7-27-2011

Characterization of Parallel Application Runtime Sensitivity on Multi-core High Performance Computing Systems

Padma Priya Veeraraghavan
Purdue University, pveerara@purdue.edu

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Padma Priya Veeraraghavan

Entitled
Characterization of Parallel Application Runtime Sensitivity on Multi-core High Performance Computing Systems

For the degree of Master of Science

Is approved by the final examining committee:

Jeffrey J Evans
Chair

Mark Jackson

Thomas Hacker

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Jeffrey J Evans

Approved by: James L Mohler 07/22/2011
Head of the Graduate Program Date

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

Characterization of Parallel Application Runtime Sensitivity on Multi-core High Performance Computing Systems

For the degree of Master of Science

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Padma Priya Veeraraghavan

Printed Name and Signature of Candidate

07/22/2011

Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

CHARACTERIZATION OF PARALLEL APPLICATION RUNTIME SENSITIVITY ON MULTI-CORE HIGH PERFORMANCE COMPUTING SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Padma Priya Veeraraghavan

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2011

Purdue University

West Lafayette, Indiana

ACKNOWLEDGEMENTS

At the outset, I would like to wholeheartedly thank my advisor Dr. Jeffrey Evans for providing me an opportunity to work in his research group. He has taught me to how not to just look at the obvious things but also thoroughly observe, understand, and analyze the research problem in hand. I would also like to thank him for allowing me to work and finish the final part of my thesis remotely, as I had to relocate because of family situation. Without his understanding and support my research would not have been a reality.

I would like to thank my committee members, Dr. Thomas Hacker and Dr. Mark Jackson for their time and support for my research work. I also deeply appreciate their feedback on my thesis. I also want to thank all the professors and teachers who taught me and helped me to be where I am now.

My special thanks to my beloved husband for his encouragement and support during this journey and I want to say many thanks to my dad and mom for their love and affection. Also, I would like to thank my brothers, nieces, nephews, and all my in-laws for their moral support.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Appendix Figure

Appendix Figure Page

LIST OF ABBREVIATIONS

MPI	Message Passing Interface
HPC	High Performance Computing
ACSL	Adaptive Computing Systems Laboratory
PACE	Parallel Application Communication Emulation
PARSE	Parallel Application Runtime Sensitivity Evaluation
AUT	Application Under Test
EA	Emulated Application
NAS	Numerical Aerodynamic Simulation
NPB	NAS Parallel Benchmark
EP	Embarrassingly Parallel
MG	Multi-Grid
LU	Lower-Upper symmetric Gauss-Seidel
PSTSWM	Parallel Spectral Transform Shallow Water Model
NOW	Network of Workstations
MCMP	Multi-core Multi-processor

ABSTRACT

Veeraraghavan, Padma Priya. M.S., Purdue University, August 2011. Characterization of Parallel Application Runtime Sensitivity on Multi-core High Performance Computing Systems. Major Professor: Jeffrey J Evans.

A commonly seen behavior of parallel applications is that their runtime is influenced by network communication load. The way a parallel application is run in a network and the presence of other applications and processes in the network can contribute to a wide range of variations in the runtime. Therefore, in order to achieve consistent and optimal runtimes, it is important to understand and characterize the runtime sensitivity of parallel applications with respect to execution under the presence of network communication load.

In this research, runtime sensitivities for various parallel applications were studied by applying additional network communication load. In particular, the focus was on the runtime sensitivity of parallel applications on a multi-core multi-processor (MCMP) system where less network switching and routing are involved compared to single-core single-processor machines.

The objective of this work was to determine if a previously developed sensitivity model for single-core single-processor machines still holds good for multi-core machines. For this purpose, previously developed tools (PACE and PARSE) were used to perturb the communication sub-system while executing several parallel application benchmarks such as the NAS benchmarks and PSTSWM. Runtime variations of these parallel applications were studied, under a specific network communication load, for different test cases by changing computing core allocation. A 10-node 80-core cluster was used as the test bed for this research purpose.

Several test cases were explored using a variety of core allocations (process locations) for the application under test (AUT) to simulate job scheduler fragmentation. To ensure statistical significance, several iterations (trial runs) were executed in each test case. Results indicate that the idea of application sensitivity to communication sub-system performance degradation holds for MCMP architectures.

CHAPTER 1. INTRODUCTION

1.1. Parallel Computing: An overview

In the past couple of decades or so, there has been an increased interest in running applications using parallel computers. Considering the benefits offered by parallel computing, and the growth of hardware capability to support parallel computing, this trend is not surprising. In fact, the highest speeds of advanced supercomputers are growing at a rate that is exceeding Moore's Law, which predicts that processor performance doubles roughly every 18-24 months (Feng, 2005) (Figure 1.1).

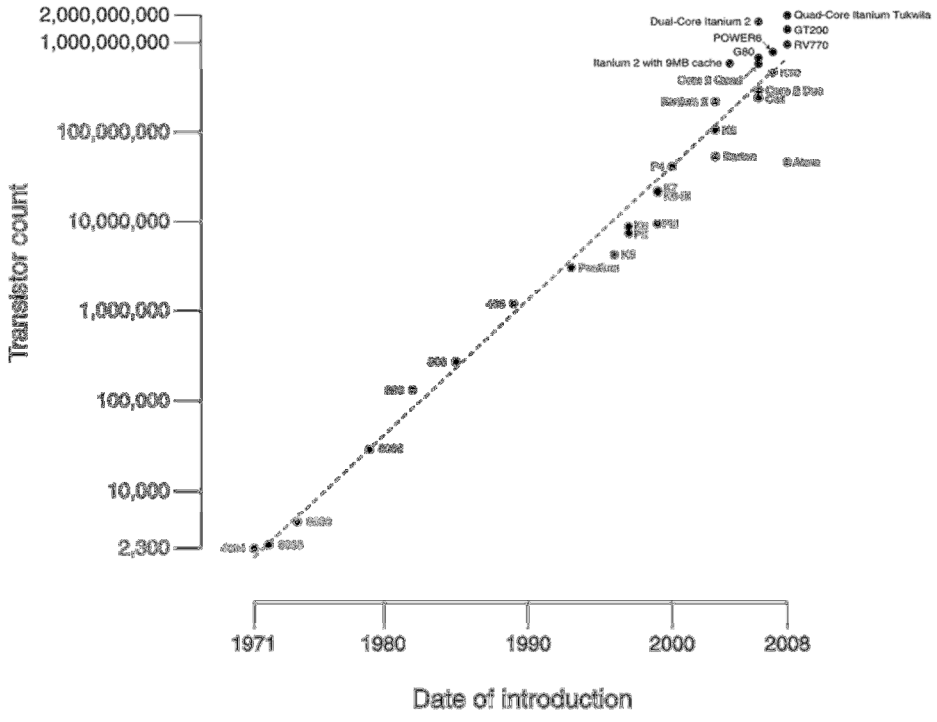

Figure 1.1 Computing power growth per Moore's law

The hardware growth to support parallel computing has been substantial in recent years. Newer systems comprise of many compute nodes, where each node contains more than 1 processor and each processor contains more than 1 processing "core". These are called multi-core multi-processor (MCMP) systems. The constructional difference between a single and a multi-core CPU chip is illustrated in Figure 1.2.

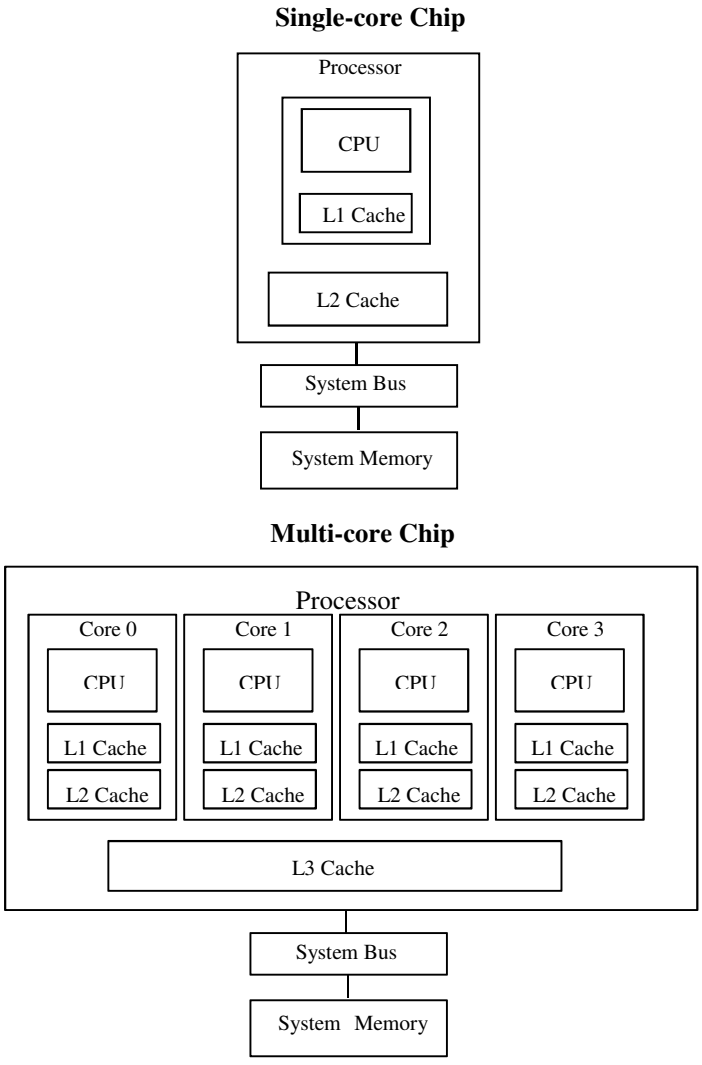

Figure 1.2 Single core chip vs. multi-core chip

Basically, multi-core architectures consist of several processing units in one chip. Due to this, they are more cost-effective. In contrast, single core architectures have physical constraints, in addition to rapidly growing power consumption. Further, they are more expensive compared to their multi-core counterparts. Considerable amount of

research is being done to assess the performance effect of these multi-core machines, especially on scientific applications (J. Carter et al., 2007).

Nowadays, scientific and engineering applications using numerical techniques make use of parallel computing more and more. Usage of parallel computers also requires development of parallel algorithms, programming models, and systems.

Parallel computers, in general, offer good deal of computational power; however on the downside they pose increased difficulty in programming. In contrast to the growth of computer peak speed, the scaling of application performance, unfortunately, has not followed suit. In general, applications do not automatically scale along with the increase in the number of processors. This puts heavy burden on programming to be used for parallel computing for the applications to run in the most efficient way. However, a well-written parallel program alone will not guarantee the expected scale-up of application runtime performance.

Apart from the programming requirements, one of the important issues that plays a key role in determining the difference between the expected and actual performance is the "communication overhead" of parallel applications. Hardware architectural attributes such as CPU speed, the number of processors, network parameters (network speed), type of the architecture used (shared or distributed memory), and system and cache memory can influence the application behavior. It is, therefore, highly imperative to understand the communication requirements of the application, to achieve the desired performance in a parallel architecture.

1.2. Statement of the Problem

Network communication plays a key role in determining the performance of a parallel application. Without properly understanding or characterizing this role, parallel applications would suffer from inconsistent runtime behavior and therefore, their performance may become unpredictable. One way to better understand the run time

performance of a parallel application under normal system operation is to evaluate its sensitivity to network communication load. This becomes even more crucial for repeatedly run applications on a given system. Many times, the focus is more on expanding the current network, adding more resources or computing power. Recently, the trend has been to adopt an MCMP system as a quick solution thereby reducing network switching and routing. However, in MCMP systems, the network communication becomes more complex because in addition to the communication that takes place between nodes via switches and routers (network resources), inter-process communication should also be considered. The inter-process communication is comprised of inter-core (within a processor) and inter-processor communication within the same node.

Therefore, it is fair to say that characterization of parallel application runtime sensitivity has not been thoroughly understood and quantified in MCMP systems. An empirical estimation is required for this purpose rather than theoretical predictions due to the fact that it is difficult, if not impossible, to model the spatial, temporal, and intensity effects of the communication sub-system during the concurrent execution of multiple parallel applications.

1.3. Significance of the Problem

Depending on the network communication load and processor allocation strategy, a poorly characterized parallel application would result in inconsistent runtimes. Again, depending on the problem size, this inconsistency could become very significant. For example, for a network loaded at 90%, the mean runtime of MG32 from Evans & C. S. Hood, (2005) was about 4.5 times the baseline (with no additional network loading) and the run time variability of nearly six times. This variation could result in unpredictable production time leading to loss of revenue, overbilling, overuse of resources, etc. Further, this could lead to increased use of energy to power and cool High Performance Computing (HPC) systems. For example, at Lawrence Livermore National Laboratory, for every Watt (W) of power consumed by an HPC system, 0.7 W is used for cooling

alone. The annual cost to power and cool the HPC system amounts to a total of $14.6 million per year (Feng, 2005). Multiplying this amount by the runtime variability could result in significant additional cost in terms of energy and money.

Therefore, understanding and characterizing the application sensitivity to existing network conditions, including inter-core and inter-processor communications, could become critical for businesses to minimize uncertainty related to application runtimes and hence save unnecessary operational costs.

1.4. Purpose of the Study

Characterization of parallel application sensitivity is essential to maximize its performance in existing network and to obtain a more consistent runtime behavior. In a parallel computing environment, network communication can adversely affect overall performance in many cases.

In this study, runtime sensitivities of parallel applications were characterized based on process allocation under an existing network communication load in an MCMP system by using network load emulation and evaluation tools (PACE and PARSE). Several parallel applications such as the NAS benchmarks and PSTSWM were tested and classified based on their runtime sensitivity factors. For each application, several iterations (trial runs) were run for different test cases (core allocations) and the resulting runtime variations were plotted and evaluated for sensitivity. The runtime sensitivity information thus obtained can be an useful input for schedulers for proper job allocation in order to achieve a more consistent runtime behavior.

1.5. Assumptions

The following are the assumptions of this study:

1. To be statistically significant, each application was run several times (at least 30 trial runs) to characterize the runtime sensitivity.

2. The tools used in the study such as PACE and PARSE are, in general, applicable to both single and multi-core systems.
3. PACE assumes that the environment is a Linux cluster.
4. The results obtained in this study to most parts can be generalized to MCMP systems of similar architecture.

1.6. Limitations

The following are the limitations of this study:

1. While the network is loaded, PACE does not communicate at all time, so there would be an influence from temporal component.
2. One limitation of loading with PACE is that, with increasing scaling, there is a natural reduction in systemic loading since contribution from PACE gets reduced.

CHAPTER 2. LITERATURE REVIEW

Parallel computers have gained considerable popularity in the recent years since they offer better performance and capability to handle large scale problems. In the 80's the distinction between supercomputers (multi-processor machines), workstations and PCs was widely known based on their performance. From the mid 90's, the gap between supercomputers and PCs (or workstations) has been widening. As pointed out by W. Feng (2005), with the advent of clustering, the goal of HPC manufacturers and adopters has been to narrow down this gap.

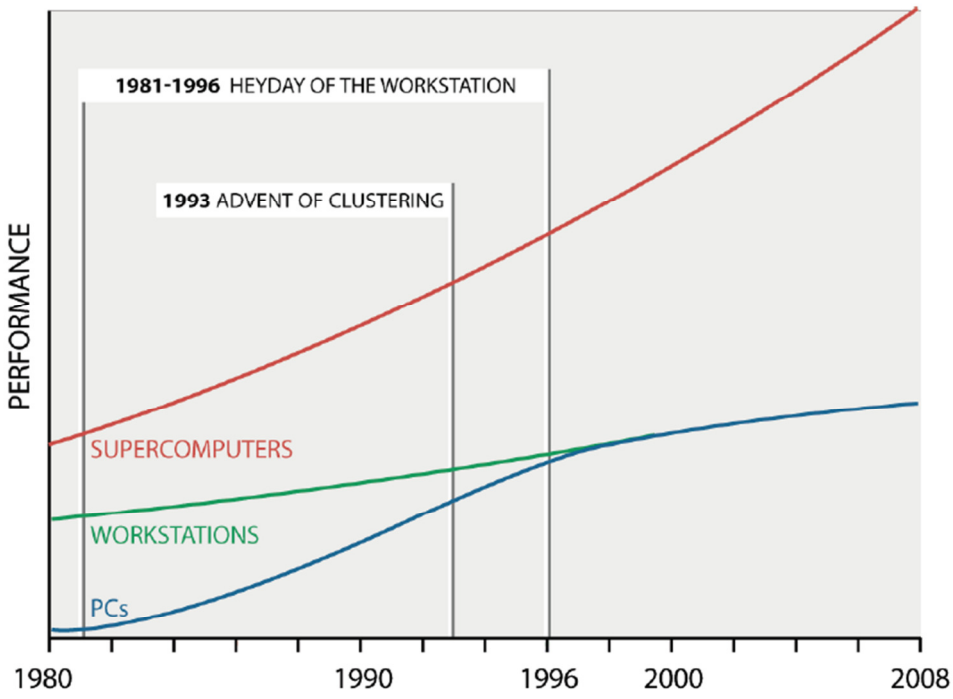

Figure 2.1 Widening gap between supercomputers and PCs

Today's super computers are essentially massively parallel processors (MPPs) – computers built using workstation-type nodes interconnected by a low latency and dedicated network. Anderson et al. (1995) observed that one of the key weaknesses of MPPs is that they lag behind 1 to 2 years in comparison to workstations that are built using equivalent parts. Assuming 50% performance improvements per year, this would lead to more than a factor of two in the bottom-line computational performance. In fact, clustered Networks of Workstations (NOWs) have become the most rapidly growing sector of supercomputing ("TOP500 Supercomputing Sites") and there has been a growing interest in clustered NOWs comprising of commodity based machines (T. E. Anderson et al., 1995).

The recent trend in the HPC world has been to move from clustering single-processor machines to multi-core multi-processor machines (MCMP systems). Kaiser et al. (2009) predict that the future high end systems will integrate thousands of 'nodes', each comprising many hundreds of cores by means of system area networks. Chip manufacturing giant, Intel, had announced to produce and release to the market an 80 core processor chip by 2011 ("Intel shows off 80-core processor - CNET News," 2007). In general, multi-core architectures prove to be a cost-effective way, since they offer more computational power through parallel processing. They also utilize less power and occupy less board space. Chai et al. (2007) observed that the scalability of multi-core cluster is more promising as compared to single-core cluster.

With the trend moving towards multi-core machines, how valid is the Amdahl's law and its assumptions? Amdahl's law or argument says that the speed up of a code or program in a parallel computing environment, using multiple processors, is restricted by the time taken to run the sequential portion or fraction of the code or program. This becomes debatable with respect to scalability of parallel processing, more so with multi-core architectures. Sun and Chen (2009) assert that "the fixed-size assumption of Amdahl's law is unrealistic and results in pessimistic predictions; this does nothing to encourage healthy growth in the scale of multicore architectures" (Sun & Chen, 2009, p. 188).

Performance improvement of parallel applications running on multi-core architectures can be a complicated job since it is affected by a large number of variables or parameters. The performance of parallel computing essentially consists of two levels: algorithm related and network related. Since the scalability is not automatic, parallel programming can become cumbersome from algorithmic standpoint to arrive at desired performance. There are several tools developed by researchers to address this. Ni and Tai (1990) have nicely summarized the available tools.

What exactly performance of a parallel computing or application mean? There can be potentially several definitions for this. Nevertheless, one of the desired attributes is to expect "systemic performance consistency". Evans (2005) defines systemic performance consistency, from the parallel application standpoint, as the one that implies minimal or at least reasonable variation in runtime when operating conditions are similar (e.g., problem size, input data, network resources etc.).

Degraded systemic performance consistency means inconsistent runtime behavior of applications. This can lead to loss of time, money, overuse of resources, overbilling, etc.

Based on algorithm, process scheduling, and network parameters, there are several ways to improve the systemic performance consistency of a parallel application. Understanding the influence of different parameters on parallel application performance is often referred to as "characterization of parallel application".

"Determining the number of processors to be allocated per job that gives rise to good system performance is important for the global scheduler employed by the operating system for parallel environments" (Majumdar & Yiu Ming Leung, 1994, p.304). Does one need to consider the importance of I/O behavior while scheduling processes? Yes. Majumdar and Leung (1994) emphasize the importance of I/O characteristics of a parallel application on process scheduling.

Several scheduling methods for improving parallel application performance have been proposed by researchers. A few of interesting approaches are worth mentioning here, Sinen et al. (2006) proposed a method for optimized scheduling based on computation to communication ratio (CCR) of an application while Berman and Wolski (1996) introduced an "application-centric" scheduling method in which everything about the system is evaluated in terms of its impact on the application.

An interesting way of combining the Maui Scheduler, as a plug-in, to the portable batch scheduler (PBS) package was proposed by Bode et al. (2000). This method was tested on many clusters of varying size, performance, and network communication patterns, focusing on maximization of resource utilization as well as the execution of large parallel applications. Dinda (2002) discusses about the interface and the approach to implement a real-time job scheduling advisor (RTSA). RTSA is mainly based on appropriately predicting the host load. Based on Dinda's results, this prediction-based strategy seems to be highly effective to improve the performance of a parallel application. Calzarossa et al. (2004) proposed a methodological approach to analyze parallel applications performance in an automated fashion essentially based on key performance metrics, load imbalance and dissimilar processor behavior. The aim of this method is to identify and address local inefficiencies. The effects of running large multiple applications and core-level interactions on MCMP systems, however, were not studied in these scheduler researches.

Another important and interesting way to study the performance of a parallel application is from the network standpoint. How sensitive is the application with respect to network parameters such as network speed, switching, routing, existence of other applications (network communication load)?

Understanding and quantifying the "communication overhead" is paramount for such a study. Singh et al. (1994) discuss the three main attributes used extensively in capturing the communication overhead in a parallel architecture, "namely temporal, spatial, and volume components. Temporal behavior is captured by the message

generation rate, spatial behavior is expressed in terms of the message distribution or traffic pattern, and the volume of communication is specified by the number of messages and the message length distribution" (Sivasubramaniam, 1997, p. 1027). All the three attributes mentioned above are widely used in performance analyses of interconnection networks. Carter et al. (2007) did a systematic study on the effects within node by using applications run at low concurrencies and node-interconnect interactions on multi-core machines.

Running parallel applications on multi-core architectures often result in resource contention. Jin et al. (2009) did a differential performance analysis to quantify this contention effect for various benchmark applications and developed a method to isolate contention for shared resources in multi-core systems. The multi-core architecture itself can be an important point of consideration. Datta et al. (2008) provided several key insights into the architectural trade-offs of emerging multi-core designs.

In multi-core multi-processor systems, communication occurs between nodes (inter-node) and also between cores (intra-node). Chai et al. (2007) in their study observed that, in a multi-core cluster, optimization of intra-node communication is equally important as optimization of inter-node communication.

The primary focus of this research work was on the characterization of runtime sensitivity of parallel applications based on different core allocation strategies under a given network communication load, specifically on multi-core multi-processor machines. A range of methodologies are available to conduct a characterization study and they can be broadly classified under four categories: analytical modeling, direct measurement (or actual execution), simulation, and emulation.

The analytical modeling method typically involves a set of characteristic equations that relates various parameters and metrics. In contrast to the other methods, applications are not 'run' per se. Hennessy and Patterson (1996) discuss in detail of approaching this in a quantitative sense. Jain (1991) also discusses how the analytical

methods are used to enhance systems performance for different applications. However, analytical methods suffer from two main weaknesses: limited accuracy and inability to model system related feedback in real systems.

While the analytical methods are considered less realistic, the most realistic of the methods is the direct measurement or actual execution method. Many times this method is used judge different network systems, for example, Hall et al. (1997) did a comparative study of NFS (Network File System) performance over different network systems such as Autonet, Ethernet, and FDDI (Fiber Distributed Data Interface). For given network conditions, direct measurement methods offer excellent accuracy, however, their potential weakness is that it may not be possible to configure the network system "properly" to extrapolate to other similar systems.

Simulation methods offer the most flexibility and are very popular because of this reason. Simulators are close approximations to real systems. While the simulation methods offer flexibility and powerfulness to test any network system, they often lack fidelity and scalability. Many times extensive validation of simulators needs to be performed as well. Woo et al. (1995) used SPLASH-2 simulators for characterization and studied the effects of scaling.

In emulation methods, typically a portion of the network is loaded or emulated while a parallel application being run in the same network in real time. Compared to simulation, emulation methods are not that flexible since full-blown control of the network cannot be achieved due to the real time run of the parallel application. However, emulation methods are scalable to a larger extent compared to simulation. Understanding or observing real time runs of applications can be challenging since real time measurements are needed. Fortunately, emulation methods can gain from real time measurement studies such as the one from Anderson et al. (1997).

Evans and Hood (2005) developed a methodology and framework for studying the impacts of network and communication performance on parallel application runtime.

They developed the Parallel Application Communication Emulator (PACE) framework which essentially executes one or several “emulated parallel applications” (EAs). In this framework, EAs are defined and executed by a combination of runtime setup parameters and the calculation of compute/communication cycles based on measured communication performance. Communication cost parameters are determined by the tool through linear interpolation. Based on the values, an overall runtime can be predicted for each EA. The difference between the predicted and actual runtime is an indication of the network performance variability.

Evans and Hood (2006) further extended the PACE framework by adding a Parallel Application Runtime Sensitivity Evaluation (PARSE) program. Further, “a necessary condition for using PACE to evaluate a parallel application under test (AUT) is to force concurrent execution while PACE emulates one or more parallel applications. PACE then can be viewed as a communication network ‘disrupter’, providing a controlled and repeatable quantity and temporal dispersion of network traffic, directly competing for network resources with the AUT” (Evans & C. S. Hood, 2006, p. 3). What PARSE does is that it redistributes the nodes allocated by job scheduler to execute both PACE and AUT simultaneously, at the same time, it ensures that the AUT execution begins only after PACE computes the related communication cost parameters.

Evans and Hood (2011), in their work, used two 48-node cluster segments, each node consisting of single-processor. They evaluated NAS parallel benchmarks and PSTSWM using the PACE and PARSE framework. They also defined run time sensitivity metrics such as coefficient of mean (COM) and coefficient of variance (COV) to quantify the sensitivity of each parallel application to network performance.

While the previous works cited here provides significant insights on understanding and characterizing parallel application runtime sensitivity, their focus was on network level (inter-node) communication. In this study, Evans and Hood’s method was implemented on MCMP systems to characterize parallel application runtime sensitivity to core allocation under a given network communication load.

As a prelude to this, a preliminary study was conducted to assess the runtime sensitivity of various NAS benchmarks (Veeraraghavan & Evans, 2010). In this, PARSE, PACE, and the sensitivity metric were used to better understand their utility on MCMP systems. The network load was kept constant at 95% using PACE and the process (core) allocation was changed. This preliminary runtime evaluation of NAS benchmarks suggested that as the core allocation "distance" increased, the runtime of the applications also increased (in some cases, there was up to a threefold increase). For some applications, such as MG, the variability also increased significantly. This warranted further detailed study (the current work) on the parallel application runtime sensitivity due to core allocation on a loaded network.

CHAPTER 3. METHODOLOGY

3.1. Introduction

A key benefit of using parallel processing is that it reduces the overall computational time required and in addition, it also allows for larger problems to be solved. This is especially true in the case of long running scientific and engineering programming codes. Both in academia and industry, there have been several research initiatives to develop algorithms to reduce computational time. On the other hand, one of the challenges faced while running parallel applications is their runtime variability. It is important that the system (a set of application codes, compute and I/O nodes, schedulers, resource managers, and the interconnection network) operate with systemic performance consistency. Systemic performance consistency is defined as a minimal or at least a reasonable variation in runtime when operating conditions such as, problem size, input data, compute and network resources etc., are similar over time. The growth in HPC tends toward solving larger problems using huge amount of data. In this situation, the runtime variability or systemic performance inconsistency is not desirable.

Understanding the influence of different parameters on parallel application performance is often referred to as “characterization of parallel application”. A range of methodologies are available to conduct such a study using analytical modeling, direct measurement (or actual execution), simulation, or emulation.

Evans (2005) in his doctoral thesis had developed tools such as PACE and PARSE and studied parallel application runtime sensitivity on clusters consisting of single-processor machines. As a complement, the primary focus of this research work was to characterize parallel application runtime sensitivity to core allocation under a given network communication load on a cluster made of multi-core multi-processor

machines using PACE and PARSE and compare the results to those of single-processor machines as applicable.

3.2. Tools used

3.2.1. PACE

In order to study the run time performance anomalies, a framework called PACE (Parallel Application Communication Emulator), that emulates one or more parallel applications, was developed by Evans (2005). PACE monitors itself, producing data that reflects the error between its prediction of an emulated application (EA) run time and the actual run time.

The time that an individual processor (p) takes to execute its portion of the parallel program can be given as,

$$T^p = T^p_{comp} + T^p_{comm} + T^p_{idle} \quad \text{(Eq.3.1)}$$

$$T = \frac{1}{P}\left(T_{comp} + T_{comm} + T_{idle}\right) \quad \text{(Eq.3.2)}$$

$$= \frac{1}{P}\left(\sum_{p=0}^{P-1} T^p_{comp} + \sum_{p=0}^{P-1} T^p_{comm} + \sum_{p=0}^{P-1} T^p_{idle}\right)$$

Where, P – Processors working on a problem

T_{comp} – Computation time

T_{comm} – Communication time

The communication cost in parallel program is given by,

$$T_{msg} = \alpha + \beta n \quad \text{(Eq.3.3)}$$

Where, α - Startup time

β - Transfer time of a unit of data

n - Number of units transferred

Combining the above two equations yield,

$$T_{comm} = \sum_{p=0}^{P-1} \sum_{k=1}^{M} (T_{msg})_k^p = \sum_{p=0}^{P-1} \sum_{k=1}^{M} (\alpha + \beta n)_k^p \qquad \text{(Eq.3.4)}$$

The PACE framework is designed to load a cluster network using EAs in a prescribed manner. As mentioned earlier, one or more EAs within the PACE framework is created and ran. Each EA emulates running a parallel application. As in any parallel application, the scheduler allocates certain number of nodes for a fixed time for PACE. For each EA, a subset of the PACE nodes is allocated. Then, each EA runs its own computation and communicates via the interconnection network of the cluster. A typical PACE run consists of two "phases". The first phase determines the communication cost T_{comm} and predicts the overall run time of each EA. The second phase is an emulated application *run* consisting of a number of compute/communicate *cycles* that are either specified or calculated according to user input. Each cycle is composed of a *computation* and *communication* component. PACE executes its performance measurement by executing many communication exchanges and timing them using hardware timers.

