoke Zoltan initialization, register Query functions and call Zoltan functions in the same way as a real parallel application does. However, it should be emphasized that there is no real load partitioning involved during the simulation. All we have is a collection of time-stamped, fine-grained workload traces, which is however enough to mimic the workload distribution in the time and space domains. Recall our original idea of ‘duplicating’ a substitute (i.e. synthetic application) to undertake complex or expensive experiments in an efficient way; here repartitioning experiments are feasible through simulations which invoke Zoltan based on the fine-grained traces instead of the application code.

4. Case study - MRTD application

In this section we illustrate the operation of the synthetic system by feeding the workload traces of a real parallel application MRTD [18] into a simulation which is configured to develop a repartitioning strategy for the

application. The strategy is based on a heuristic algorithm to find appropriate repartitioning steps.

We have used a 3-dimensional Multi-Resolution Time-domain (MRTD) Electromagnetics code for experiments. The MRTD modeling application performs time domain modeling of large scale cosite interference problems arising in wireless communications. The MRTD application employs Haar Wavelets [18] to achieve multi-resolution. As magnetic and electric fields propagate across the mesh domain, the active and thresholded regions are changed based on values of the wavelet coefficients. Therefore, the amount of work per cell varies across the domain and varies across time steps. To account for this, each cell in the mesh is assigned a weight [13] which corresponds to its update time per iteration. The MRTD code uses runtime equations to calculate each cell's weight based on current wavelet coefficient thresholding results.

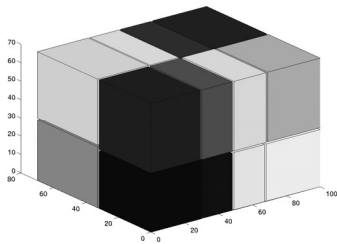


Figure 4. MRTD domain partition

Because the domain of MRTD is represented by a cube composed of rectangular cells, a recursive coordinate bisection (RCB) algorithm [19] in the Zoltan library is used for load partitioning. The RCB algorithm cuts a 3-dimensional domain with a plane along one axis which produces two 3-dimensional rectangular blocks which are then recursively divided by similar cuts until the desired number of partitions is obtained. The coordinates of cuts are determined by the total weight of cells in both resulting partitions.

4.1. MRTD profiling

In the MRTD code, each cell is assigned a weight corresponding to its workload. To verify and illustrate

the relationship between cell weight and its computation cost, we profiled the computational time associated with an arbitrarily chosen mesh block and an arbitrarily chosen mesh cell individually. The results from figure 5 to 8 show that the computation time of a mesh element can be characterized by its weight as the two measurements match very well.

