CERIAS Tech Report 2008-30

Memory Balancing for Large-scale Network Simulation in Power-law Networks

by Hyojeong Kim
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

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 HyoJeong Kim

Entitled Memory Balancing for Large-scale Network Simulation in Power-law Networks

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Kihong Park
Chair

Sonia Fahmy

Eugene Spafford

Cristina Nita-Rotaru

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): Kihong Park

Approved by: Aditya Mathur / William J. Gorman 12/05/08
Head of the Graduate Program Date

Graduate School Form 20
(Revised 10/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

Memory Balancing for Large-scale Network Simulation in Power-law Networks

For the degree of Doctor of Philosophy

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.

HyoJeong Kim
Signature of Candidate

12/08/08
Date

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

MEMORY BALANCING FOR LARGE-SCALE NETWORK SIMULATION IN POWER-LAW NETWORKS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

HyoJeong Kim

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2008

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Kihong Park for his persistent guidance for seven years of my graduate study at Purdue. He has been always available for meetings even nights and weekends. His keen criticism on research and his earnest devotion to science have improved my attitude of exploring science. I would like to thank Professor Sonia Fahmy, Professor Cristina Nita-Rotaru, and Professor Eugene Spafford for serving on my advisory committee and helping me with the dissertation. I thank Dr. William J. Gorman for helping me to solve many administrative problems. Thanks to Steve Plite and Dan Trinkle for helping me manage and set up experimental environments.

I would like to thank my lab mates at Network Systems Lab: Bhagya Bethala, Humayun Khan, Asad Awan, and Hwanjo Heo. They have given me technical feedback about my research and also moral supports when needed. Since they have been there always, the long journey of Ph.D. study has been enjoyable. Many thanks to my friends at Purdue: Mercan Topkara, Umut Topkara, and Nauman Rafique. With them, I shared many memorable moments outside the lab. Special thanks to my friends, Wonhong Nam and Hyunyoung Kil, for their constant warm moral supports. Finally, I would like to express my gratitude to my parents and brothers for their life-long love and support. I also thank my friends in Korea, Eun-Ju Jwa and Seung-Hyub Jeon, who have given me warm encouragements throughout my study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

ABSTRACT

Kim, HyoJeong. Ph.D., Purdue University, December 2008. Memory Balancing for Large-scale Network Simulation in Power-law Networks. Major Professor: Kihong Park.

Large-scale network simulation has grown in importance due to a rapid increase in Internet size and the availability of Internet measurement topologies with applications to computer networks and network security. A key obstacle to large-scale network simulation over PC clusters is the memory balancing problem, where a memory-overloaded machine can slow down a distributed simulation due to disk I/O overhead. Network partitioning methods for parallel and distributed simulation are insufficiently equipped to handle new challenges brought on by memory balancing due to their focus on CPU and communication balancing.

This dissertation studies memory balancing for large-scale network simulation in power-law networks over PC clusters. First, we design and implement a measurement subsystem for dynamically tracking memory consumption in DaSSFNet, a distributed network simulator. Accurate monitoring of memory consumption is difficult due to complex protocol interaction through which message related events are created and destroyed inside and outside a simulation kernel. Second, we achieve efficient memory cost monitoring by tackling the problem of estimating peak memory consumption of a group of simulated network nodes in power-law topologies during network partitioning. In contrast to CPU balancing where the processing cost of a group of nodes is proportional to their sum, in memory balancing this closure property need not hold. Power-law connectivity injects additional complications due to skews in resource consumption across network nodes. Third, we show that the maximum memory cost metric outperforms the total cost metric for memory balancing under multilevel re-

cursive partitioning but the opposite holds for CPU balancing. We show that the trade-off can be overcome through joint memory-CPU balancing—in general not feasible due to constraint conflicts—which is enabled by network simulation having a tendency to induce correlation between memory and CPU costs. Fourth, we evaluate memory balancing in the presence of virtual memory (VM) management which admits larger problem instances to be run over limited physical memory. VM introduces complex memory management dependencies that make understanding and evaluating simulation performance difficult. We provide a performance evaluation framework wherein the impact of memory thrashing in distributed network simulation is incorporated which admits quantitative performance comparison and diagnosis. Fifth, we show that improved memory balancing under the maximum cost metric in the presence of VM manifests as faster completion time compared to the total cost metric despite the CPU balancing advantage of the latter. In the cases where the CPU balancing advantage of the total cost metric is strong, we show that joint memory-CPU balancing can achieve the best of both worlds.

We carry out performance evaluation using benchmark applications with varying traffic characteristics: BGP routing, worm propagation under local and global scanning, and distributed client/server system. We use a testbed of 32 Intel x86 machines running a measurement-enhanced DaSSFNet over Linux.

1 INTRODUCTION

1.1 Motivation

Large-scale network simulation is a multi-faceted problem spanning synchronization, network modeling, simulator design, partitioning, and resource management [1, 2]. Large-scale network simulation has grown in importance due to a rapid increase in Internet size and the availability of Internet measurement topologies, with applications to computer networks and network security. They include Border Gateway Protocol (BGP) routing in Internet inter-autonomous system (AS) networks, protection against distributed denial-of-service (DDoS) attacks, Internet worm epidemics, peer-to-peer systems, and multicasting [3–13].

A key obstacle of large-scale network simulation over PC clusters is the memory balancing problem where a memory-overloaded machine can slow down a distributed simulation due to disk I/O overhead. Figure 1.1 illustrates the impact of memory overload on a BGP routing simulation running on a single Linux PC configured with 2 GB and 1 GB physical memory ("BGP A"). There is a factor 9.4 slow down when memory is 1GB. "BGP B" compares completion time of a different BGP simulation instance on 512 MB and 256 MB memory configurations which may be found in older PC clusters. There is a factor 52.3 difference. Slow down due to disk I/O overhead can vary significantly depending on physical memory, application memory referencing behavior, and operating system support [14, 15].

From a user's perspective, parallel and distributed computing clusters are widely available, and so are parallel and distributed simulation environments such as SSFNet, DaSSFNet, Parallel/Distributed NS (PDNS), and JavaSim (J-Sim), that provide software support for utilizing hardware resources [16–19]. Network simulators assume that network partitioning where a network graph is partitioned into subgraphs and

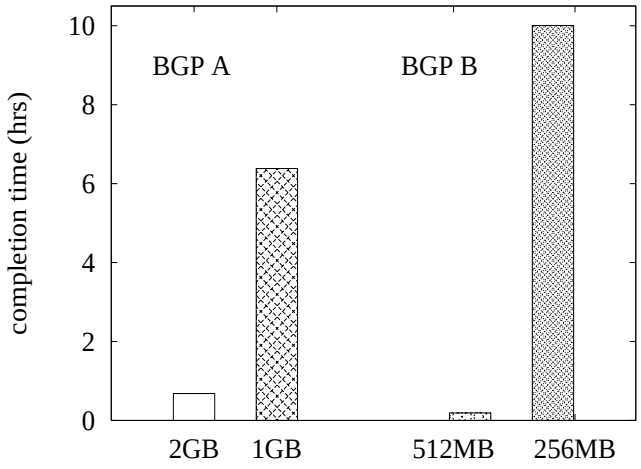


Figure 1.1. Completion time slow down of BGP simulations due to disk I/O overhead for different memory configurations.

subgraphs are mapped to processing nodes is done beforehand and given as part of the input of a simulation. Network partitioning methods for parallel and distributed simulation are insufficiently equipped to handle new challenges brought on by memory balancing due to their hereto focus on CPU and communication balancing where the main goal has been to facilitate parallel speed-up by balancing CPU load while reducing network communication cost [20, 21].

1.2 Memory Balancing Issues

We consider several issues that need to be taken into account when performing memory balancing for large-scale network simulation with power-law topologies.

1.2.1 Systems Issue: Accurate Memory Cost Estimation

The first issue is a systems related feature where, to balance memory load, we require accurate information about the memory needs of simulated network nodes for a given problem instance. Due to the dynamic nature of network events, run-time

measurement of resource consumption behavior is required which includes memory, CPU, and communication cost. Accurate gauging of memory cost is difficult because message related simulation events are created and destroyed via complex protocol interactions inside and outside a simulation kernel. Whereas the processing cost of a simulated network node is proportional to the total number of messages processed over time, its memory cost is determined by the maximum footprint over time which involves tracking of dynamic memory allocation/deallocation and event synchronization.

1.2.2 Algorithmic Issue: Efficient Memory Cost Estimation

A second issue is algorithmic in nature which highlights a key difference between network partitioning for memory and CPU balancing. In CPU balancing, when the processing cost of a group of simulated network nodes is considered, it is proportional to the sum of the processing cost of the individual nodes. There is a closure property with respect to summation (i.e., integration) since an individual node's processing cost is the sum of its processing cost over time. This allows CPU cost accounting to be done using constant space per network node. Since the memory cost of a network node is the maximum cost over time but the collective memory cost of a group of nodes is the maximum over time of the sum of individual memory costs, memory cost estimation during network partitioning requires that a node's memory consumption time series be logged to calculate the maximum over time of the sum of individual costs. Space complexity dictates that this is infeasible.

1.2.3 Graph Connectivity Issue: Power-law Topology

Advances in Internet measurement research have yielded measurement-based real-world network topologies [22–27]. They are not only large in size (Figure 1.2 shows the growth of the Internet AS topology during the past 10 years) but their connectivity tends to follow a power-law structure [28–33]. Irrespective of whether large-scale

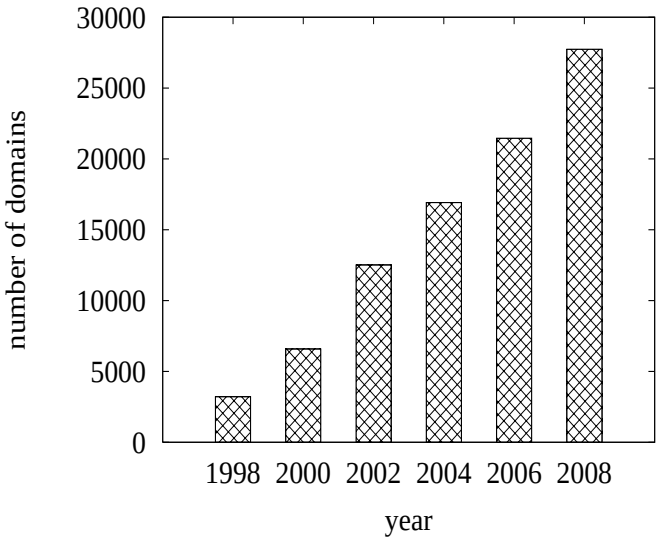


Figure 1.2. Growth of the Internet inter-AS network during 1998–2008 based on RouteViews/NLANR measurement data [23].

measurement topologies are mathematically power-law, they exhibit a characteristic skewness that impacts memory balancing: most nodes are connected to a few nodes and a few nodes are connected to many nodes. In network simulation, high-degree nodes ("elephants") tend to consume significantly more memory and CPU resources than low-degree nodes ("mice") which introduces difficulties during network partitioning due to the large disparity in resource consumption. In random graphs [34] all nodes have approximately equal degree which implies that their resource consumption during simulation is similar and therefore interchangeable during network partitioning. Network partitioning in random graphs is significantly easier than network partitioning in power-law graphs.

1.2.4 Performance Issue: Resource Balancing Trade-off

The difference in the cost metrics between memory balancing and CPU balancing—i.e., maximum vs. total—suggests that there may be a performance trade-off between memory and CPU balancing. That is, to balance memory well we may have to incur

a penalty in CPU balancing, and vice versa. The trade-off relation may depend on a number of factors including application type and problem size. In addition to investigating the trade-off relation, we may ask whether it is possible to achieve the best of both worlds.

1.2.5 Operating System Issue: Virtual Memory

The preceding issues pertain to memory balancing as a goal unto itself without specific regard to the performance consequences that memory imbalance may bring about. If a simulator is run on an operating system (OS) without virtual memory (VM) support and memory demand exceeds available physical memory, the simulation crashes. With VM support that allows larger problem instances to be run, the performance bottleneck tends to be onset of memory thrashing where an OS expends significant effort swapping pages in and out while blocked on the resultant disk I/O. Higher memory imbalance can translate to earlier onset of thrashing which results in slower completion time. VM introduces complex memory management dependencies that make understanding and evaluating simulation performance difficult.

1.3 Thesis Statement

The thesis statement of this dissertation is as follows. It is possible to achieve memory balancing for facilitating large-scale network simulation in power-law networks over PC clusters that achieves significant performance improvement over the existing state-of-the-art in network simulation partitioning. This is accomplished by taking a memory-centric approach that tackles key issues spanning accurate and efficient memory cost measurement, influence of power-law connectivity, memory-CPU balancing trade-off, and impact of virtual memory and thrashing.

1.4 Technical Challenges

The key technical challenges addressed in this dissertation are as follows.

(i) *Design, implement, and evaluate a measurement subsystem in DaSSFNet that achieves accurate memory cost estimation.* Accurate estimation of per-node memory cost is difficult because of the need to keep track of dynamic memory allocation/deallocation that are the result of complex protocol interaction inside and outside the DaSSF simulator kernel. The size of data structures of various event types, their lifetime, synchronization across different simulation events to compute the maximum memory footprint over time at per-node granularity, and garbage collection are challenges that need to be handled. When a frequently instantiated message type is maintained as a message pool to reduce memory allocation overhead, calling the destructor may not free memory from the viewpoint of the OS. `malloc`() performs its own internal memory pool management so that `free`() need not necessarily return memory to the OS. The focus on memory balancing should not come at the expense of CPU and communication balancing. That is, the measurement subsystem must provide accurate CPU and communication cost estimation which is also needed for performance comparison and trade-off analysis.

(ii) *Achieve efficient memory cost estimation such that space complexity per node is $O(1)$.* For a graph with n nodes the processing cost of node $i \in [1, n]$, denoted C_i, is proportional to the total number of messages $X_i(t)$ processed at the node over time, i.e., $C_i \propto \sum_t X_i(t)$. Hence the sum of the CPU costs of two nodes $i, j \in [1, n]$ is given by their sum $C_{ij} = C_i + C_j$ due to closure under summation. The per-node space complexity for CPU cost is constant. The memory cost of node i, denoted M_i, is determined by the maximum footprint over time, i.e., $M_i = \max_t X_i(t)$. The memory cost of two nodes $i, j \in [1, n]$, however, is not their sum $M_i + M_j$, but $M_{ij} = \max_t(X_i(t) + X_j(t))$ which need not equal the sum. Only when peak memory usage across nodes are synchronized does $\max_t(X_i(t) + X_j(t)) = \max_t X_i(t) + \max_t X_j(t)$ hold. We refer to this as the max-of-sum vs. sum-of-max problem. Our network

simulation results show that memory peak synchronization is an exception, not the rule. Network partitioning to balance memory requires that the collective memory cost of a group of nodes be computed so that different balancing choices can be weighed. Maintaining a per-node time series $X_i(t)$ incurs per-node space complexity $O(t)$ which in large-scale network simulation with many nodes and long simulation times is not feasible. How to achieve space efficient memory cost estimation for network partitioning is a key challenge of memory balancing.

(iii) *Power-law connectivity and resource consumption skews.* Network simulation is different from other simulation domains in that the central event is a message that travels from node to node. All else being equal, a node that is highly connected is more likely to encounter and process messages. In applications running over Internet measurement networks that exhibit power-law connectivity, there is a tendency for the power-law connectivity skew to manifest as severe skews in message load. That is, if i is a high-degree node and j is a low-degree node, then $X_i(t) \gg X_j(t)$[1]. This stands in stark contrast to random graphs where all nodes have approximately the same degree and the probability of deviating significantly from the mean is exponentially small. Balancing equal size objects across a number of bins/partitions is easier than balancing unequal size objects since equal size objects are interchangeable. When the number of objects is many such that an individual object takes up a fraction of the total space ("grain of sand"), balancing becomes easier compared to the case where an object takes up a larger space in a bin ("pebbles"). Large-scale Internet measurement topologies with power-law connectivity skews produce a small but non-negligible number of very large size objects ("elephants") which work against balancing. However, most objects in power-law networks are small ("mice"), and given their large number, work in favor of balancing. Understanding the impact of power-law connectivity in memory and CPU balancing is a new challenge.

(iv) *Memory-CPU balancing trade-off and joint optimization.* Memory and CPU cost metrics are different since the former depends on the maximum resource footprint

[1]Recall that $X_i(t)$ is defined as the total number of messages processed at node i

over time whereas the latter is proportional to the total footprint over time. This may result in a conflict between memory and CPU balancing where balancing one well creates imbalance in the other. Understanding the trade-off relationship between memory and CPU balancing is important for effective network partitioning. A related question is joint memory-CPU balancing that aims to achieve the best of both worlds. Given a graph of n nodes that need to be partitioned into k groups, in the space of all k-partitions of $[1, n]$ the total cost metric favors a subset of partitions $\mathcal{S}_T$ where CPU is well-balanced and the maximum cost metric selects partitions $\mathcal{S}_M$ where memory is well-balanced. The joint memory-CPU balancing question asks whether there are partitions where both memory and CPU are well-balanced (i.e., $\mathcal{S}_M \cap \mathcal{S}_T \neq \emptyset$), and, if so, how to find them.