Each process within an EA determines communication cost β using linear interpolation,

$$\beta = \frac{\frac{t_l}{m} - \frac{t_s}{m}}{n_l - n_s} \qquad \text{(Eq.3.5)}$$

Where, t_l - Time to run all the communication exchanges on the large message size

t_s - Time for the small message size

m - Number of messages

n_l and n_s - Lengths in bytes for large and small messages respectively

Values of the startup cost α are calculated for each message size and is given by,

$$\alpha_l = \frac{t_l}{m} - \beta * n_l \qquad \text{(Eq.3.6)}$$

$$\alpha_s = \frac{t_s}{m} - \beta * n_s \qquad \text{(Eq.3.7)}$$

These values should always be equivalent. Each process determines their own α and β values independently. They are then gathered from the processes, and after computing an average, gets redistributed to the processes prior to the start of the run. The PACE benchmark, for all participating processes, calculates a single average value. Each PACE EA process uses the same communication performance metric as the basis for calculating their compute component time.

$$T_{comp} = \frac{T_{comm} * 100}{L} - T_{comm} \qquad \text{(Eq.3.8)}$$

Where, T_{comp} - Calculated compute time

T_{comm} - Calculated communication time based on timing measurements $\alpha + \beta$

L - Percent communication load

The PACE software is written in 'C' language and uses MPI for message passing. PACE is built using the GNU gcc compiler. The PACE system consists of several functional modules such as, input parameter processing, run-time configuration, communicators (EAs) and communication patterns, emulated computation, communication cost measurement, and data collection and logging.

Before PACE executes an EA, it performs a communication cost measurement using the communication type and pattern to be executed by EA. Then, using the communication cost measurement, communication and compute times, PACE calculates the per cycle time. Overall, EA runtime is then determined using the combination of user input and measured communication cost.

With MCMP systems, PACE is used in the context of cores (instead of nodes/processors) and the cluster intercommunication comprise of, in addition to inter-node communication, inter-core and inter-processor communication through the network. In this case, PACE is not just acting as a network load 'disrupter' but also acts as a communication sub-system load.

3.2.2. PARSE

To evaluate a parallel AUT using PACE, it is necessary to run them concurrently while PACE emulates one or more EAs. Here, "PACE can be viewed as a communication network 'disrupter', providing a controlled and repeatable quantity and temporal dispersion of network traffic, directly competing for network resources" (Evans & C. S. Hood, 2006, p. 3). In order to achieve this, an interface to the PACE framework was developed (Evans & C. S. Hood, 2006) called Parallel Application Runtime Sensitivity Evaluation (PARSE). PARSE is designed to address the two runtime aspects. They are:

1. To accommodate and run PACE and AUT by distributing the nodes allocated by the scheduler.
2. To guarantee that the AUT runs concurrently with PACE EAs and to ensure that AUT is not executed during the time when PACE is computing communication cost parameters.

Initial version of PARSE was written as a perl script that parses the machine file created by scheduler and creates two new machine files namely, pace.mach for running PACE and aut.mach for running AUT. PARSE can be run using a PBS script. There are several command line arguments such as

–m for the machine file

–N for the total number of nodes

–P for the PACE nodes

–a for the AUT nodes

–A for the AUT executable

–p to specify alternate mpi path (optional)

–S for the "stride" factor

–i for number of iterations or trial runs

–d for delay in seconds

–I for specifying input file for PACE.

An updated version of PARSE (python script) now takes advantage of the Hydra process manager (discussed in section 3.2.4) used in MPICH2 (discussed in section 3.2.5), which supports strict process binding. The following additional options are available in the updated version of PARSE.

-W argument is used to specify AUT path

-H argument is used to invoke Hydra

-B argument is used to run a baseline test

-S argument is used to run a sensitivity test

In addition, with the modified version the user creates the machinefile and specifies using the –m argument on the command line for baseline tests. While running sensitivity tests, machinefiles- autsens.mach and pacesens.mach should be provided by the user.

In order to ensure concurrent execution of PACE and AUT, "PARSE script uses system calls embedded in parent and child branches of a fork() system call. Essentially each branch performs its own mpirun. The parent process executes PACE while the child process runs the AUT " (Evans & C. S. Hood, 2006, p. 4). Before running the EAs, PACE performs its communication cost benchmark.

3.2.3. PBS Resource Manager and Maui Scheduler

The cluster used in this study employs OpenPBS as the resource manager. OpenPBS is an open source version of Portable Batch System (or simply PBS). Open PBS is a NASA's Ames research center developed, POSIX compliant batch software. It was developed originally for large parallel computers (Symmetric multiprocessing (SMP) system). The primary function of PBS is to allocate and manage resources for computational tasks along with effective job scheduling.

An effective scheduler is responsible for:

1. Managing traffic by properly allocating resources to a job and by avoiding the other jobs from using the same resources.
2. Maintaining site mission goals by providing a suite of policies that can be mapped into scheduling behavior.
3. Implementing intelligent scheduling decisions to maximize cluster performance.

There are several built-in schedulers available within PBS and they can be customized depending on individual site requirements. The FIFO scheduler is the default PBS scheduler, which enforces maximum CPU utilization. It searches through the queue and starts jobs based on available resources. This can be a limitation for large jobs since the resources could not be met by certain nodes even though they become available. In situations like this, other schedulers can be used depending on cluster and problem size. To achieve this, PBS supports interfacing with other meta schedulers (or plug-in schedulers) like Maui scheduler.

Maui scheduler is an open-source job scheduler and it can be readily used for clusters and supercomputer systems. It can accommodate a large array of policies for job scheduling since it is optimized and highly configurable. It also incorporates features such as dynamic priorities, fair share, and extensive reservations. “The Maui Scheduler can be thought of as a policy engine which allows sites control over when, where, and how resources such as processors, memory, and disk are allocated to jobs. In addition to this control, it also provides mechanisms which help to intelligently optimize the use of these resources, monitor system performance, help diagnose problems, and generally manage the system” (“Maui Scheduler - Administrator’s Guide,” p. 1).

The combination of PBS and Maui scheduler is highly successful in scheduling parallel applications, and in general, improves the manageability and efficiency of cluster computing in many cases as with the cluster system used in this study.

3.2.4. Hydra Process Manager

In order to ensure core binding in MCMP systems, it is necessary to use a tool that can be invoked via PARSE. Hydra process manager developed by Argonne National Laboratory was used in this study for this purpose.

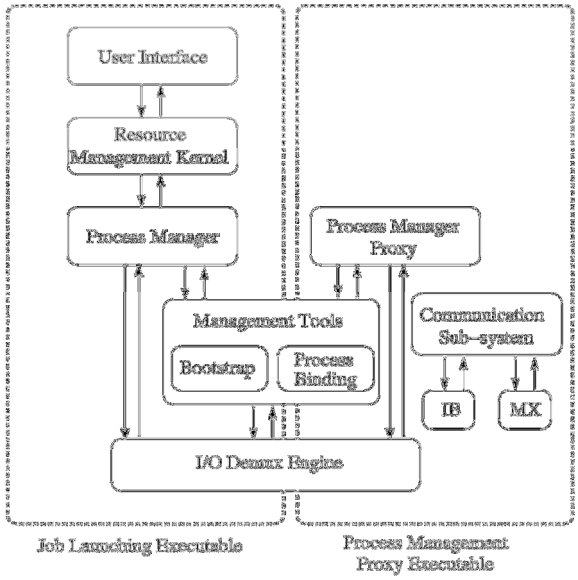

Figure 3.1 Hydra Process Management Framework

Figure 3.1 shows the schematic of the Hydra process management framework which consists of the following basic components:

1. User Interface
2. Resource Management Kernel
3. Process manager
4. Bootstrap server
5. Process Binding
6. Communication Sub-system
7. Process Management proxy
8. I/O demux engine

The process binding component essentially extracts the system architecture information (such as the number of processors, cores etc.) and binds processes to different cores in a portable manner.

3.2.5. MPICH2

MPICH2 is the Message Passing Interface (MPI) used by the programs in this study. MPICH2 is a portable MPI implementation. As per Argonne National Laboratory, "the goals of MPICH2 are: (1) to provide an MPI implementation that efficiently supports different computation and communication platforms including commodity clusters (desktop systems, shared-memory systems, multicore architectures), high-speed networks (10 Gigabit Ethernet, InfiniBand, Myrinet, Quadrics) and proprietary high-end computing systems (Blue Gene, Cray, SiCortex) and (2) to enable cutting-edge research in MPI through an easy-to-extend modular framework for other derived implementations" ("MPICH2 : about MPICH2").

3.3. Benchmarks used in the study (AUT)

3.3.1. NAS Parallel Benchmark

The Numerical Aerodynamic Simulation (NAS) Program is based of NASA Ames Research Center. The program objective is to advance computational aerodynamics to new levels. To help "measure the performance of highly parallel computers and to compare their performance with that of conventional supercomputers, NAS developed the NAS Parallel Benchmarks (NPB 1.0) in 1991. These benchmarks, which are derived from computational fluid dynamics codes" (D. Bailey, Harris, Saphir, Van Der Wijngaart, A. Woo, & Yarrow, 1995, p.3), consist of two main parts: a total of five "parallel kernel" benchmarks and a total of three "simulated application" benchmarks. Each of the five kernels benchmark denote a particular numerical simulation or computation. The CFD applications (numerical simulations) are a "representative of

the types of data movement and computation required in state-of-the-art CFD application codes" (Saini & D. H. Bailey, 1996).

For years, the high performance computer systems have grown significantly in size and capabilities. Total numbers of processors have gone up, computer clock speed and memory limits have increased, and network bandwidths have increased. Therefore, the needs for more challenging benchmark sizes to rate the performance of the parallel machines have grown. For this reason, the NAS benchmarks come in different problem sizes given as "class". The NAS Parallel Benchmarks consist of 6 different problem sizes, they are, Class "S", "A", "B", "C", "D", and "W". The class "A" benchmarks can be run on a medium powered workstation, class "B" on high-end computers or smaller parallel systems, and class "C" on high-end parallel systems. In order to study the runtime variations with network communication load, classes "A", "B", and "C" are selected and their problem sizes are shown in Table 3.1 (D. Bailey, Harris, Saphir, Van Der Wijngaart, A. Woo, & Yarrow, 1995b).

Table 3.1 *Parallel Benchmark Problem Sizes*

Benchmark	Abbreviation	Class A	Class B	Class C
Embarrassingly Parallel	EP	2^{28}	2^{30}	2^{32}
MultiGrid	MG	256^3	256^3	512^3
LU solver	LU	64^3	102^3	162^3

Two kernel benchmark and one simulated application benchmark (Saini & D. H. Bailey, 1996) are used for this study. They are:

- The first kernel benchmark is the "Embarrassingly Parallel problem. In this benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudorandom numbers, which are generated according to a particular scheme that is well-suited for parallel computation. This problem is typical of many Monte Carlo applications" (Saini & D. H. Bailey, 1996)

- Simplified MultiGrid (MG) problem is the second of the kernel benchmarks that solves three-dimensional Poisson PDE. MG can be considered as a good test for both, short distance and long distance communication that is highly structured. Even though Class A and Class B problems have the same size, Class B uses more iterations for inner loop calculations than Class A.
- The third benchmark, which is a simulated application benchmark, is the Lower-Upper diagonal (LU) benchmark. Basically, "it does not perform a LU factorization but instead employs a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block *5x5* lower and upper triangular system. This problem represents the computations associated with a newer class of implicit CFD algorithms, typified at NASA Ames by the code INS3D-LU" (Saini & D. H. Bailey, 1996)

3.3.2. PSTSWM Benchmark

"The Parallel Spectral Transform Shallow Water Model (PSTSWM) is a message-passing application and parallel algorithm testbed that solves the nonlinear shallow water equations on a rotating sphere using the spectral transform method. It is a parallel implementation of STSWM, developed by J. J. Hack and R. Jacob at the National Center for Atmospheric Research (NCAR) and used to generate reference solutions for the shallow water test cases" (Smith, Vetter, & Xuejun Liang, 2005, p. 3). Within PSTSWM, there are several multiple parallel algorithms which can be chosen during run-time. Parameters such as the number of processors, problem size, and decomposition (data) can also be specified.

As per the user guide, "PSTSWM is written in Fortran 77 with VMS extensions and a small number of C preprocessor directives. Message passing is implemented using MPI, MPI/SHMEM, PICL, PVM, and/or native message passing libraries, with the choice being made at compile time. Additionally, all message passing is encapsulated in three high level routines for broadcast, global minimum and global maximum, and in two

classes of low level routines representing variants and/or stages of the swap operation and the send/receive operation" (Worley & Toonen, 1995).

The recent version of PSTSM v6.9 was used in this research work. PSTSWM is, in general, provides large problem size and communication intensive allowing it to be a good testing algorithm for HPC systems.

3.4. HPC cluster at ACSL

The Adaptive Computing Systems Lab (ACSL) consists of two cluster systems, the Chiba City cluster and HP cluster. Chiba cluster consists of several 16 node cluster segments plus login and administrative machines. Each node is a dual-processor Pentium III with a 9GB of hard drive space and a RAM of 512MB. A fast Ethernet (100Mbps) connects all the nodes. The HP cluster consists of 10 nodes and an administrative machine. It is a HP Proliant DL185G5 and DL165G5p machines. Each node has 2 quad core AMD Opteron processors with 16GB of RAM and a 500GB SATA HD and the nodes are interconnected with two Gigabit Ethernet network ports.

In this research, the parallel application runtime sensitivity to network communication load for the MCMP systems was studied. For this purpose, the HP cluster of ACSL was used (Figure 3.2). Open PBS (Torque) is used as the resource manager and Maui scheduler is used for job scheduling for ACSL cluster.

The operating system used by the ACSL cluster is RedHat Linux version 5.5. The MPI is through MPICH2 version 1.3.2. For studying NAS benchmarks, gcc compiler version 4.4.0 is used and for PSTSWM, pgi compiler version 9.0-3 is used.

Figure 3.2 HP cluster of ACSL

3.5. Experimental Matrix

AUT's sensitivity to network communication load by varying core allocation in a MCMP machine was studied in this project. For this purpose, a series of test cases were conducted under two scenarios. In the first scenario, baseline tests for each benchmark were conducted by varying the number of cores from 2 to 32 in powers of two. Each benchmark was run 30 to 100 times to ensure the results were statistically significant enough to capture the runtime and variability.

In the second scenario, sensitivity tests were conducted by running each benchmark (AUT) concurrently with PACE using PARSE. PACE was used as a communication "disruptor" against AUT. The entire HP cluster of ACSL (a total of 10 nodes) was used. Each node has 2 quad-core processors (Figure 3.3), which comprises of a total of 80 cores.

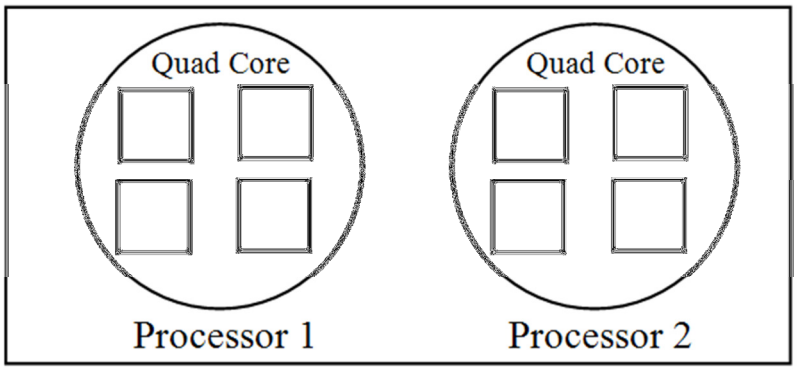

Figure 3.3 Layout of a HP cluster node with 2 processors of quad-core each

For sensitivity testing, PACE was run on cores that were not used by AUT so as to load the network at 95% communication load (using 512K messages in a all-to-all communication pattern). The different core allocation policies were implemented by PARSE. For single-core single-processor machines PARSE uses "stride" factor to allow user to specify almost any node distribution for PACE and AUT. However, in multi-core multi-processor machines, the "stride" factor, in general, does not ensure core binding as per allocation. In our study, it is important to achieve proper core binding in MCMP so that core allocation influence of AUT runtime sensitivity can be properly determined. For this purpose, PARSE was modified to invoke Hydra process manager for core binding. In the related preliminary study (Veeraraghavan & Evans, 2010), core binding was found to be not effective since Hydra was not used.

The experimental matrix of AUT and PACE is given in Table 3.2. with stride factor and respective core allocation policy for 2 to 32 (in powers of 2) core runs. For each set of cores, three test cases TC1, TC2, and TC3 were run for both baseline and sensitivity tests. These test cases were designed to allocate jobs across the cores based on following core allocation policies:

1. Run on contiguous cores (TC1)
2. Run on cores across contiguous nodes (TC2)
3. Run on cores across nodes based on largest "stride" factor (TC3)

Table 3.2 *Experimental Matrix*

# of Cores	Test Case	Stride Factor	AUT Nodes	AUT Cores	PACE Nodes	PACE Cores
2	TC1	2	1	0,2	1	1,3,4,5,6,7
					2-4	0-7
	TC2	4	1	0,4	1	1,2,3,5,6,7
					2-4	0-7
	TC3	79	1	0	1	1-7
			10	7	2-9	0-7
					10	0-6
4	TC1	1	1	0-3	1	4-7
					2-4	0-7
	TC2	8	1-4	0	1-4	1-7
	TC3	26	1	0	1	1-7
			4	2	2,3,5,6,8,9	0-7
			7	4	4	0-1, 3-7
			10	6	7	0-3,5-7
					10	0-5,7
8	TC1	1	1	0-7	2-10	0-7
	TC2	8	1-8	0	1-8	1-7
					9,10	0-7
	TC3	11	1	0	1	1-7
			2	3	2	0-2,4-7
			3	6	3	0-5,7
			5	1	4,8	0-7
			6	4	5	0,2-7
			7	7	6	0-3,5-7
			9	2	7	0-6
			10	5	9	0-1,3-7
					10	0-4,6-7
16	TC1	1	1,2	0-7	3-10	0-7
	TC2					
	TC3	5	1,6	0,5	1,6	1-4,6,7
			2,7	2,7	2,7	0,1,3-6
			3,8	4	3,8	0-3,5-7
			4,9	1,6	4,9	0,2-7
			5,10	3	5,10	0-2,4-7
32	TC1	1	1-4	0-7	5-10	0-7
	TC2					
	TC3	2	1-8	0,2,4,6	1-8	1,3,5,7
					9,10	0-7

The above test cases were chosen to allow varying “distance” between cores starting from the most intuitive (contiguous cores) to the widest “distance”. This will also allow for any intermediate cases, to be estimated from these test cases. As shown in

Table 3.2, for both baseline and sensitivity testing scenarios, these three different test cases (core allocation policies) were evaluated.

As a special case, for 2-core benchmark runs, the following core allocation policies were used:

1. TC1- Allocate the AUT to run within the same processor (core 0 and 2). In the case of 2-core runs, TC1 could be allocating to any of the 4 cores since architecturally they all are within the same processor and hence can be considered contiguous.
2. TC2- Allocate the AUT to run across processors (core 0 and 4).
3. TC3- Allocate the AUT to run across different nodes (core 0 and core 79).

3.6. Sensitivity Factor

Once the test runs were completed as per the experimental matrix, run time data were extracted from the benchmark output files. The collected data was further examined for outliers and they were removed using Grubbs method through Minitab. The following terms and definitions from Evans (2005) are worthwhile to be included here as the same terms and definitions apply to the current study also.

"***Coefficient of Mean (COM):*** The ratio of the average values of multiple application executions under a given pair of network load operating conditions" (Evans, 2005, p. 132)

$$COM_{ji} = \frac{x_{mean\ j}}{x_{mean\ i}} = \frac{\sum_1^n \frac{T_{run\ j}}{n}}{\sum_1^n \frac{T_{run\ i}}{n}} \qquad \text{(Eq.3.9)}$$

Where n is the number of trial runs of the application

"***Coefficient of Variation (COV):*** The ratio of the standard deviation of multiple application executions under a given network load operating condition to the mean of those executions" (Evans, 2005, p. 133)

$$COV = \frac{\sigma}{x_{mean}} . 100 \quad \text{(Eq.3.10)}$$

The COM normalizes the runtime scale between applications, while the COV normalizes the differences between applications. COV values can also be represented as a ratio between two data points (say i and j), in which case it indicates the relative increase or decrease. This quantity is termed as ROV_{ji} (Ratio of Variations).

$$ROV_{ji} = \frac{COV_j}{COV_i} \quad \text{(Eq.3.11)}$$

Both these factors are used to arrive at the sensitivity factor as shown below:

$$S_{ji} = ROV_{ji}\, COM_{ji} \quad \text{(Eq.3.12)}$$

Sensitivity factors were calculated using the above method for different core-allocations, namely, S_{TC1}, S_{TC2} and S_{TC3} and averaged to arrive at a single sensitivity factor for core-allocation, namely S_{alloc}. Even though there are other possible core allocation policies, apart from those tested in this study, an average of S_{TC1}, S_{TC2} and S_{TC3} should give a good estimate of the sensitivity of allocation since it covers a wide range of possibilities. The allocation sensitivity factor, S_{alloc} was computed for each parallel application by varying the number of cores from 2 to 32 in powers of two. Providing the allocation sensitivity factor to schedulers for a particular application can be very useful since S_{alloc} covers varying possibilities of core allocations and provides a sense of the application behavior as a function of core allocation. Results of the different parallel applications and related discussions are presented in Chapter 4.

3.7. Experiment Execution

For executing the experimental matrix shown in Table 3.2, the below procedure was followed in this study:

1. Build and compile PACE code
2. Build and compile NAS benchmarks and PSTSWM benchmark
3. Using a script file
    a. Request for nodes/cores and wall time

b. Load mpich2, gcc compiler (for NAS benchmarks), and pgi compiler (for PSTSWM benchmark)

```
# Setup and select the run-time environment
source /opt/admintools/Modules/etc/profile.modules
module load mpich2-1.3.2/64/nemesis-gcc-4.4.0/4.4.0
module load mpich2-1.3.2/64/nemesis-pgi-9.0-3/9.0-3
```

c. Invoke and run PARSE by providing required options and inputs.

4. PARSE in turn, according to provided options, runs baseline and sensitivity tests and collects data.
5. Post processing of the collected data

An example of invoking PARSE for a 4-core job run is given below:

Initialization of AUT variables

```
AUT="mg"
CLASS="A"
AUTPROC="4"
AUTFILE="mg.A.4"
AUTPATH="./$CLASS/$AUTPROC"
```

PARSE command line for baseline test

```
./parse.py -N 4 -P 0 -a 4 -W $AUTPATH -A $AUTFILE -i 100 -d
1 -m base4.mach -H -B
```

While running baseline test, the following base4.mach machinefile is provided to PARSE

```
hpn01:1 binding=user:0
hpn02:1 binding=user:0
hpn03:1 binding=user:0
hpn04:1 binding=user:0
```

PARSE command line for sensitivity test

```
./parse.py -N 32 -P 28 -a 4 -W $AUTPATH -A $AUTFILE -i 100
-d 1 -I pace_input.sens -H -S
```

While running sensitivity test, the following machine files and input files are provided to PARSE:

autsens.mach

```
hpn01:1 binding=user:0
hpn02:1 binding=user:0
hpn03:1 binding=user:0
hpn04:1 binding=user:0
```

pacesens.mach

```
hpn01:7 binding=user:1,2,3,4,5,6,7
hpn02:7 binding=user:1,2,3,4,5,6,7
hpn03:7 binding=user:1,2,3,4,5,6,7
hpn04:7 binding=user:1,2,3,4,5,6,7
```

PACE input file:

pace_input.sens

```
BENCHPERPROC:1
BENCHAVG:1
PERFONLY:0
EAS:1
LOG:1
INTERCONNECT:1
COMMLOAD:95
MESSAGES:1000
SIZESTART:524288
SIZEEND:1048576
SYNCHRONIZE:0
RUNTIME:600
PATTERN1:P2P
TEST1:ALLTOALL
ALPHA1:
BETA1:1.
PATTERN2:
TEST2:
ALPHA2:
BETA2:
PATTERN3:
TEST3:
```

```
ALPHA3:
BETA3:
PATTERN4:
TEST4:
ALPHA4:
BETA4:
FILEPATH:./A/4/
```

PSTSWM input files:

Algorithm:

```
8                / NPLON
4                / NPLAT
1                / MESHOPT
1                / RINGOPT
1                / FTOPT
0                / LTOPT
11               / COMMFFT
11               / COMMIFT
20                / COMMFLT
20                / COMMILT
0                / BUFSFFT
0                / BUFSIFT
0                / BUFSFLT
0                / BUFSILT
6                / PROTFFT
6                / PROTIFT
6                / PROTFLT
6                / PROTILT
0                / SUMOPT
0                / EXCHSIZE
```

Measurment:

```
.TRUE.          / TIMING
.FALSE.          / TRACING
.FALSE.          / TRACEFILE
0                / VERBOSE
```

```
100000          / TRSIZE
1               / TRSTART
10000           / TRSTOP
0               / TL1
0               / TL2
0               / TL3
'timings'       / TOUTPUT
                / TMPNAME
                / PERMNAME
1               / INITSTEPS
```

Problem:

```
'0002'              / CHEXP
42                  / MM
42                  / NN
42                  / KK
64                  / NLAT
128                 / NLON
16                  / NVER
                    / NGRPHS
                    / A
                    / OMEGA
                    / GRAV
                    / HDC
0.0                 / ALPHA
1800.0              / DT
999.0               / EGYFRQ
0.01                / ERRFRQ
999.0               / SPCFRQ
12000.0             / TAUE
                    / AFC
.TRUE.              / SITS
                    / FORCED
                    / MOMENT
2                   / ICOND
```

CHAPTER 4. DATA ANALYSIS AND RESULTS

4.1. Introduction

Using the methodology discussed in Chapter 3, the runtime for the baseline tests of each parallel application (AUT) with no competing traffic and the runtime for sensitivity tests with network load were obtained from application output files. Using these baseline and sensitivity runtimes, sensitivity factors were computed.

Each parallel application was run several times (50-100 times for NAS benchmarks and 30 times for PSTSWM benchmark) for different core allocation strategies (test cases) and number of cores. The runtime data obtained for baseline and sensitivity runs were plotted against trials to represent in a graphical format. Visual inspection of the plots obtained for several benchmarks show existence (or non-existence) of patterns amongst various runs. For example, the difference in the means between baseline and sensitivity runs denotes the energy consumption difference. Similarly, difference in the variations about the means denote the difference in predictability.

For each application, sensitivity factors for different test cases (TC1, TC2 and TC3) were computed based on the ratio of means (COM) and ratio of variance (ROV) as described before. Then an average of the sensitivity factors for different test cases was computed to arrive at a single allocation sensitivity factor, S_{alloc}. The applications were run on 2 to 32 cores (in powers of 2) and corresponding allocation sensitivity factors were computed.

4.2. Baseline and Sensitivity runtime plots

Baseline and sensitivity runtime plots were created based on the data collected for various benchmarks. All the PSTSWM benchmark plots are discussed in this section. For NAS benchmarks, three different classes (Class A, B, and C) were run to assess the sensitivities with respect to the problem size. However, the plots shown in this section focus on Class C (largest problem size) only. In many cases, Class A and Class B plots remained similar to Class C. Existence of any difference with respect to problem size is highlighted and discussed accordingly. The rest of the plots obtained from this study for Class A and B are attached to Appendix.

EP Benchmark runs:

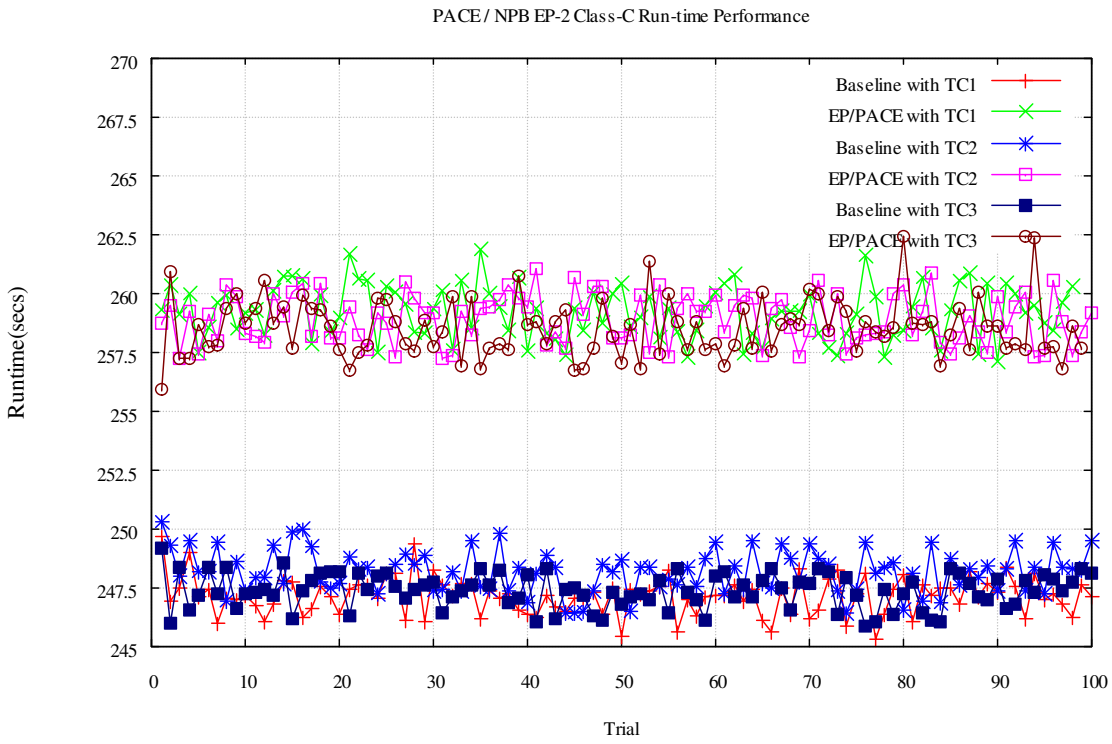

Figure 4.1 2-core allocations: Baseline and Sensitivity runtimes of EP Class C

From Figure 4.1 above, for 2-core allocations of EP Class C, the baseline runtimes of all the three test cases, TC1, TC2 and TC3, were very similar. The sensitivity

runtimes were about 5% higher than baseline and also were similar for all the three test cases. This shows that the application EP class C 2-core job is equally sensitive with respect to different core allocation policies. EP Class A and Class B also showed equivalent behavior for 2-core jobs.