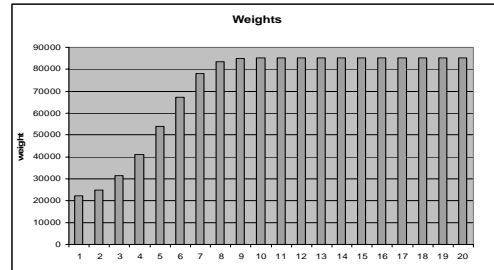


Figure 5. Time-varying load (weights) of sampled block in the mesh

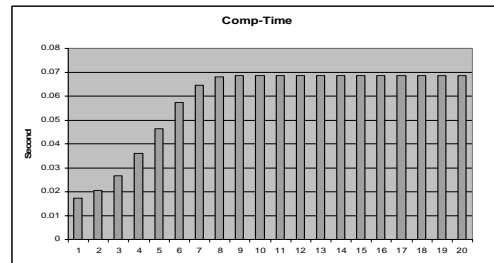


Figure 6. Time-varying load (computation time) of sampled block in the mesh

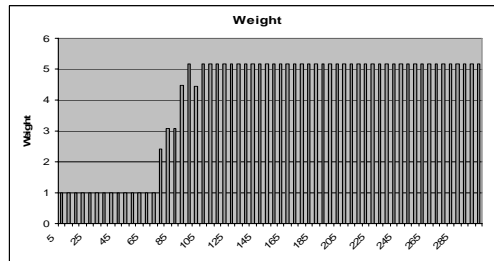


Figure 7. Time-varying weight of a mesh cell

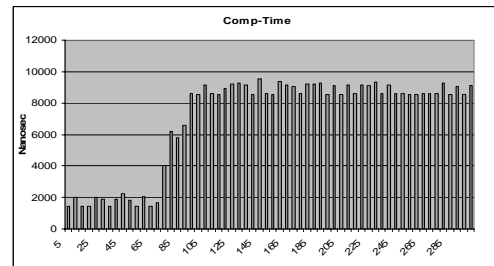


Figure 8. Time-varying load (computation time) of a sampled mesh cell

4.2. Repartitioning policy

As mentioned above, MRTD must employ dynamic repartitioning to maintain a balanced workload distribution. The load imbalance is usually quantified as $(L_{max} - L_{ave})/L_{max}$. L_{max} and L_{ave} represent maximum and average load across all processors respectively.

There have been attempts to develop a good load balancing policy which could adapt the repartitioning interval dynamically. Simply speaking, it is expected that frequent repartitioning should only occur when the imbalance situation is serious, otherwise the repartitioning rate should be relaxed to avoid unnecessary overhead. However, runtime experiments are limited by two factors: 1) lack of knowledge of future situations; predictions based on history may be far from accurate and the tuning decision can be inapplicable due to the time delay. 2) the algorithms are unable to learn the long run tuning consequence, i.e. they are shortsighted.

The above limitations do not exist in our simulation, because time-stamped load traces can be visited in an arbitrary manner and, most importantly, load information is attached to mesh elements, not processors. This means that global or local views of workload status can be easily gained by specifying the interested domain.

The algorithm described here attempts to find the best time steps to invoke repartitioning through adjusting the repartitioning interval. The following pseudo code provides a description of the approach.

```

while (step <= time_step_end)
{
    /* calculate imbalance */
    imbalance = (max_weights - avg_weights) / ave_weights;
    if (imbalance < imbalance_limit) /* Does not need repartition
    */
    {
        /* double rate and march */
        if (partition_interval <= max_partition_interval/2)
            partition_interval = partition_interval * 2;
        step += partition_interval;
    }
    else if (partition_interval > min_partition_interval)
        /* Need repartition; reduce rate and backtrack */
        {
            partition_interval = partition_interval / 2;
            step -= partition_interval;
        }
}

```

```

}
else /* Do repartition here */
{
    Do_Repartition;
    Do_DateMigration;
    Step += partition_interval;
}
}

```

It can be seen that the algorithm takes advantages of the ability to move forward and backward in time. Such an experiment is not possible during runtime but can be easily realized in our simulator based on workload profiles mapping to the mesh structure. While the algorithm is not feasible for runtime use, it provides a best case for methods using adaptive intervals. In the next section we show results of using this algorithm for a particular MRTD simulation.

Figure 9 illustrates how the repartitioning interval can be adapted based on above algorithm, assuming the maximum interval is 20 and the minimum interval is 5. Step 20 and 65 are recommended for repartitioning.