(v) *Framework for memory balancing under VM.* Memory balancing aims to minimize memory imbalance during network partitioning without regard to the ultimate effect this will have on simulation performance with respect to completion time. When a simulation run consumes significantly more memory than available physical memory, a VM may enter thrashing where an OS continually swaps pages in and out trying to satisfy application memory references. CPU utilization tends to be low since the simulation process spends most of its wall clock time blocked on disk I/O. A goal of memory balancing under VM is to delay the onset of thrashing so that the benefit of VM—ability to run larger problem instances for longer periods—can be maximally harnessed. However, this is not the only possible goal. VM introduces complex memory management dependencies since there is no one-to-one relation between memory imbalance, page fault rate, and performance slow down. For example, if memory demand on a PC in distributed simulation exceeds available physical memory by, say, 1 GB, a number of other factors contribute to determining whether thrashing sets in. Memory occupied by routing tables tend to be more conducive to locality of reference than memory consumed by messages. Messages, which distinguish network simulation from other simulation domains, tend to be transient in the sense of being dynamically created and destroyed. A 1 GB excess demand may not induce thrashing if memory

consumption is dominated by routing tables but trigger thrashing if messages are the dominant factor. Thrashing is a qualitative descriptor that may be accompanied by a range of page fault rates that in turn lead to severe or less severe simulation performance degradation. A performance evaluation framework with effective metrics is needed to gauge and diagnose distributed network simulation performance.

(vi) *Memory balancing performance under VM.* At the end of the day, we are interested in evaluating how much memory balancing contributes toward facilitating large-scale network simulation over PC clusters whose operating systems implement virtual memory management. The impact of the trade-off between memory and CPU balancing needs to be evaluated after the effect of memory imbalance is quantified with respect to distributed simulation performance slow down. The same goes for joint memory-CPU balancing.

1.5 New Contributions

The contributions of the dissertation are as follows. First, we design and implement a measurement subsystem for dynamically tracking memory consumption in DaSSFNet, a distributed network simulator that acts as our benchmarking software environment. We show that the measurement subsystem achieves accurate monitoring of memory consumption at per-node granularity. The measurement subsystem also accurately tracks CPU and communication cost. Although the specific implementation details are DaSSFNet dependent, the underlying measurement methodology is applicable to other distributed network simulation environments.

Second, we achieve efficient memory cost monitoring by showing that the maximum cost metric, which has constant per-node space complexity, enables effective memory balancing during network partitioning. We show that although the max-of-sum vs. sum-of-max problem cannot be solved in general because peak memory consumption across simulated nodes is not synchronized, the maximum cost metric suffices to achieve relative memory balancing. That is, predicting the absolute mem-

ory consumption of a group of nodes is inherently difficult but comparing the relative memory consumption between two or more groups of nodes is solvable. Our performance evaluation focuses on PC clusters with uniform memory but the maximum cost metric approach is extensible to the non-uniform memory case.

Third, we evaluate the impact of power-law connectivity of large-scale measurement networks on memory balancing. On the one hand, high-degree nodes contribute a large fraction of overall memory consumption and need to be spread out across partitions to avoid hot spots that lead to memory imbalance. On the other hand, the preponderance of low-degree nodes with similarly small memory requirements admits filling in the gaps left by high-degree nodes in the partitions by virtue of interchangeability and smallness. Since the memory cost estimate of a single high-degree node is accurate, combined memory balancing of high- and low-degree nodes is conducive to relative memory balancing. We devised and evaluated custom network partitioning methods that are sensitive to power-law connectivity and spread out high-degree nodes followed by filling in via low-degree nodes. However, we found that multilevel recursive partitioning tends to find partitions that obey this property which obviates the need to advance new network partitioning methods aimed at power-law topologies. This also holds for CPU balancing.

Fourth, we show that the maximum memory cost metric outperforms the total cost metric for memory balancing under multilevel recursive partitioning but the opposite holds for CPU balancing. The extent of the trade-off depends on the application type—BGP routing and worm epidemics under global (i.e., random) scanning exhibit small memory balance gain but high CPU balance gain, and vice versa for worm epidemics under local (i.e., topological) scanning and distributed client/server—and problem size. We explain the difference in trade-off across application type by showing that BGP and worm global exhibit a more pronounced memory cost skew induced by power-law connectivity that makes balancing more difficult.

Fifth, we show that the performance trade-off can be overcome through joint memory-CPU balancing which is, in general, not feasible due to constraint conflicts.

In network simulation we have $\mathcal{S}_M \cap \mathcal{S}_T \neq \emptyset$ which allows achieving the best of both worlds. This is facilitated by network simulation having a tendency to induce correlation between memory and CPU costs. The correlation is not strong enough to prevent maximum cost metric based network partitioning from favoring memory balancing and total cost metric based partitioning favoring CPU balancing. That is, individually the former finds partitions in $\mathcal{S}_M - \mathcal{S}_T$ and the latter finds partitions in $\mathcal{S}_T - \mathcal{S}_M$. However, if network partitioning is carried out in joint 2-dimensional search space, the correlation guides the search to partitions in their intersection $\mathcal{S}_M \cap \mathcal{S}_T$. We use multi-dimensional multilevel recursive partitioning to perform joint memory-CPU balancing.

Sixth, we advance a performance evaluation framework for evaluating memory balancing under VM. We define metrics for quantifying distributed simulation slow down under thrashing and the performance gain achieved by one network partitioning method over another. We show that onset of thrashing in network simulation is effected not only by the amount of excess memory demand (beyond physical memory) but also its composition with respect to message versus routing table memory. We find that an idiosyncrasy of network simulation is that messages are not conducive to locality of reference. As a consequence, in a distributed network simulation a PC that is most message-overloaded may be the weak link that slows down overall completion, not the PC that is most memory-overloaded.

Seventh, we show that improved memory balancing under the maximum cost metric in the presence of VM manifests as faster completion time compared to the total cost metric despite the CPU balancing advantage of the latter. The performance gain is most pronounced in worm local and distributed client/server which is consistent with the general trade-off between memory and CPU balancing and its dependence on application type. We show that in the cases where the CPU balancing advantage of the total cost metric is strong, joint memory-CPU balancing can achieve the best of both worlds.

1.6 Organization of the Dissertation

The dissertation is organized as follows. In Chapter 2, we present related works. In Chapter 3, we describe the design of our measurement subsystem in DaSSFNet for accurate resource usage estimation. In Chapter 4, we propose an efficient memory balancing mechanism targeted at large-scale network simulation in power-law networks. In Chapter 5, we evaluate memory balancing performance of the proposed mechanism. We establish a memory-CPU balancing trade-off, and we show that joint memory-CPU balancing can overcome the performance trade-off. In Chapter 6, we evaluate memory balancing performance under virtual memory. In Chapter 7, we summarize the contributions of this dissertation and discuss the future directions.

2 RELATED WORK

2.1 Approaches to Large-scale Network Simulation

2.1.1 Memory Requirement Optimization

The importance of reducing memory footprint for large-scale network simulation is well recognized, and research has been carried out to reduce application and simulator memory requirements [13, 35–37]. In BGP routing, it is known that routing tables are the main memory-consuming component in BGP simulation. In [35], the authors leverage the redundancy of information in BGP routing tables to reduce BGP routing table memory requirement. In [36, 37], efforts are directed at designing methods with the aim of reducing IP routing table memory requirement. In [13], the authors identify the main memory-consuming components in multicast simulations—multicast routing states and IP routing tables—and propose a set of techniques to compress their memory requirement.

2.1.2 Topological Down-scaling

Several works have been carried out in down-scaling large network topologies such that the size of the resultant network topology is small enough to be simulated within a reasonable timeline while preserving important characteristics of the original topology [38–40]. In [38], the authors advanced a number of reduction algorithms targeted at the Internet inter-AS networks and analyzed the fidelity of the reduced network topology against the original one with respect to power-law topology characteristics shown in [28]. In [39], the authors investigated reduction mechanisms also targeted at the Internet inter-AS networks, where selected vertices are removed and edges are added to preserve the original topology's routing path properties. In [40], the authors

have focused on down-scaling of an arbitrary network topology that is shared by TCP flows. They show that end-to-end TCP performance is preserved by retaining congested links in the sampled topology.

2.1.3 Distributed Simulation

Substantial efforts are directed at designing and implementing parallel and distributed network simulators with the aim of providing scalable solution to the growing need of large-scale network simulation. DaSSF/DaSSFNet [17] and SSFNet [16] implement Scalable Simulation Framework (SSF) API [41] in C++ and in Java, respectively. Parallel/Distributed NS (PDNS) [18] extends the Network Simulator (ns-2) [42] to enable the parallel and distributed simulation. JavaSim (J-Sim) [19] is a component-based, compositional simulation environment. The Georgia Tech Network Simulator (GTNetS) [43] provides distinct separation of protocol stack layers as actual networks are structured. PDNS [18], DaSSF/DaSSFNet [17], and GTNetS [43] provide packet-level network simulation environment over parallel and distributed machine environments. JavaSim (J-Sim) [19] and SSFNet [16] support packet-level network simulation environment over parallel processor machine environment. Genesis [44] proposes improved synchronization mechanism for distributed simulation.

Distributed simulation research [16–19, 41, 43, 44] has focused on provisioning transparent network simulation environment to users, addressing issues on simulation time management, synchronization, communication mechanism between remote machines, and model description mechanism. Nonetheless, in order to run large-scale network simulation across workstation clusters, users have to overcome resource barriers. This dissertation focuses on providing effective network partitioning to facilitate distributed memory and computing power for large-scale network simulation. Section 2.3 describes main related work in this perspective.

2.2 Power-law Network Connectivity

Internet inter-AS and intra-AS networks are shown to have power-law like connectivity [28] where most are connected to a few, but a few are connected to many. Recent measurements of various information networks, including the World Wide Web [29], metabolic networks [30], and various social networks [31–33], have shown such pattern. These networks are sometimes collectively referred to as *power-law networks* as there is a power-law relation between the degree and frequency of nodes of that degree: $\Pr\{\deg(u) = k\} \propto k^{-\beta}$. The impact of power-law network connectivity on a network security mechanism's performance has been studied [9]. Recent research has focused on improvement of multilevel graph partitioning algorithms—specifically, new clustering-based coarsening scheme—to accommodate power-law network connectivity characteristics [45].

2.3 Network Simulation Partitioning

2.3.1 Graph Partitioning Tools

Multilevel recursive bisection methods have been shown to yield improved partitioning vis-à-vis spectral methods [20], and their fast running time has made Metis and Chaco [46] commonly used benchmark tools for network partitioning. We use Metis [47] which implements a multilevel recursive k-way partitioning algorithm [21] as the default network partitioning tool. We use Chaco [46] as well to confirm our results with Metis. Some other available tools are as follows. The PARTY partitioning library [48] is a software library for graph partitioning which provides efficient implementations of approximation heuristics. JOSTLE [49] is a software tool designed for mapping unstructured mesh calculations to parallel computers for parallel speed-up. SCOTCH [50] is a software package for graph partitioning based on the dual recursive bipartitioning algorithm. Among all these tools, Metis [47] is the only tool which supports multi-dimensional node weights.

2.3.2 Recent Benchmark-driven Approaches

Network simulation partitioning has been studied in several papers, perhaps the most relevant to our work being Benchmark-based, Hardware and Model-Aware Partitioning (BencHMAP) [51]. BencHMAP is a general framework for network simulation partitioning whose scope included memory balancing. The main concern was managing CPU, communication, and synchronization cost (i.e., lookahead), with memory balancing receiving a tangential treatment as part of total message balancing given as node weight input during benchmarking. The latter roughly corresponds to total cost metric C_i defined in Chapter 4. In [52] topological partitioning is studied for scaling network emulation where component partitioning is driven by estimated communication cost. In [53] focus is also placed on communication cost with respect to cross traffic and parallel speed-up in PDNS for partitioning simple topologies. In [54], the authors demonstrate gradual improvement of network simulation partitioning following a benchmark driven approach to scalable network partitioning. In [55], the authors propose a scheme combining static partitioning and dynamic load balancing so that run-time resource usage can be reflected adaptively in load balancing.

2.4 Load Balancing in Parallel and Distributed Computing

2.4.1 Memory Load Balancing

Paucity of memory-centric load balancing also holds in the parallel distributed computing community where focus has been on computation and communication balancing [20,21]. In recent work [56], CPU-memory balancing has been studied using an adaptive application-driven approach targeted at scientific applications. In [57,58], CPU-memory balancing is evaluated from a dynamic load balancing perspective with job assignment and migration incorporating CPU and memory balancing constraints. Note that there were earlier research efforts from operating system (OS)'s point of view for utilization of distributed memory resources using a remote memory server

mechanism [59,60]. The remote memory server mechanism provides dynamic memory load balancing at OS level, with which a memory-overloaded machine can utilize other machine's unused memory transparently.

2.4.2 Static Load Balancing

Graph partitioning is a type of static load balancing, which has been widely used to map scientific computations onto parallel computers aiming for parallel speed-up [61]. Another type of static load balancing focuses on the task (or job at a coarser level) assignment problem to processors. In [62], the authors approach the problem modeling a parallel program as a task interaction graph, which is appropriate for iterative parallel programs. In [63], the authors model a parallel program as a task precedence graph, where computation time and explicit execution dependences are known a priori.

3 DASSFNET MEASUREMENT SUBSYSTEM

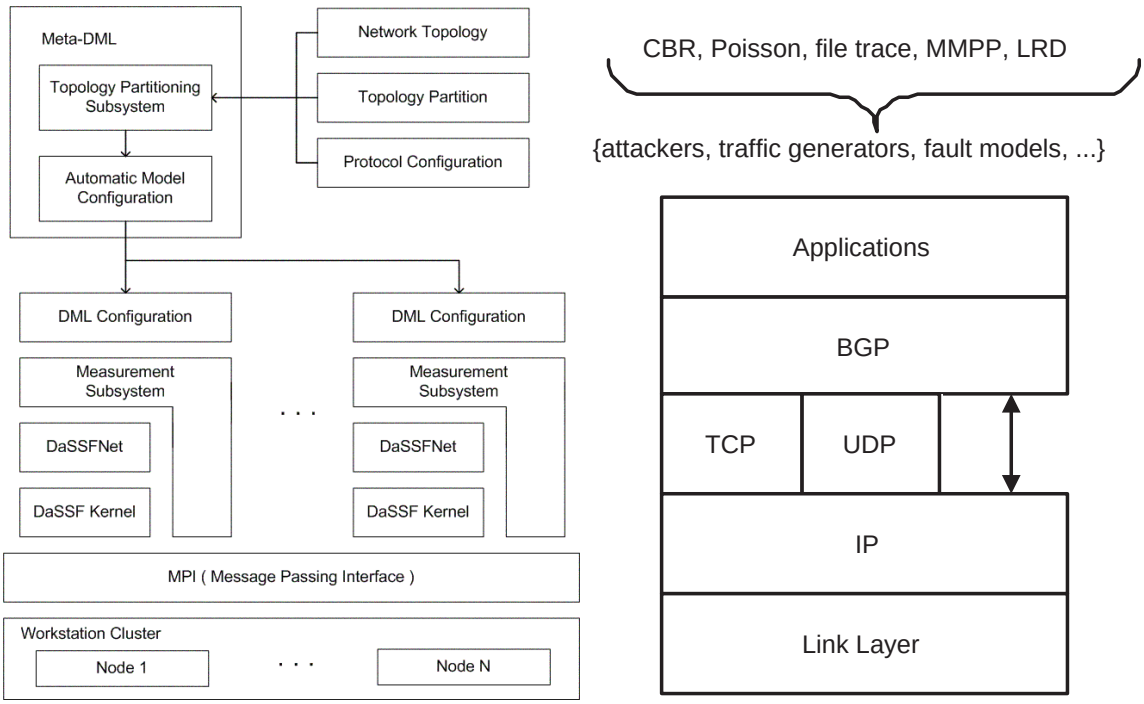
3.1 Architecture of Measurement-Enhanced DaSSFNet

3.1.1 Background on DaSSFNet

DaSSFNet is a DaSSF-based implementation of SSFNet. As a general-purpose simulation environment, the Dartmouth Scalable Simulation Framework (DaSSF) [17] provides a C++ implementation of the Scalable Simulation Framework (SSF). SSF [41] defines a unified, object-oriented application programming interface (SSF API) as a standard user interface for discrete-event simulation, considering usability and performance as important design goals. Supporting a process-oriented world-view of discrete-event simulation, SSF helps make full-fledged design and implementation of network models possible. SSFNet [16] provides simulation models of various network elements and network protocols on top of a Java-based implementation of SSF.