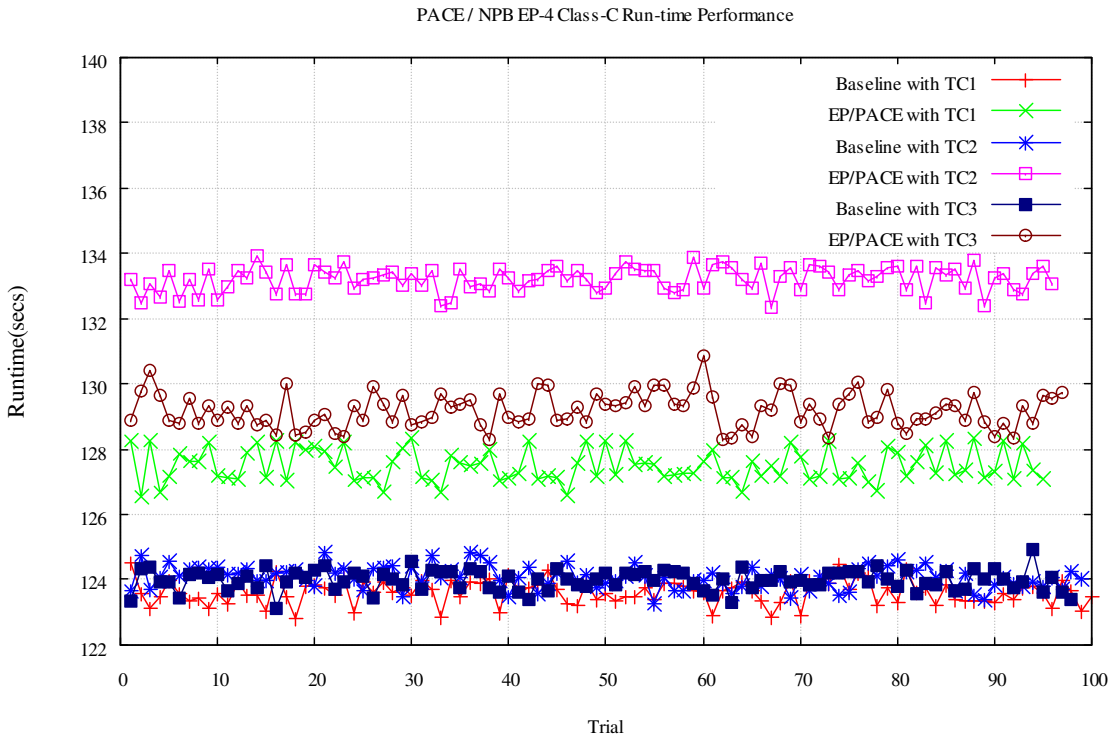

Figure 4.2 4-core allocations: Baseline and Sensitivity runtimes of EP Class C

Figure 4.2 above shows the baseline and sensitivity runtimes of 4-core allocations done using the three test cases, TC1, TC2 and TC3 for EP Class C problem. The baseline runtimes of these tests were very similar. For sensitivity tests, TC2 showed the most runtime, about 7% higher than the baseline runtime, while TC1 and TC3 showed about 3% and 4% higher runtime than their respective baselines. EP Class A and Class B also showed equivalent behavior for 4-core jobs under all three test cases.

Figure 4.3 shows the baseline and sensitivity runtimes of EP Class C 8-core runs. The baseline runtimes for TC2 and TC3 were similar. However, TC1 baseline was slightly lower than TC2 and TC3.

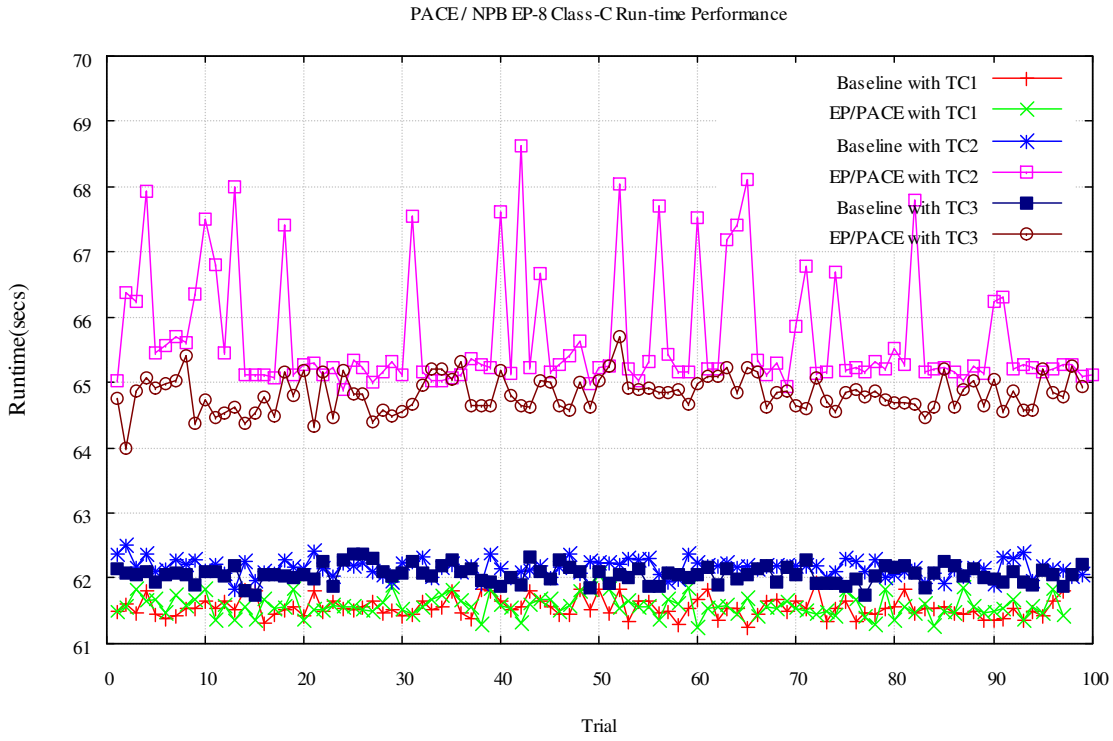

Figure 4.3 8-core allocations: Baseline and Sensitivity runtimes of EP Class C

The sensitivity runtime of TC1 remained more or less the same as its baseline suggesting insensitivity for this core allocation strategy (contiguous core allocation). This is somewhat intuitive and expected behavior since the application was allocated on contiguous cores, it had no influence from the network traffic due to PACE running on other cores in the cluster. On the other hand, TC2 and TC3 showed sensitivity due to the influence of network communication load. The sensitivity runtime of TC2 (about 6% more than its baseline) was slightly higher than TC3 sensitivity runtime (about 5% more than its baseline). TC2 also showed higher variation (more fluctuation) in runtime as compared to TC3. For Class A and Class B problems, TC2 was about 25% and 9% higher than its respective baseline runtimes, while TC3 was about 23% and 5% higher than its baseline runtimes.

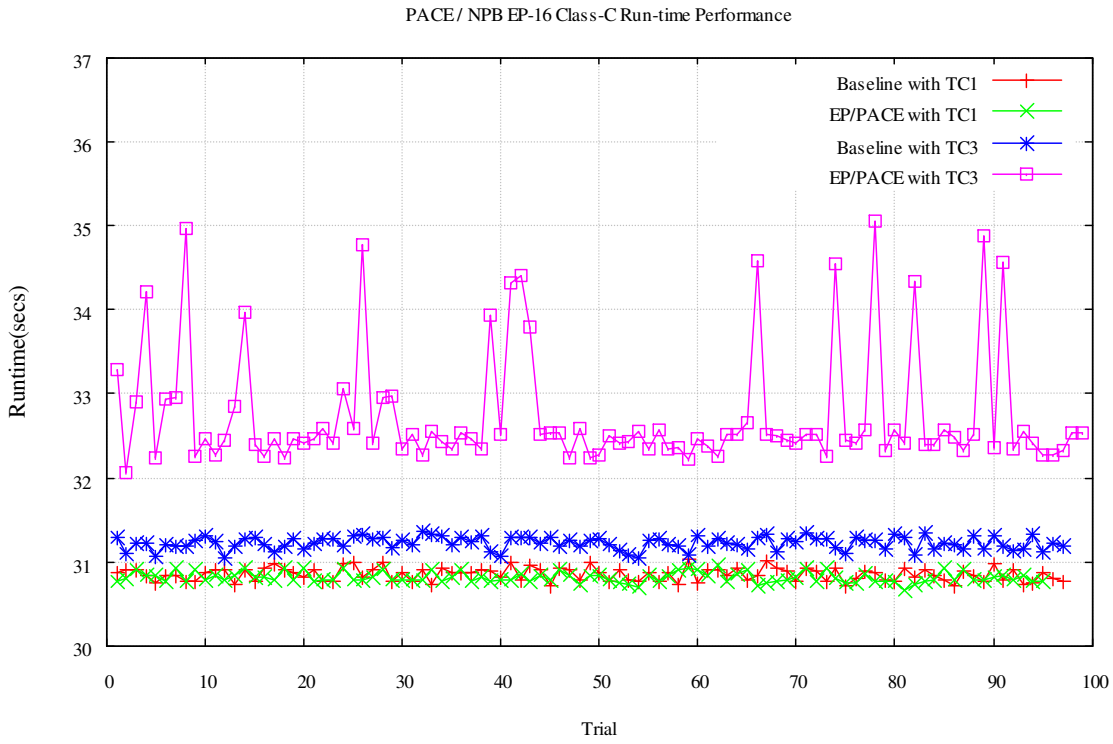

Figure 4.4 16-core allocations: Baseline and Sensitivity runtimes of EP Class C

In the ACSL cluster used in this study, there were only 10 nodes; therefore, running on cores across contiguous nodes (TC2) does not apply for 16 and 32 core runs.

Figure 4.4 shows the baseline and sensitivity runtimes for 16-core allocations (TC1 and TC3) of EP Class C problem. TC3 baseline runtime was slightly higher than TC1. Under network loaded condition, TC1 remained insensitive. Again, this is somewhat intuitive as running on contiguous cores was not affected by the load on other cores since the network traffic of the AUT never left the nodes.TC3, however, showed sensitivity since the allocated cores were widespread. For EP Class C problem size, TC3 showed about 4% higher runtime than its baseline.

Classes A and B showed equivalent behavior for TC1 (i.e. insensitive). TC3 showed about 43% and 4% higher runtimes compared to its respective baselines.

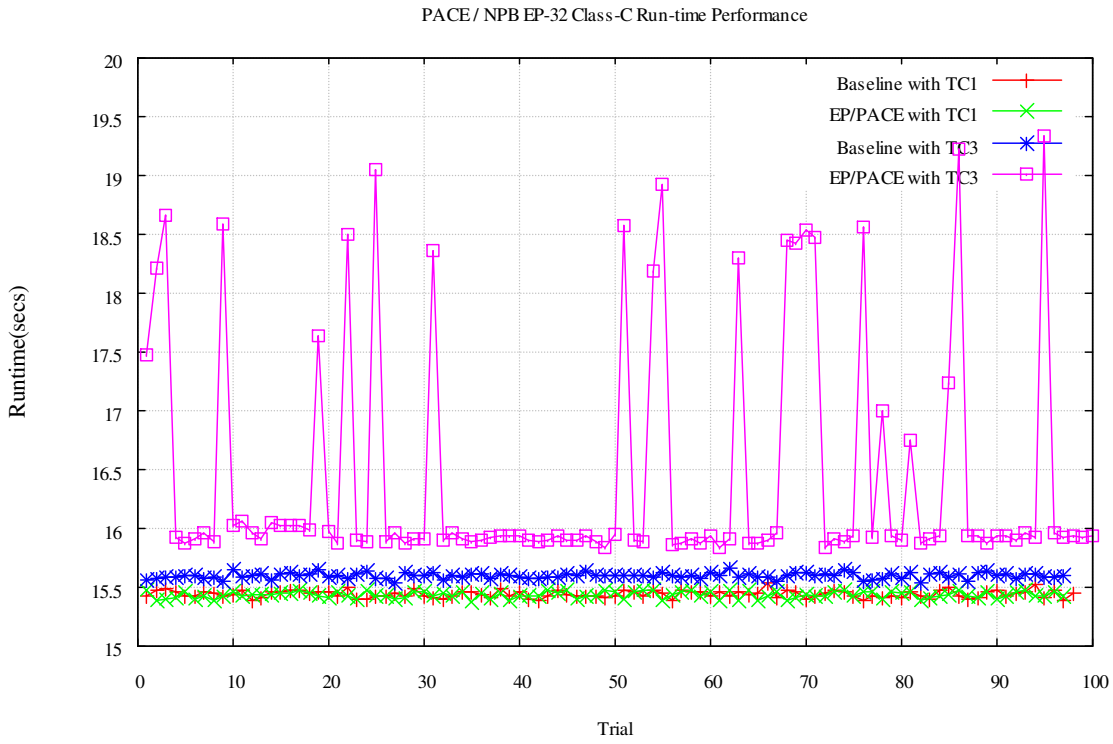

Figure 4.5 32-core allocations: Baseline and Sensitivity runtimes of EP Class C

Figure 4.5 shows the baseline and sensitivity runtimes for 32-core allocations (TC1 and TC3) of EP Class C problem. As seen with 16-core allocation, TC3 baseline runtime was slightly higher than TC1. Also, under network loaded condition, TC1 remained insensitive. TC3, however, showed sensitivity and was about 2% higher runtime than its baseline.

Classes A and B showed equivalent behavior for TC1 (i.e. insensitive). TC3 showed about 42% and 17% higher runtimes compared to its respective baselines.

Summary of EP Benchmark runs:

Based on the data obtained for EP benchmark runs, the following observations were made:

- In contrast to the general idea of how EP benchmark works, in MCMP system, they do not appear to be truly EP since the runtimes at network loaded conditions

were not truly insensitive for certain cases (EP 8-core, 16-core and 32-core runs). This shows that for these cases when PACE is loading the network, EP could be competing with the entire system thereby making it sensitive to network load.

- For 8-core, 16-core, and 32-core runs, TC1 (contiguous core allocation) remained insensitive. This is due to the fact that the cores are all from adjacent cores/nodes and are not influenced by network communication load on other cores in the system. This not the case with 2-core and 4-core runs, where TC1 showed sensitivity, since the other cores in the nodes were loaded with PACE.
- As the problem size increases, TC2 and TC3 appear to be less sensitive. This is evident when looking at the sensitivity runtimes of EP Class A, B, and C problems. This behavior is somewhat intuitive, as the cores are apart, communication load becomes more dominant with smaller size problems.

A statistical summary of the runtime data collected for EP benchmark runs, both for baseline and under network loaded condition (EP/PACE) are shown in Table 4.1. For each problem size (Class A, B, and C), mean, standard deviation, and coefficient of variation (COV) of the runtime are shown for different core allocations, TC1, TC2 and TC3.

Based on the statistical data, sensitivity factors were computed for TC1, TC2 and TC3 using the previously described methodology (Chapter 3). An average of these quantities was taken to arrive at a single sensitivity factor for core allocation, S_{alloc}. Allocation sensitivity factors thus obtained for EP benchmark are shown in Table 4.2.

The allocation sensitivity factors computed for EP shows, in general, an increasing trend with scaling. This might sound somewhat counter-intuitive as one would expect a natural reduction in systemic loading with scaling since contribution from PACE gets reduced. Inspecting further, it is interesting to see how the individual sensitivity factors (S_{TC1}, S_{TC2} and S_{TC3}) contribute to this trend. S_{TC1}, for example, shows a downward trend and in-line with intuition. However, S_{TC2} and S_{TC3} show an upward trend with scaling. Also, when looking at this in a generic sense, one would expect any

Table 4.1 *Statistical Data of EP Benchmark Runs*

Test Case	Class A			Class B			Class C		
	x_{mean} (sec)	σ (sec)	COV	x_{mean} (sec)	σ (sec)	COV	x_{mean} (sec)	σ (sec)	COV
EP2									
Baseline-TC1	15.456	0.046	0.300	61.959	0.180	0.290	247.140	0.797	0.320
EP/PACE-TC1	16.206	0.062	0.390	64.826	0.279	0.430	259.200	1.130	0.440
Baseline-TC2	15.542	0.059	0.380	62.117	0.205	0.330	248.180	0.894	0.360
EP/PACE-TC2	16.196	0.066	0.410	64.756	0.254	0.390	258.920	1.060	0.410
Baseline-TC3	15.457	0.046	0.290	61.875	0.203	0.330	247.380	0.728	0.290
EP/PACE-TC3	16.146	0.081	0.500	64.622	0.291	0.450	258.530	1.220	0.470
EP4									
Baseline-TC1	7.722	0.026	0.340	30.905	0.086	0.280	123.610	0.367	0.300
EP/PACE-TC1	7.971	0.031	0.390	31.887	0.131	0.410	127.500	0.495	0.390
Baseline-TC2	7.752	0.021	0.270	31.014	0.080	0.260	124.080	0.341	0.270
EP/PACE-TC2	8.318	0.045	0.540	33.332	0.101	0.300	133.200	0.380	0.290
Baseline-TC3	7.751	0.022	0.280	31.006	0.076	0.250	123.990	0.307	0.250
EP/PACE-TC3	8.098	0.060	0.740	32.350	0.158	0.490	129.190	0.530	0.410
EP8									
Baseline-TC1	3.855	0.010	0.250	15.406	0.041	0.260	61.552	0.150	0.240
EP/PACE-TC1	3.853	0.011	0.300	15.402	0.038	0.240	61.576	0.159	0.260
Baseline-TC2	3.887	0.010	0.250	15.549	0.041	0.260	62.160	0.126	0.200
EP/PACE-TC2	4.885	1.241	25.400	16.953	1.039	6.130	65.691	0.931	1.420
Baseline-TC3	3.887	0.011	0.270	15.520	0.035	0.220	62.059	0.134	0.220
EP/PACE-TC3	4.780	1.085	22.690	16.262	0.154	0.950	64.828	0.277	0.430
EP16									
Baseline-TC1	1.933	0.007	0.340	7.717	0.020	0.250	30.853	0.079	0.260
EP/PACE-TC1	1.932	0.007	0.350	7.715	0.019	0.250	30.814	0.062	0.200
Baseline-TC3	1.952	0.005	0.260	7.798	0.020	0.260	31.223	0.077	0.250
EP/PACE-TC3	2.784	1.069	38.410	8.467	0.791	9.340	32.754	0.732	2.230
EP32									
Baseline-TC1	0.970	0.004	0.380	3.865	0.009	0.240	15.441	0.031	0.200
EP/PACE-TC1	0.969	0.004	0.450	3.866	0.011	0.270	15.434	0.028	0.180
Baseline-TC3	0.978	0.004	0.440	3.898	0.006	0.150	15.598	0.024	0.150
EP/PACE-TC3	1.449	0.581	40.100	4.556	0.974	21.380	16.443	1.037	6.300

Table 4.2 *EP Benchmark: Sensitivity Factors*

AUT	S_{TC1}	S_{TC2}	S_{TC3}	$\mathbf{S_{alloc}}$
EPA2	1.363	1.124	1.801	1.429
EPA4	1.184	2.146	2.761	2.030
EPA8	1.199	127.686	103.341	77.409
EPA16	1.029	n/a	210.741	105.885
EPA32	1.184	n/a	135.106	68.145
EPB2	1.551	1.232	1.424	1.403
EPB4	1.511	1.240	2.045	1.599
EPB8	0.923	25.706	4.525	10.384
EPB16	1.000	n/a	39.005	20.002
EPB32	1.125	n/a	166.603	83.864
EPC2	1.442	1.188	1.694	1.441
EPC4	1.341	1.153	1.709	1.401
EPC8	1.084	7.503	2.042	3.543
EPC16	0.768	n/a	9.357	5.063
EPC32	0.900	n/a	44.275	22.587

intermediate allocation possibility to behave different from S_{TC1} and hence the allocation sensitivity factor is a good measure to capture the overall trend of the allocations. This also signifies how the core allocation policies can make a big difference in the way a parallel application behaves in an MCMP system.

In the case of EPA16, it appears that the sensitivity factor (S_{TC3}) might be skewing the S_{alloc} value to a higher number. This might require one to take a deeper look into the individual components of S_{TC3} such as COM and ROV. For cases like these, some possibilities have been discussed as part of future work in Chapter 5.

MG Benchmark runs:

Figure 4.6 shows the baseline and sensitivity runtimes of MG Class C 2-core job for TC1, TC2 and TC3 allocations. TC1 showed a higher baseline runtime when compared to TC2. This is somewhat counter-intuitive since one might think that contiguous core allocation strategy (TC1) would have the least baseline runtime. In this

case, TC1 baseline runtime was the highest (about 16% higher than TC2), TC2 baseline runtime was the lowest, and TC3 baseline runtime was in-between TC1 and TC2.

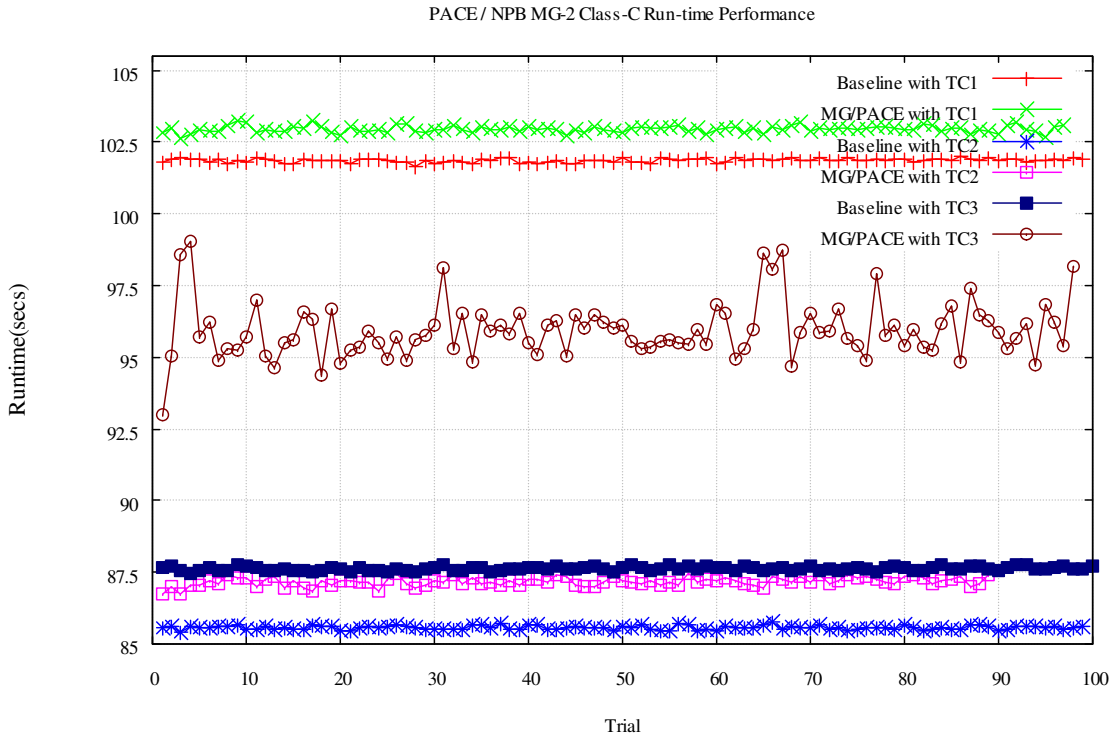

Figure 4.6 2-core allocations: Baseline and Sensitivity runtimes of MG Class C

Under sensitivity tests, TC1 and TC2 were sensitive to a similar extent when compared with their baselines. TC3 runtime was much higher than TC1 and TC2 and remained the most sensitive (about 10% higher runtime than its baseline). Class A and Class B problem sizes exhibited more or less similar behavior as Class C.

Figure 4.7 shows the baseline and sensitivity runtimes of MG Class C run on 4 cores using TC1, TC2 and TC3 allocation policies. As observed with the 2 core runs, TC1 allocation resulted in a much higher baseline runtime (about 76% more) compared to TC2 or TC3 allocations. TC2 and TC3 baseline runtime were about the same, indicating that as the scale (number of cores) increases, the TC2 and TC3 allocations performed very similar to each other in the absence of additional network communication

load. Class A and Class B MG 4 core runs also showed that the baseline runtimes of TC2 and TC3 were very similar.

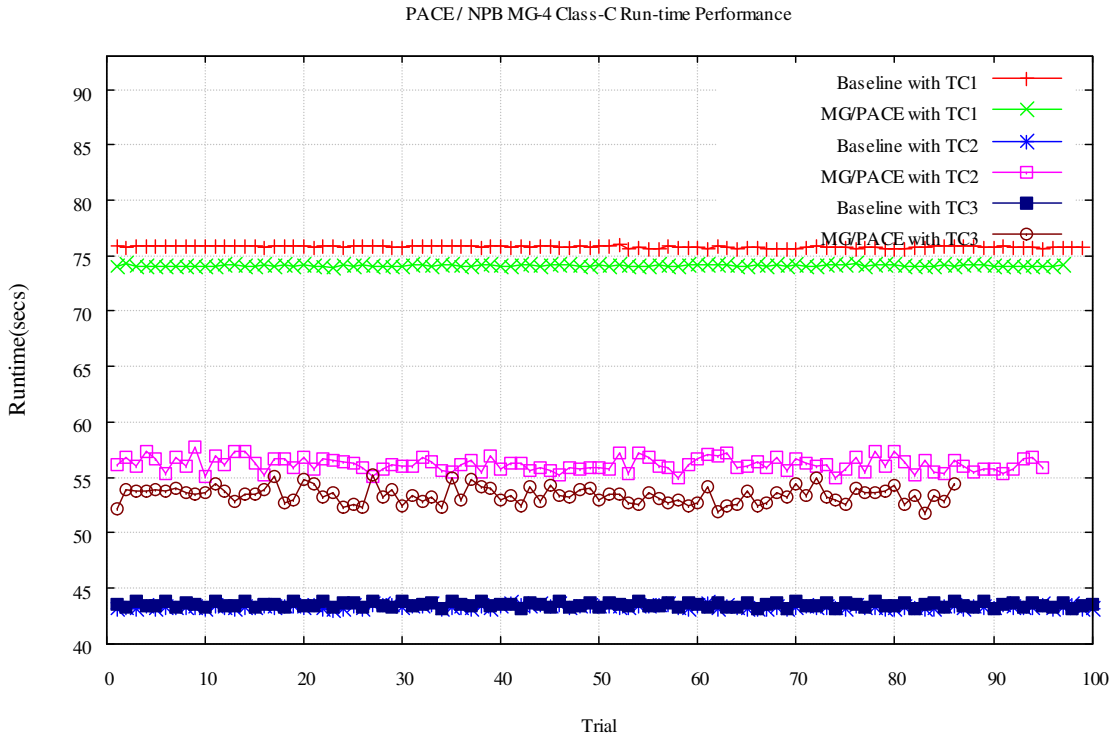

Figure 4.7 4-core allocations: Baseline and Sensitivity runtimes of MG Class C

For sensitivity tests, TC1 remained almost insensitive whereas TC2 runtime was about 30% higher than its baseline and TC3 runtime was about 23% higher than its baseline.

For Class A and Class B problems of MG 4-core runs, TC1 remained insensitive also. TC2 was 60% (Class A) and 52% (Class B) higher than its respective baselines while TC3 was 82% (Class A) and 66% (Class B) higher than its respective baselines. This shows that with increasing problem size, the sensitivity of MG 4-core runs showed decreasing sensitivity to network communication load for TC2 and TC3 allocations.