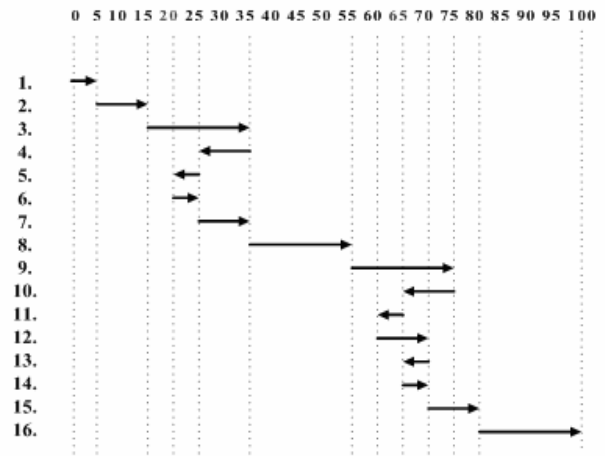


Figure 9. Adaptive repartitioning interval based on the heuristic algorithm

The execution platform of the MRTD application is the Itanium II cluster at the Ohio Super Computer Center. This cluster provides 128 compute nodes, each of which contains two 900MHz Itanium-2 processors and 4GB main memory. The nodes are connected using Myrinet, which is a switched 2.0GB/s network. The simulation is executed on a 4-CPU Solaris workstation at the University of Cincinnati and is not bound to the same specific platform as the real application.

4.3. Results analysis

The performance of the heuristic strategy can be analyzed from two aspects: imbalance improvement and repartition overhead. Due to space limitation, the performance data illustrated here is specific to a small MRTD case with a domain of 30x20x20 cells and a Gaussian source signal.

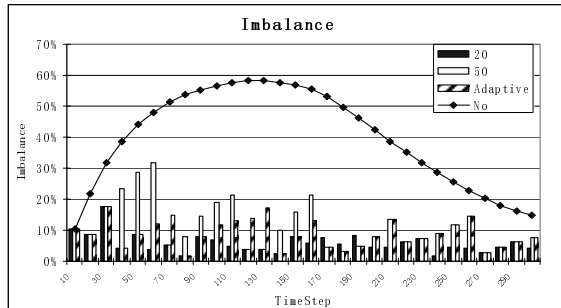


Figure 10. **Comparison of different repartitioning choices**

Figure 10 illustrates the effects of various repartitioning strategies on workload imbalance. Results are shown for fixed repartitioning intervals of 20 and 50 and an adaptive interval. It can be seen that the adaptive strategy clearly reduces workload imbalance, even compared to a dense fixed repartitioning interval of 20.

The overhead of invoking dynamic repartitioning is expensive. A small fixed interval may achieve better balance but at a high cost which will offset gains due to the better balance in the overall execution time. The adaptive policy is superior since it avoids unnecessary repartition invocation. Specific to MRTD_beta_case1 running up to 300 steps, a fixed interval of 20 means 14 repartitions. But the heuristic algorithm (with imbalance threshold 0.15 and window size 5 to 20) suggests only 7 repartitions. The corresponding experiments with the real application show that a 19.9% to 40% improvement in overall execution time can be achieved by applying the adaptive repartitioning steps instead of even intervals.

It can be seen that the heuristics of backtracking and rate adaptation could be easily implemented and tested through simulation. Our simulator enables convenient

development and evaluation of policies suitable for use in the real application.

5. Limitations & future work

Commonly, with a profiling-based approach, one set of traces corresponds to one specific application input. Application behavior may be different when input parameters are changed. Fortunately, different inputs can usually be partitioned into a few categories with similar behavior such that only one representative case needs to be profiled for analysis and simulation. The returned tuning strategy is expected to achieve good performance when applied to cases in this category. In the long term, we hope to discover a collection of ‘patterns’ through simulation which can be utilized for online tuning. That is, if the runtime application behavior matches a certain pattern, then corresponding tuning policies can be invoked.

Another issue is that scientific applications usually involve large computational domains, which result in either highly compact traces (and hence less precision) or very large trace sizes. At the profiling stage, one choice would be to adopt a coarser profiling granularity; at the simulation stage, efficient decompression choices such as dynamic or partial decompression could greatly reduce trace I/O. These considerations will be part of our future work.

6. Conclusion

In this paper we utilized a novel profiling approach oriented to mesh-based applications, which could associate performance metrics to the mesh structure with fine granularity. The resulting trace can be fed into a synthetic simulator to replay the application dynamics, providing a convenient environment for analyzing the application behavior and developing appropriate optimization policies. Our initial experiments, testing a heuristic repartitioning strategy in the synthetic environment, have shown a good performance.