3.1.2 Simulator Architecture

One of the main features of DaSSFNet is that its simulation kernel, DaSSF, supports distributed PC clusters as well as shared-memory multiprocessor machines, incorporating advanced parallel simulation techniques. DaSSF uses Message Passing Interface (MPI) for synchronization and communication between simulation kernels on distributed PCs. Figure 3.1(a) shows DaSSFNet's system architecture on top of a distributed PC cluster. DaSSFNet's collection of protocol models is shown in Figure 3.1(b). DaSSFNet takes a simulation configuration written in Domain Modeling Language (DML) as input. The DML specification includes network topology, protocol configuration, and network simulation partitioning. We designed and implemented a script for automatic partitioning and model description generation. For



(a) System Architecture (b) Protocol Stack

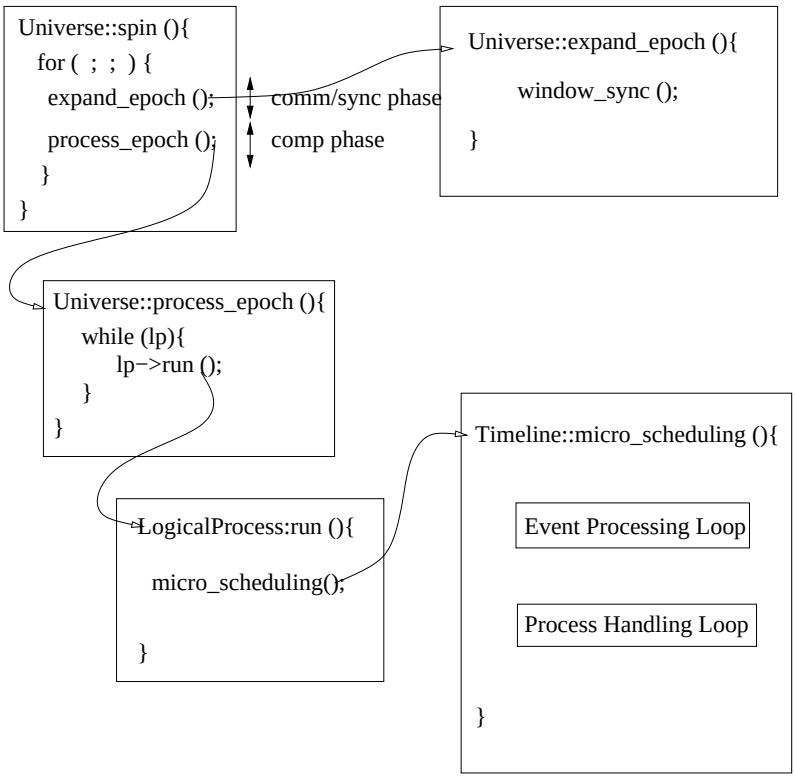
Figure 3.1. (a) DaSSFNet's system architecture on top of a distributed PC cluster with our measurement subsystem enhancement. (b) DaSSFNet protocol stack.

network simulation partitioning, the script internally calls Metis [47] library routines. Our measurement subsystem spans across the DaSSFNet protocol stack and DaSSF simulation kernel distributed across each PC. The protocol stack in DaSSFNet is comprised of IP/NIC, TCP, UDP, and BGP which are accessed via a socket API-like interface.

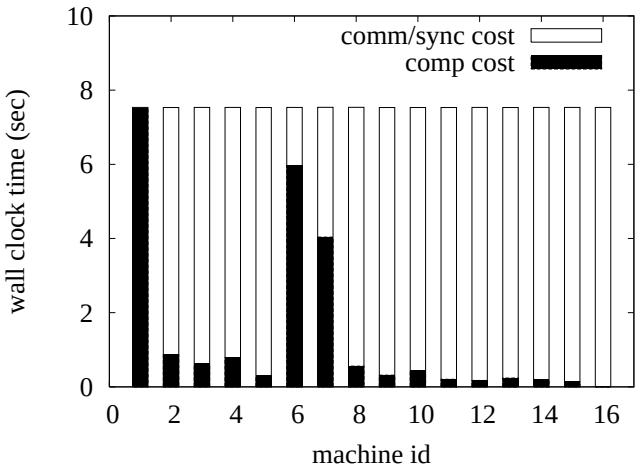
3.1.3 Overview of DaSSF's Distributed Synchronization

Distributed synchronization across network partitions is effected through a barrier mechanism that occurs at fixed time granularity (called epoch) which, by default,

is determined by the minimum link latency of inter-partition links. Although conservative, this method assures causal ordering of distributed simulation events [64, 65]. Each epoch is composed of two phases: (1) computation and (2) communication/synchronization. Figure 3.2(a) shows the structure of distributed simulation execution from a simulation kernel's perspective. During the computation phase, each simulation kernel processes simulation events scheduled to occur within the epoch and handles SSF processes nonpreemptively until there are no more ready to run processes. By definition, the computation phase proceeds independently of events processed in the same epoch at other machines. During the communication/synchronization phase, intermediate channel events—carrying remotely protocol messages—and synchronization messages are exchanged over MPI. Simulation kernels block until all of them complete the communication/synchronization phase. Figure 3.2(b) demonstrates the time dynamics of a single epoch in a BGP simulation over 16 PCs. This snapshot is obtained from our measurement subsystem, which can monitor computation and communication/synchronization costs at per-epoch granularity. Note that machine 1 with the most computation load determines the overall completion time of the epoch. Other machines are blocked in the communication/synchronization phase once they finish processing their computation load.



(a) Epoch-loop



(b) Single Epoch

Figure 3.2. (a) Structure of distributed simulation execution from the simulation kernel's perspective. (b) Demonstration of time dynamics of a single epoch across 16 PCs.

3.2 Message-centric Resource Usage Estimation

Due to the dynamic nature of network events, run-time measurement of resource consumption is required with respect to memory, CPU, and communication cost. Memory cost estimation is difficult due to message related simulation events which are generated, processed, stored, forwarded, and deleted via complex protocol interactions both inside and outside a simulation kernel. Accurate memory cost estimation requires tracking of dynamic memory allocation and deallocation, and synchronization with respect to time is critical. Since simulation time acts as a global clock for distributed network simulation, it allows synchronization of distributed simulation events and their resource footprint.

3.2.1 Message Event Types

We define major message event types which contribute to significant resource usage. They are listed in Table 3.1. We track all message related events in DaSSFNet both inside the simulation kernel and the network stack outside the kernel. Inside the DaSSF simulation kernel, three simulation events—KEVT-OUTCHANNEL, KEVT-INCHANNEL, and KEVT-CHANNEL—are used for implementing message exchange between nodes located at different machines. In the network stack outside the kernel, we define the application message event type, representing message events instantiated in a network application above TCP/IP. We define TCP, TCP-send-buffer, and TCP-receive-buffer message event types for TCP reflecting DaSSFNet's "full-fledged" implementation of TCP; similarly for UDP and UDP-receive-buffer in the case of UDP. We define IP/NIC and IP/NIC-buffer representing message events inside IP layer and network interface card (NIC) device driver buffer. We define FRAME message event type representing IP packets which consists of network protocol headers and application payload.

There are important, non-message-related events which contribute to significant resource usage. First, network protocol tables such as IP route table and BGP table

Table 3.1
Major message event types.

Inside DaSSFNet protocol stack:
Application
TCP
TCP-send-buffer
TCP-receive-buffer
UDP
UDP-receive-buffer
IP/NIC
IP/NIC-buffer
FRAME
Inside DaSSF simulation kernel:
KEVT-OUTCHANNEL
KEVT-INCHANNEL
KEVT-CHANNEL

are one of the main memory-demanding components. We take their memory usage into account when accounting memory cost. Second, non-message-related simulator events—KEVT-TIMER, KEVT-SEMSIGNAL, and KEVT-TIMEOUT—are created for implementing timer and process coordination mechanisms. We track their memory cost and computation cost.

3.2.2 Message Event Count vs. Memory Usage

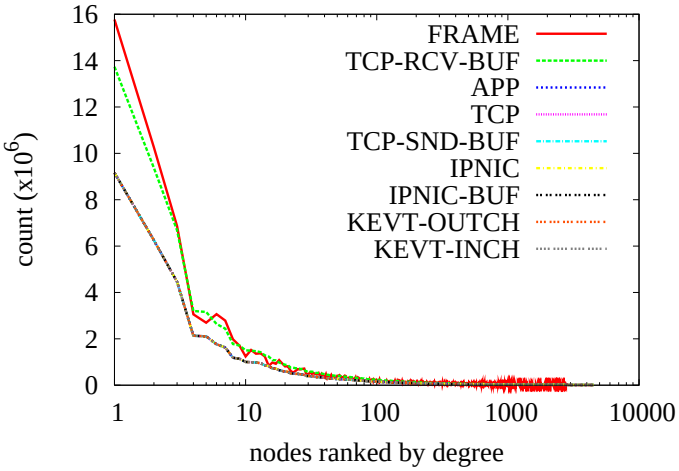
We distinguish between the count and the memory usage of message events for each message event type. Though the count of message events is often used as a measure of computation cost, it is not sufficient to be a measure of memory cost. We demonstrate a case where the count of message events is not a good measure of

memory usage in Figure 3.3 from a BGP simulation of an AS topology with 4512 nodes. Figure 3.3(a) shows per-node the total count of message events for each message type as a function of nodes ranked by their degree, where the highest degree node has the lowest rank. The abscissa is shown in log-scale to highlight high-degree nodes. In terms of event count, FRAME and TCP-receive-buffer messages are the most frequent. Figure 3.3(b) shows per-node memory usage for each message type as a function of nodes ranked by their degree. In terms of memory usage, TCP-send-buffer is the dominant component, not FRAME or TCP-receive-buffer messages. Note that we aggregate message event count and memory usage at node granularity, which is described in more detail in the next section.

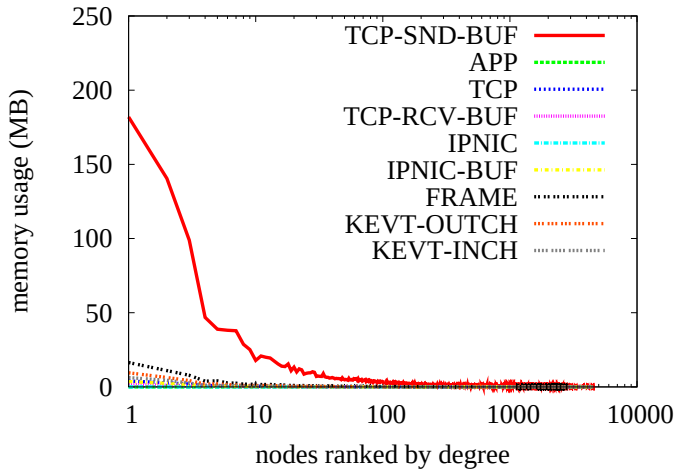
3.2.3 Message Event Aggregation and Synchronization

Memory cost of a node or a machine is estimated by aggregating message events which occur at node or machine granularity, respectively. Since memory cost is determined by the maximum footprint over time, temporal synchronization of allocation and deallocation of message events is important. We track per-node maximum memory usage for each message event type at run time which requires per-node O(1) memory complexity. We also track per-node total event count for each message event type which requires per-node O(1) memory complexity. Per-machine maximum memory usage and total event count for each message event type are measured by aggregating over nodes assigned to a machine.

Simulation time acts as a global clock for distributed network simulation. It allows synchronization of distributed simulation events and their resource footprint. In terms of wall clock time, time skew across distributed machines, which may be due to time of day clock drift for handling high priority interrupts that interfere with the system's hardware clock, is an issue for distributed time synchronization. We normalize wall clock time measurements relative to the wall clock time measured right after *main*() is called at each machine. Note that in network simulation partitioning nodes are



(a)



(b)

Figure 3.3. (a) Per-node event count for each message type. (b) Per-node memory usage for each message type.

mapped to machines, meaning that all message events of a node are simulated in one machine where the node is mapped to. Hence, when aggregating message events which occur at node or machine granularity, distributed time synchronization across machines is not an issue.

3.2.4 Message Event Evolution and Footprint

We create message event evolution diagrams that trace the time dynamics of event creation, processing, queueing, and deletion effected by complex protocol mechanisms, in particular, TCP.

Message Event

A message related event m is represented by a data structure which has a size attribute $s(m)$ that is dependent on the event type and whether it is copied or pointer (i.e., reference) based. m has a start time $t_s(m)$ at which it is generated and a finish time $t_f(m)$ at which it is deleted. Time stamps are recorded with respect to both simulation time and wall clock time. Simulation time acts as a global clock. m has a location attribute $l(m)$ representing a node where it is assigned.

Message Event Evolution Diagram

Figure 3.4 shows TCP based message event evolution diagram spanning application, DaSSFNet protocol stack, and DaSSF simulation kernel. The starting point is the application layer where a message is created and passed to TCP at simulation time t. The application message is copied to the TCP send buffer. All instances where message copying is instituted are highlighted in bold. A queued TCP message is processed after a delay d_1 determined by TCP and forwarded to the IP/NIC layer. Another copy operation results when a frame message is created which is enqueued in the IP/NIC output buffer. Queueing delay and transmission time are computed and passed to the simulation kernel along with the frame message by writing to outChannel of the network link[1]. This triggers creation of kernel event KEVT_OUTCH that is enqueued in the kernel event queue. These steps occur in zero simulation time. KEVT_OUTCH is dequeued at time $t + d_1 + d_2$ where d_2 is the sum of queueing delay

[1]outChannel and inChannel are message interface classes defined as a part of SSF API.

and transmission time. If the receiving node is on the same network partition as the sending node, KEVT_OUTCH is transformed to inChannel event KEVT_INCH that is enqueued in the kernel event queue. KEVT_OUTCH is deleted. When link latency d_3 has elapsed, KEVT_INCH is dequeued and the frame message is popped up the protocol stack at the receiving node. If the receiving node is on a different network partition, KEVT_OUTCH is mapped to an intermediate channel event that is held at the machine simulating the sending node until the epoch barrier completes (i.e., MPI packs and transfers the channel event to the machine where the receiving node is simulated). At the receiving machine the intermediate channel event is mapped to KEVT_INCH and enqueued in its kernel event queue with time stamp $t + d_1 + d_2 + d_3$.

UDP based message event evolution spanning application, DaSSFNet protocol stack, and DaSSF simulation kernel is shown in Figure 3.5. The starting point is the application layer where a message is created and passed to UDP at simulation time t. The pointer of the application message is forwarded from UDP to the IP/NIC layer of DaSSFNet. A frame message is created by copying the application message and network protocol headers, and it is enqueued in the IP/NIC output buffer. Queueing delay and transmission time are computed and passed to the simulation kernel along with the frame message by writing to outChannel of the network link. The following steps are analogous to that of TCP triggered frame messages.

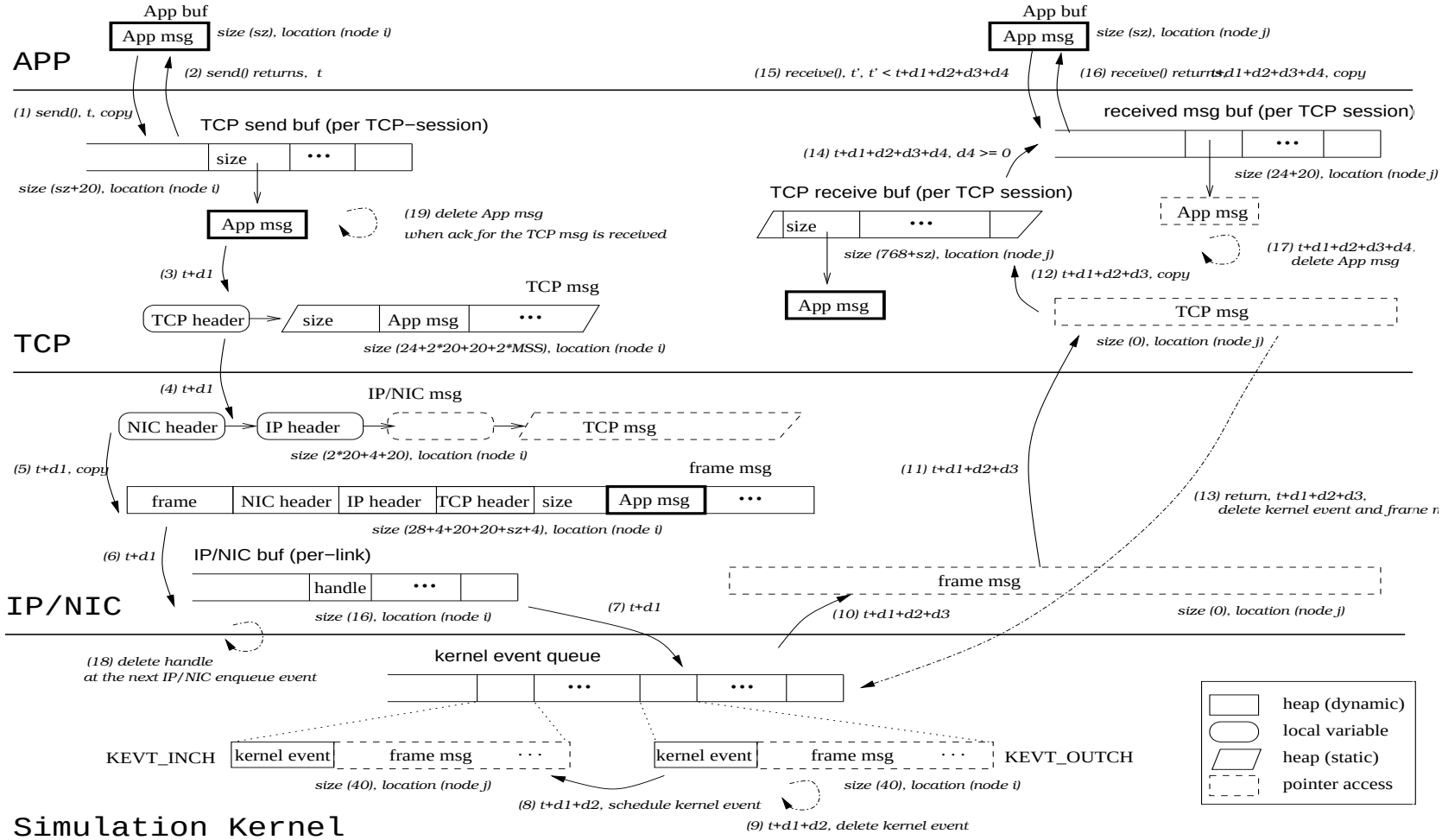


Figure 3.4. TCP based message event evolution and footprint spanning application layer, DaSSFNet protocol stack, and DaSSF simulation kernel.