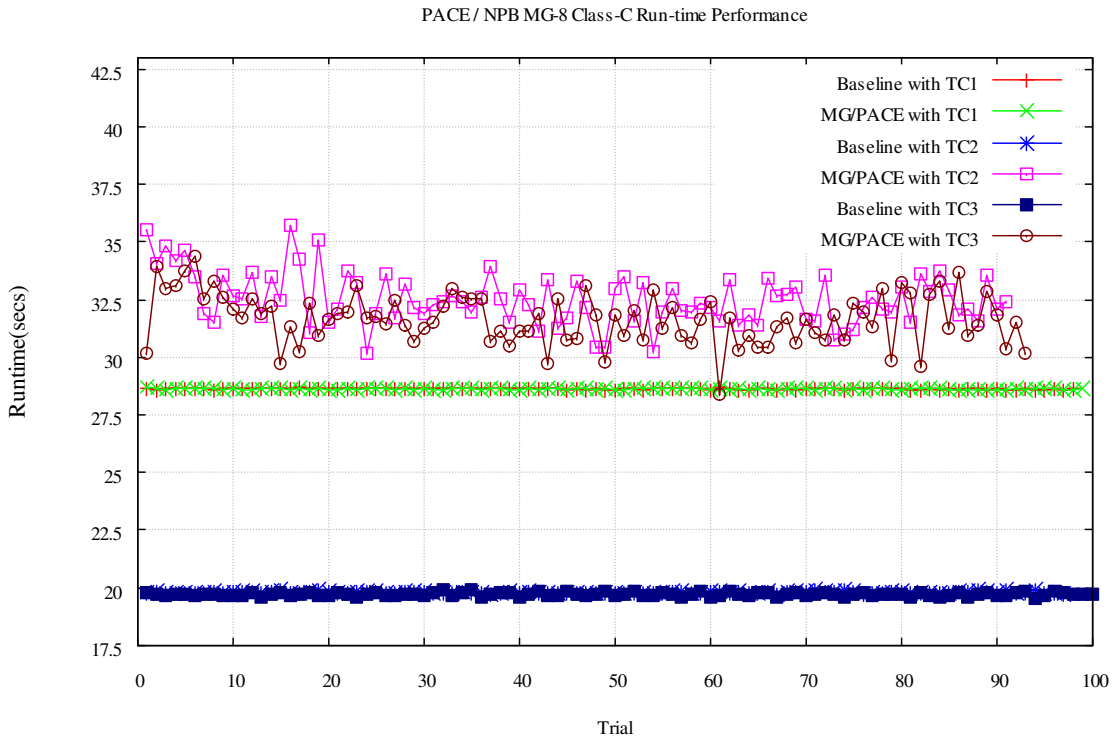

Figure 4.8 8-core allocations: Baseline and Sensitivity runtimes of MG Class C

MG Class C 8-core job baseline and sensitivity runtimes are shown in Figure 4.8 above. Again, as observed with the 2 and 4 core runs, TC1 allocation resulted in a much higher baseline runtime (about 45% more) compared to TC2 or TC3 allocations. TC2 and TC3 baseline runtime remained about the same, indicating that as the scale (number of cores) increases, the TC2 and TC3 allocations performed very similar to each other in the absence of additional network communication load.

Interestingly, in Class A and Class B MG 8-core runs, TC1 showed no appreciable difference in its baseline runtime when compared to TC2 and TC3 (all the three test cases showed almost the same runtimes). This shows that MG 8-core TC1 runs exhibit some size dependency.

For sensitivity tests, Class C MG 8-core TC1 remained almost insensitive while TC2 and TC3 were almost equally sensitive (about 60% higher than their baseline runtimes).

For Class A and Class B problems of MG 8-core runs, TC1 remained insensitive also. TC2 was 220% (Class A) and 190% (Class B) higher than its respective baselines while TC3 was 160% (Class A) and 145% (Class B) higher than its respective baselines. This shows that with increasing problem size, the sensitivity of MG 8-core runs showed decreasing sensitivity to network communication load for TC2 and TC3 allocations.

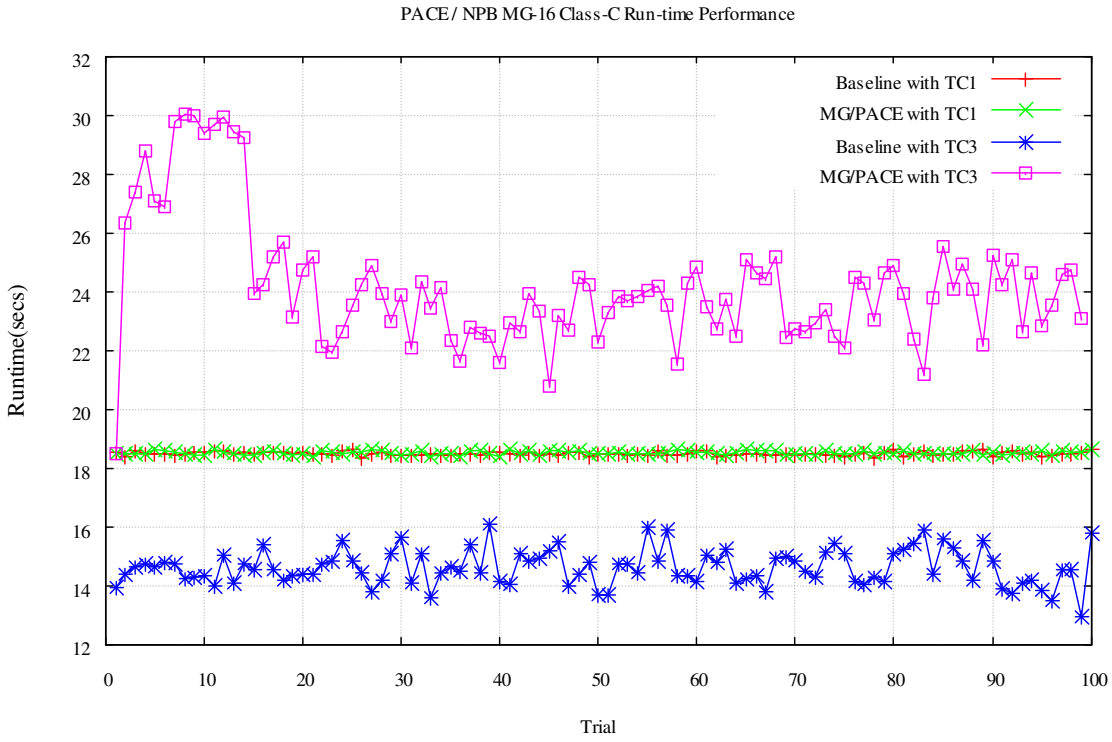

Figure 4.9 16-core allocations: Baseline and Sensitivity runtimes of MG Class C

Figure 4.9 shows the baseline and sensitivity runtimes for MG Class C problem run on 16 cores using TC1 and TC3 core allocation policies. Here TC2 does not apply since there were fewer nodes in the cluster (10 nodes) to allocate 16 cores under TC2 (core allocation on contiguous nodes).

The baseline runtime of TC1 was higher (about 26%) than TC3 baseline runtime. Class A and B of MG 16-core run showed about 27% (Class A) and 15% (Class B) higher TC1 compared to TC3.

For sensitivity tests, TC1 remained insensitive as the traffic never left the nodes of the contiguous cores allocated. However, TC3 showed a 62% higher runtime than its baseline. TC1 was also insensitive for Class A and Class B problem sizes of MG run on 16-cores. TC3 was about 400% (Class A) and 237% (Class B) than its respective baselines.

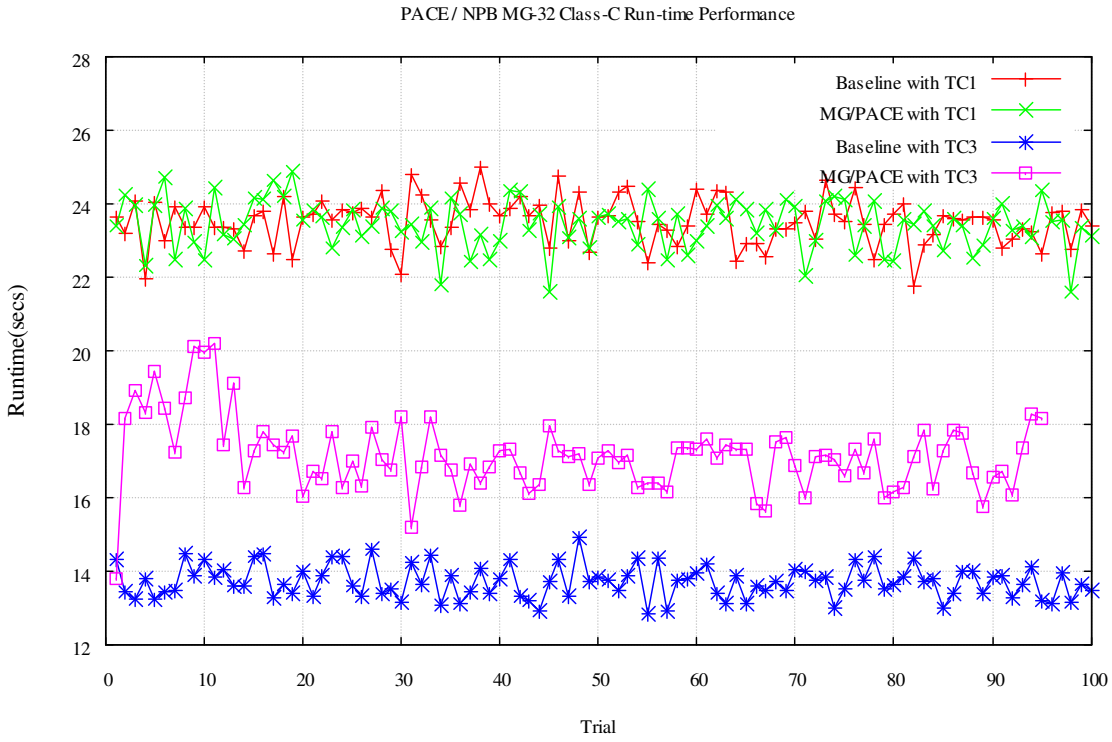

Figure 4.10 32-core allocations: Baseline and Sensitivity runtimes of MG Class C

The baseline and sensitivity runtimes of TC1 and TC3 are shown in Figure 4.10 for MG Class 32-core run. TC1 baseline was much higher than TC3 baseline (about 72% higher). Even with MG Class A and Class B problems, run on 32 cores, TC1 baseline runtime was higher than TC3 baseline.

For sensitivity tests, runtime of TC1 did not change much and remained insensitive. TC3, however, was sensitive and its runtime was about 25% higher than its baseline. TC1 remained insensitive for MG Class A and B also, indicating that this behavior is irrespective of problem size. TC3 was 278% (Class A) and 214% (Class B) higher than its respective baselines.

Summary of MG Benchmark runs:

Based on the data obtained for MG benchmark runs, the following observations were made:

- In contrast to the general intuition that TC1 allocation would yield the best (lowest) runtime, the results of MG benchmark runs show that TC1 baseline runtime, in fact, was not the lowest. It turned out that TC2 or TC3 yielded much lower runtimes than TC1. This gives a unique perspective of how even baseline runs (without additional network communication load), in some cases, might be different in MCMP systems.
- For 8-core, 16-core, and 32-core runs, TC1 (contiguous core allocation) remained insensitive. This again, (as seen with EP runs) due to the fact that the allocated cores are all from adjacent cores/nodes and are not influenced by network communication load on other cores in the system.
- 4-core TC1 runs, even though the other cores in the nodes are loaded with PACE, showed insensitivity. In fact, if looked closely at the data, there was some slight improvement in runtime (lower than baseline) under loaded condition. This again is counter-intuitive and seems to be problem dependent (since EP 4-core run did not show this behavior) and needs further investigation in future studies.
- 2-core TC1 runs, showed sensitivity for network communication load since the other cores in the nodes were loaded with PACE.

A statistical summary of the runtime data collected for MG benchmark runs, both for baseline and under network loaded condition (MG/PACE) are shown in Table 4.3. For each problem size (Class A, B, and C), mean, standard deviation, and coefficient of

variation (COV) of the runtime are shown for different core allocations, TC1, TC2 and TC3.

Table 4.3 *Statistical Data of MG Benchmark Runs*

Test Case	Class A			Class B			Class C		
	x_{mean} (sec)	σ (sec)	COV	x_{mean} (sec)	σ (sec)	COV	x_{mean} (sec)	σ (sec)	COV
MG2									
Baseline-TC1	2.185	0.005	0.240	10.090	0.009	0.090	101.860	0.069	0.070
MG/PACE-TC1	2.235	0.020	0.890	10.338	0.038	0.370	102.950	0.116	0.110
Baseline-TC2	1.879	0.003	0.140	8.614	0.006	0.070	85.568	0.067	0.080
MG/PACE-TC2	1.936	0.021	1.070	8.907	0.037	0.420	87.171	0.153	0.180
Baseline-TC3	2.068	0.008	0.370	9.529	0.033	0.340	87.622	0.069	0.080
MG/PACE-TC3	2.645	0.293	11.060	11.724	0.584	4.980	95.932	0.994	1.040
MG4									
Baseline-TC1	1.540	0.004	0.230	7.200	0.016	0.220	75.806	0.088	0.120
MG/PACE-TC1	1.519	0.009	0.570	7.100	0.022	0.300	74.099	0.058	0.080
Baseline-TC2	1.171	0.008	0.650	5.415	0.017	0.320	43.339	0.134	0.310
MG/PACE-TC2	1.880	0.297	15.780	8.232	0.439	5.330	56.170	0.630	1.120
Baseline-TC3	1.192	0.007	0.620	5.514	0.030	0.550	43.517	0.201	0.460
MG/PACE-TC3	2.159	0.505	23.390	9.128	1.211	13.270	53.403	0.766	1.440
MG8									
Baseline-TC1	0.672	0.010	1.420	3.104	0.007	0.210	28.634	0.026	0.090
MG/PACE-TC1	0.672	0.009	1.300	3.107	0.007	0.230	28.624	0.026	0.090
Baseline-TC2	0.694	0.005	0.760	3.225	0.010	0.300	19.821	0.040	0.200
MG/PACE-TC2	2.210	0.712	32.210	8.362	2.304	27.550	32.521	1.124	3.460
Baseline-TC3	0.690	0.002	0.260	3.205	0.009	0.280	19.718	0.079	0.400
MG/PACE-TC3	2.018	0.709	35.160	8.127	2.322	28.570	31.660	1.103	3.480
MG16									
Baseline-TC1	0.552	0.010	1.890	2.593	0.026	0.990	18.497	0.064	0.350
MG/PACE-TC1	0.554	0.012	2.170	2.590	0.029	1.120	18.523	0.058	0.310
Baseline-TC3	0.465	0.075	16.090	2.264	0.229	10.110	14.628	0.602	4.110
MG/PACE-TC3	2.213	0.729	32.960	7.808	2.095	26.840	24.187	2.169	8.970
MG32									
Baseline-TC1	0.750	0.260	34.700	2.710	0.695	25.640	23.511	0.636	2.710
MG/PACE-TC1	0.759	0.269	35.380	2.735	0.696	25.460	23.435	0.668	2.850
Baseline-TC3	0.283	0.011	3.800	1.321	0.003	0.210	13.713	0.444	3.230
MG/PACE-TC3	1.061	0.426	40.170	4.146	1.283	30.940	17.149	1.008	5.880

Table 4.4 *MG Benchmark: Sensitivity Factors*

AUT	S_{TC1}	S_{TC2}	S_{TC3}	$\mathbf{S_{alloc}}$
MGA2	3.793	7.873	38.234	16.633
MGA4	2.445	38.961	68.342	36.583
MGA8	0.915	134.980	395.385	177.093
MGA16	1.152	n/a	9.746	5.449
MGA32	1.032	n/a	39.691	20.361
MGB2	4.212	6.204	18.020	9.479
MGB4	1.345	25.321	39.941	22.202
MGB8	1.096	238.104	258.775	165.992
MGB16	1.130	n/a	9.157	5.143
MGB32	1.002	n/a	462.480	231.741
MGC2	1.588	2.292	14.233	6.038
MGC4	0.652	4.683	3.842	3.059
MGC8	1.000	28.385	13.969	14.451
MGC16	0.887	n/a	3.609	2.248
MGC32	1.048	n/a	2.277	1.662

Based on the statistical data, sensitivity factors were computed for TC1, TC2 and TC3. An average of these quantities was taken to arrive at a single sensitivity factor for core allocation, S_{alloc}. Allocation sensitivity factors thus obtained for MG benchmark are shown in Table 4.4.

The allocation sensitivity factors computed for MG, in general, show an increasing trend with scaling up to 8-cores, then drop for 16-cores and then show an increase for 32-cores. With this varying trend, it is not possible to generalize the allocation sensitivity factor, S_{alloc}, as the individual sensitivity factors (S_{TC1}, S_{TC2} and S_{TC3}) contribute differently to affect the trend.

LU Benchmark runs:

Figure 4.11 above shows the baseline and sensitivity runtimes of LU Class C 2-core run. As seen with MG, the TC1 baseline runtime was not the lowest and remained higher (about 17%) than TC2 and TC3 baseline runtimes. In this case, TC2 baseline runtime was the lowest and TC3 runtime was slightly higher than TC2. In Class A and

Class B problems for 2-core runs, TC1 was 4% (Class A) and 28% (Class B) higher than its baseline.

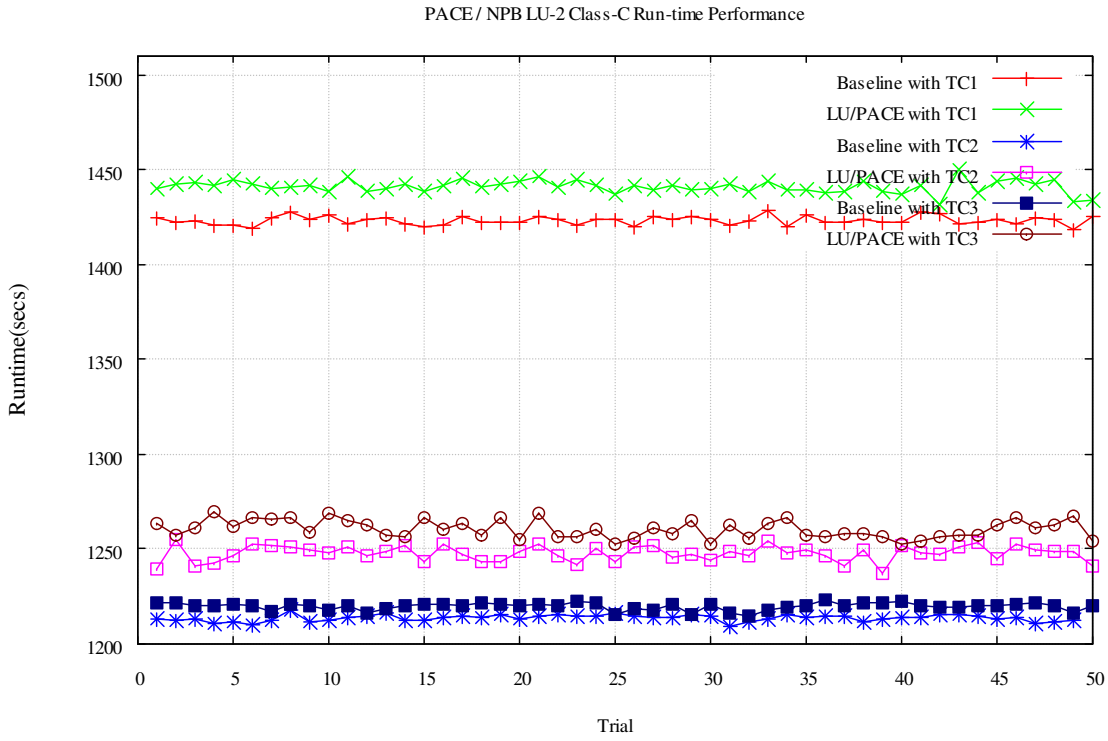

Figure 4.11 2-core allocations: Baseline and Sensitivity runtimes of LU Class C

Under loaded condition, for LU Class C 2-core run (sensitivity test), TC1 runtime was slightly higher compared to its baseline. TC2 and TC3 runtimes were about 3% higher than their respective baselines. In Class A and Class B problems, for 2-core runs, TC1 runtime was 3% higher than its respective baselines. TC2 was 4% (Class A) and 5% (Class B) higher than its baseline while TC3 was 10% (Class A) and 8% (Class B) higher than its baseline. This shows that with increasing problem size, the sensitivity of LU 2-core runs, in general, showed decreasing sensitivity to network communication load for TC2 and TC3 allocations.

Figure 4.12 above shows the baseline and sensitivity runtimes of LU Class C 4-core run. TC2 and TC3 baseline runtimes remained the same. Again in this case, TC1 baseline runtime was not the lowest and remained higher (about 68%) than TC2 and TC3

baseline runtimes. In Class A and Class B problems for 4-core runs, TC1 was 7% (Class A) and 31% (Class B) higher than its respective baselines.

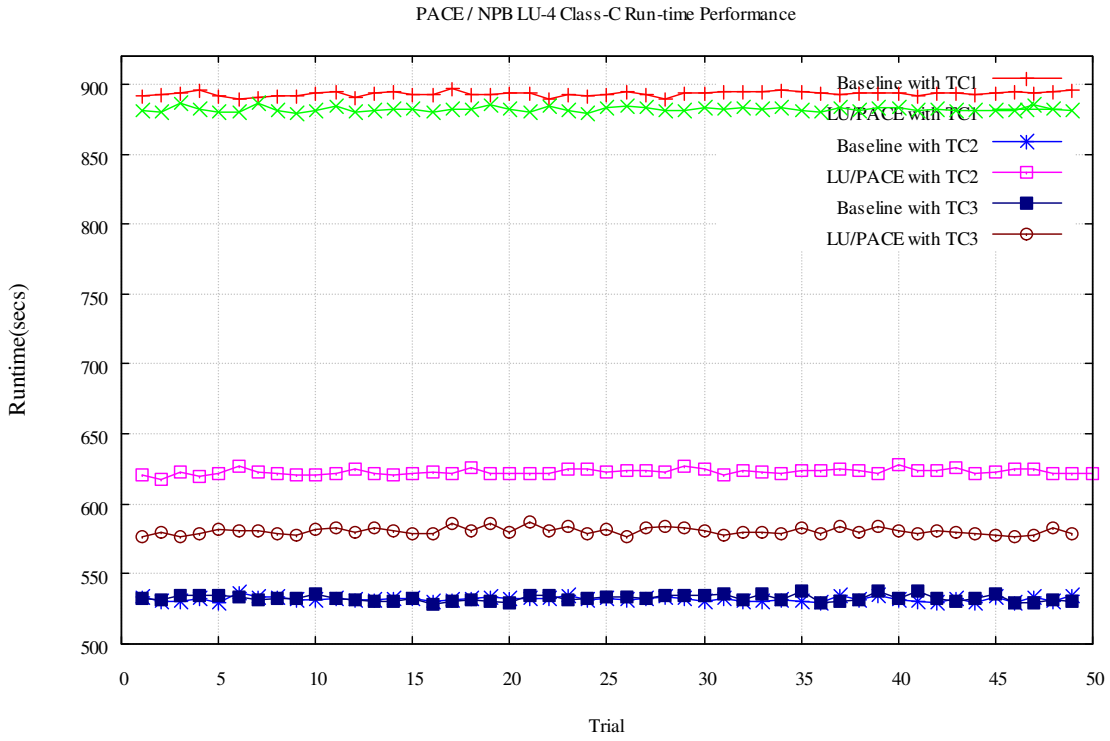

Figure 4.12 4-core allocations: Baseline and Sensitivity runtimes of LU Class C

Under loaded condition, for LU Class C 4-core run (sensitivity test), TC1 runtime was slightly lower compared to its baseline which is somewhat counter-intuitive and the reason for this needs further investigation in future studies. TC2 runtime was about 17% higher than its baseline and TC3 runtime was about 9% higher than its baseline.

In Class A and Class B problems, for 4-core runs, TC1 runtime was almost similar to its respective baselines. TC2 was 36% (Class A) and 20% (Class B) higher than its baseline while TC3 was 26% (Class A) and 12% (Class B) higher than its baseline. This shows that with increasing problem size, the sensitivity of LU 4-core runs, showed decreasing sensitivity to network communication load for TC2 and TC3 allocations.

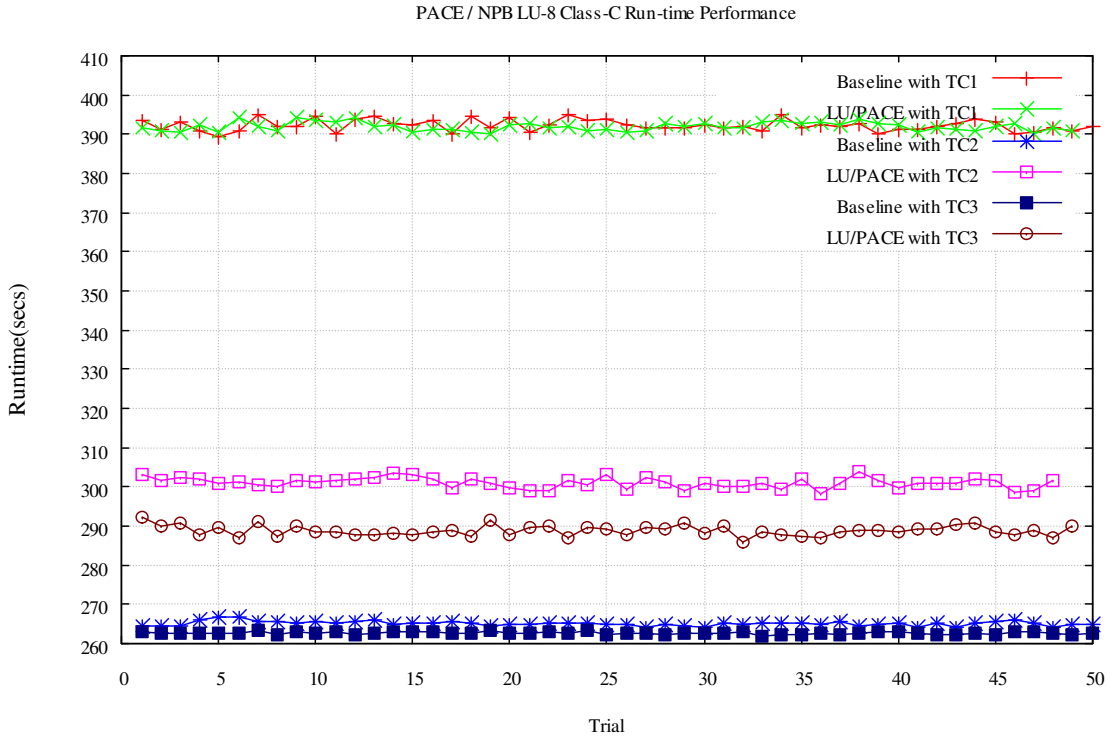

Figure 4.13 8-core allocations: Baseline and Sensitivity runtimes of LU Class C

The baseline and sensitivity runtimes of LU Class C problem run on 8 cores are shown in Figure 4.13 above. TC2 and TC3 baseline runtimes remained very similar. As seen before, TC1 baseline runtime was not the lowest and remained higher (about 48%) than TC2 and TC3 baseline runtimes. However, in Class A and Class B problems TC1 baseline runtimes were different from this observation. In Class A 8-core runs, TC1 baseline runtime was about the same as TC2 or TC3. In Class B 8-core runs, TC1 baseline runtime was lower than TC2 or TC3 baselines.

Under loaded condition, for LU Class C 8-core run (sensitivity test), TC1 runtime was similar to its baseline, suggesting that it was insensitive. TC2 runtime was about 14% higher than its baseline and TC3 runtime was about 10% higher than its baseline.

In Class A and Class B problems, for 8-core runs, TC1 runtimes were almost similar to their respective baselines. TC2 was about 25% (for both Class A and Class B) higher than its baseline while TC3 was about 63% (for both Class A and Class B) higher

than its baseline. This shows that with increasing problem size, mainly from size B to C, the sensitivity of LU 8-core runs, showed decreasing sensitivity to network communication load for TC2 and TC3 allocations.

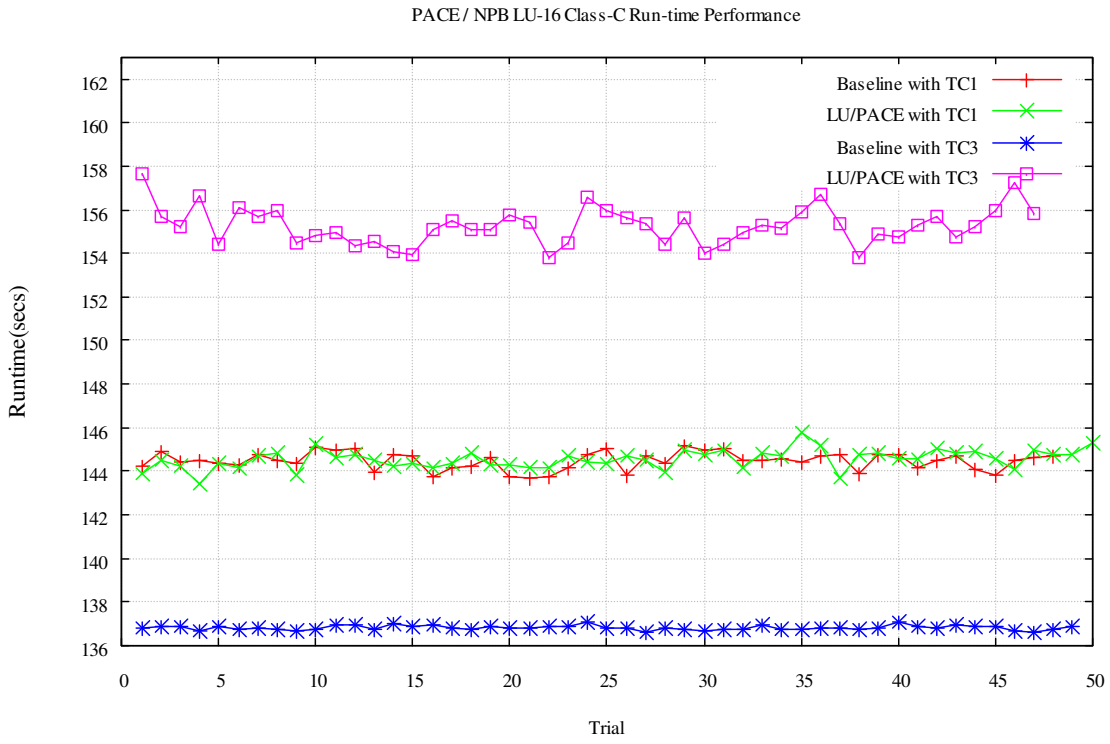

Figure 4.14 16-core allocations: Baseline and Sensitivity runtimes of LU Class C

The baseline and sensitivity runtimes of LU Class C problem run on 16 cores are shown in Figure 4.14 above. TC1 baseline runtime was higher (about 6%) than TC3. In Class A and Class B problems also TC1 baseline runtimes were slightly higher than TC3.

