Sensitivity analysis of parallel applications to local and non-local interference

Vaddadi P. Chandu[*], Karandeep Singh[#]
[*]Georgia Institute of Technology, Atlanta GA
vchandu@cc.gatech.edu
[#] University of Maryland, College Park MD
ksingh@mail.umd.edu

Abstract

The environment in which a parallel application is executed has high impact on the performance of the application due to interference caused by various factors in the execution environment. A detailed understanding of the sensitivity of the application to the parameters describing the execution environment can be of great help in (a) predicting a suitable target machine model for the application, (b) predicting its performance on the target machine, and (c) any algorithmic bottlenecks. In this paper, we analyze a suite of parallel applications for their sensitivity to local and non-local interference arising due to various factors in a parallel environment. We create a test bed consisting of five different parallel applications taken from different sources and analyze their sensitivity to single node and multi-node perturbations and show that parallel applications can behave very differently under different conditions of interference in the environment in which they are running. The main contributions of this paper are: (a) studying a suite of parallel algorithms for their sensitivity to local and non-local interference, (b) demonstrate that an application can behave differently to different interference levels in the environment, (c) demonstrate that the sensitivity of an application can be quantified as its absorption ratio at a given interference level.

1. Introduction

The performance of a parallel application in distributed memory machines is highly dependent on the local and non-local interference existing in the environment which can be due to operating system interference or communication latency. Interference of any kind in the environment is highly detrimental to the performance of a parallel application. Hence, the ability to understand the sensitivity of a parallel application to interference at different levels is very important to understand its behavior in that environment. One of the techniques to analyze the sensitivity of a parallel application is to simulate it for compute time and messaging latency and see the effect on the execution time by changing these parameters. Petrini et al in [8], describe a method to analyze the performance of parallel applications on a parallel machine through modeling the application and study its behavior before actually building the machine. In [11], a methodology to analyze a parallel application through trace-based analysis is presented. In this method, the message-passing events in a parallel application are extracted during its execution and are linked in the same order as they occurred during the execution. This method provides an actual execution model of the application which is free from any assumptions or conceptual errors. Dimemas [3] and Vampir [14] are also similar methodologies but this approach significantly differs from them in the way that interference from local and non-local sources can also be simulated.

Trace-based analysis of a parallel application can be visualized as creating a message-passing graph (although it need not be actually created) in which any two message-passing events are interconnected with an edge that represents communication in the original application during its execution time. Each edge contains a start time stamp, an end time stamp, and an edge weight. Changing the weights on the edges simulates perturbations corresponding to local or non-local interference such as operating system noise or communication latency. Since the trace is extracted from the actual execution of the application, this method guarantees the correctness of the application and retains the actual ordering of the messages in the application.

In this paper, we focus on analyzing a suite of parallel applications for interference from local and non-local sources such as CPU throttling, network congestion, overloading of a single node, and misconfiguration of the operating system on all nodes. Although, our work is in the context of MPI [4] library, this method can be easily extended to any implementation ARMCI [6], PVM [12] of the message passing interface. The rest of the paper is organized as follows. In Section 2, we describe the message-passing graph concept. In Section 3, we describe our experimental dataset, and in Section 4 we present our results. In Section 5, we present the conclusions and future research.

2. Message Passing Graph Concept

The execution behavior of a parallel application, which is written using a message passing library, can be modeled as a message-passing graph by tracing it during runtime. The fundamental idea behind modeling it as a message passing graph is that, during execution, most parallel programs alternate between computation and communication.

Hence, by time stamping the events of computation and communication and joining them through directed edges, an exact representation of the execution model of the

T. Sobh (ed.), Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, 469–474.

application can be produced. In such a message-passing graph, the edges connecting any two events in the same trace are called as local edges and the edges connecting any two events in different traces are called as message edges.

After representing the execution of an application as a message-passing graph, perturbations can be injected into a local or a message edge and propagated along the graph ensuring that the execution order of all events is preserved and no two events violate each other's time stamps. However, if there room to absorb the perturbation instead of propagating it without violating the execution order, it is absorbed. Perturbing a message-passing graph in this manner increases the execution time by a definite amount thereby providing an insight into the amount of interference that the application can absorb. In this manner, the sensitivity of the application to that perturbation can be computed.