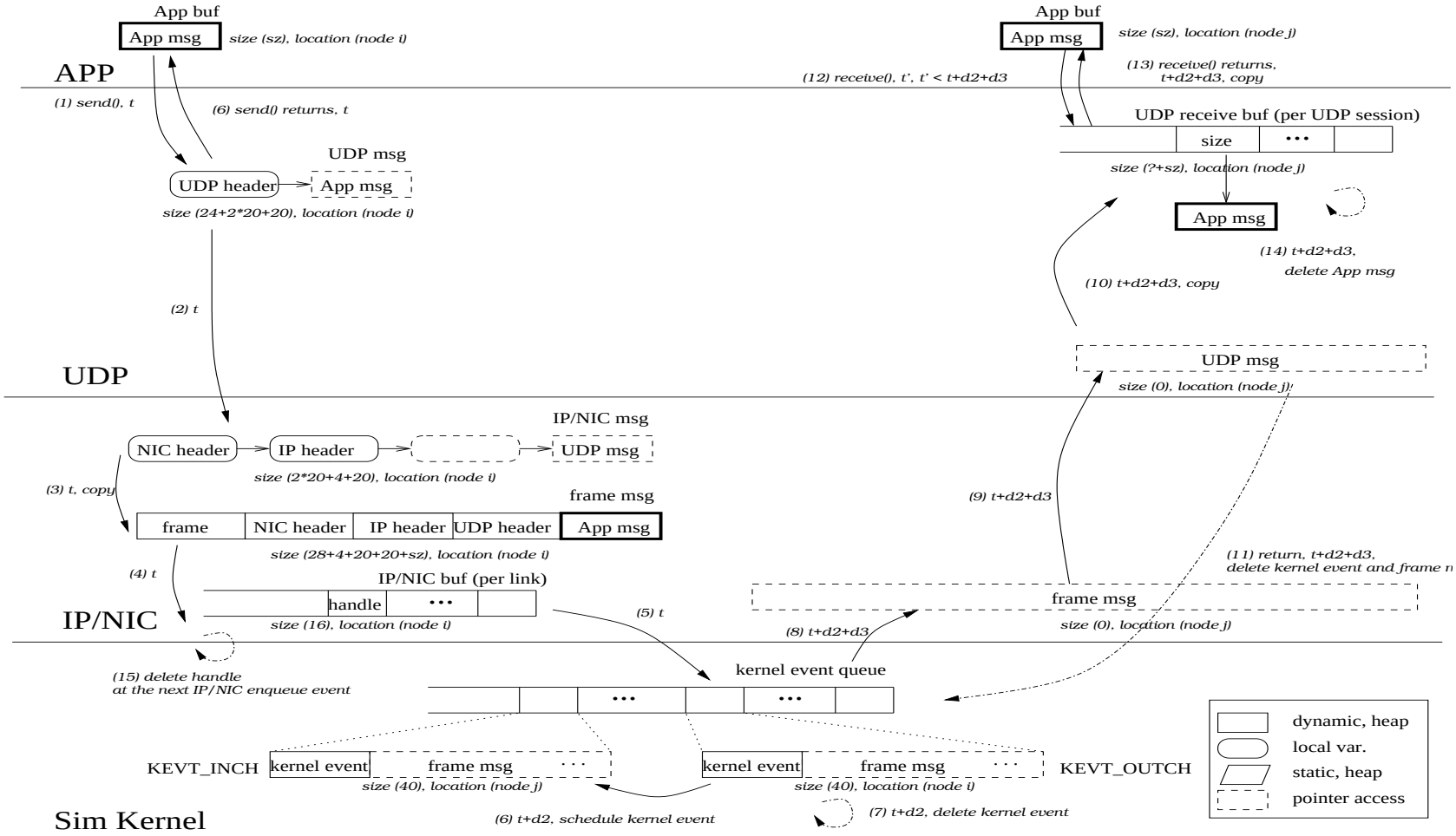


Figure 3.5. UDP based message event evolution and footprint spanning application layer, DaSSFNet protocol stack, and DaSSF simulation kernel.

3.2.5 Demonstration of Memory Cost Estimation

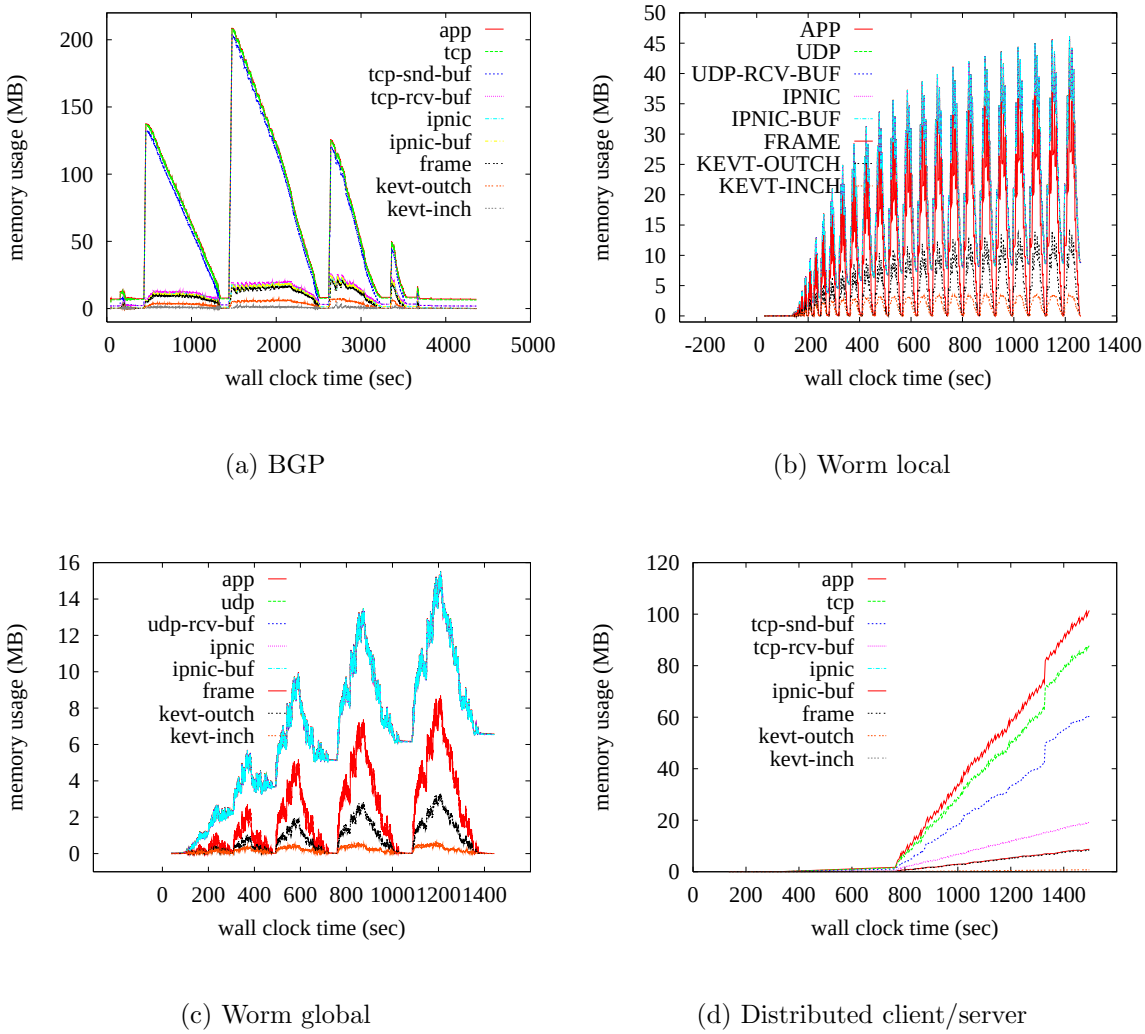
We demonstrate memory consumption of message related events over time for benchmark applications BGP, Internet worm propagation under local and global scanning, and distributed client/server. A description of the benchmark applications is given in Section 5.1.2. Figure 3.6(a) shows monitored memory consumption of message related events of a BGP simulation from a 16-machine distributed simulation. Message related events are aggregated over nodes assigned to the same machine. Figure 3.6(a) shows that memory consumption is variable reaching its peak at 1477.81 sec wall clock time (90.0062 sec simulation time). Figure 3.6(b) demonstrates memory consumption of message related events of a local worm propagation simulation. Figure 3.6(c) demonstrates memory consumption of message related events of a global worm propagation simulation. Figure 3.6(d) demonstrates memory consumption of message related events of a distributed client/server simulation.

Events are stacked (i.e., cumulative at a time instance). Memory consumption is dominated by TCP send buffer queueing in BGP. The main memory-demanding components in local worm propagation are ipnic-buf, frame, kevt-outch, and kevt-inch message types. The main memory-demanding components in global worm propagation are ipnic-buf, frame, and kevt-outch message types. The main memory-demanding components in distributed client/server are app, tcp, tcp-snd-buf, tcp-rcv-buf, and frame message types.

3.2.6 Measurement Accuracy and Overhead

Measurement Accuracy

Garbage collection issues make accurate memory cost estimation difficult. Figure 3.7 quantifies measurement accuracy of memory consumption monitoring for the BGP simulation of Figure 3.6. In our measurement enhanced version of DaSSF we use the malloc library (part of standard C library) to manage dynamic memory al-



(a) BGP

(b) Worm local

(c) Worm global

(d) Distributed client/server

Figure 3.6. Dynamic monitoring of message related events. (a) tcp-snd-buf dominates peak memory usage. (b) ipnic-buf, frame, kevt-outch, and kevt-inch dominate peak memory usage. (c) ipnic-buf, frame, and kevt-outch dominate peak memory usage. (d) app, tcp, tcp-snd-buf, tcp-rcv-buf, and frame dominate peak memory usage.

location and deallocation. We disable DaSSF's support of the Hoard multiprocessor memory allocator [66] and memory-pool mechanism for optimizing memory allocation and deallocation speed. In Figure 3.7 "in-use" indicates memory used by a user process. The gap between our estimation and in-use represents memory that is not

tracked as part of the message event evolution diagrams (i.e., BGP, TCP, and UDP). The gap tends to be small, in this instance, less than 5.2%. /proc specifies total memory that malloc has allocated. The gap between in-use and /proc arises when deallocation calls are made which do not necessarily result in actual garbage collection due to malloc's internal memory pool management. The gap between the two is not small and variable. Since at time instances of maximum memory usage they coincide, impact on memory estimation and balancing is limited.

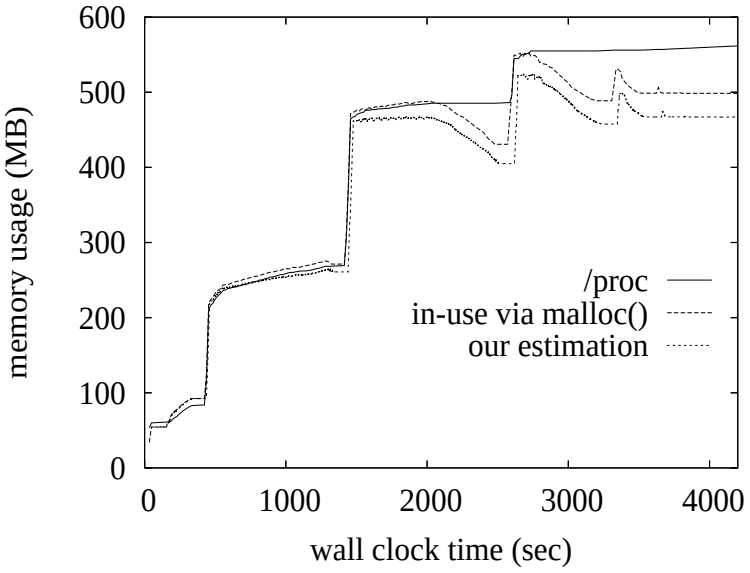
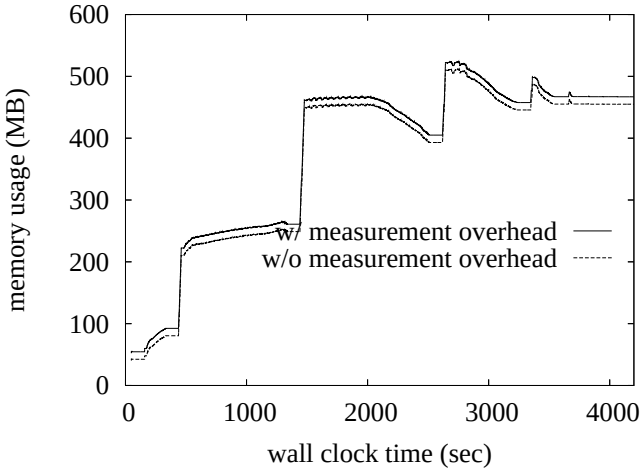


Figure 3.7. Measurement subsystem memory monitoring accuracy.

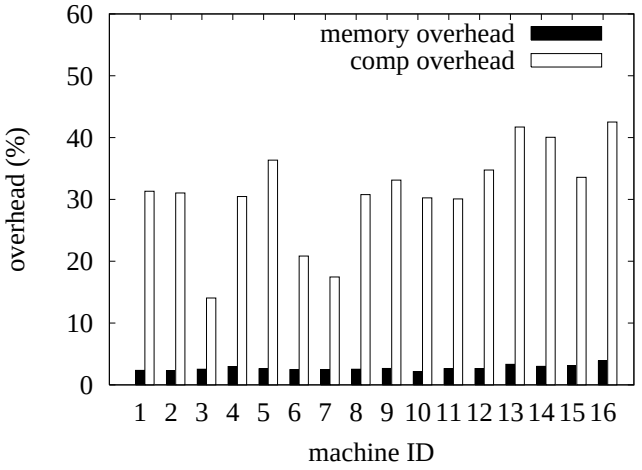
Measurement Overhead

Figure 3.8(a) quantifies measurement overhead with respect to memory consumption monitoring for the BGP simulation of Figure 3.6. The two curves show measured memory usage with and without measurement overhead needed to maintain per-node memory, computation, and communication load. Overhead is small due to constant per-node space needed of memory cost and computation cost. The same goes for communication cost which requires constant memory per link. In power-law measurement

topologies the number of links is about 2–3 times the number of nodes. Figure 3.8(b) summarizes measurement subsystem's monitoring overhead with respect to memory usage and completion time at each participating machine from the same BGP simulation. The memory overhead is about 5% across all machines, but the computation overhead is about 30% across all machines. The non-trivial computation overhead is due to detailed per-epoch computation and communication/synchronization cost measurement. It can be turned off for memory and CPU balancing purposes.



(a)



(b)

Figure 3.8. (a) Measurement subsystem memory monitoring overhead. (b) Measurement subsystem monitoring overhead with respect to memory usage and completion time at each participating machine.

4 MEMORY BALANCING PROBLEM

4.1 Space Efficiency in Per-partition Memory Cost Estimation

4.1.1 Per-node Memory Cost Estimation

Computation cost in network simulation is dominated by messages, being proportional to the total number of messages processed over the course of a simulation. If processing cost varies significantly between message types, weighting may be necessary to arrive at normalized cost. The processing cost, C_i, associated with a single node i—for sending, receiving, and processing messages—is proportional to the sum of all messages $X_i(t)$ processed at the node over time, $C_i = \sum_t X_i(t)$, which requires constant space to record. The memory cost, M_i, of node i depends on the maximum number of messages over time, $M_i = \max_t X_i(t)$, which can significantly differ from the total C_i. If the memory footprint of different message types varies, weighting is necessary for accurate accounting. All else being equal, we expect M_i to be superior for memory balancing while C_i is expected to favor CPU balancing.