For sensitivity tests, TC1 of LU Class C 16-core run was similar to its baseline, suggesting that it was insensitive. TC3 runtime was about 14% higher than its baseline. In Class A and Class B problems, for 16-core runs, TC1 runtimes were similar to their respective baselines (i.e. remained insensitive). TC3 was about 162% (Class A) and 43% (Class B) higher than its respective baselines.

The baseline and sensitivity runtimes of LU Class C problem run on 32 cores are shown in Figure 4.15 above. TC1 baseline runtime was higher (about 21%) than TC3. In

Class A problem TC1 baseline runtimes was similar to TC3, while in Class B, TC1 baseline runtime was about 52% higher than TC3.

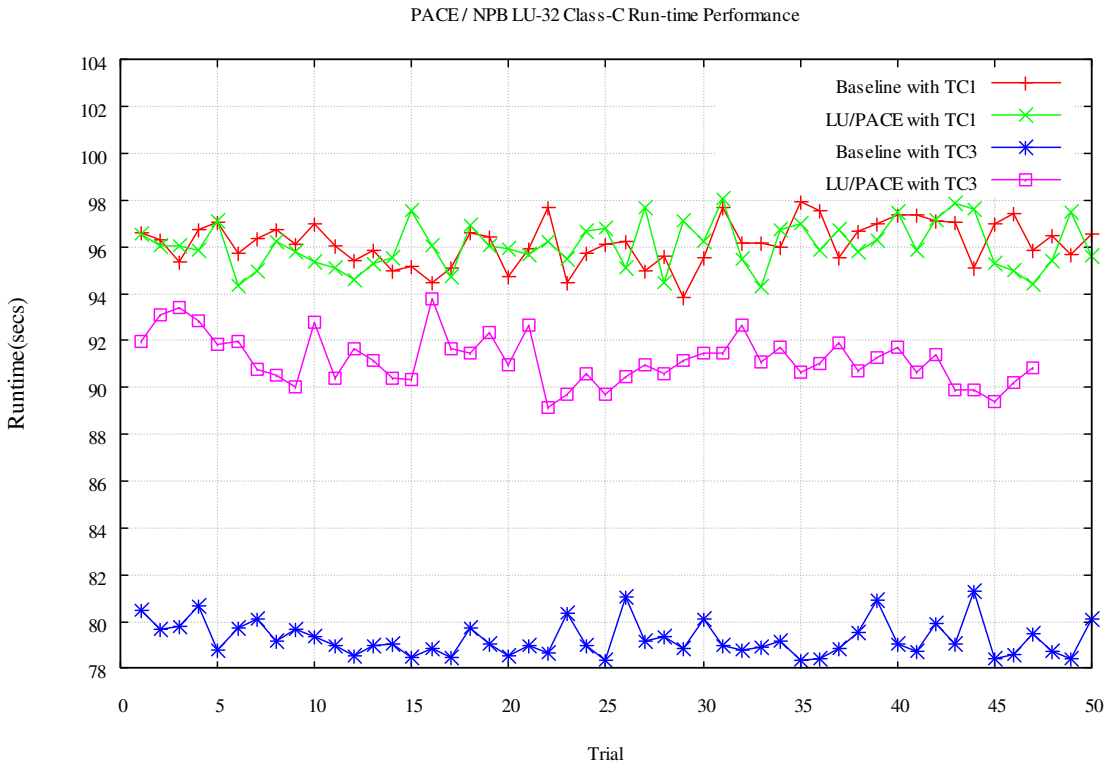

Figure 4.15 32-core allocations: Baseline and Sensitivity runtimes of LU Class C

For sensitivity tests, TC1 of LU Class C 32-core run was similar to its baseline, suggesting that it was insensitive. TC3 runtime was about 15% higher than its baseline.

In Class A and Class B problems, for 16-core runs, TC1 runtimes were similar to their respective baselines (i.e. remained insensitive). TC3 was about 155% (Class A) and 90% (Class B) higher than its respective baselines.

Summary of LU Benchmark runs:

Based on the data obtained for LU benchmark runs, the following observations were made:

- Similar to what was seen with MG benchmark, and in contrast to the general intuition that TC1 allocation would yield the best (lowest) runtime, the results of

LU benchmark runs show that TC1 baseline runtime was not the lowest. It turned out that TC2 or TC3 yielded much lower runtimes than TC1.

- For 8-core, 16-core, and 32-core runs, TC1 (contiguous core allocation) remained insensitive (as seen with EP and MG runs). This is due to the fact that the allocated cores are all from adjacent cores/nodes and are not influenced by network communication load on other cores in the system.
- 4-core TC1 runs, even though the other cores in the nodes were loaded with PACE, showed insensitivity. In fact, for Class C problem, there was some slight improvement in runtime (lower than baseline) under loaded condition.
- 2-core TC1 runs, showed sensitivity for network communication load. This is somewhat intuitive since the other cores in the nodes were loaded with PACE.

A statistical summary of the runtime data collected for LU benchmark runs, both for baseline and under network loaded condition (LU/PACE) are shown in Table 4.5. For each problem size (Class A, B, and C), mean, standard deviation, and coefficient of variation (COV) of the runtime are shown for different core allocations, TC1, TC2 and TC3.

Based on the statistical data, sensitivity factors were computed for TC1, TC2 and TC3. An average of these quantities was taken to arrive at a single sensitivity factor for core allocation, S_{alloc}. Allocation sensitivity factors thus obtained for LU benchmark are shown in Table 4.6.

The allocation sensitivity factors computed for LU Class A and Class B show an increasing trend with scaling. However, Class C shows a mixed trend of decrease and increase. Again, the individual sensitivity factors (S_{TC1}, S_{TC2} and S_{TC3}) contribute differently to affect the trend of the allocation sensitivity factor, $S_{alloc.}$

Table 4.5 *Statistical Data of LU Benchmark Runs*

Test Case	Class A			Class B			Class C		
	x_{mean} (sec)	σ (sec)	COV	x_{mean} (sec)	σ (sec)	COV	x_{mean} (sec)	σ (sec)	COV
LU2									
Baseline-TC1	58.953	0.037	0.060	306.380	1.530	0.500	1423.200	2.290	0.160
LU/PACE-TC1	60.751	0.099	0.160	315.620	0.918	0.290	1441.100	3.460	0.240
Baseline-TC2	56.869	0.039	0.070	238.680	0.379	0.160	1213.600	1.680	0.140
LU/PACE-TC2	58.963	0.097	0.160	250.240	0.587	0.230	1247.700	4.130	0.330
Baseline-TC3	58.842	0.046	0.080	244.560	0.752	0.310	1219.700	2.000	0.160
LU/PACE-TC3	64.613	0.548	0.850	263.020	1.100	0.420	1260.500	4.790	0.380
LU4									
Baseline-TC1	31.638	0.018	0.060	171.320	0.342	0.200	893.150	1.670	0.190
LU/PACE-TC1	31.830	0.036	0.110	173.590	0.380	0.220	881.950	1.710	0.190
Baseline-TC2	29.683	0.047	0.160	130.500	0.074	0.060	531.900	1.640	0.310
LU/PACE-TC2	40.537	0.713	1.760	156.380	0.668	0.430	622.860	2.040	0.330
Baseline-TC3	29.421	0.043	0.140	130.060	0.123	0.090	532.630	2.250	0.420
LU/PACE-TC3	37.002	0.637	1.720	145.710	0.949	0.650	580.480	2.590	0.450
LU8									
Baseline-TC1	15.715	0.033	0.210	65.805	0.044	0.070	392.250	1.480	0.380
LU/PACE-TC1	15.705	0.028	0.180	65.819	0.048	0.070	391.880	1.140	0.290
Baseline-TC2	15.560	0.059	0.380	67.143	0.078	0.120	265.080	0.597	0.230
LU/PACE-TC2	25.758	1.279	4.970	85.254	1.237	1.450	301.010	1.340	0.450
Baseline-TC3	15.222	0.066	0.430	66.104	0.117	0.180	262.620	0.319	0.120
LU/PACE-TC3	24.755	1.220	4.930	81.877	1.125	1.370	288.810	1.350	0.470
LU16									
Baseline-TC1	8.502	0.041	0.480	37.680	0.034	0.090	144.480	0.402	0.280
LU/PACE-TC1	8.497	0.037	0.440	37.693	0.040	0.110	144.550	0.436	0.300
Baseline-TC3	8.161	0.031	0.380	35.475	0.082	0.230	136.810	0.101	0.070
LU/PACE-TC3	21.384	1.635	7.650	50.716	1.196	2.360	155.280	0.865	0.560
LU32									
Baseline-TC1	5.872	0.037	0.630	30.956	1.511	4.880	96.170	0.929	0.970
LU/PACE-TC1	5.863	0.031	0.520	30.636	1.579	5.160	96.054	1.002	1.040
Baseline-TC3	5.832	0.039	0.670	20.425	0.077	0.380	79.267	0.756	0.950
LU/PACE-TC3	15.313	2.495	16.290	38.890	1.631	4.190	91.193	1.060	1.160

Table 4.6 *LU Benchmark: Sensitivity Factors*

AUT	S_{TC1}	S_{TC2}	S_{TC3}	$\mathbf{S_{alloc}}$
LUA2	2.748	2.370	11.667	5.595
LUA4	1.844	15.022	15.451	10.773
LUA8	0.857	21.651	18.645	13.718
LUA16	0.916	n/a	52.748	26.832
LUA32	0.824	n/a	63.842	32.333
LUB2	0.597	1.507	1.457	1.187
LUB4	1.115	8.588	8.091	5.931
LUB8	1.000	15.343	9.427	8.590
LUB16	1.223	n/a	14.669	7.946
LUB32	1.046	n/a	20.995	11.020
LUC2	1.519	2.423	2.454	2.132
LUC4	0.987	1.247	1.168	1.134
LUC8	0.762	2.222	4.307	2.430
LUC16	1.072	n/a	9.080	5.076
LUC32	1.071	n/a	1.405	1.238

PSTSWM Benchmark runs:

Figure 4.16 shows the baseline and sensitivity runtimes of PSTSWM 2-core job for TC1, TC2, and TC3 allocations. TC1 showed a (about 20%) higher baseline runtime when compared to TC2 (as seen with MG and LU runs). However, in this case, TC3 baseline runtime was the highest (about 128% higher than TC2). TC1 was in-between TC2 and TC3.

Under sensitivity test, TC1 and TC2 runtimes were slightly higher when compared to their baselines, suggesting slight sensitivity. TC3 runtime was about 66% higher than its baseline.

Figure 4.17 shows the baseline and sensitivity runtimes of PSTSWM 4-core job for TC1, TC2, and TC3 allocations. In this case, TC1 showed the least baseline runtime. Both TC2 and TC3 baseline runtimes were similar and about 70% higher than TC1 baseline.

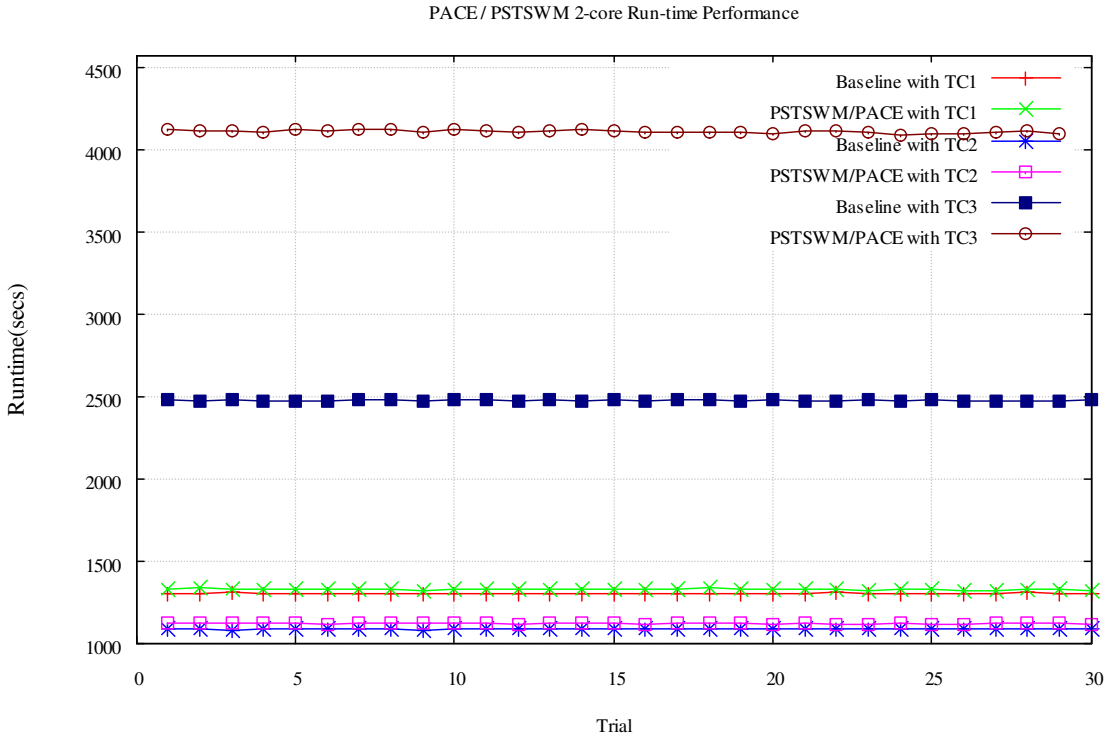

Figure 4.16 2-core allocations: Baseline and Sensitivity runtimes of PSTSWM

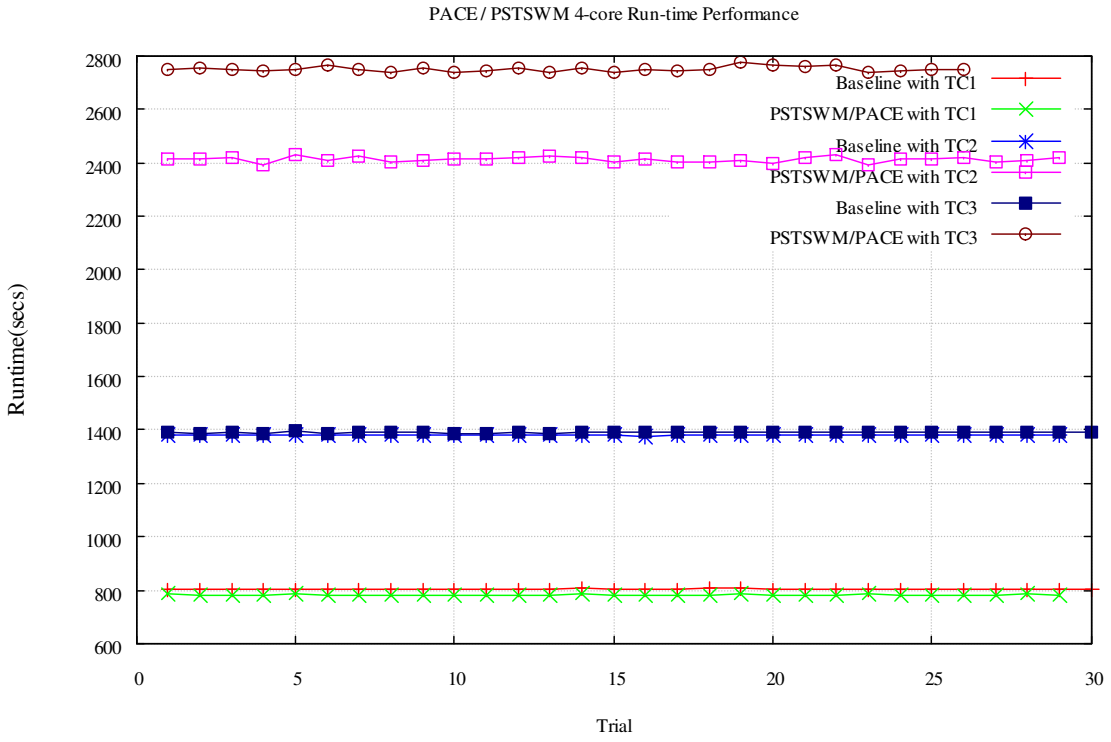

Figure 4.17 4-core allocations: Baseline and Sensitivity runtimes of PSTSWM

Under sensitivity test, TC1 runtime was similar to its baseline, suggesting insensitivity. TC2 runtime was about 75% higher than its baseline while TC3 runtime was 98% higher than its baseline.

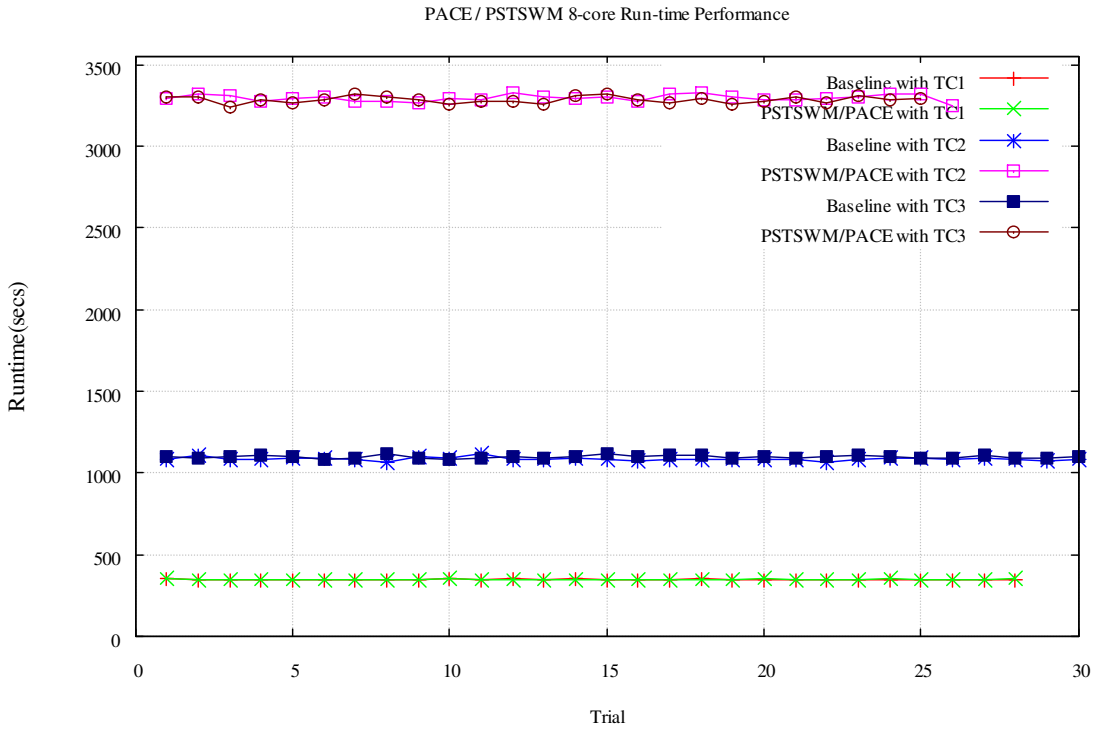

Figure 4.18 8-core allocations: Baseline and Sensitivity runtimes of PSTSWM

Figure 4.18 shows the baseline and sensitivity runtimes of PSTSWM 8-core job for TC1, TC2, and TC3 allocations. Similar to the 4-core run, in this case also the baseline runtime of TC1 was the lowest and baseline runtimes of both TC2 and TC3 baseline were similar and about 210% higher than TC1 baseline.

Under sensitivity test, TC1 runtime was similar to its baseline, suggesting insensitivity. Both TC2 and TC3 runtimes were about 200% higher than their similar baseline runtime

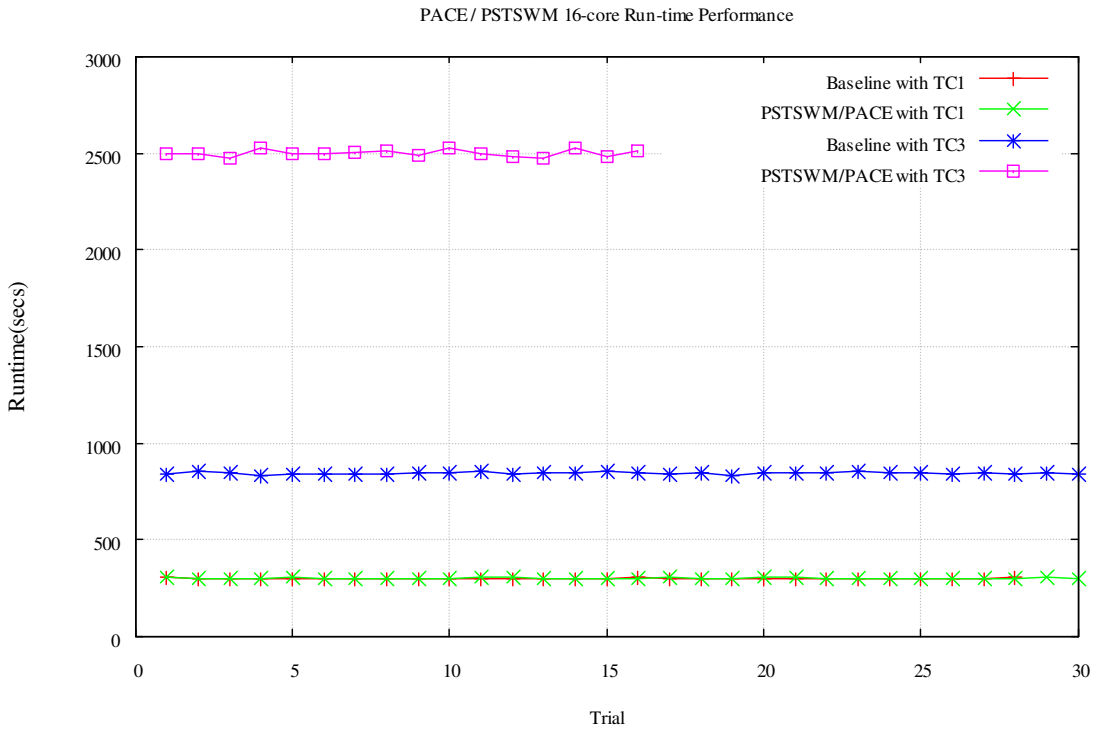

Figure 4.19 16-core allocations: Baseline and Sensitivity runtimes of PSTSWM

Figure 4.19 shows the baseline and sensitivity runtimes of PSTSWM 16-core job for TC1 and TC3 allocations. TC1 baseline runtime was the lowest and TC3 baseline runtime was about 178% higher than TC1.

Under sensitivity test, TC1 runtime was similar to its baseline, suggesting insensitivity. TC2 runtime was about 196% higher than its baseline runtime.

Figure 4.20 shows the baseline and sensitivity runtimes of PSTSWM 16-core job for TC1 and TC3 allocations. TC1 baseline runtime was the lowest and TC3 baseline runtime was about 126% higher than TC1.

Under sensitivity test, TC1 runtime was similar to its baseline, suggesting insensitivity. TC2 runtime was about 121% higher than its baseline runtime.

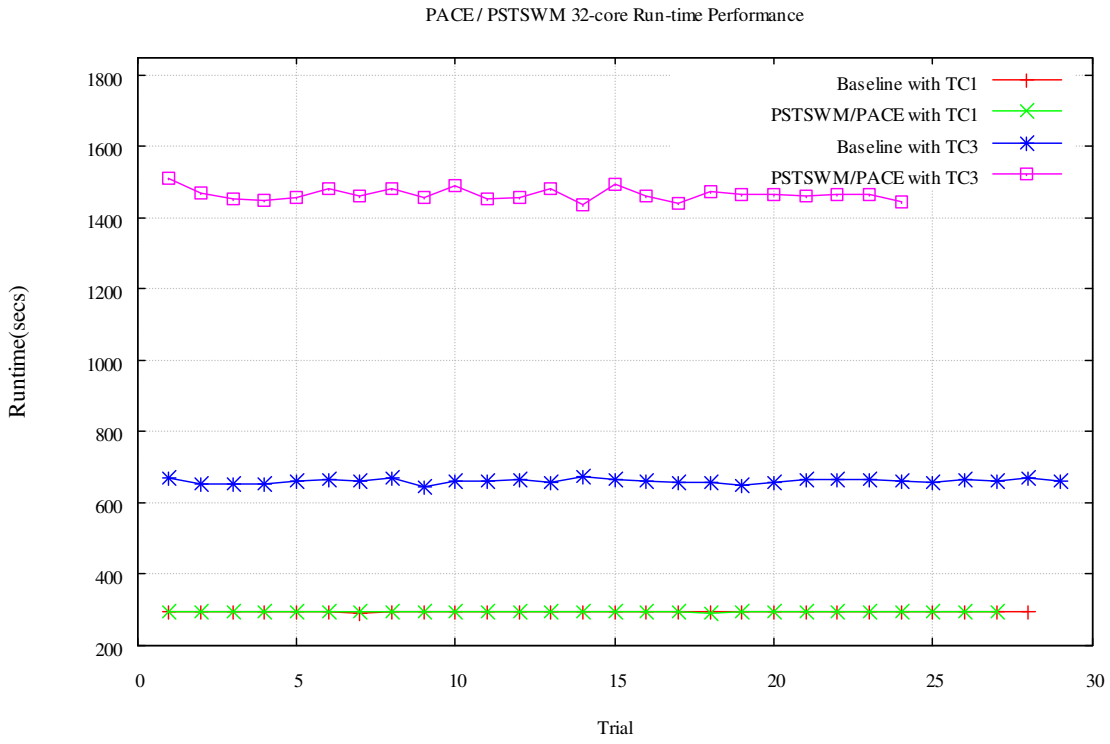

Figure 4.20 32-core allocations: Baseline and Sensitivity runtimes of PSTSWM

Summary of PSTSWM Benchmark runs:

Based on the data obtained for PSTSWM benchmark runs, the following observations were made:

- Unlike what was observed with MG and LU benchmarks, TC1 (contiguous core allocation) yield the best (lowest) runtime for PSTSWM benchmark runs (except for 2-core runs).
- In all the cases TC3 baseline runtime was the highest and remained similar to TC2 (except for the 2-core run, where TC3 was not similar to TC2).
- For 2-core run of PSTSWM, TC1 baseline was higher than TC2, which showed the lowest baseline runtime. TC3 showed the highest baseline runtime.
- Under network loaded condition, TC1 remained insensitive for all the PSTSWM cases irrespective of scaling or number of cores. TC2 and TC3 were sensitive for all cases.

Table 4.7 *Statistical Data of PSTSWM Benchmark Runs*

Test Case	x_{mean} (sec)	σ (sec)	COV
PSTSWM2			
Baseline-TC1	1303.400	2.420	0.190
PSTSWM/PACE-TC1	1327.800	3.170	0.240
Baseline-TC2	1087.500	2.140	0.200
PSTSWM/PACE-TC2	1122.000	3.310	0.300
Baseline-TC3	2475.400	2.460	0.100
PSTSWM/PACE-TC3	4109.200	8.930	0.220
PSTSWM4			
Baseline-TC1	804.640	0.799	0.100
PSTSWM/PACE-TC1	782.060	1.530	0.200
Baseline-TC2	1379.900	1.100	0.080
PSTSWM/PACE-TC2	2412.900	10.100	0.420
Baseline-TC3	1390.800	1.990	0.140
PSTSWM/PACE-TC3	2750.800	9.810	0.360
PSTSWM8			
Baseline-TC1	349.110	0.564	0.160
PSTSWM/PACE-TC1	349.340	1.050	0.300
Baseline-TC2	1084.200	11.300	1.040
PSTSWM/PACE-TC2	3294.800	19.400	0.590
Baseline-TC3	1098.000	9.240	0.840
PSTSWM/PACE-TC3	3282.700	21.100	0.640
PSTSWM16			
Baseline-TC1	302.600	0.452	0.150
PSTSWM/PACE-TC1	303.310	1.760	0.580
Baseline-TC3	843.420	5.600	0.660
PSTSWM/PACE-TC3	2499.000	17.300	0.690
PSTSWM32			
Baseline-TC1	293.300	0.465	0.160
PSTSWM/PACE-TC1	293.050	0.340	0.120
Baseline-TC3	661.710	6.330	0.960
PSTSWM/PACE-TC3	1466.700	17.300	1.180

Based on the statistical data, sensitivity factors were computed for TC1, TC2 and TC3. An average of these quantities was taken to arrive at a single sensitivity factor for core allocation, S_{alloc}. Allocation sensitivity factors thus obtained for PSTSWM benchmark are shown in Table 4.8.

Table 4.8 *PSTSWM Benchmark: Sensitivity Factors*

AUT	S_{TC1}	S_{TC2}	S_{TC3}	$\mathbf{S_{alloc}}$
PSTSWM2	1.287	1.548	3.652	2.162
PSTSWM4	1.944	9.180	5.086	5.403
PSTSWM8	1.876	1.724	2.278	1.959
PSTSWM16	3.876	n/a	3.098	3.487
PSTSWM32	0.749	n/a	2.724	1.737

The allocation sensitivity factors computed for PSTSWM show an increasing trend with scaling up to 4-cores, then drop for 8-cores and then show an increase for 16-cores, and then drops again for 32-cores. With this varying trend, it is not possible to generalize the allocation sensitivity factor, S_{alloc}, as the individual sensitivity factors (S_{TC1}, S_{TC2} and S_{TC3}) contribute differently to affect the trend.