An application can be traced during execution time by wrapping the MPI primitives with a lightweight PMPI wrapper using the standard PMPI interface which is defined in the MPI specification. Each node generates an event trace that includes all the events that occurred at that node, a time stamp for each event, the type of the event, and corresponding metadata. It is important to mention here that although events are paired in a message-passing graph, they only represent the ordering of the execution and not the actual synchronized distributed-clock timing. However, this is not an issue in our analysis because we require only the actual execution ordering of the events and not the actual synchronized timing. Another vital issue is to confirm the correctness of the concept. Since the injected perturbations follow the same path as in the original execution and propagate through the message passing graph along the edges, the dependencies are automatically taken care of and hence the correctness is preserved.

However, it is important to note that there are two classes of applications where this method may not provide accurate results. First, the class of applications that is non-deterministic, such as the Branch and Bound algorithm where the communication pattern is highly non-deterministic, and second, the class of applications which is embarrassingly parallel such as SETI@home [7], where there is very little or practically no communication between the nodes. In both cases tracing the application will not provide enough information to generate the correct execution model of the application and hence cannot provide accurate results. However, this method works well for all the applications that do not fall in the above two classes.

2.1 Analyzing the sensitivity of an application using trace-based analysis method

After creating the execution model of an application, the sensitivity of the application to local and non-local interference on particular platform can be measured by using the signature of that particular parallel platform. Signature of a parallel platform can be constructed using microbenchmarks. Microbenchmarks are capable of probing a given platform to a very minute detail. By assuming a distribution on these results, a sufficiently accurate parameter set for a given parallel platform can be estimated. One such microbenchmark to estimate the operating system noise is the Fixed Time Quantum microbenchmark [10], which probes the operating system for periodic perturbations in a large number of fine grained workloads. Using a simple message-passing program as suggested in [5] is another method to estimate the operating system noise for a particular platform.

Communication parameters can be benchmarked in a similar way. Intel Cluster Toolkit [13] provides a suite of benchmarks which provides three classes of benchmarking functionalities- Single Transfer, Parallel Transfer, and Collective. The single transfer class benchmarks the performance of a single message transfer between two processes using PingPing and PingPong style messaging. The parallel transfer class benchmarks the performance the message transfer performance under global load using a periodic chain and periodic exchange pattern styled messaging and the collective class benchmarks the performance the performance of the collective primitives of the message passing interface using the style as defined by the collective primitives in the MPI library.

3. Experimental Test Bed

In our experimental test bed, we consider five different applications for the sensitivity analysis using the trace based analysis method. The applications are Parallel Sorting using Regular Sampling [9], Cycle Detection of Partitioned Planar Digraphs [1], Parallel Selection and Median Finding [2], Parallel Heap Sorting, and A Grid based problem-Jacobi iteration, each of which is described in the next subsection. All the applications in our test bed are written using the MPI library (version 1.2.6) and are compiled using level three optimization enabled for a distributed memory machine which uses Infiniband interconnect in Linux environment. Each node is a 3.20 GHz Intel Xeon dual core CPU with 1024 KB cache memory and 1 GB main memory.

3.1 Parallel Sorting using Regular Sampling

Parallel Sorting by Regular Sampling is a sorting algorithm for distributed memory machines. This algorithm uses techniques to reduce the bus and memory contention by ensuring a good pivot selection through regular sampling of the data. It fits into the partition-based sorting methodology, where the data set is partitioned into smaller subsets such that all elements in one subset are no greater than any element in another, and each subset is sorted in parallel.

This algorithm, through regular sampling of the data, is capable of finding pivots to partition a dataset into similar sized smaller datasets. Due to uniform partition of the data, load balancing is also uniform. This algorithm consists of the following three phases.

Consider p processors. In the first phase, the input dataset is uniformly distributed among the p processors using sequential Quicksort algorithm. Each one of the p processors sorts the list of data elements assigned to it. From each of these lists (which are p in number), p-1 samples are chosen such that they are evenly spaced in the list. Each one of the (p-