4.1.2 Per-partition Memory Cost Estimation

An additional issue that arises in the case of maximum cost metric, but not total cost metric, is when network partitioning based on M_i provided by the measurement subsystem is carried out. Suppose that a partitioning algorithm $\mathcal{A}$ which receives $M_1, \ldots, M_n$ as node weight input assigns two nodes i and j to the same partition based on their sum $M_i + M_j$. The problem with doing so is that although their individual memory peaks are M_i and M_j, their collective memory footprint is given by $\max_t\{X_i(t)+X_j(t)\}$ which need not equal $\max_t X_i(t)+\max_t X_j(t)$. This is depicted in Figure 4.1 for memory consumption dynamics of two nodes in a worm propagation

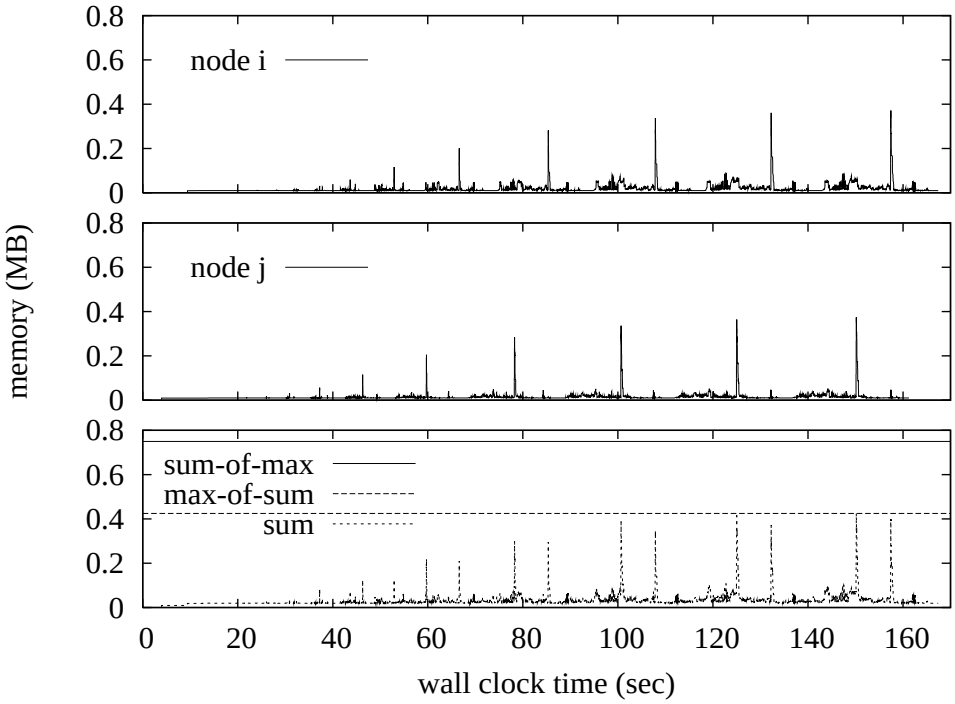


Figure 4.1. Network partitioning: sum-of-max vs. max-of-sum problem.

simulation where their memory peaks are not synchronized—both wall clock and simulation time—leading to significant overestimation of sum-of-max over max-of-sum. This problem does not arise for the total cost metric due to its closure property. The sum-of-max vs. max-of-sum problem cannot be addressed by logging per-node time series of memory consumption due to prohibitive space complexity (i.e., $O(nT)$ where T is simulation time). We use M_i as input to partitioning algorithms to evaluate memory and CPU balancing performance and compare it against C_i. In Section 4.1.4 we explain why M_i is effective despite the sum-of-max overestimation problem.

4.1.3 Relative Memory Balancing

As illustrated in the sum-of-max overestimation problem shown in Figure 4.1, accurate estimation of per-partition memory cost—the max-of-sum, which we name absolute memory cost estimation—is inherently difficult. In this dissertation, we propose relative memory balancing, where we aim to balance the sum-of-max across all

machines. In the case of relative memory balancing, absolute memory cost estimation is not necessary. Although sum-of-max does not achieve absolute memory cost estimation, performance results in Chapter 5 show that it is sufficient to achieve balancing of per-partition memory cost. Figure 4.2 compares sum-of-max (our estimation) and max-of-sum (actual memory usage) as a function of machine ID for worm local simulation with 21460-node topology using 10 machines. We observe that relative memory balancing is able to approximate the performance of absolute memory balancing. We describe relative memory balancing with M_i in Section 4.1.4.

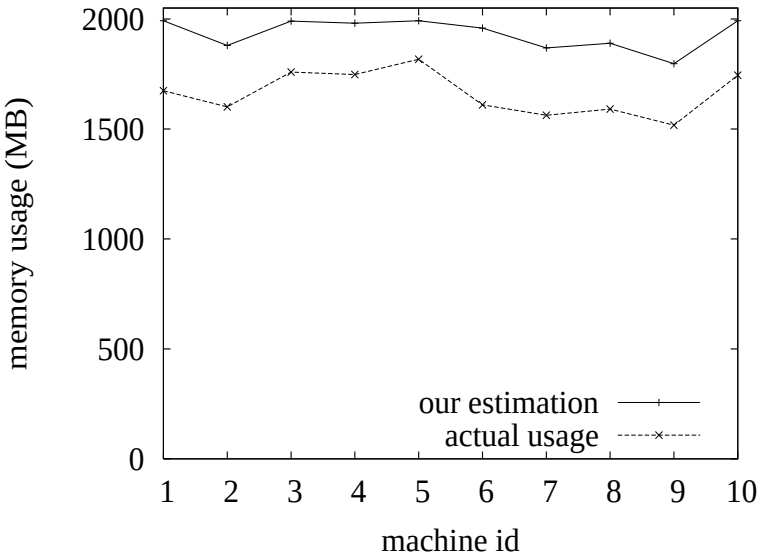


Figure 4.2. The sum-of-max (our estimation) and max-of-sum (actual memory usage) as a function of machine ID for worm local simulation with 21460-node topology using 10 machines.

4.1.4 Effect of Scale

In Section 4.1.2 we described the sum-of-max vs. max-of-sum problem where the maximum memory metric can significantly overestimate actual memory load during partitioning. Despite the fact that the total cost metric does not have this problem due to its closure property, performance results presented in Chapter 5 show that

the maximum cost metric, overall, outperforms the total cost metric with respect to memory balancing. This is due to two factors.

First, in large-scale network simulation where the number of nodes n is large, the law of large numbers helps mitigate the sum-of-max overestimation problem by averaging out the time instances where individual nodes reach peak memory. Figure 4.3 quantifies this effect by showing the correlation coefficient between the sum-of-max and the max-of-sum across machines as problem size is increased from 300 to 3213 for worm global in random graphs. The number of partitions is held constant at 24 (i.e., 24 machines participate in distributed simulation). This demonstrates the increasing prowess of sum-of-max in predicting actual memory consumption (i.e., max-of-sum) across machines as n is increased. To highlight the law of large number effect, we use random graphs of the same edge density as power-law Internet measurement graphs where all nodes have approximately similar degrees. This removes complications introduced by high-degree nodes in power-law graphs whose memory peaks are significantly higher than other nodes (the "elephants and mice" feature).

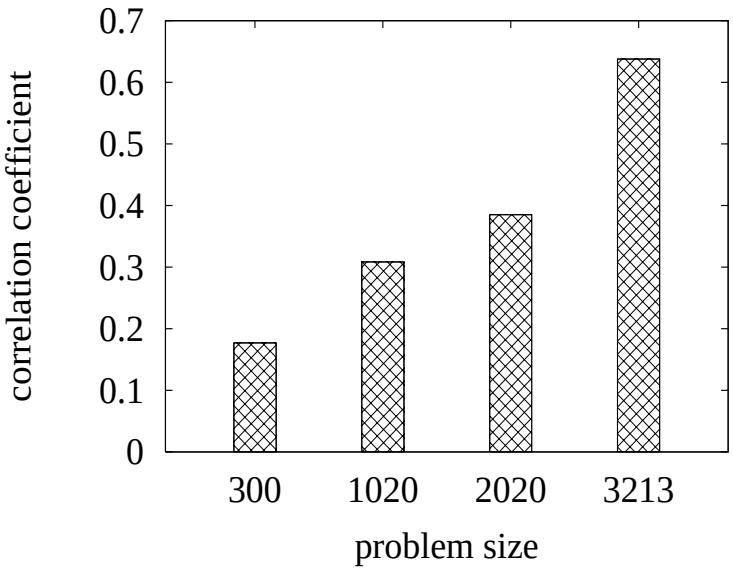


Figure 4.3. Correlation between sum-of-max and max-of-sum balancing as a function of problem size in random graphs.

The second factor is the increased role of table memory in large problem instances. In worm local, per-node route table size is fixed (in BGP table sizes are variable), and in the absence of table memory reduction optimization per-node table memory grows linearly in n. Figure 4.4 shows per-node cumulative M_i load distribution as a function of node rank—nodes are ranked by their degree with rank 1 indicating a node with the largest number of neighbors—for a worm local simulation. The abscissa is shown in log-scale to highlight the load values of high-degree nodes. We observe that per-node message memory is about twice that of table memory. Although this is true on a per-node basis, when multiple nodes are mapped to a common partition the sum-of-max overestimation problem kicks in which reduces their actual collective message memory footprint.

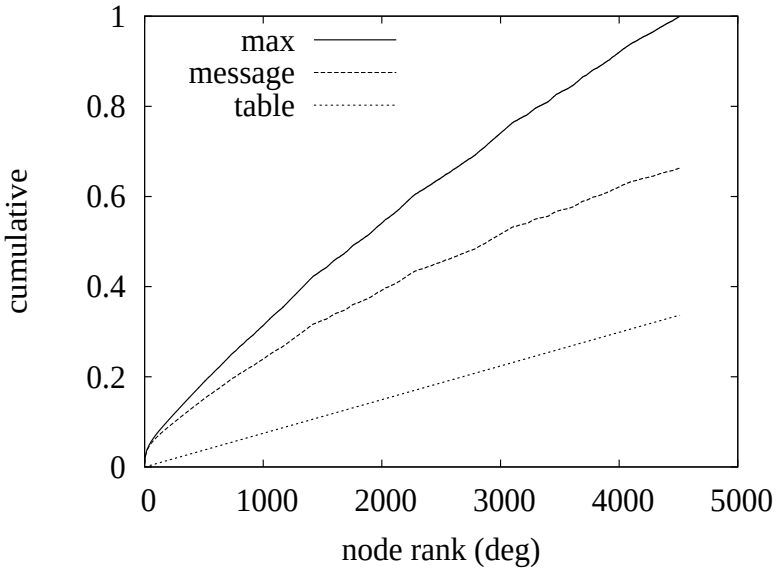


Figure 4.4. Per-node cumulative M_i load distribution: worm local.

As shown in Figure 4.5, the per-partition make-up of message vs. table memory of Figure 4.4 is about half-half. This dampens node load skewness during partitioning which is conducive to easier memory balancing[1].

[1]If table memory were dominant, memory balancing would be trivial since all nodes would have approximately equal weight.

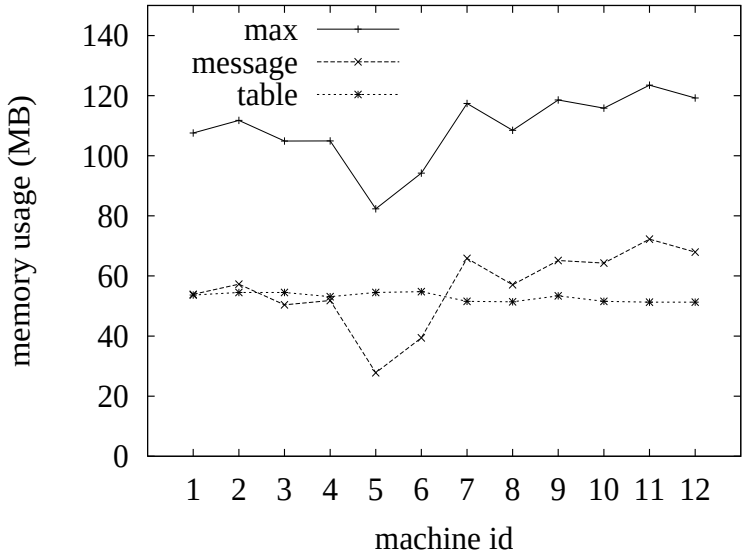


Figure 4.5. Per-partition make-up of message vs. table memory: worm local.

4.2 Impact of Power-law Connectivity on Memory Balancing

4.2.1 Overview of Metis's Multilevel Recursive Partitioning

Metis [47] takes a graph $G = (V, E)$ and the number of partitions k as input. Weights on nodes and weights on edges can be given optionally. The default weight of each node and each edge is set 1. Metis generates a network partitioning $f : V \rightarrow [1..k]$, where it tries to balance the sum of node weights in each partition over partitions and minimize weighted edgecut. `METIS_WPartGraphKway()` implements the multilevel graph partitioning described in [21] which consists of three phases—coarsening, initial partitioning, and uncoarsening/refinement. Metis provides multiple algorithms for coarsening and uncoarsening/refinement phases. We use the default algorithm for each phase—Sorted Heavy-Edge Matching (SHEM) for coarsening and random boundary refinement that also minimizes the connectivity among the subdomains for uncoarsening/refinement.

First, the coarsening phase consists of multiple levels. At each level, a smaller scale graph $G'' = (V'', E'')$ is generated from an input graph $G' = (V', E')$ by finding

a maximal matching, where a node is matched to an unmatched node with heavier edge. At the first level, $G' = G$. This phase ends when (a) the size of the smaller scale graph is below a certain number of nodes, (b) a maximal matching fails to reduce the size of the input graph effectively more than a certain percentage, or (c) the smaller scale graph is too sparse, not having enough edges to use for finding a maximal matching at the next level.

Second, the initial partitioning phase takes the smallest-scale graph generated by the coarsening phase $G^* = (V^*, E^*)$ and the number of partitions k as input. It outputs a network partitioning $f : V^* \rightarrow [1..k]$. It uses the multilevel recursive bisection described in [20], bisecting the input graph—splitting the input graph into two—until the total number of partitions reaches k. In each bisection, additional coarsening, initial two-way partitioning, and uncoarsening/refinement algorithms are applied.

Third, the uncoarsening/refinement phase consists of multiple levels. At each level, a smaller-scale graph G'' is projected to the next level finer graph G'. Then, a randomized boundary refinement algorithm, that reduces the connectivity among partitions, is performed. An important role of this phase is balancing per-partition sum of node weights. This phase ends when the given graph is projected back to the original graph (i.e., $G' = G$).

4.2.2 Benchmark-based Memory Balancing

In a typical scenario, a user runs a partitioning algorithm $\mathcal{A}$ whose input is a graph $G = (V, E)$, a vector of node weights W_V, a vector of edge weights W_E, and the number of partitions k. The output is a function $f : V \rightarrow [1..k]$, mapping each node in V to a partition ID, where the sum of node weights in each partition is balanced across partitions.

In our approach, we need an additional benchmark-based cost estimation step, where we estimate W_V and W_E using a benchmark simulation run. Note that we set

all edge weights 1, unless otherwise stated. First, we run a partitioning algorithm $\mathcal{A}$ inputting a graph $G = (V, E)$, a vector of node weights whose elements are all 1, and the number of partitions k. We call the output function the initial partitioning. Next, we run a benchmark simulation, where nodes are mapped to partitions with respect to the initial partitioning. At the end of a simulation, we obtain per-node memory cost M_i and per-node CPU cost C_i. For memory balancing, we provide M_i to the partitioning algorithm $\mathcal{A}$ as W_V. For CPU balancing, we give C_i to the partitioning algorithm $\mathcal{A}$ as input. Per-node memory cost M_i and per-node CPU cost C_i are logical costs that do not depend on partitioning; hence a uniform weight vector for the initial partitioning suffices. Figure 4.6 illustrates the memory load balancing procedure which includes the benchmark-based per-node cost estimation step.

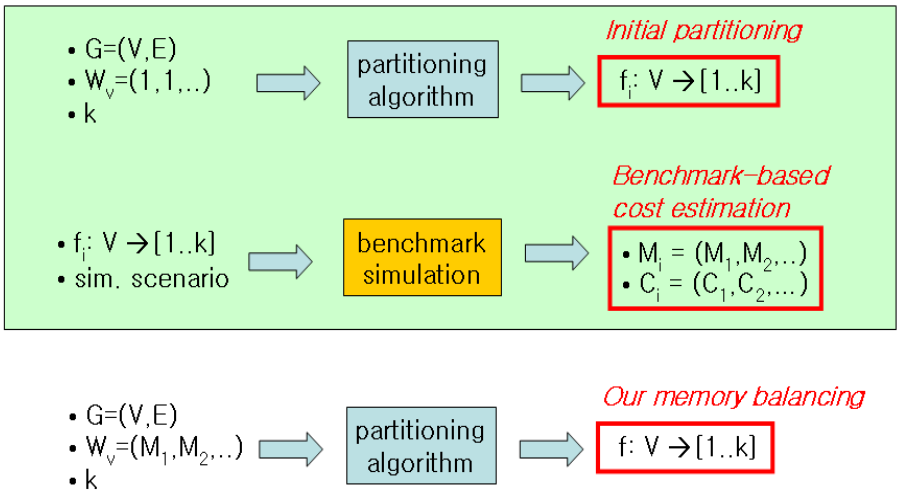


Figure 4.6. Memory load balancing procedure which includes the benchmark-based cost estimation step.

A drawback of our benchmark-based cost estimation is that the completion time of a benchmark simulation run can be significant. A given simulation scenario may require significant computation inherently, or CPU load imbalance may delay the overall simulation completion. In some cases, memory load imbalance may cause the benchmark simulation to crash before its completion—if VM paging is disabled—or to run prohibitively long if VM paging is enabled and trashing occurs.

We resolve this issue by showing that per-node memory cost M_i and per-node CPU cost C_i obtained from a partial run are still effective in memory balancing and CPU balancing, respectively. In particular, in the cases when thrashing occurs during the benchmark simulation run due to memory load imbalance, we show that node weights measured at the starting point of thrashing suffice for effective memory and CPU balancing.