4.3. Overall Summary

From the data analysis performed on the runtime data collected for NAS and PSTSWM benchmarks, under different test scenarios (core allocations), the overall results can be summarized as follows:

- For EP benchmark, the runtimes at network loaded conditions were not truly insensitive for certain cases (EP 8-core, 16-core and 32-core TC2 and TC3 cases). This shows that for these cases when PACE is loading the network, EP could be competing with the entire system thereby making it sensitive to network communication load.
- For EP, MG, and LU benchmarks, TC1 (contiguous core allocation test case) remained insensitive under network loaded conditions for 8-core, 16-core, and 32-core runs. This is due to the fact that the allocated cores are all from adjacent cores/nodes and are not influenced by network communication load on other cores in the system. In the case of PSTSWM, TC1 remained insensitive irrespective of scaling or number of cores.
- In contrast to the general intuition that TC1 allocation would yield the best (lowest) runtime, the results of MG and LU benchmark runs show that TC1 baseline runtime, in fact, was not the lowest. It turned out that TC2 or TC3

yielded much lower runtimes than TC1. This gives a unique perspective of how even baseline runs (without additional network communication load), in some cases, might be non-intuitive in MCMP systems. One possibility why TC1 was not the lowest could be due to contention of resources (such as a common cache) between contiguous cores, which again could be due to the core-level architectural differences. The exact reason why TC1 was not the lowest needs further investigation and beyond the scope of this study.

Table 4.9 *Sensitivity Factors of AUTs grouped under bins*

AUT	Less Sensitive ($S < 5$)	Moderately Sensitive ($5 \leq S \geq 20$)	Highly Sensitive ($S > 20$)
EPA2,4	**x**		
EPA8,16,32			**x**
EPB2,4	**x**		
EPB8		**x**	
EPB16,32			**x**
EPC2,4,8	**x**		
EPC16		**x**	
EPC32			**x**
MGA2,16		**x**	
MGA4,8,32			**x**
MGB2,16		**x**	
MGB4,8,32			**x**
MGC2,8		**x**	
MGC4,16,32	**x**		
LUA2,4,8		**x**	
LUA16,32			**x**
LUB2	**x**		
LUB4,8,16,32		**x**	
LUC2,4,8,32	**x**		
LUC16		**x**	
PSTSWM2,,8,16,32	**x**		
PSTSWM4		**x**	

- Appropriate sensitivity metrics such as coefficients of mean (COM) and ratio of variations (ROV) were determined and based on these metrics, sensitivity factors were computed for each application under different core allocation test cases (TC1, TC2 and TC3).
- From the results, it can be concluded that the sensitivity model used in single-processor machines, still holds on multi-core machines, which showed varying levels of sensitivity for parallel applications depending on the core allocation.
- The parallel applications (AUTs) tested in the current study can be grouped based on the level of sensitivity (less, moderate or high) as shown in Table 4.9. This information can be an important input for schedulers to allocate these applications properly in MCMP systems.
- In general, from single-processor machines results (Evans, 2005), the parallel applications can be ranked in the order of least to most sensitive: EP, LU, MG, and PSTSWM. In MCMP systems, based on the results of this study, the sensitivity is dependent on both the problem type and size. For a given problem type, in general, the sensitivity decreases as the problem size increases.

CHAPTER 5. CONCLUSION AND FUTURE WORK

5.1. Conclusion

The growth of parallel applications has been significant in the recent years and poised to grow even more in the coming years. This offers exciting and promising innovations both in the software and hardware development. At the same time, the growth brings enough challenges for researchers to make sure parallel applications run as intended. It is important to understand how these applications perform, with the existing network conditions and loads, on parallel computers such as a cluster environment.

In this study, the focus was on parallel application performance on MCMP systems. Previously developed sensitivity model for single-processor machines was adopted in this research to study parallel applications on a MCMP system. The runtime sensitivity of parallel applications such as NAS and PSTSWM benchmarks were studied by applying a specific network communication load and by varying the core allocations. Three different core allocation policies were employed to capture the runtime variations. Each test case (core allocation) was designed based on the "distance" between the number cores used for running the applications. Several iterations were run for each test case to gather statistically significant runtime data.

Based on the various tests conducted, the following are the findings of this study:

- Parallel applications showed runtime sensitivity due to network communication load on MCMP system. Therefore, the sensitivity model developed for single-core single-processor machines still holds for MCMP systems.
- The results of the sensitivity tests indicate that the parallel applications run on MCMP systems exhibit varying sensitivity based on core allocation and problem size as discussed in Chapter 4.

- It was found that EP benchmark was not behaving as per definition, since in some cases; it exhibited sensitivity as discussed in Section 4.2.
- It was also found that the contiguous core allocation policy (TC1) was not the best strategy (to get least runtime) for some applications (For example, MG and LU).
- A single allocation sensitivity factor was computed for each parallel application based on several core allocations for a given number of cores.
- The sensitivity factors obtained for the parallel applications tested in this study were categorized based on their values (less sensitive vs. moderately or highly sensitive).

The information obtained from this study can be a useful input for job schedulers to properly allocate parallel applications on MCMP systems. In addition, the information on runtime sensitivity could provide new insights for parallel application programmers, system administrators and hardware architectural designers to optimize application runtime under network communication loaded scenarios and achieve the desired scale-up with MCMP systems.

5.2. Future work

The following topics could be investigated as part of future work to this current study:

- The results of this current study indicate EP benchmark does not truly behave as EP since it showed sensitivity to network communication load in some cases when the core allocations are set to be wide apart. By employing a few more core allocation policies, apart from what was done, and by looking at how EP works at the core level (inter-core communication) could provide useful information.
- In many cases, it was found that the contiguous core allocation strategy (TC1) was not the best way to run a baseline, rather, distributing the job on non-contiguous cores yielded lower runtimes. This could be investigated further to understand what causes this behavior.

- In some cases, it was found that loaded case of TC1 showed improved performance compared to its baseline run. This is an interesting observation and more detailed investigation could be conducted to better under this phenomenon.
- The sensitivity factors computed, in some cases, seem to be slightly skewed thereby producing large values. An alternate method, addressing any anomalies with the individual statistical components, could be developed. One possible method for calculating sensitivity factor that may be used is shown below:

$$S_{ji} = \frac{\left|\frac{x_{mean\ j}+2\sigma_j}{x_{mean\ j}-2\sigma_j}\right|}{\left|\frac{x_{mean\ i}+2\sigma_i}{x_{mean\ i}-2\sigma_i}\right|} \frac{x_{mean\ j}}{x_{mean\ i}} \qquad \text{(Eq.5.1)}$$

From Statistics, $x_{mean} + 2\sigma$ and $x_{mean} - 2\sigma$ represent the upper and lower limits respectively covering 95% of the data spread. The ratio of these two quantities represents the ratio of the data variation in the context of the value of mean. This method, however, needs to be thoroughly tested and confirmed for different possible scenarios.

LIST OF REFERENCES

LIST OF REFERENCES

Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S. T. A., Sites, R. L., et al. (1997). Continuous profiling: where have all the cycles gone? *ACM SIGOPS Operating Systems Review*, *31*(5), 1-14.

Anderson, T. E., Culler, D. E., & Patterson, D. (1995). A case for NOW (Networks of Workstations). *IEEE micro*, *15*(1), 54–64.

Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., & Yarrow, M. (1995a). *The NAS parallel benchmarks 2.0*.

Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., & Yarrow, M. (1995b). *The NAS parallel benchmarks 2.0* (No. Report NAS-95-020).

Berman, F., & Wolski, R. (1996). Scheduling from the perspective of the application. *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing HPDC96* (pp. 100–111).

Bode, B., Halstead, D. M., Kendall, R., Lei, Z., & Jackson, D. (2000). The portable batch scheduler and the maui scheduler on linux clusters. *Proceedings of the 4th Annual Linux Showcase and Conference- Volume 4* (pp. 27–27). Atlanta, Georgia: USENIX Association.

Calzarossa, M., Massari, L., & Tessera, D. (2004). A methodology towards automatic performance analysis of parallel applications. *Parallel Computing*, *30*(2), 211-223. doi:10.1016/j.parco.2003.08.002

Carter, J., He, Y., Shalf, J., Shan, H., Strohmaier, E., & Wasserman, H. (2007). The Performance Effect of Multi-core on Scientific Applications. Lawrence Berkeley National Laboratory. LBNL Paper LBNL-62662.

Chai, L., Gao, Q., & Panda, D. K. (2007). Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)* (pp. 471-478). Presented at the Seventh IEEE International Symposium on Cluster Computing and the Grid, Rio de Janeiro, Brazil. doi:10.1109/CCGRID.2007.119

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, Jonathan, Oliker, L., Patterson, David, et al. (2008). Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 1-12). Presented at the 2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, TX, USA. doi:10.1109/SC.2008.5222004

Dinda, P. A. (2002). A prediction-based real-time scheduling advisor. *Proceedings 16th International Parallel and Distributed Processing Symposium* (pp. 10-17). Presented at the 16th International Parallel and Distributed Processing Symposium. IPDPS 2002, Ft. Lauderdale, FL, USA. doi:10.1109/IPDPS.2002.1015480

Evans, J. J. (2005, December). *Modeling Parallel Application Sensitivity to Network Performance*. Doctoral Thesis, Illinois Institute of Technology, Chicago, Illinois.

Evans, J. J., & Hood, C. S. (2011). A Network Performance Sensitivity Metric for Parallel Applications. *International Journal of High Performance Computing and Networking*, *7, Number 1*, 8-18.

Evans, J. J., & Hood, C. S. (2005). Network performance variability in NOW clusters. *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.* (pp. 1047-1054). Presented at the CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005., Cardiff, Wales, UK. doi:10.1109/CCGRID.2005.1558676

Evans, J. J., & Hood, C. S. (2006). PARSE: A Tool for Parallel Application Run Time Sensitivity Evaluation. *12th International Conference on Parallel and Distributed Systems - (ICPADS'06)* (pp. 475-484). Presented at the 12th International Conference on Parallel and Distributed Systems - (ICPADS'06), Minneapolis, MN, USA. doi:10.1109/ICPADS.2006.78

Feng, W. (2005). The importance of being low power in highperformance computing. *Cyberinfrastructure Technology Watch (CTWatch)*, *3*.

Hall, J., Sabatino, R., Crosby, S., Leslie, I., & Black, R. (1997). Counting the cycles: a comparative study of NFS performance over high speed networks. *22nd Annual Conference on Local Computer Networks (LCN'97)* (pp. 8-19). Minneapolis, MN, USA.

Hennessy, J. L., & Patterson, D. A. (1996). *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1996.

Intel shows off 80-core processor - CNET News. (2007, February). *http://news.cnet.com/Intel-shows-off-80-core-processor/*. Retrieved May 5, 2010, from http://news.cnet.com/Intel-shows-off-80-core-processor/2100-1006_3-6158181.html

Jain, R. (1991). *The Art of Computer Systems Performance Analysis. 1991*. John Wiley, New York.

Jin, H., Hood, R., Chang, J., Djomehri, J., Jespersen, D., Taylor, K., Biswas, R., et al. (2009). *Characterizing Application Performance Sensitivity to Resource Contention in Multicore Architectures* (No. NAS Technical Report NAS-09-002). Citeseer.

Kaiser, H., Brodowicz, M., & Sterling, T. (2009). ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications. *2009 International Conference on Parallel Processing Workshops* (pp. 394-401). Presented at the 2009 International Conference on Parallel Processing Workshops (ICPPW), Vienna, Austria. doi:10.1109/ICPPW.2009.14

Majumdar, S., & Yiu Ming Leung. (1994). Characterization of applications with I/O for processor scheduling in multiprogrammed parallel systems. *Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing* (pp. 298-307). Presented at the 1994 6th IEEE Symposium on Parallel and Distributed Processing, Dallas, TX, USA. doi:10.1109/SPDP.1994.346154

Maui Scheduler - Administrator's Guide. Retrieved from http://www.adaptivecomputing.com/resources/docs/maui/mauiadmin.php

MPICH2 : about MPICH2. Retrieved from http://www.mcs.anl.gov/research/projects/mpich2/about/index.php?s=about

Ni, L. M., & Tail, K. C. (1990). Special issue on software tools for parallel programming and visualization: Guest editors' introduction. *Journal of Parallel and Distributed Computing*, *9*(2), 101–102.

Saini, S., & Bailey, D. H. (1996). *NAS parallel benchmark (version 1.0) results 11-96* (No. Report NAS-96-18).

Singh, J. P., Rothberg, E., & Gupta, A. (1994). Modeling communication in parallel algorithms: A fruitful interaction between theory and systems? *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures* (pp. 189-199). New Jersey, United States.

Sinnen, O., Sousa, L. A., & Sandnes, F. E. (2006). Toward a realistic task scheduling model. *IEEE Transactions on Parallel and Distributed Systems*, *17*(3), 263-275. doi:10.1109/TPDS.2006.40

Sivasubramaniam, A. (1997). Execution-driven simulators for parallel systems design. *Proceedings of the 29th conference on Winter simulation - WSC '97* (pp. 1021-1028). Presented at the the 29th conference, Atlanta, Georgia, United States. doi:10.1145/268437.268735

Smith, M. C., Vetter, J. S., & Xuejun Liang. (2005). Accelerating Scientific Applications with the SRC-6 Reconfigurable Computer: Methodologies and Analysis. *19th IEEE International Parallel and Distributed Processing Symposium* (p. 157b-157b). Presented at the 19th IEEE International Parallel and Distributed Processing Symposium, Denver, CO, USA. doi:10.1109/IPDPS.2005.75

Sun, X. H., & Chen, Y. (2009). Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing, 70*(2), 183-188. doi:10.1016/j.jpdc.2009.05.002.

"TOP500 Supercomputing Sites". http://www.top500.org/.

Veeraraghavan, P. P., & Evans, J. J. (2010). Parallel Application Communication Performance on Multi-Core High Performance Computing Systems. *IASTED International Conference Proceedings*. Presented at the Informatics 2010, Marina del Rey, USA. doi:10.2316/P.2010.724-059

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., & Gupta, A. (1995). The SPLASH-2 programs: Characterization and methodological considerations. *Proceedings of the 22nd annual international symposium on Computer architecture* (pp. 24–36).

Worley, P. H., & Toonen, B. (1995). *A users' guide to PSTSWM* (No. ORNL Technical Report ORNL/TM-12779). ORNL Technical Report ORNL/TM-12779.