Acknowledgements

The authors would like to express their thanks to Costas Sarris, Linda Katehi and Pawel Czarnul et al. for the Parallel Multi-Resolution Time-Domain Electromagnetics simulation (MRTD) and also to the Ohio SuperComputer Center for computer resources.

References

- [1] D. Reed, R. Aydt, L. DeRose, C. Mendes, R. Ribler, E. Shaffer, H. Simitci, J. Vetter, D. Wells, S. Whitmore, and Y. Zhang. "Performance Analysis of Parallel Systems: Approaches and Open Problems. *Proceedings of the Joint Symposium on Parallel Processing(JSPP)*, pp.239-256, 1998.
- [2] A. Downey and D. Feitelson. The Elusive Goal of Workload Characterization. *Technical report*, Hebrew University, Jerusalem, March 1999.
- [3] J. Yan, S. Sarukkai and P. Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software Practice & Experience*, Vol.25, pages 429-461, April 1995.
- [4] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37-46, 1995.
- [5] G. Weiming, G. Eisenhauer and K. Schwan, J. Vetter. Falcon: On-line Monitoring for Steering Parallel Programs. *Concurrency-Practice and Experience*, Vol.10,No.9, 1998.
- [6] L. Rose and D. Reed. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. *Proceedings of ICPP' 99*, 1999.
- [7] R. Ribler, J. Vetter, H.S imitci and D. Reed. Autopilot: Adaptive Control of Distributed Applications. *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*,1998.
- [8] Tapus, I. Chung and J. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. *Proceedings of SC'02*, 2002.
- [9] M. Heath and J. Finger. ParaGraph: A Performance Visualization Tool for MPI.
- [10] W. M. Jr, T. LeBlanc and A. Poulos. Waiting Time Analysis and Performance Visualization in Carnival. *ACM SIGMETRICS Symp. On Parallel and Distributed Tools*, pp.1-10, 1996.
- [11] S. Shende, A. Malony, J. Cuny, K. Lindlan, P. Beckman and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. *Proceedings of SPDT'98*, 1998.
- [12] A. Snavely, N. Wolter and L. Carrington. Modeling Application Performance by Convolving Machine Signatures with Application Profiles. *IEEE 4th Annual Workshop on Workload Characterization*, 2001.
- [13] G. Rodriguez, R. Badia and J. Labartaq. Generation of Simple Analytical Models for Message Passing Applications. *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference*, 2004.
- [14] R. Badia, J. Labarta, J. Gimenez and F.Escale. DIMEMAS: Predicting MPI applications behavior in Grid environments. *Workshop on Grid Applications and Programming Tools(GGF8)*, 2003.
- [15] A. Lu and D. Reed. Compact Application Signatures for Parallel and Distributed Scientific Codes. *Proceedings of SC'2002*, 2002.
- [16] A. Deshmukh, Q. Y. Liu and K. Tomko. An approach for Fine-grained Profiling of Mesh-based Parallel Programs. *Parallel and Distributed Computing Systems(PDCS'04)*, 2004.
- [17] K. Devine, B. Hendrickson, E. Boman, M. St. John and C.Vaughan. Zoltan: A Dynamic Load_Balancing Library for Parallel Applications; User's Guide. *Sandia National Laboratories Tech. Rep. SAND99-1377*, 1999.
<http://www.csar.uiuc.edu/software/paragraph/userguide.pdf>
- [18] C. D. Sarris, K. Tomko, P. Czarnul, S. H. Hung, R. L. Robertson, D. Chun, E. S. Davidson, and L.P.B Katehi. Multiresolution Time Domain Modeling for Large Scale Wireless Communication Problems. *2001 IEEE AP-S International Symposium on Antennas and Propagation*, 2001.
- [19] K. Schloegel, G. Karypis and V. Kumar. *CRPC Parallel Computing Handbook*, chapter Graph Partitioning for High Performance Scientific Simulations. Morgan Kaufmann, 2000