4.2.3 Impact of Power-law Connectivity

Power-law networks have a distinct connectivity structure where a few high degree nodes are connected to many low-degree nodes and many low-degree nodes are connected to a few high degree nodes resulting in a power-law relation between the degree and frequency of nodes of that degree: $\Pr\{\deg(u) = k\} \propto k^{-\beta}$. Figure 4.7 shows examples of power-law network and random network. Figure 4.7 left shows a 300-node subgraph of the 3023-node Internet AS topology dated on 1997/11/08 based on RouteViews/NLANR measurement data [23]. Figure 4.7 right shows a random network which has the same number of nodes and the same number of edges as the power-law network.

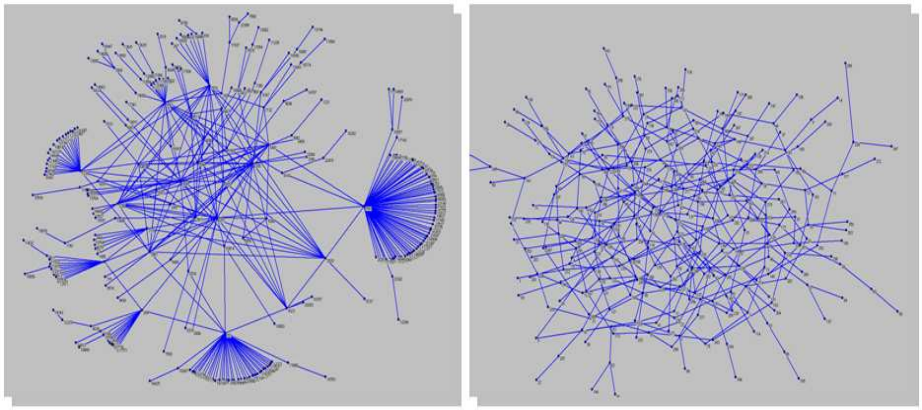


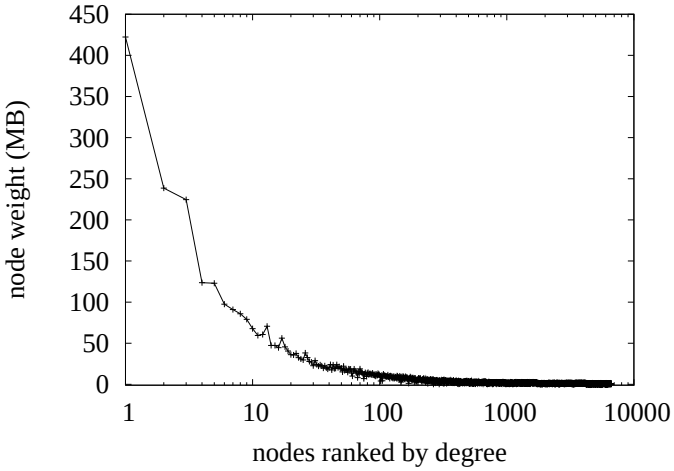
Figure 4.7. Power-law network vs. random network.

The problem of partitioning a power-law network with balancing per-partition total weight is difficult because of the skewness of per-node memory and CPU cost distributions. The per-node CPU cost at high degree nodes is higher than that of low degree nodes since high degree nodes tend to process forwarded messages for many neighboring low degree nodes. Hence, messages are queued at high degree nodes, and it causes the per-node memory cost at high degree nodes to be higher than that of low degree nodes. Figure 4.8(a) shows an example of per-node memory cost as a function of nodes ranked by their degree from a BGP simulation of an AS topology with 6582 nodes. The abscissa is shown in log-scale to highlight high-degree nodes. We observe a high skew in per-node memory cost distribution. Figure 4.8(b) shows an example of per-node CPU cost as a function of nodes ranked by their degree from the same BGP simulation. The abscissa is shown in log-scale to highlight high-degree nodes. We observe a high skew in per-node CPU cost distribution as well.

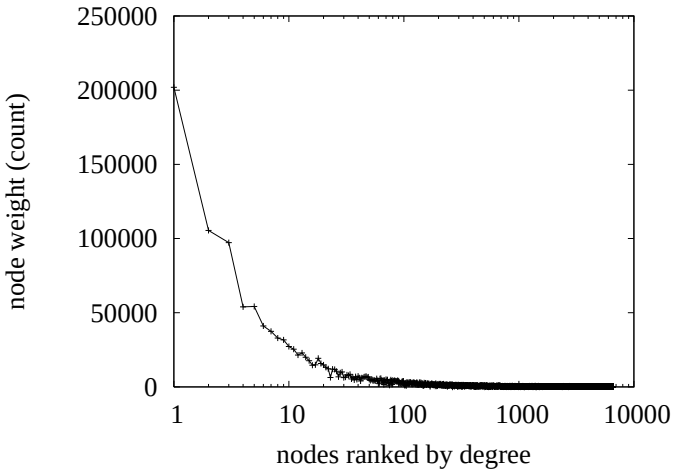
4.2.4 Custom Network Partitioning

In selecting a network partitioning algorithm, we first devised a heuristic power-law network partitioning algorithm, which targets power-law networks and focuses on balancing per-partition sum of node weights without specific consideration for minimizing edgecut.

Our heuristic consists of three parts: power-law balancing, post-processing, and refinement. The power-law balancing algorithm takes as input a network topology $G = (V, E)$, per-node cost estimation W_v, and the number of partitions k. It outputs a partitioning instance, mapping each node into a partition ID in the range $[1..k]$. The power-law balancing algorithm tries to balance the sum of node weights across partitions, placing a node with heaviest weight into a partition with the least sum of node weights. In power-law networks, per-node memory or CPU cost distribution tends to exhibit high skewness as a consequence of the underlying topology's connectivity structure. Hence, a key element of the heuristic is to distribute high-degree



(a)



(b)

Figure 4.8. (a) Per-node memory cost skew from a BGP simulation of an AS topology with 6582 nodes. (b) Per-node CPU cost skew from a BGP simulation of an AS topology with 6582 nodes.

nodes evenly across the partitions, reducing the chance of putting heavy-weight nodes into the same partition to the extent possible. Figure 4.9 shows a pseudo code of the algorithm.

```
powerlaw-balancing
Input: G=(V,E), Wv, k
Output: f:V → [1..k]
Algorithm:
▷ Sort each node i by its weight in non-increasing order
▷ Initialize each machine j's sum of node weights
    j.sumweight ← 0;
▷ Put each machine j into a min priority queue (PQ) w.r.t. j.sumweight
▷ Visit each node i in sorted order starting from the highest
    j ← PQ.get();
    j.sumweight  += i.weight;
    i.part = j;
    PQ.put(j);
```

Figure 4.9. Pseudo code of the power-law balancing algorithm.

We have seen cases where memory balancing with the M_i cost metric achieves balanced memory usage overall, but there is a partition with non-negligible higher memory usage than others. In the cases we have seen, such a partition contains heavy-weight nodes as well as light-weight nodes. Under our per-partition cost estimation, the case of a partition containing a node with heavy-weight w and the case of a partition containing many light-weight nodes where the sum of node weights is w are not distinguishable. The problem here is that when there are two such partitions—one containing a heavy-weight node and the other containing light-weight nodes equal total weight, the actual per-partition memory load is not balanced.

Targeting such cases, we devised a post-processing algorithm to improve memory balancing by redistributing light-weight nodes at overloaded machines with respect to peak memory usage measurement onto underloaded machines. Figure 4.10 shows a pseudo code of the post-processing algorithm. It takes a network topology $G = (V, E)$, the number of partitions k, the network partitioning f, per-node memory cost M_i,

```
post-processing
Input: G=(V,E), k, f:V→[1..k], Mi, per-machine peak memory usage, threshold
Output: f':V → [1..k]
Algorithm:
▷ Set node i's partitioning assignment i.part using the input f
▷ Set machine j's peak memory usage j.memusage using the input
▷ Classify each node into big or small using the input threshold
    if i.weight > threshold then i.class ← big;
    else i.class ← small;
▷ Calculate per-machine average weight of small nodes in each machine
▷ Assign each node i the average node weight
    if i.class == small then i.avgweight ← i.part.avgweight;
    else i.avgweight ← 0;
▷ Sort each node i by its average weight in non-increasing order
▷ Put each machine j into a min priority queue (PQ) w.r.t. j.memusage
▷ Classify each machine into overloaded or underloaded w.r.t. average usage
    if j.memusage > avgmemusage then j.overloaded ← true
    else j.overloaded ← false
▷ Visit each node i in sorted order starting from the highest
    if i.class == big then do nothing;
    else if i.class == small and i.part.overloaded == false then do nothing;
    else
          j ← PQ.get();
          j.memusage += i.avgweight;
          i.part.memusage -= i.avgweight;
          PQ.put(j);
          PQ.put(i.part);
```

Figure 4.10. Pseudo code of the post-processing algorithm.

per-machine peak memory usage, and a threshold to classify nodes into heavy-weight nodes and light-weight nodes. It outputs another network partitioning instance f'. The algorithm first loads the given partitioning assignment for each node and peak memory usage for each machine. It classifies each node into a heavy (big) one or a light (small) one. It calculates the average weight of light nodes in each partition, and assign each node the average weight; in the cases of heavy nodes, 0 is assigned. Then visiting each light node in sorted order from the heaviest average weight, the algorithm moves a node into a machine with the least memory usage. If the currently visited node is not in an overloaded machine, the node is skipped.

When designing the power-law balancing algorithm and post-processing algorithm, we focused only on node weights, ignoring connectivity information between nodes. As a result, a degree-1 node may be located in a partition different from the partition where its neighboring node is located. In this case, messages between the two nodes have to travel across different partitions. Inter-partition communication results in network communication which incurs higher overhead and delay. We designed a refinement algorithm, which is aimed at reducing inter-partition communication without negatively affecting memory balancing. Figure 4.2.4 presents a pseudo code of the refinement algorithm. It takes a network topology $G = (V, E)$, the number of partitions k, and the network partitioning f. It outputs another network partitioning instance f'. Two matrices $A[\ ][\ ]$ and $B[\ ][\ ]$ are used. $A[i][j]$ stores the number of degree-1 nodes at machine i such that one of its neighboring nodes exists at machine j. $B[i][j]$ stores the number of degree-1 nodes to be exchanged between i and j as a result of this algorithm. The algorithm first initializes $A[\ ][\ ]$, visiting each node. Traversing $A[\ ][\ ]$, the algorithm fills in $B[\ ][\ ]$, such that $B[i][j]$ contains the minimum of $A[i][j]$ and $A[j][i]$. Lastly, the algorithm exchanges $B[i][j]$ number of degree-1 nodes between i and j, traversing $B[i][j]$. Assuming that weights of degree-1 nodes are uniform, we try to maintain per-partition sum of node weights the same before and after the refinement algorithm.

```
refinement-deg1
Input: G=(V,E), k, f:V←[1..k]
Output: f':V → [1..k]
Data structure: A[1..k][1..k], B[1..k][1..k]
Algorithm:
▷ Visiting each node i, fill A[1..k][1..k]
    if i.deg == 1 and i.part != i.adj.part then A[i.part][i.adj.part]++;
    else do nothing;
▷ Traverse A[1..k][1..k], fill B[1..k][1..k] such that
    B[i][j] ← min(A[i][j],A[j][i]);
▷ Traversing B[1..k][1..k],
    exchange B[i][j] number of degree-1 nodes between i and j;
```

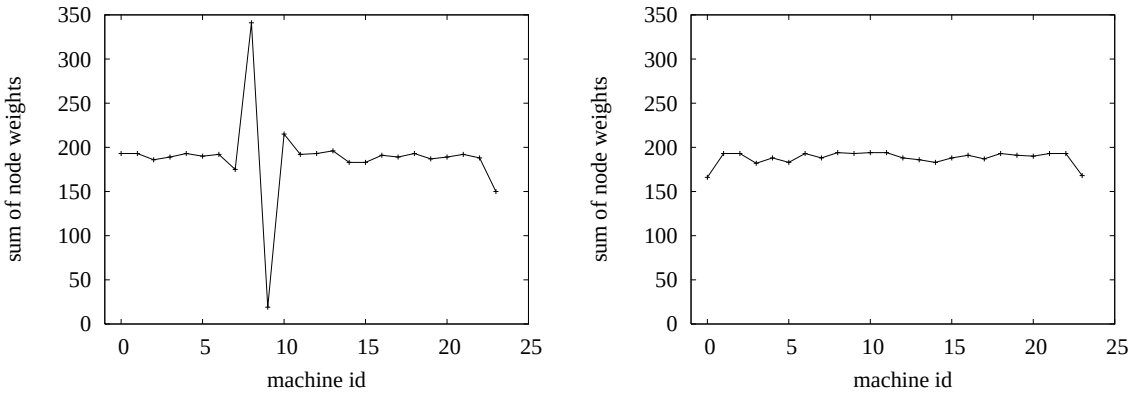
Figure 4.11. Pseudo code of the refinement algorithm.

4.2.5 Issues with Metis's Memory Balancing

In this section, we describe Metis's problems in its implementation which affect memory balancing performance. After fixing the problems, Metis produces comparable memory balancing performance to our power-law network partitioning algorithm.

1. Problem in the uniform ("1") node weight case: `EliminateComponents()`, called during the uncoarsening/refinement phase, tries to reduce the number of connected components within each partition by moving nodes to other partitions. When the function tries to move a node to another partition, it makes sure that the resulting sum of node weights at the other partition does not exceed a certain limit. However, if a node's weight is less than `delta`, which is set to 5, the node is allowed to be moved even though the resulting sum of node weights at the other partition exceeds the given limit.

   This is an issue in the case of our initial partitioning which uses the uniform ("1") node weight, and the problem is more prevalent and severe in power-



(a) (b)

Figure 4.12. Worm global simulation of 4512-node topology with k=24. The uniform ("1") node weights are used for network partitioning. (a) Before fixing the problem. (b) After fixing the problem.

law networks than random networks, since many nodes are low degree and it is common for low-degree nodes to be located at a different partition from those of their neighboring high-degree nodes. Hence, we removed the `delta` mechanism. Figure 4.12 demonstrates this problem using a worm global simulation of 4512-node AS topology with k=24. Figure 4.12(a) shows per-partition sum of node weights as a function of machine ID before fixing the problem. Figure 4.12(b) shows the result after fixing the problem. We observe that the imbalance at machine 8 and machine 9 in Figure 4.12(a) is smoothed out in Figure 4.12(b).

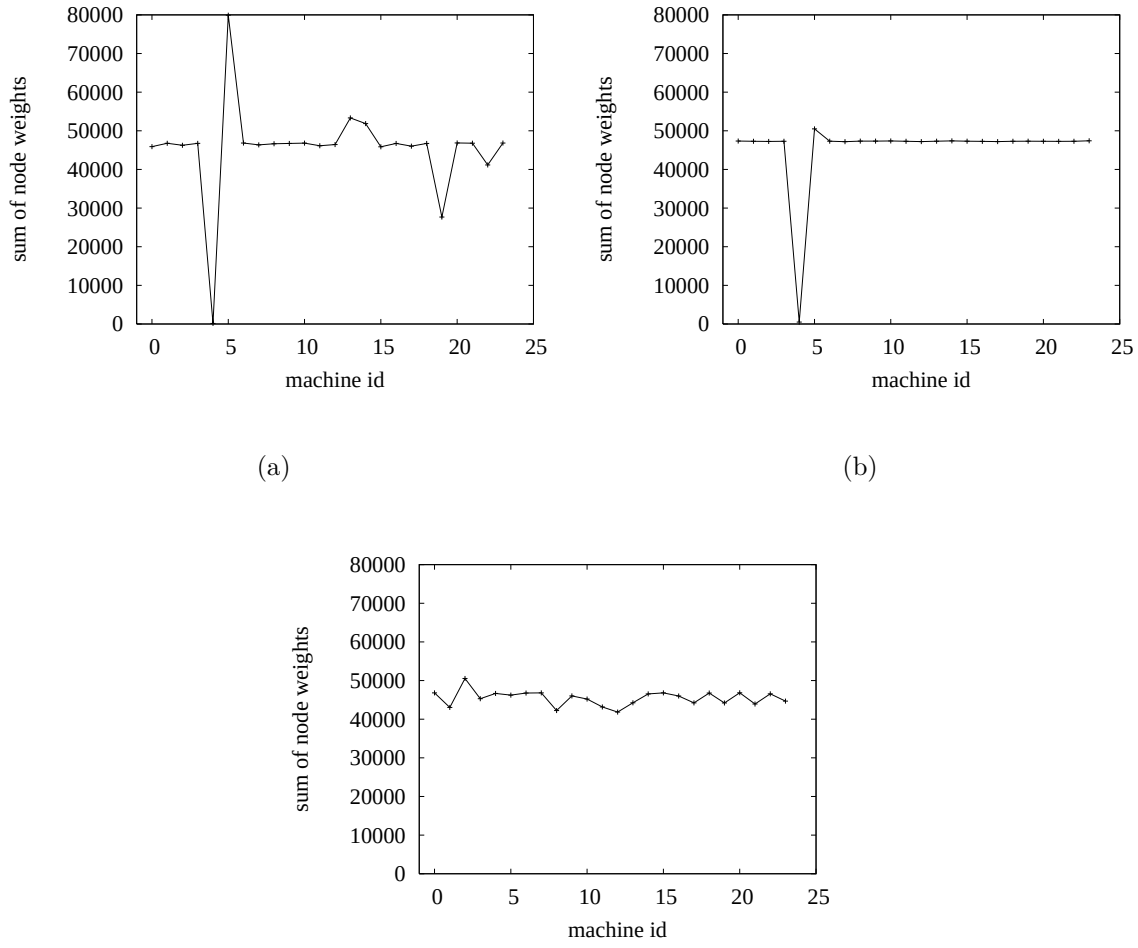
2. Problem in balancing per-partition sum of node weights during the uncoarsening/refinement phase: We found that the uncoarsening/refinement phase does not balance per-partition sum of node weights as well as our power-law network balancing algorithm. We tracked down two sources of this problem. The first is in a programming optimization while implementing a main balancing condition in `Random_KWayEdgeRefineMConn()`. The second is a programming error in `MoveGroupMConn()`. Both of them are minor problems, but the impact to

memory balancing is significant. Figure 4.13 demonstrates this problem using a worm global simulation of 4512-node AS topology with k=24. Figure 4.13(a) shows per-partition sum of node weights as a function of machine ID before fixing this problem. Figure 4.13(b) shows the result after fixing the problem. We observe that the peak at machine 5 in Figure 4.13(a) is smoothed out in Figure 4.13(b).