APPENDIX

APPENDIX

BASELINE AND SENSITIVITY RUNTIME PLOTS OF NAS CLASS A AND B

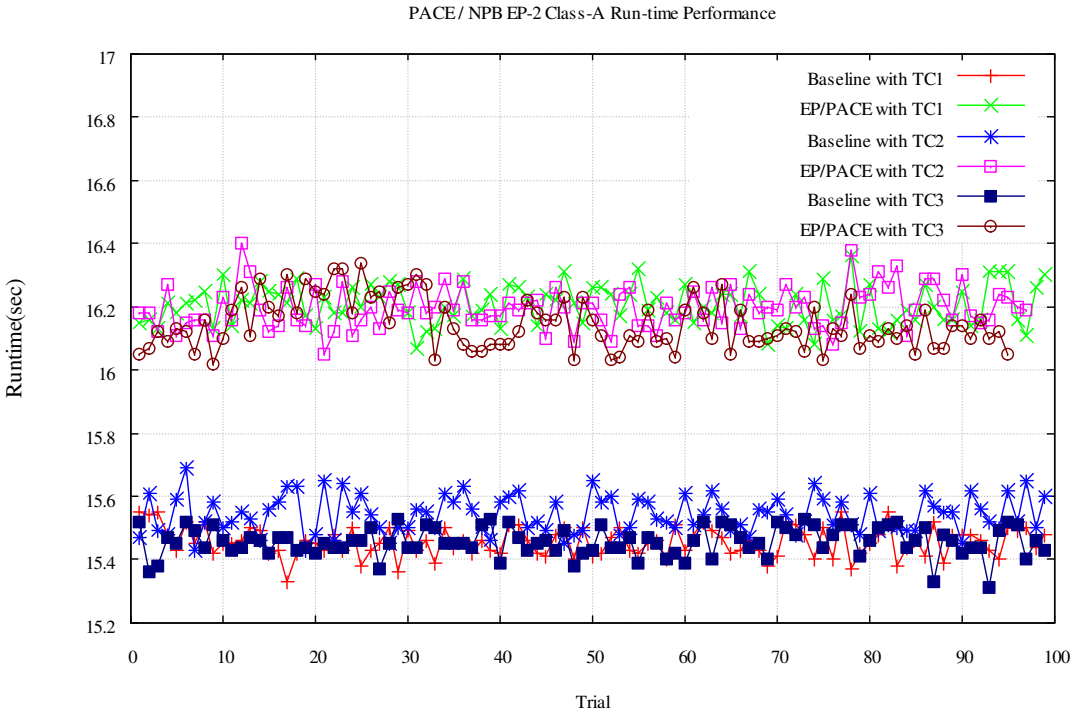

Figure 0.1 Class A EP2 Baseline and Sensitivity runtimes

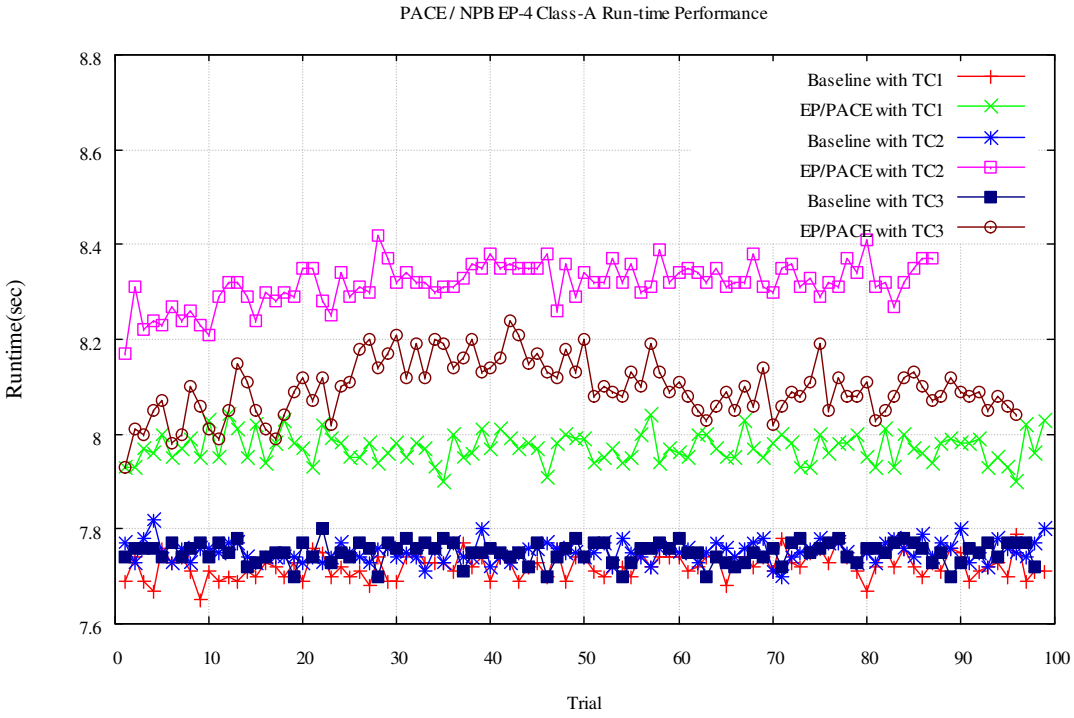

Figure 0.2 Class A EP4 Baseline and Sensitivity runtimes

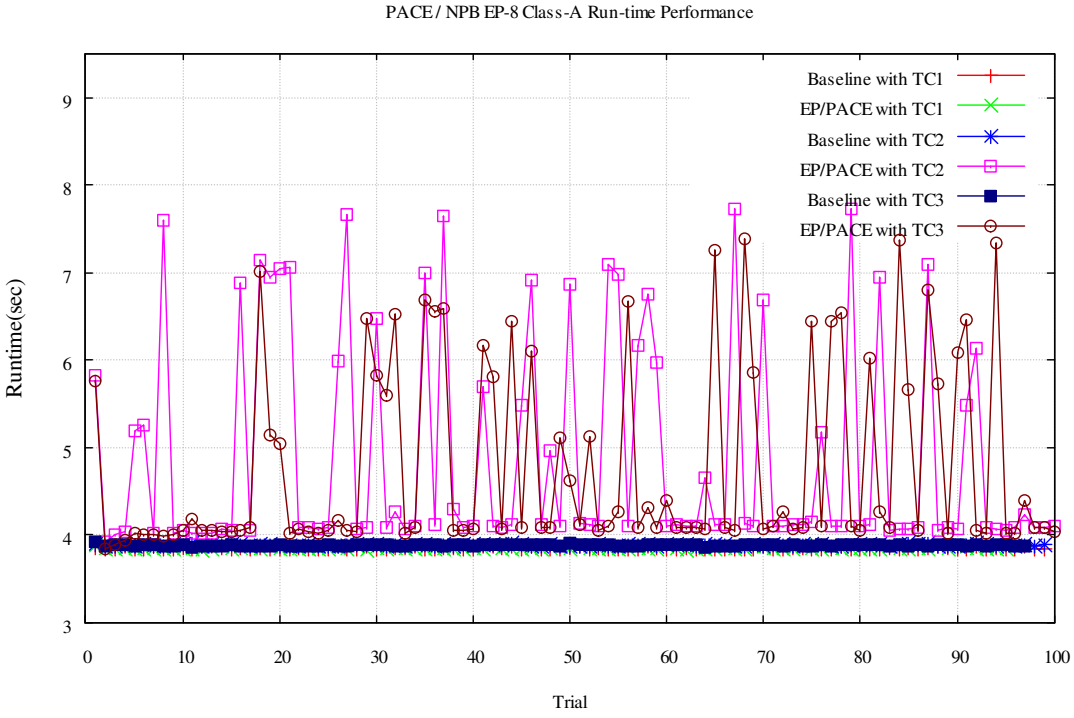

Figure 0.3 Class A EP8 Baseline and Sensitivity runtimes

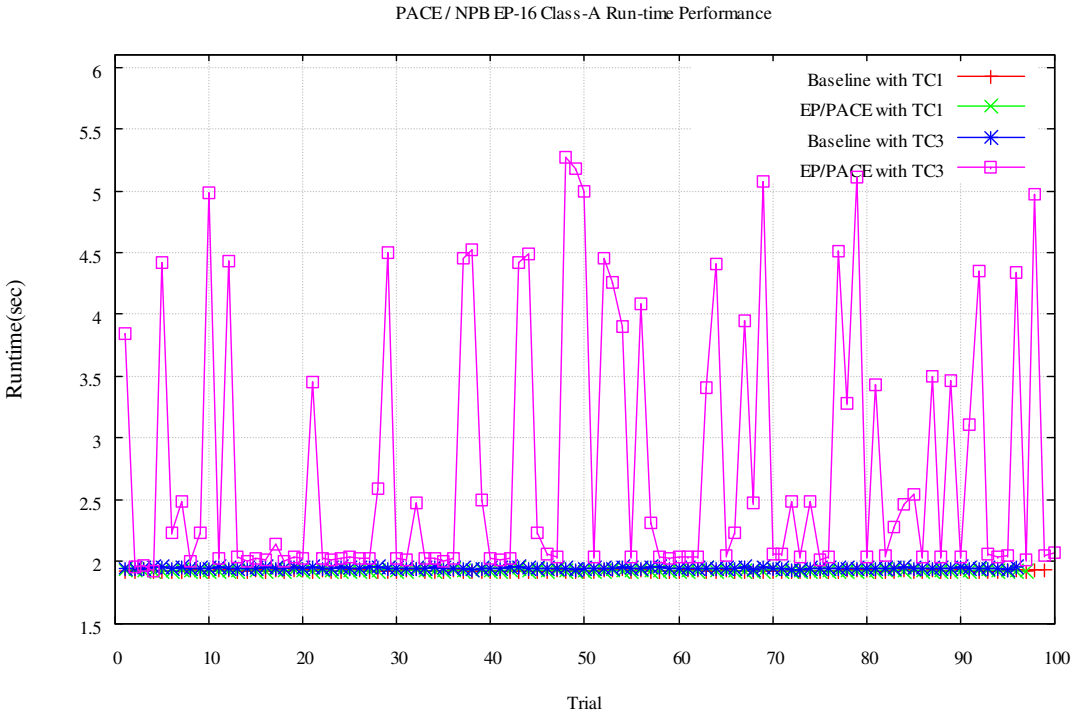

Figure 0.4 Class A EP16 Baseline and Sensitivity runtimes

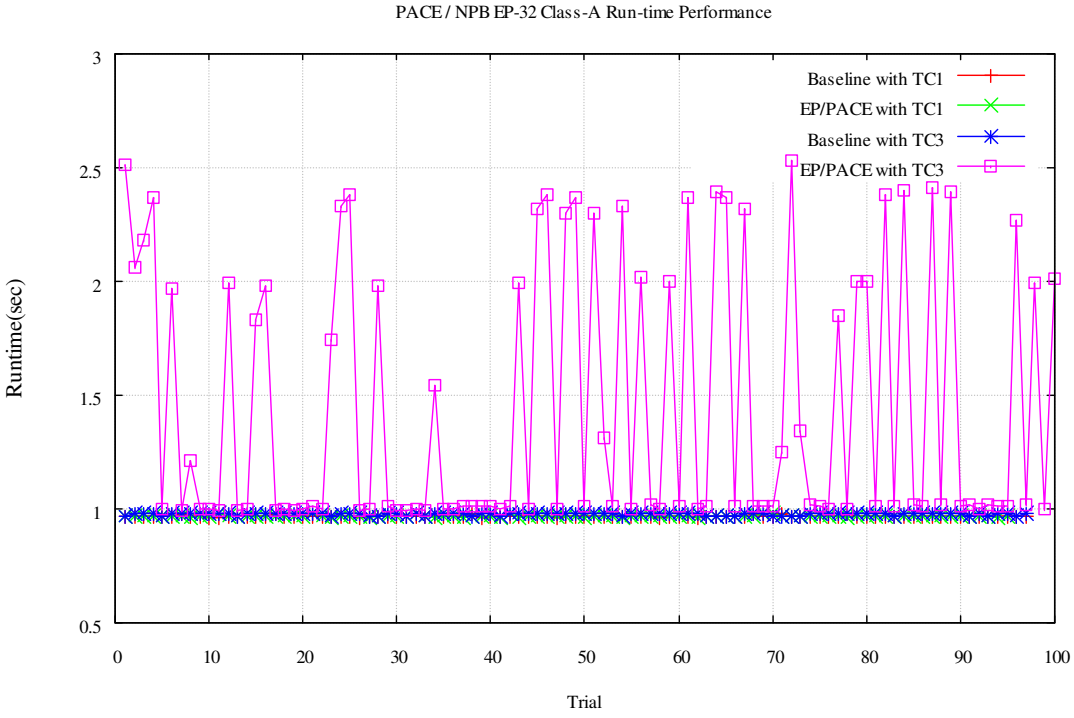

Figure 0.5 Class A EP32 Baseline and Sensitivity runtimes

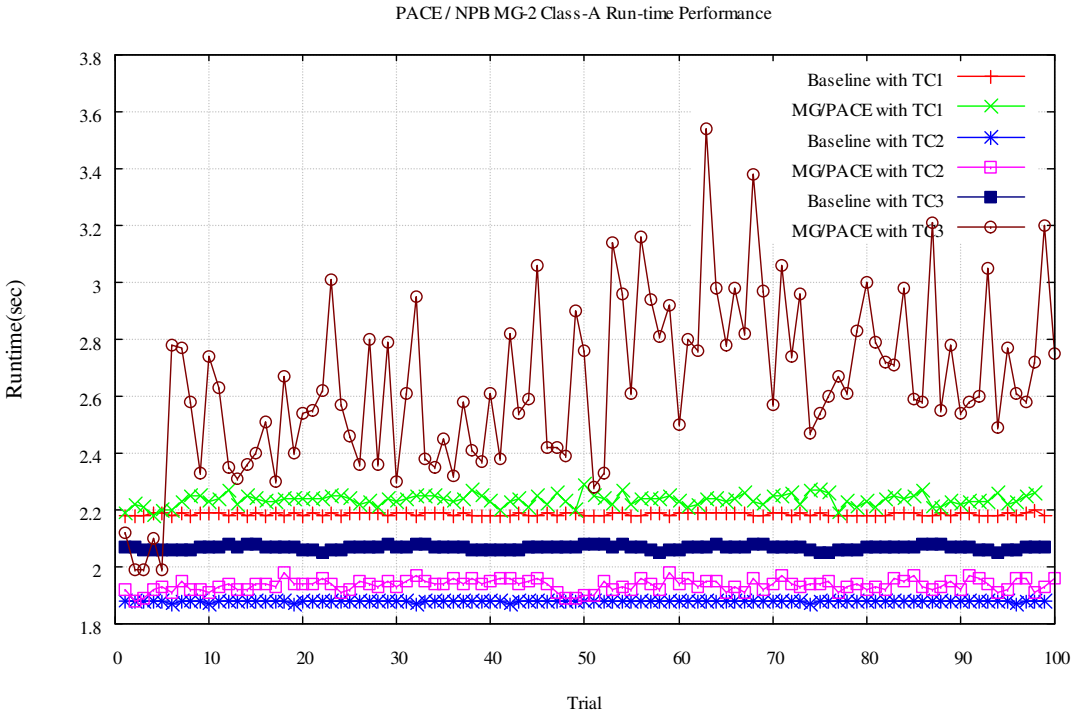

Figure 0.6 Class A MG2 Baseline and Sensitivity runtimes

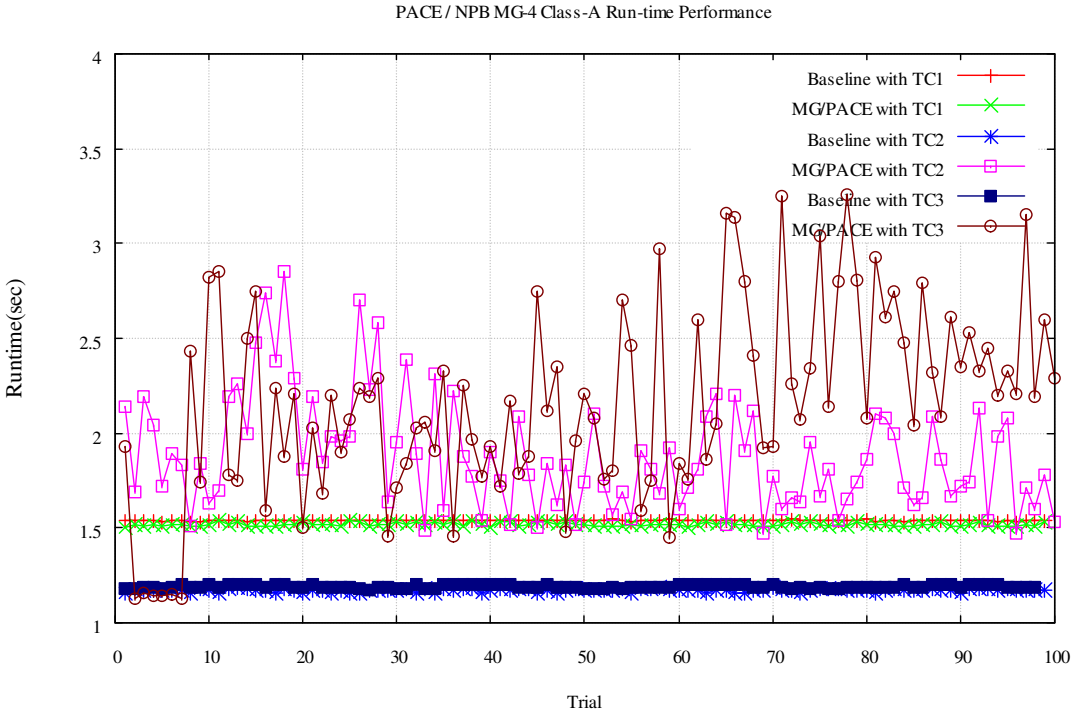

Figure 0.7 Class A MG4 Baseline and Sensitivity runtimes

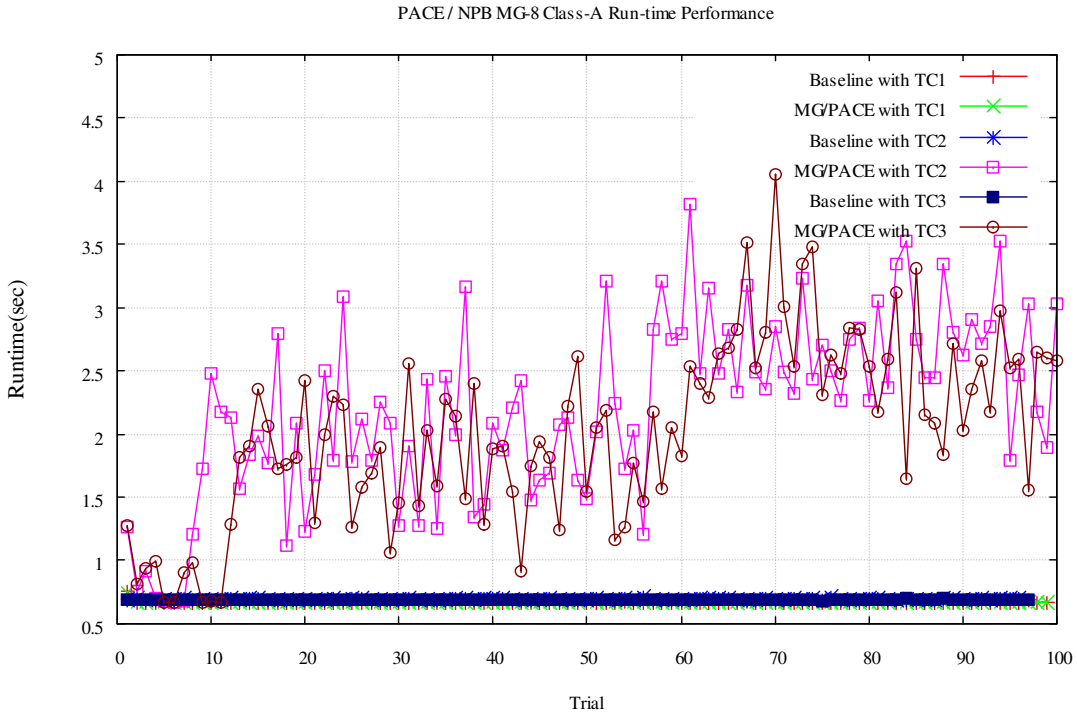

Figure 0.8 Class A MG8 Baseline and Sensitivity runtimes

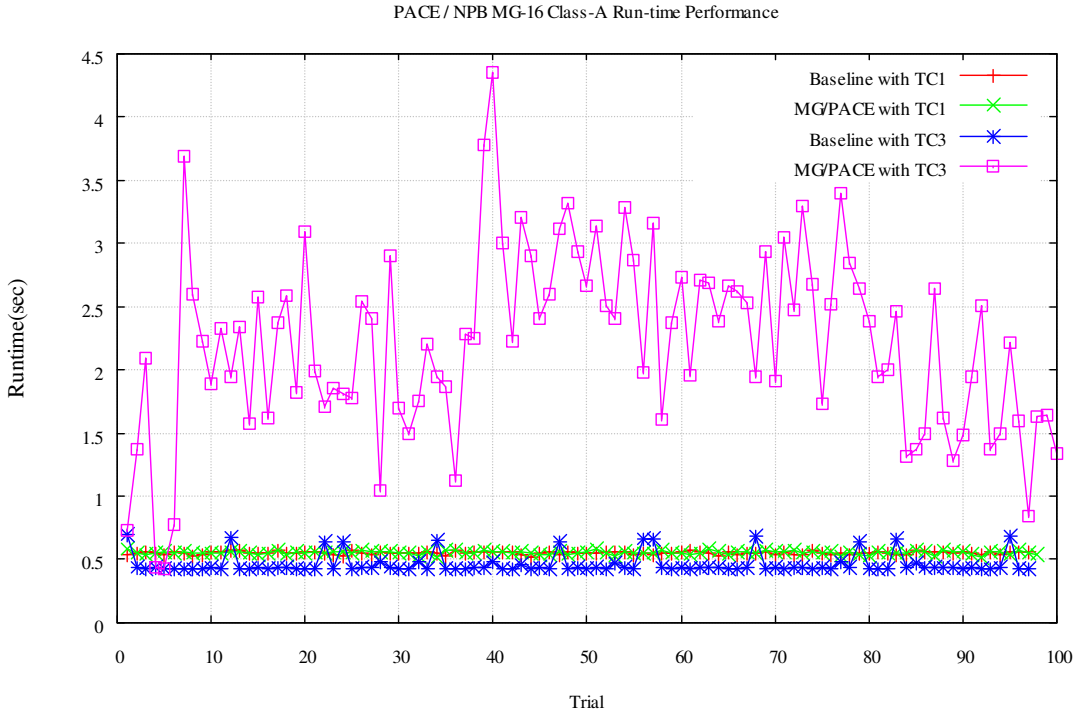

Figure 0.9 Class A MG16 Baseline and Sensitivity runtimes

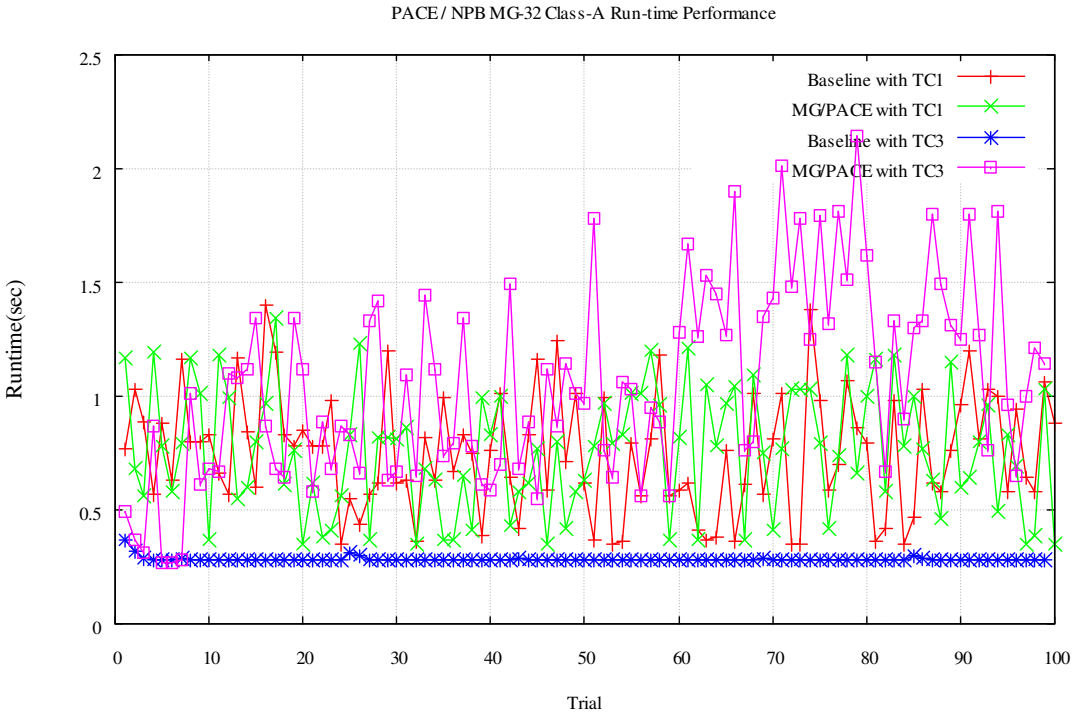

Figure 0.10 Class A MG32 Baseline and Sensitivity runtimes

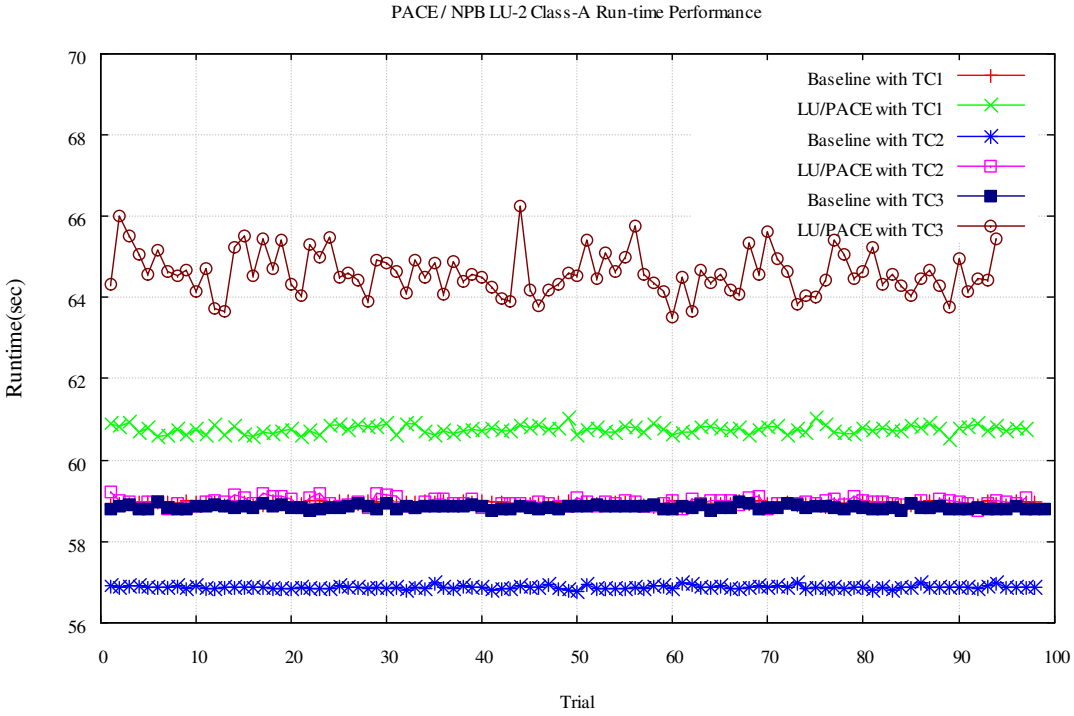

Figure 0.11 Class A LU2 Baseline and Sensitivity runtimes

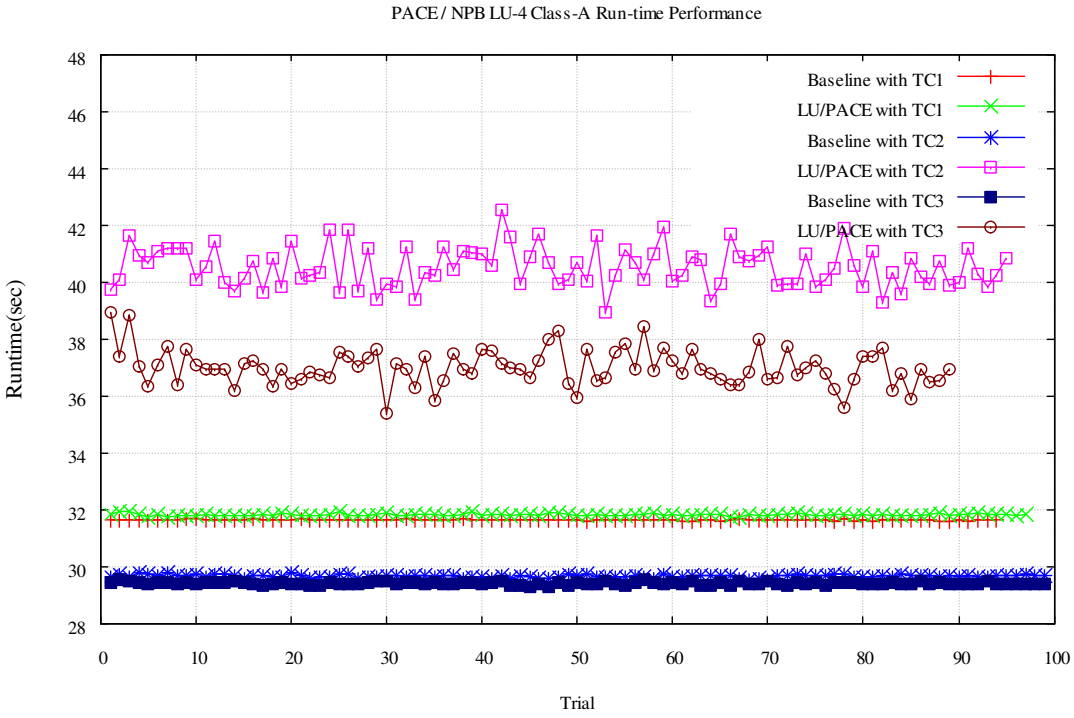

Figure 0.12 Class A LU4 Baseline and Sensitivity runtimes

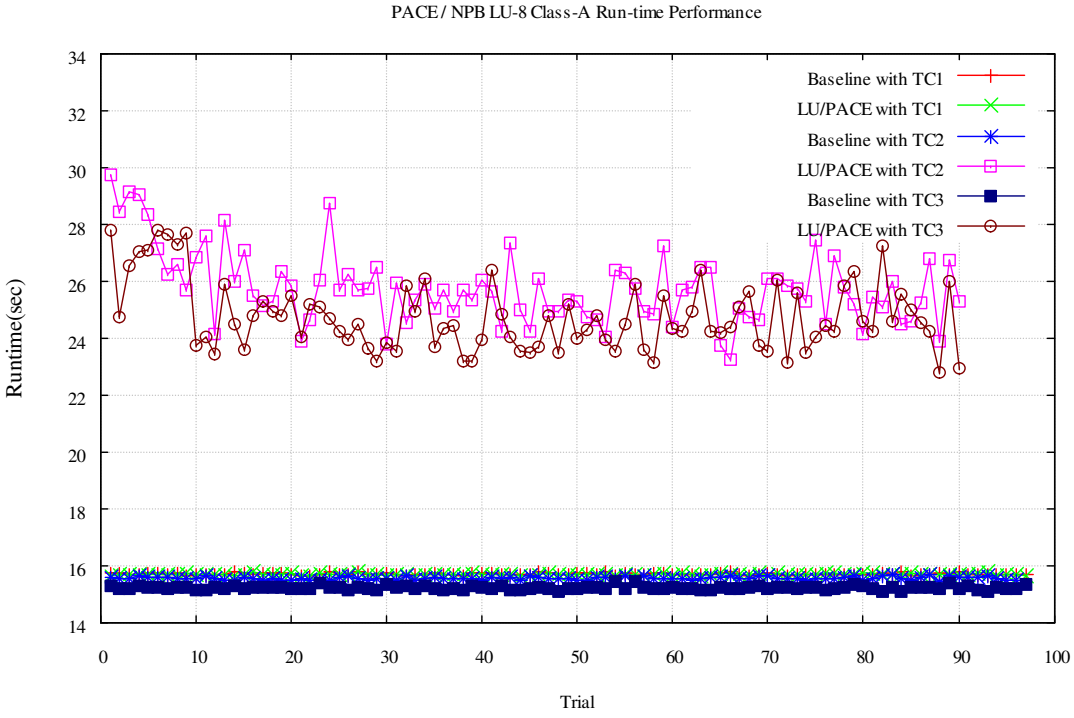

Figure 0.13 Class A LU8 Baseline and Sensitivity runtimes

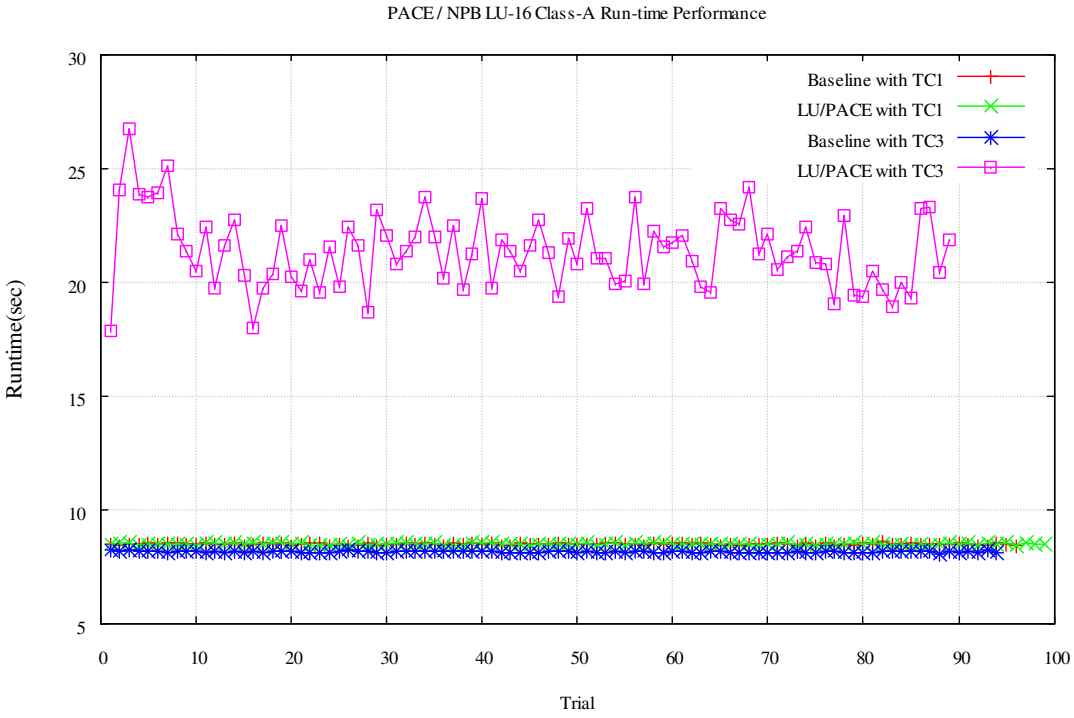

Figure 0.14 Class A LU16 Baseline and Sensitivity runtimes

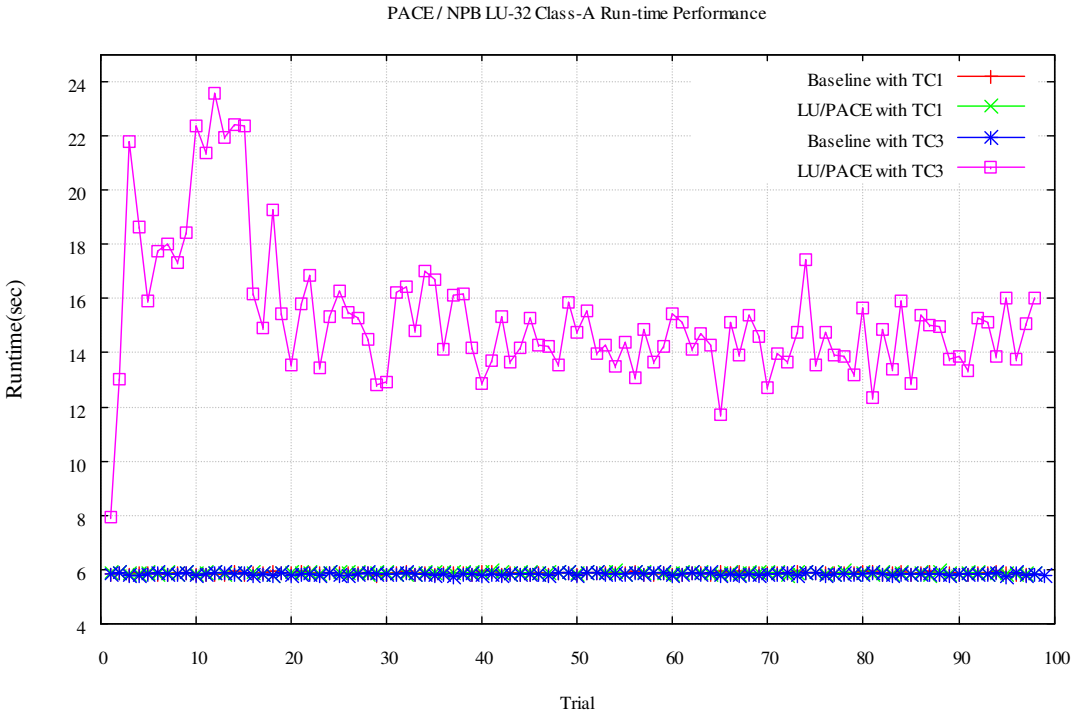