3. Starvation problem: We have frequently encountered cases where some machine has few nodes assigned to it and the sum of node weights is negligible. We say such partitions have a "starvation problem". Figure 4.13(b) has the starvation problem at machine 4. The starvation problem occurs during the initial partitioning phase, more precisely the initial two-way partitioning (`GrowBisection()` phase).

   `GrowBisection()` takes a graph and produces a bisection using a region growing algorithm. It first puts all nodes in one partition, moves a randomly-selected node into the other partition, and does breath-first-search (BFS) starting from the randomly-selected node trying to move each visited node into the other partition. A node is allowed to be moved if the resulting sum of node weights at the initial partition remains above a certain limit. Metis tries to improve completion time by stopping BFS if a trial of moving a node fails.

   This becomes a problem in the case when a star topology is given as input to `GrowBisection()`. In power-law networks, the high-degree center nodes of star-like clusters generally are heavy-weight whereas low-degree nodes are light-weight. Let us consider a case when a center node's weight is around half of the total weight of all nodes in a given star topology. When the center node is visited after visiting a low-degree node, the trial of moving the center node may fail. Due to Metis's optimization BFS stops even though the total weights of the two partitions are not balanced yet. Hence we removed this feature from Metis.



(a)

(b)

(c)

Figure 4.13. Worm global simulation of 4512-node AS topology with k=24. Per-node memory cost estimation is used for network partitioning. (a) Before fixing the problem in balancing per-partition sum of node weights during the uncoarsening and refinement phase. (b) After fixing problem in balancing per-partition sum of node weights during the uncoarsening and refinement phase. (c) After fixing starvation problem during the initial partitioning phase.

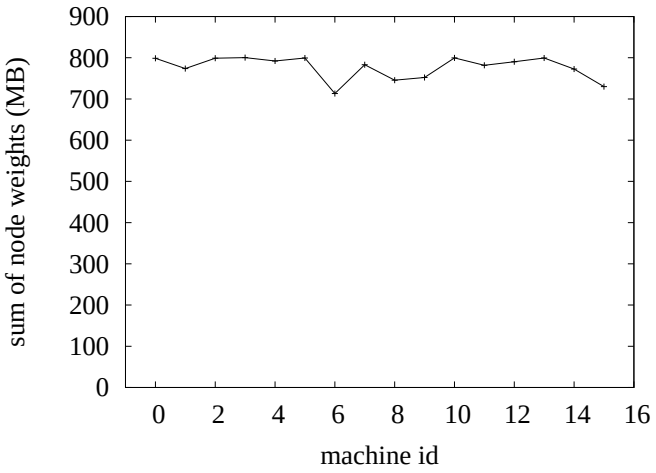
Figure 4.13(c) shows the result after removing this feature where starvation is gone.

4.2.6 Memory and CPU Balancing Performance of Metis

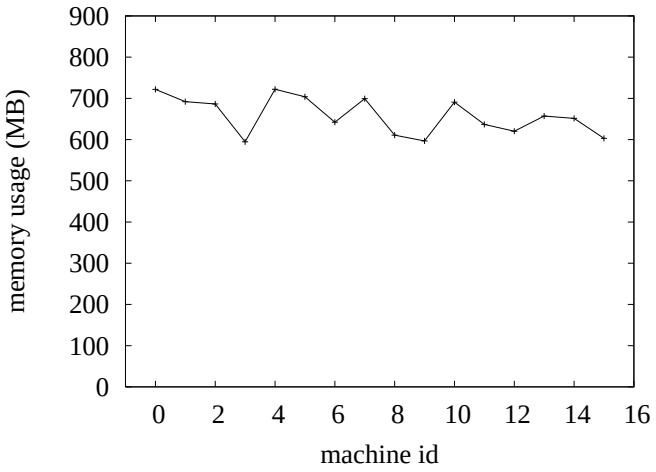
Although per-node memory and CPU costs exhibit high skews as shown in Figure 4.8, the multilevel recursive partitioning of Metis produces good memory and CPU balancing performance with our accurate and efficient cost estimation as input. Figure 4.14 shows memory balancing performance with Metis for a BGP simulation of an AS topology with 6582 nodes using 16 PCs. Figure 4.14(a) shows per-partition total weight distribution, an output of Metis. Figure 4.14(b) shows run-time measurement of per-partition memory cost. Figure 4.15 shows CPU balancing performance with Metis for a BGP simulation of an AS topology with 6582 nodes using 16 PCs. Figure 4.15(a) shows per-partition total weight distribution, an output of Metis. Figure 4.15(b) shows run-time measuremnt of per-partition CPU cost.

How does the multilevel recursive partitioning of Metis achieve good memory and CPU balancing? Recall that large-scale Internet measurement topologies with power-law connectivity skews produce a few severely unequal size objects (elephants), (elephants)—which work against balancing—and many small objects (mice) that work in favor of balancing. With our node weights as input, Metis is able to spread out the high-degree nodes across the machines. Figure 4.16 shows the details of partitioning assignment plotting machine ID as a function of nodes ranked by degree. The abscissa is shown in log-scale to highlight high-degree nodes. We observe that high-degree nodes are balanced across machines.

In the case of many small nodes, we established in Section 4.1.4 that small nodes can achieve relative memory balancing in large-scale network simulation. Since the memory cost estimate of a single high degree node is accurate and many small degree nodes can fill in the gap effectively, combined memory balancing of high- and low-degree nodes achieves relative memory balancing. Note that our custom network partitioning algorithm is based on this property.

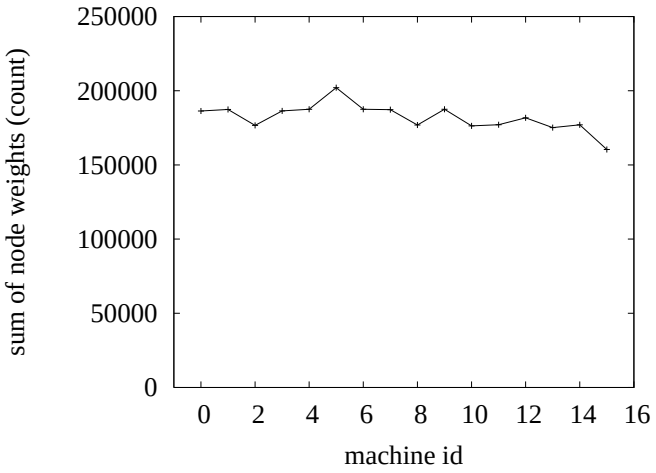


(a)

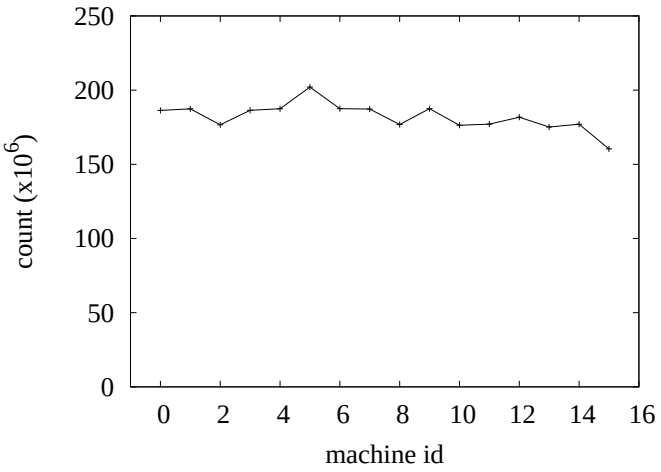


(b)

Figure 4.14. Memory balancing performance with Metis for a BGP simulation of an AS topology with 6582 nodes using 16 PCs. (a) Output of Metis. (b) Run-time measurement.



(a)



(b)

Figure 4.15. CPU balancing performance with Metis for a BGP simulation of an AS topology with 6582 nodes using 16 PCs. (a) Output of Metis. (b) Run-time measurement.

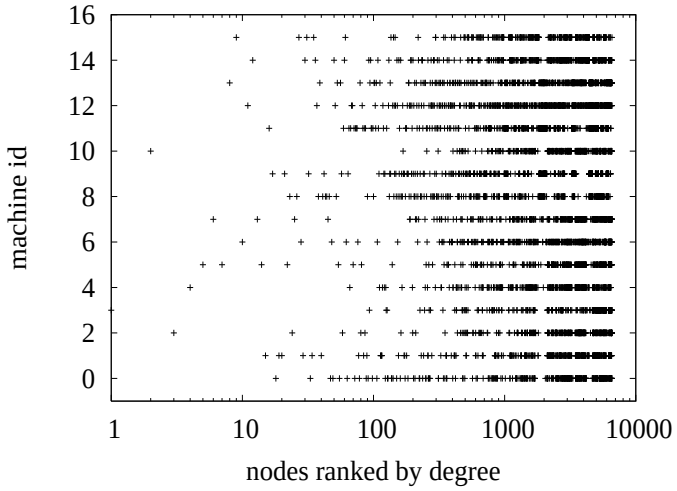


Figure 4.16. Partitioning assignment: machine ID as a function of nodes ranked by degree.

5 MEMORY BALANCING PERFORMANCE

In this chapter, we investigate memory balancing performance for large-scale network simulation which admits solutions for memory estimation and balancing not availed to small-scale or discrete-event simulation in general. First, we evaluate basic memory balancing performance under the maximum cost metric with comparison to the uniform and total cost metrics, and we establish a trade-off between memory and CPU balancing under the maximum and total cost metrics. Second, we show that joint memory-CPU balancing can overcome the performance trade-off—in general not feasible due to constraint conflicts—which stems from network simulation having a tendency to induce correlation between maximum and total cost metrics. We note that the impact of VM paging is not considered in this performance evaluation. We present memory balancing performance with VM paging in Chapter 6.

5.1 Experimental Set-up

5.1.1 Distributed Simulation Environment

We use a modified version of DaSSFNet as our distributed network simulation environment. DaSSF (Dartmouth SSF) [17] is a realization of SSF [41] written in C++ that is well suited for distributed simulation over PC clusters. DaSSFNet is an extension of DaSSF that implements a network stack and user API over its simulation kernel. The main modification we have added is a measurement subsystem that keeps track of dynamic simulation events spanning memory, computation, and communication both inside the simulation kernel and in the protocol stack outside the kernel. Distributed synchronization across network partitions is effected through a barrier mechanism that occurs at fixed time granularity (i.e., epoch)—the minimum

link latency of inter-partition links—which assures causal ordering of distributed simulation events [64, 65]. To focus on memory and CPU balancing, we set link latency to a uniform value which limits dependence of synchronization cost with respect to lookahead on network partitioning. Distributed coordination is implemented over MPI. Synchronization of distributed simulation events logged by the measurement subsystem is achieved through the shared simulation time which acts as a global clock.

5.1.2 Benchmark Applications and Network Models

We consider benchmark applications with varying traffic characteristics—BGP, worm propagation under local (i.e., topological) and global scanning, and distributed client/server system—that engage different aspects of the DaSSFNet protocol stack. BGP is a port of the Java SSF implementation of BGP-4 with both hash and trie based route table support. In BGP simulations ASes are treated as BGP router nodes. Worm propagation simulation is done at host IP granularity where IP addresses are mapped to individual ASes. Thus the higher the number of infected hosts at an AS, the higher the collective scan rate of the AS. The distributed client/server system assigns file server nodes to transit ASes that are accessed by clients at stub ASes. File servers possess heavy-tailed file sizes (Pareto with tail index 1.35) [67] that induce self-similar burstiness of aggregated traffic [68]. Session arrivals at a client are Poisson. BGP and distributed client/server system run over TCP whereas local and global worm propagation use UDP.

5.1.3 Network Topology

We use RouteViews/NLANR Internet autonomous system topologies [23] as our default benchmark network graphs. Problem size refers to the number of nodes in the network graphs. Table 5.1 shows a summary of network topologies used for our

performance evaluation specifying the date of measurement, the number of nodes, and the number of edges.

Table 5.1
Summary of network topologies used in performance evaluation.

date (yyyy/mm/dd)	nodes	edges
1999/01/11	4512	8383
1999/09/01	5663	10898
2000/01/14	6582	13194
2001/01/01	8063	16520
2001/02/03	9068	18233
2001/07/19	11555	24231
2002/01/01	12514	26030
2002/07/01	13532	28082
2003/01/01	14577	30046
2003/06/01	15465	34874
2004/02/24	16921	36767
2004/03/20	17243	37411
2004/08/01	18036	38934
2005/01/01	18960	40782
2006/01/01	21460	45712
2008/03/01	27738	57098

5.1.4 Hardware and OS Set-up

The PC cluster used in benchmark experiments consists of 32 Intel `x86` PCs running Linux 2.4.x and 2.6.x. Ten are Pentium 4, 2 GHz machines with 1 GB memory, six are 2.4 GHz with 1 GB memory, and six are 2.53 GHz with 1 GB memory. Five machines are Pentium 2.4 GHz with 2 GB memory and five are Xeon 2.4 GHz with

4 GB memory. L1 cache is 8 KB on all machines, and L2 cache is 512 KB except on the ten 2 GHz machines where it is 256 KB. Table 5.2 and Table 5.3 show CPU and memory configuration of the testbed machines. When comparing CPU balancing performance across different partitions processor speed can become a factor. Testing has shown that the 16 2.4 GHz machines yield approximately similar and predictable performance. They are used for CPU balancing comparisons. The PCs form a dedicated testbed connected by 2 GigE and 2 FE switches shown in Figure 5.1. Network congestion is not an issue, i.e., there are no packet drops and end-to-end latency is in the sub-millisecond range.

Table 5.2
CPU configuration of 32 participating machines.

host name	CPU model name	clock speed	L2 cache	L1 cache
6 dpf	Intel(R) Pentium(R) 4	2524MHz	512KB	8KB
10 infopod	Intel(R) Pentium(R) 4	1993MHz	256KB	8KB
5 4GB-mem greeks	Intel(R) Xeon(TM)	2392MHz	512KB	8KB
5 2GB-mem greeks	Intel(R) Pentium(R) 4	2391MHz	512KB	8KB
6 others	Intel(R) Pentium(R) 4	2391, 2399MHz	512KB	8KB

Table 5.3
Memory configuration of 32 participating machines.

host name	RAM size	total avail. mem	VM paging
6 dpf	1GB	1008MB	off
10 infopod	1GB	1009MB	off
5 4GB-mem greeks	4GB	3539MB	off
5 2GB-mem greeks	2GB	2021MB	off
6 others	1GB	1009–1012MB	off

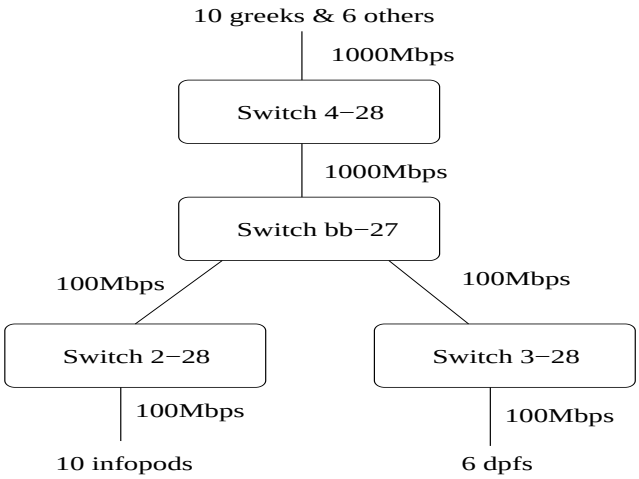


Figure 5.1. Network configuration of 32 participating machines.