Figure 0.15 Class A LU32 Baseline and Sensitivity runtimes

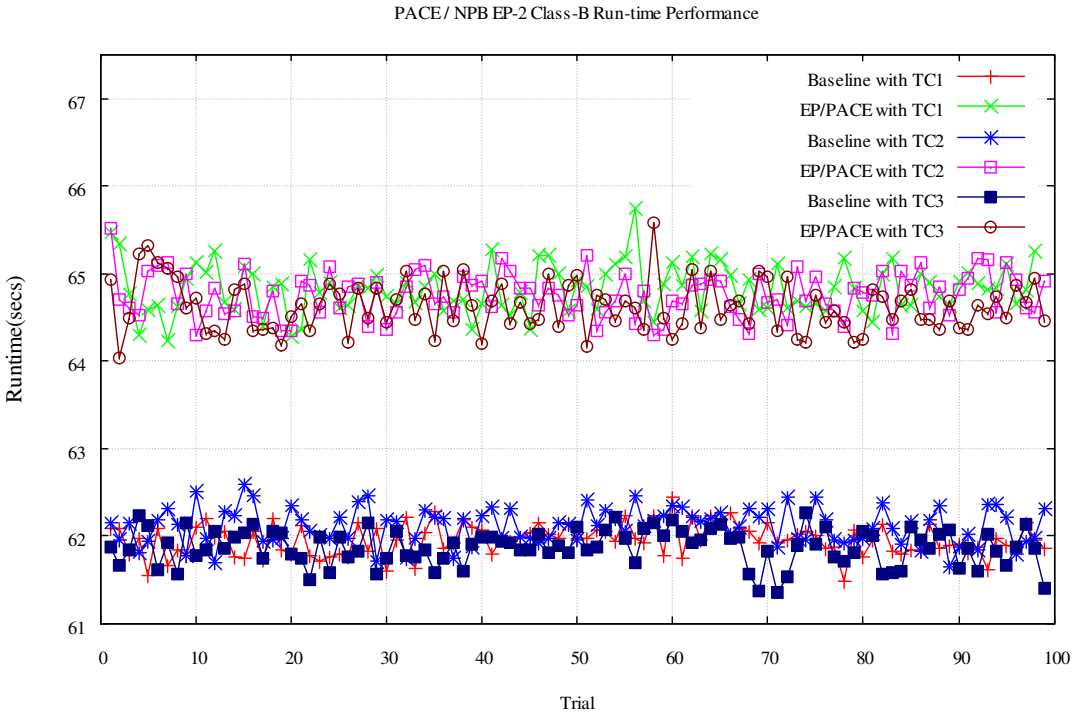

Figure 0.16 Class B EP2 Baseline and Sensitivity runtimes

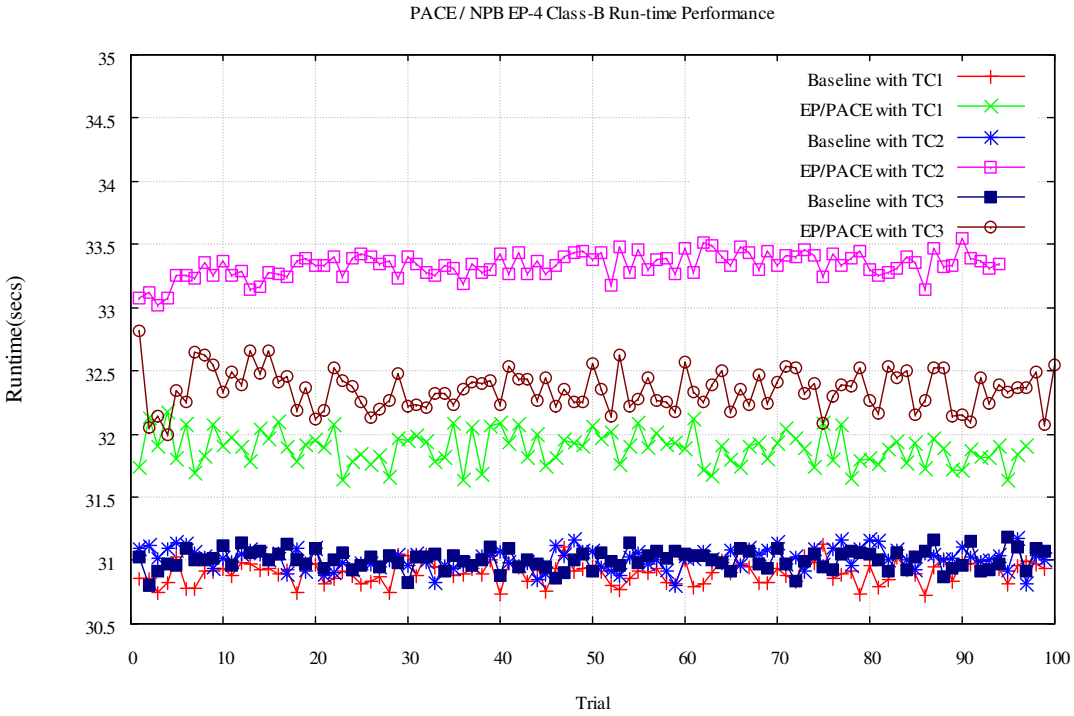

Figure 0.17 Class B EP4 Baseline and Sensitivity runtimes

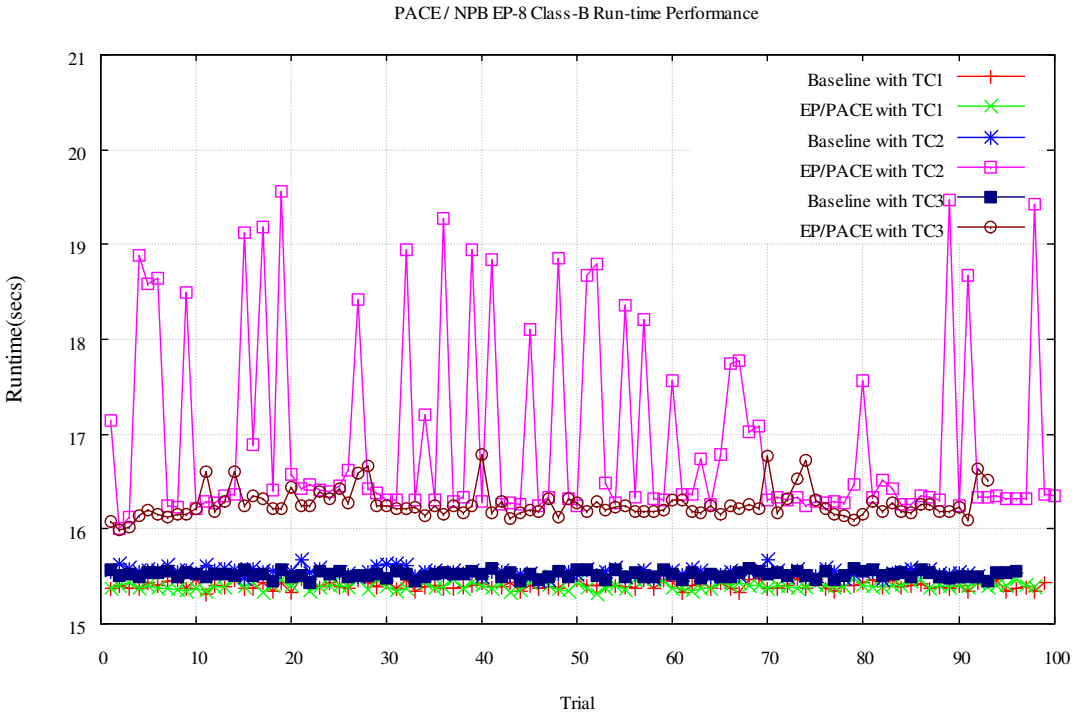

Figure 0.18 Class B EP8 Baseline and Sensitivity runtimes

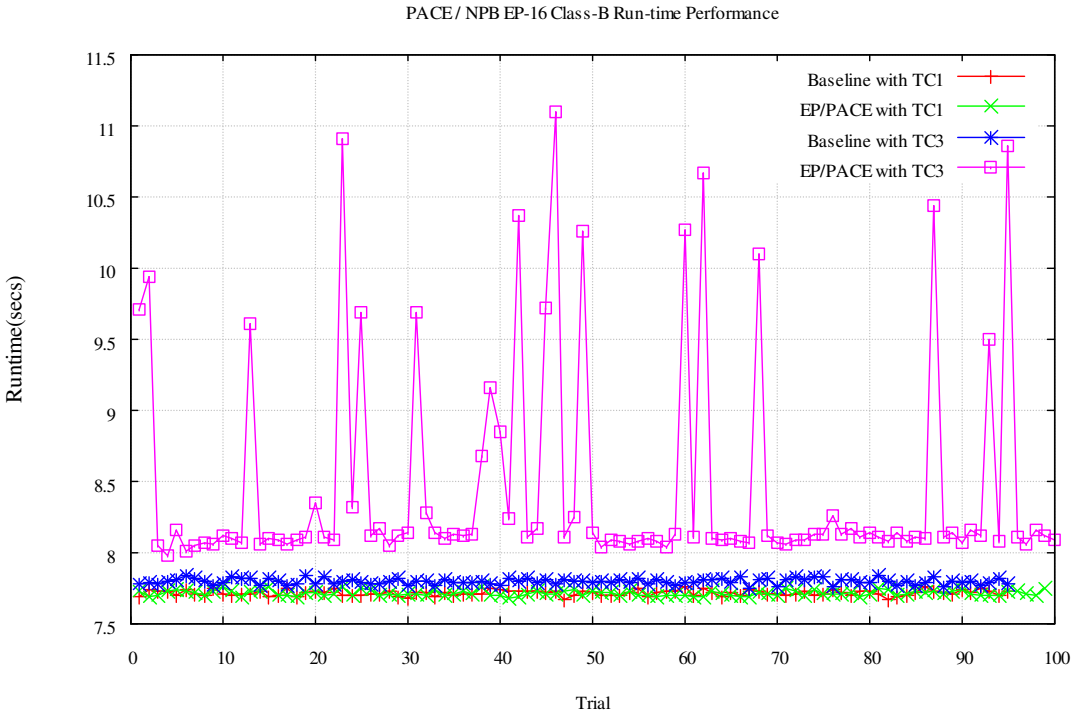

Figure 0.19 Class B EP16 Baseline and Sensitivity runtimes

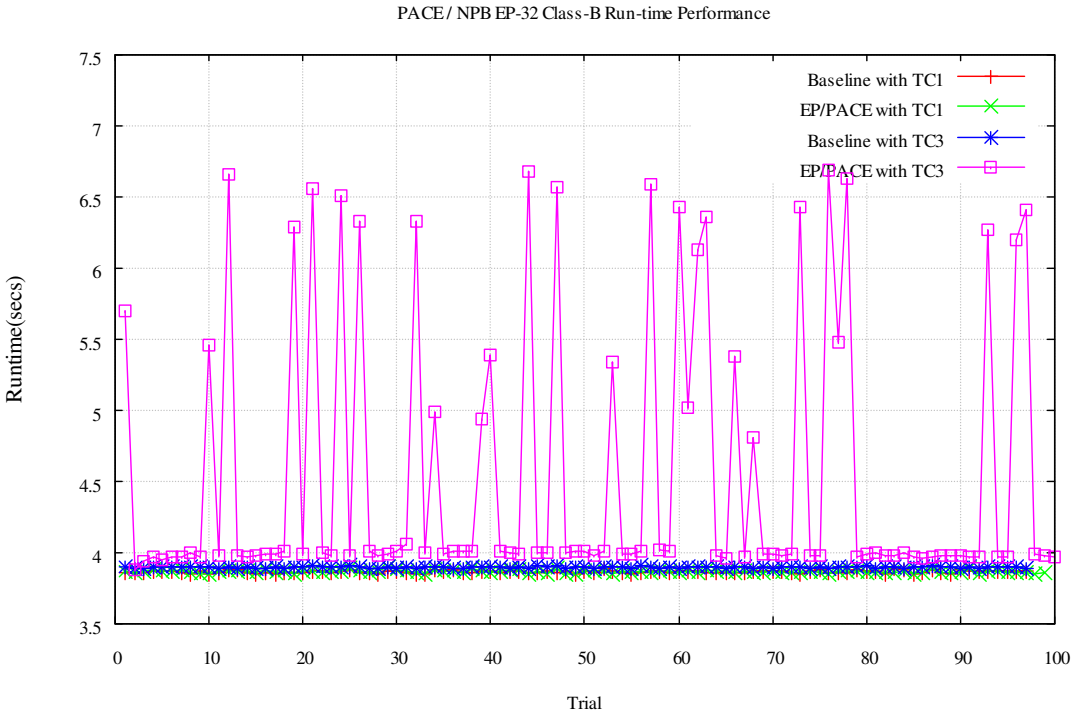

Figure 0.20 Class B EP32 Baseline and Sensitivity runtimes

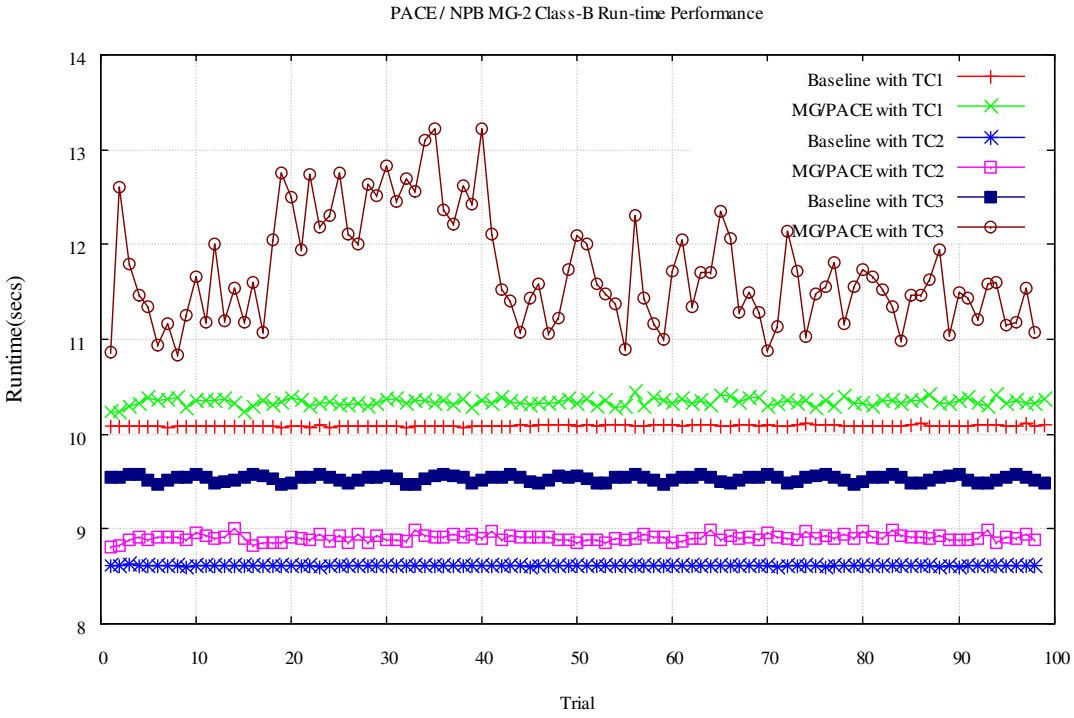

Figure 0.21 Class B MG2 Baseline and Sensitivity runtimes

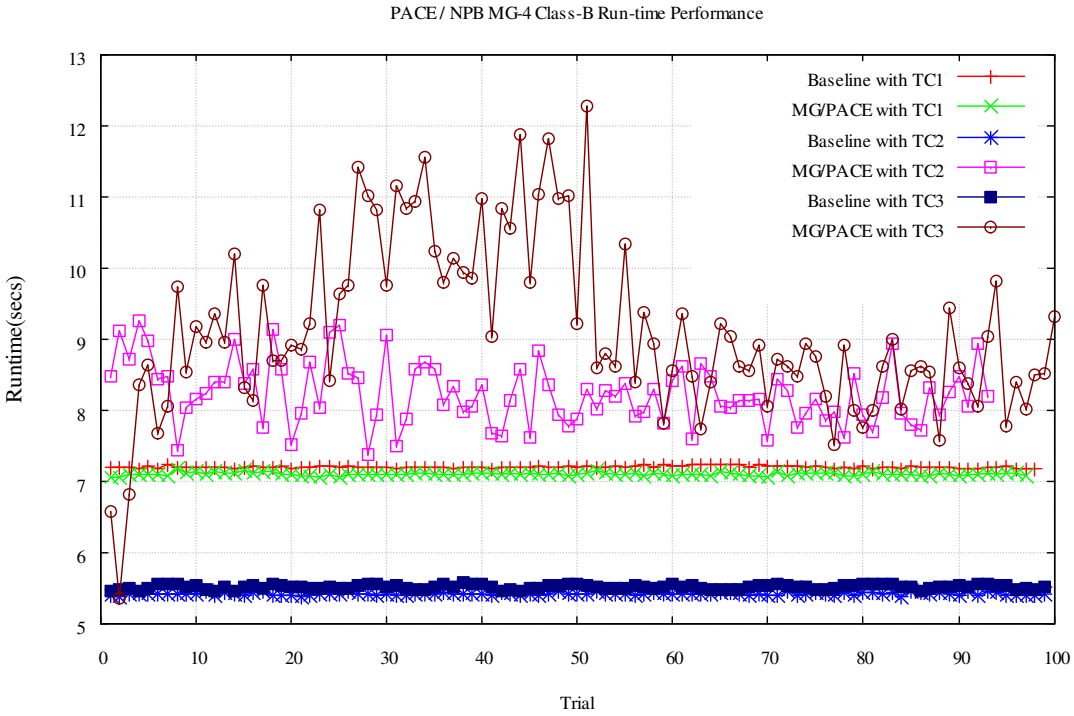

Figure 0.22 Class B MG4 Baseline and Sensitivity runtimes

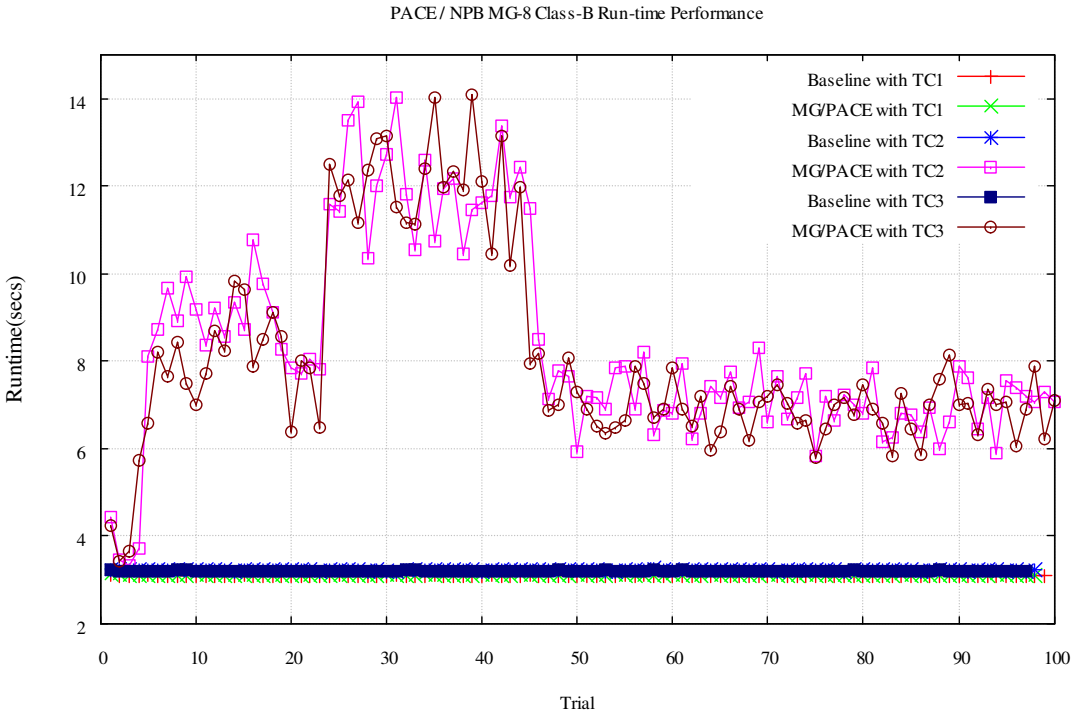

Figure 0.23 Class B MG8 Baseline and Sensitivity runtimes

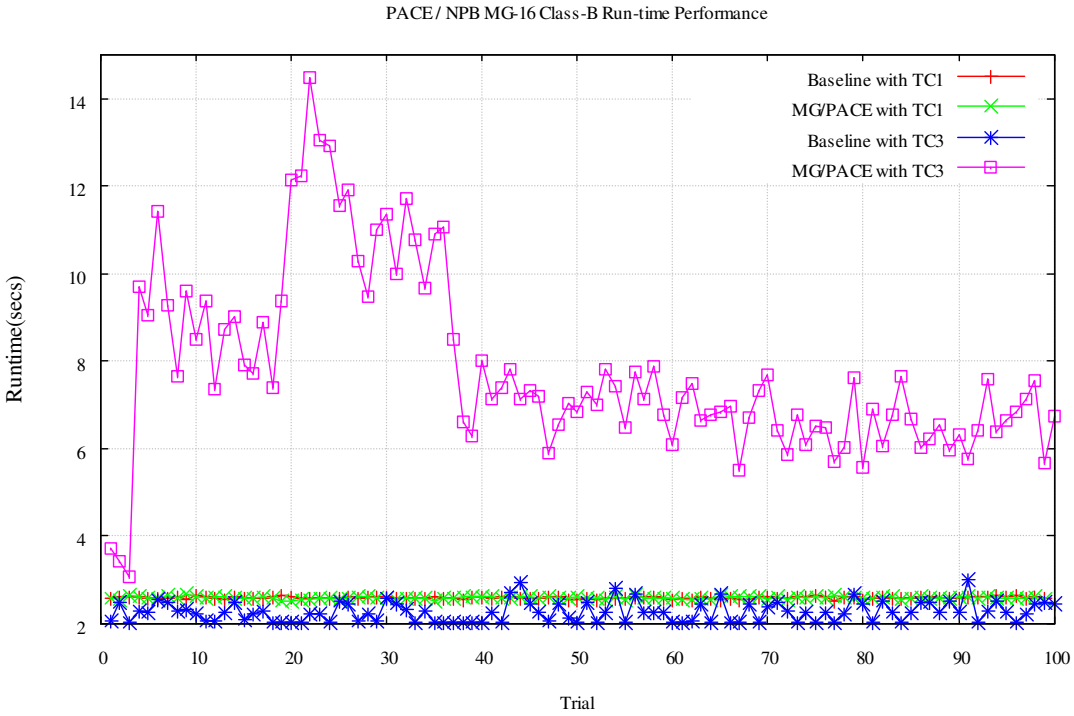

Figure 0.24 Class B MG16 Baseline and Sensitivity runtimes

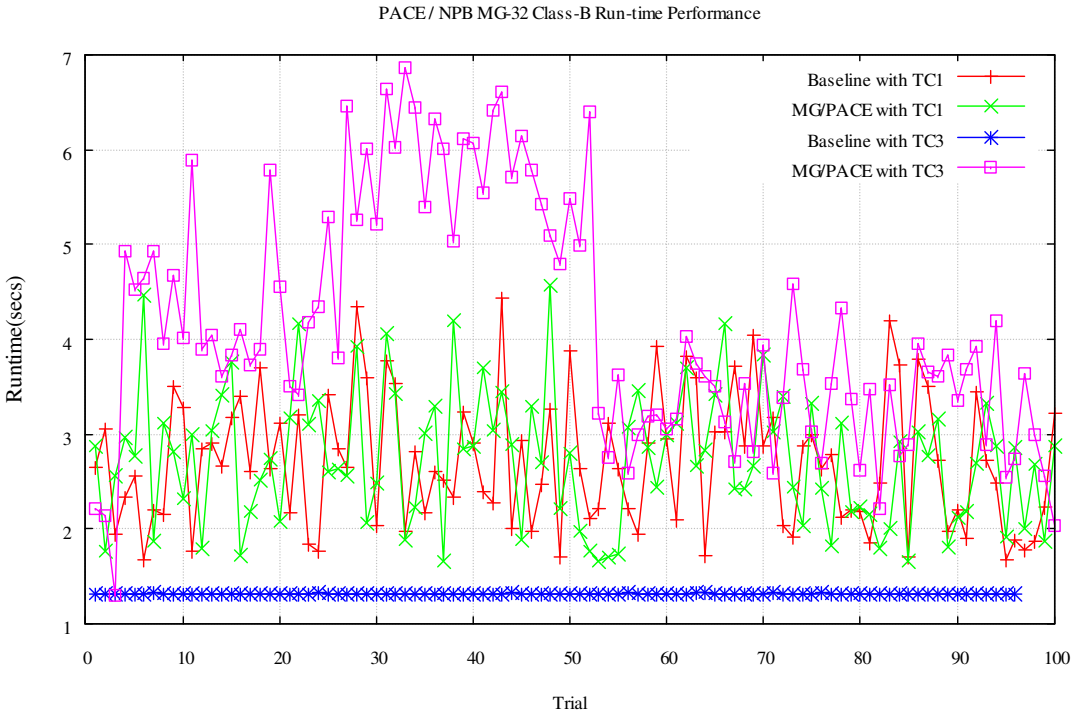

Figure 0.25 Class B MG32 Baseline and Sensitivity runtimes

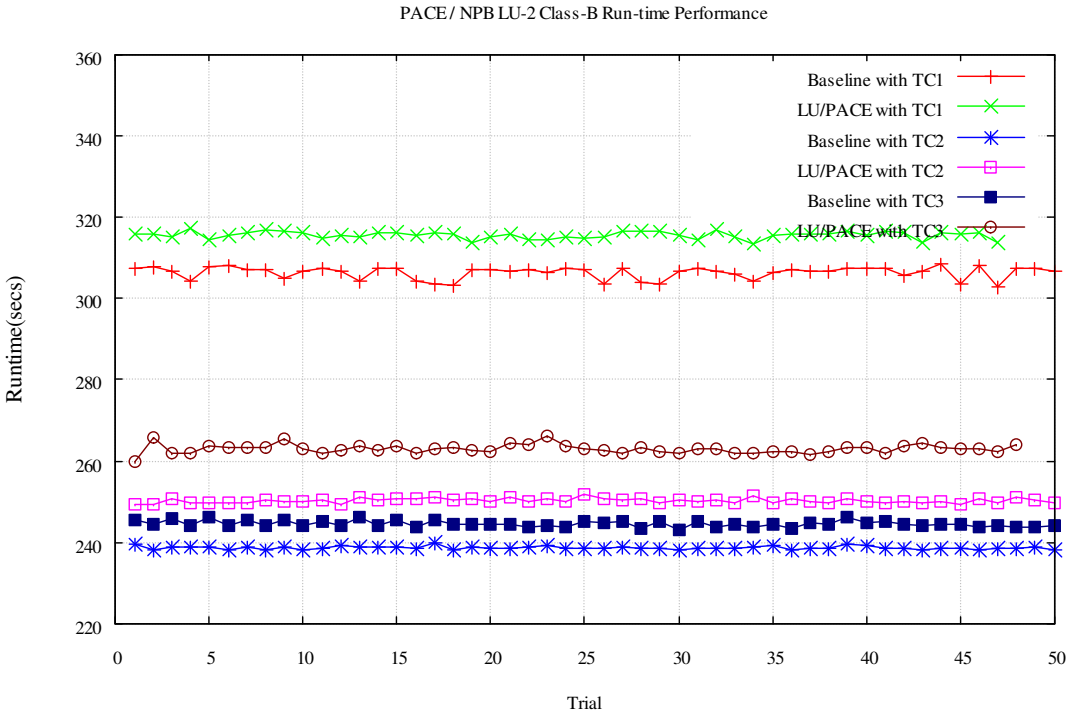

Figure 0.26 Class B LU2 Baseline and Sensitivity runtimes

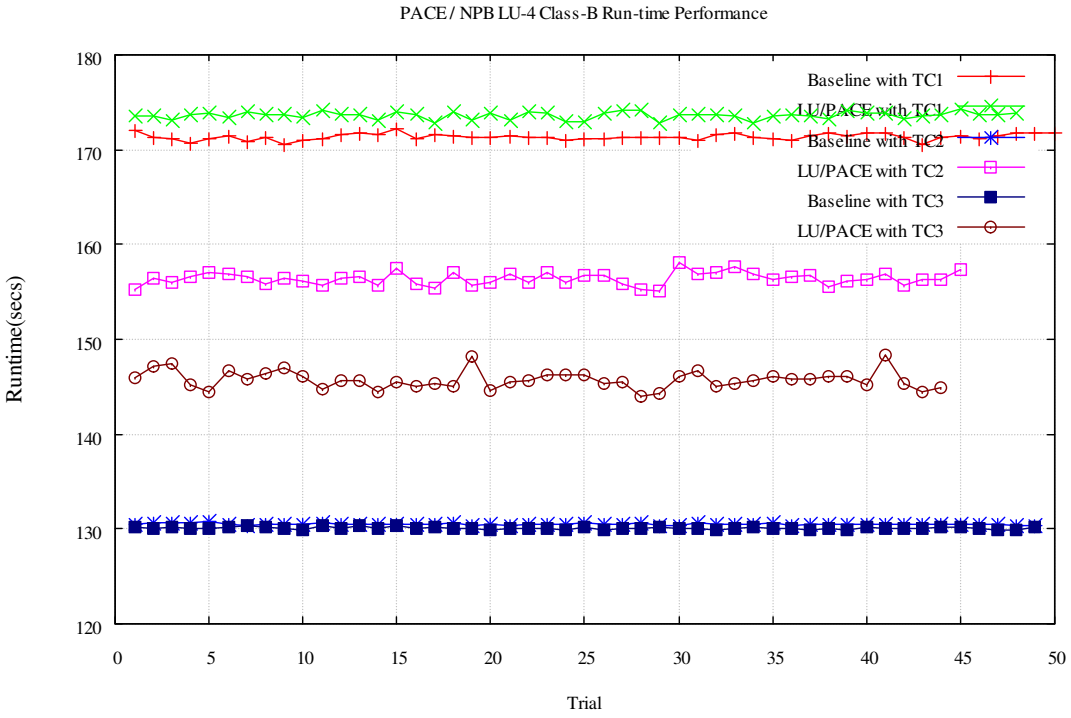

Figure 0.27 Class B LU4 Baseline and Sensitivity runtimes

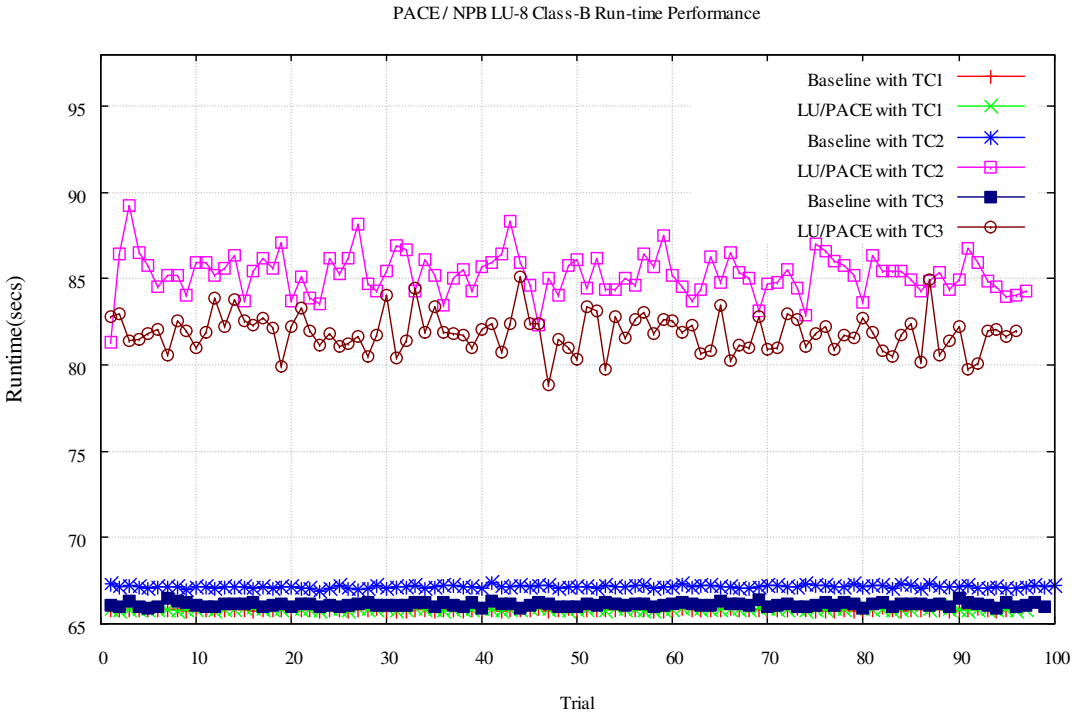

Figure 0.28 Class B LU8 Baseline and Sensitivity runtimes

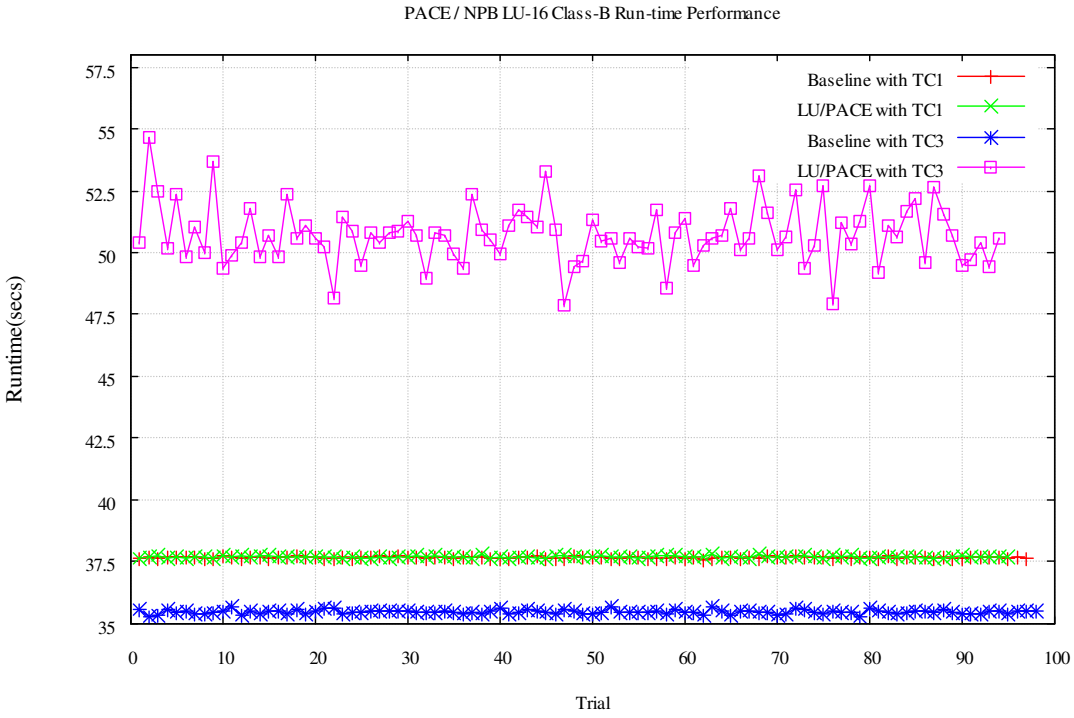

Figure 0.29 Class B LU16 Baseline and Sensitivity runtimes

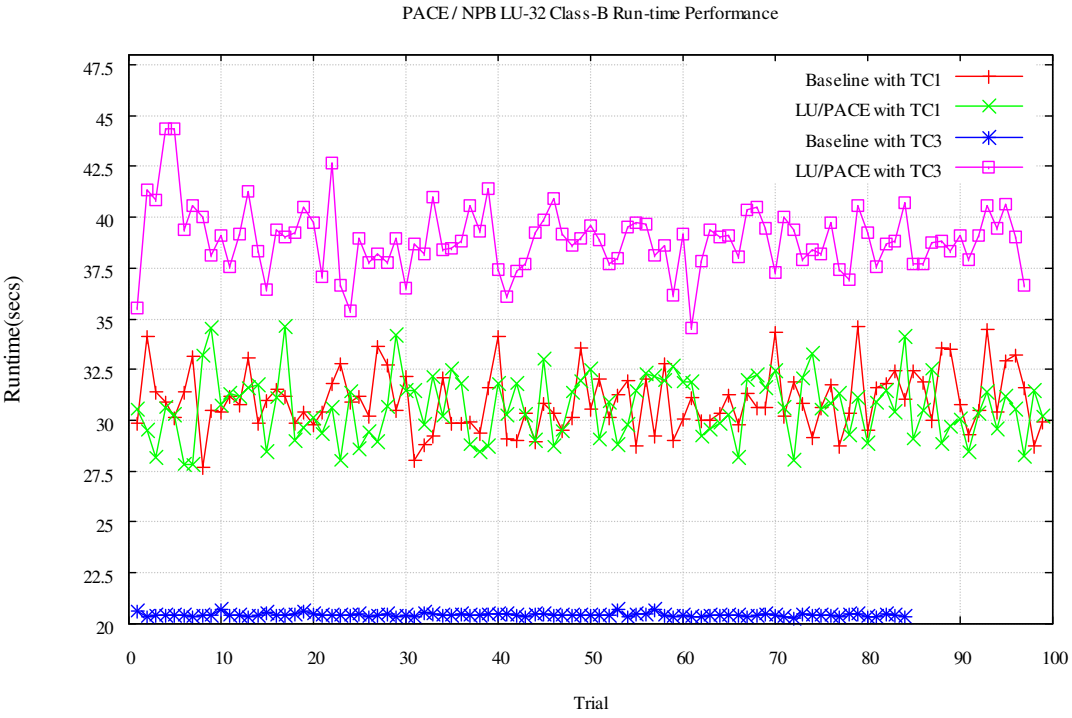

Figure 0.30 Class B LU32 Baseline and Sensitivity runtimes