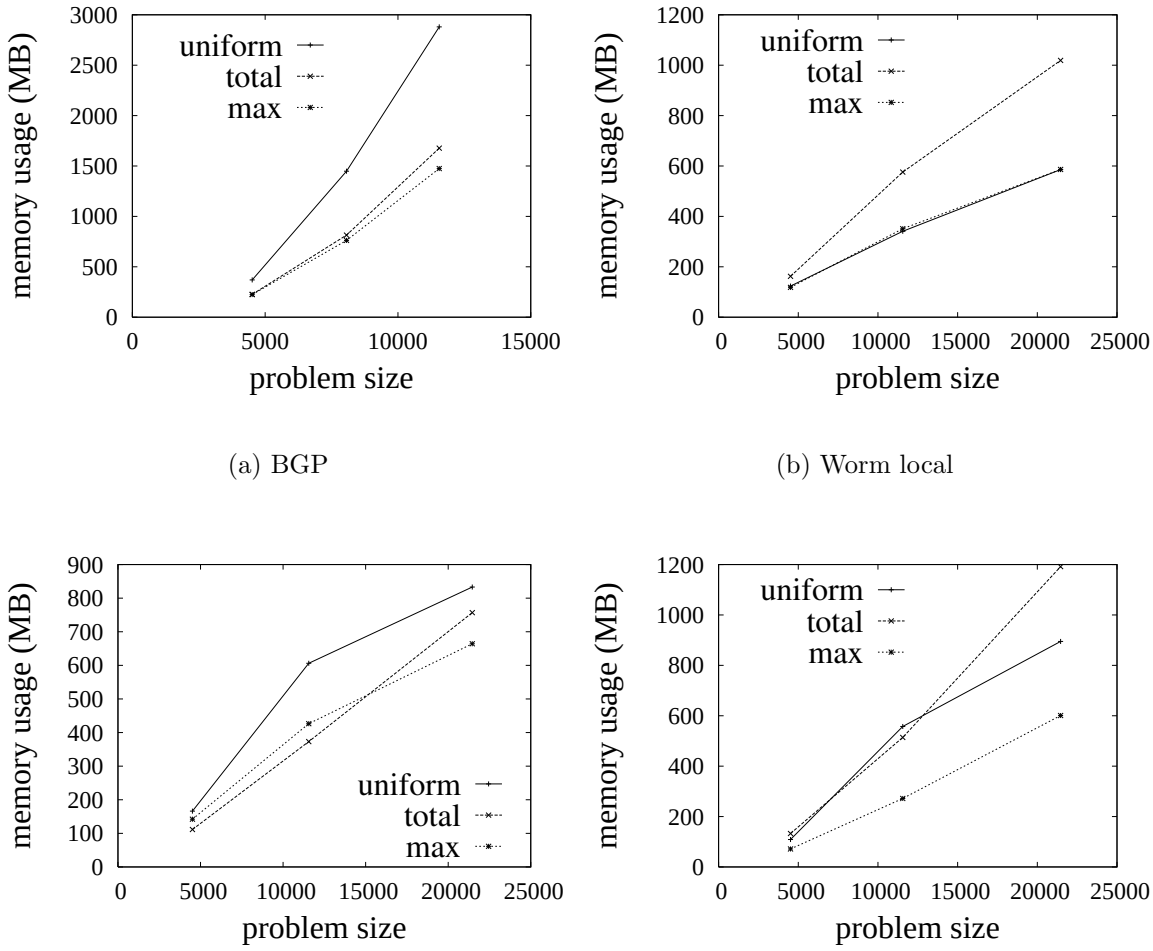
5.2 Memory Load Balancing

In this section, we evaluate the basic features and performance traits of memory-centric load balancing.

5.2.1 Performance Results

Figure 5.2 compares memory balancing performance with respect to memory usage between M_i (max) and C_i (total) for the BGP, worm, and client/server benchmark applications for a range of problem sizes. Memory usage is with respect to the maximum across all machines. We give $M_1, \ldots, M_n$ or $C_1, \ldots, C_n$ as node weight input to Metis which tries to find a k-way partitioning that minimizes edge cut while balancing node weight across the k partitions. As a reference point, we include memory balancing under uniform node weight in which premium is assigned to reducing edge cut.

We use up to 32 machines (i.e., $k \leq 32$) in the distributed simulations, with 32 machines used for the largest problem instances. Figure 5.2 shows that, overall, M_i outperforms C_i with the magnitude of the gap depending on benchmark application



(a) BGP

(b) Worm local

(c) Worm global

(d) Distributed client/server

Figure 5.2. Memory balancing performance of M_i (max), C_i (total), and uniform cost metrics as a function of problem size for different benchmark applications.

and problem size. Worm local and distributed client/server show the biggest gaps with BGP and worm global exhibiting marginal difference between maximum and total cost metrics. We also find that C_i can lead to memory imbalance that is significantly worse than the uniform cost metric (cf. Figure 5.2(b)).

5.2.2 Effect of Topology and Application Type

In this section, we analyze the memory balancing performance of the maximum cost metric compared to the uniform cost metric shown in Figure 5.2.

Influence of Power-law Connectivity

We show memory balancing performance in a random topology for different benchmark applications in Figure 5.3. We use a 4512-node random topology with the same edge density as that of a power-law topology of the same size. We note that the set of benchmark applications does not include worm local propagation. Since worm local propagation simulation shows negligible memory gaps between the uniform and maximum cost metrics in power-law networks, we expect negligible memory gaps between them in random networks as well. We observe that the memory balancing performance of the uniform and maximum cost metrics are comparable in BGP and worm global propagation simulations which is contrary to the results shown in Figure 5.2.

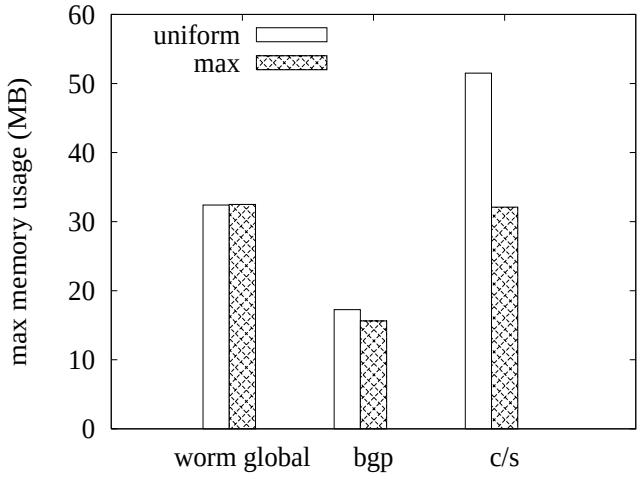
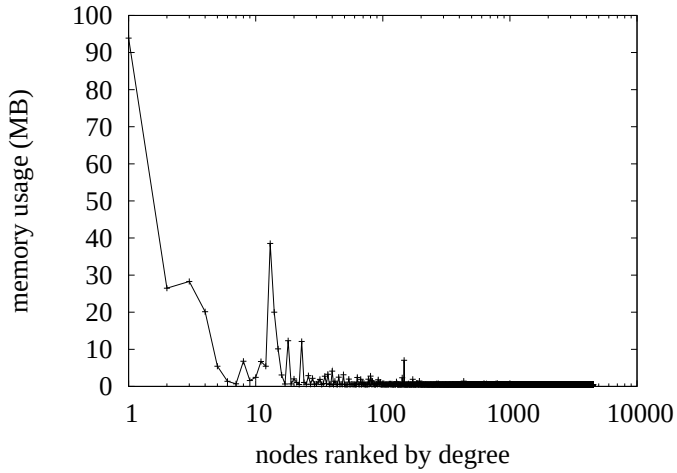
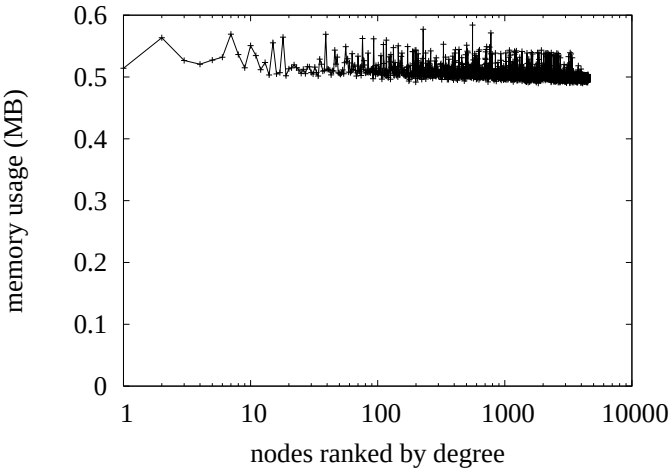


Figure 5.3. Memory balancing performance in a 4512-node random topology for different benchmark applications.

Figure 5.4 shows per-node M_i load distribution of a worm global propagation simulation, contrasting the power-law topology case and the random topology case. The x-axis represents nodes ranked by degree in decreasing order shown in log-scale. We observe a significant skew in load distribution of power-law topology in Figure



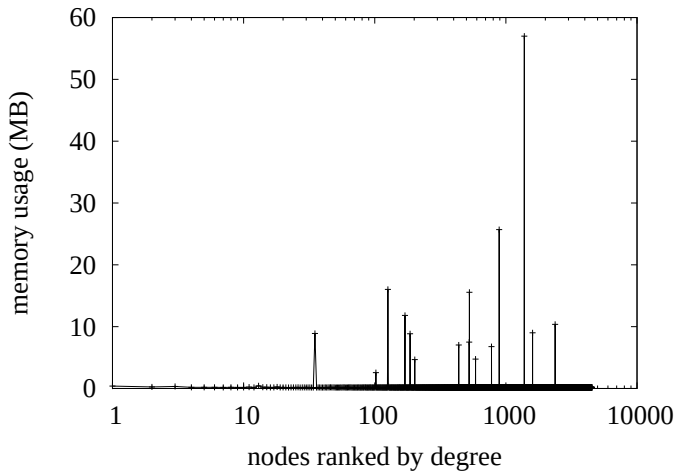
(a) Power-law



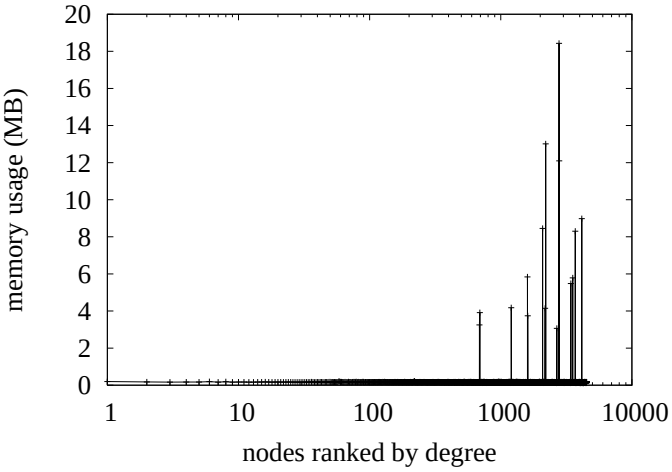
(b) Random

Figure 5.4. Per-node M_i load distribution of worm global: power-law topology vs. random topology.

5.4(a), which stands in stark contrast to the uniform distribution of the random topology in Figure 5.4(b). The skewness in power-law topology is caused by traffic concentration at high-degree nodes.



(a) Power-law



(b) Random

Figure 5.5. Per-node M_i load distribution of distributed client/server application: power-law topology vs. random topology.

Influence of Application Behavior

In the case of distributed client/server simulation, we observe that the maximum cost metric outperforms the uniform cost metric both for power-law topology and random topology as seen in Figure 5.2 and Figure 5.3.

Figure 5.5 shows per-node M_i load distribution of a distributed client/server simulation, comparing the power-law topology case and the random topology case. We observe high memory load at select nodes—i.e., location of file servers—in both power-law and random topologies. We do not see high memory load at high-degree nodes in Figure 5.5(a) which differs from memory load distribution for worm global simulation shown in Figure 5.4(a). The main reason is that the most memory-demanding component in distributed client/server simulation is packet queueing at TCP send buffers whereas the most memory-demanding component in worm global simulation is queueing of IP packets at link output buffers which is not directly governed by TCP's ARQ and congestion control[1].

5.3 Memory vs. CPU Balancing Trade-off

5.3.1 Performance Results

As indicated in Section 5.1, to carry out meaningful CPU balancing comparison we need to ensure that processor speeds across different PCs are comparable. This limits us to 16 PCs which also curbs the largest problem instances we can run without engaging VM paging.

Figure 5.6 shows memory balancing performance comparing the maximum and total cost metrics. Problem sizes are specified in parentheses. The memory and CPU balancing results in this section are averages of 5 runs, per problem instance, under different random seeds in Metis where randomization is used to affect improved bisectioning and matching. We find that the maximum metric, overall, outperforms the to-

[1]Note that worm application runs on top of UDP, not TCP.

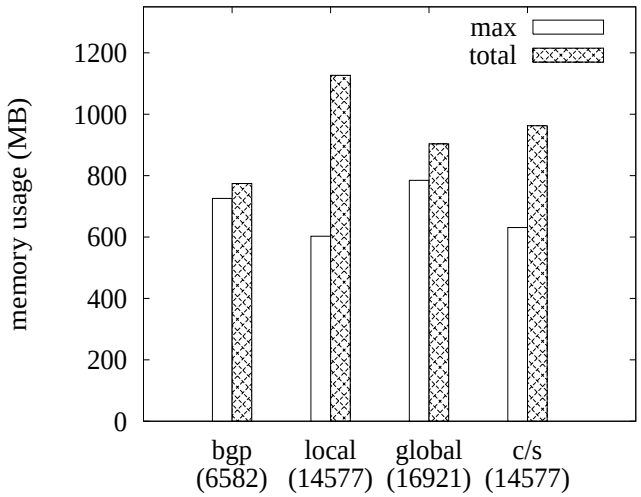
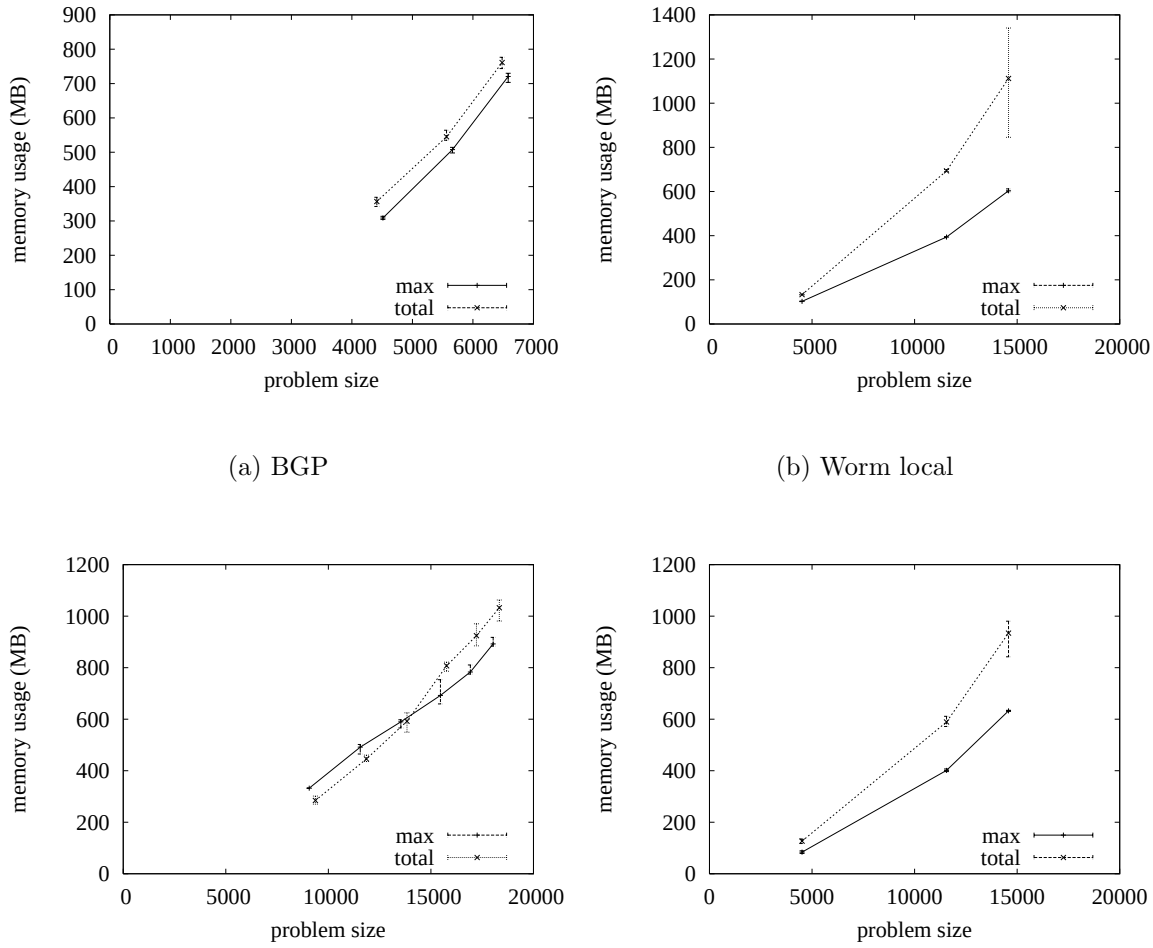


Figure 5.6. Memory balancing performance with $k = 16$ homogeneous machines.

tal cost metric with the gap being highest for worm local and distributed client/server benchmark applications consistent with the results in Section 5.2.

Figure 5.7 shows memory balancing performance of the maximum and total cost metrics as a function of problem size for different benchmark applications. For each data point, the minimum and maximum values are shown along with the average value of 5 runs under different random seeds in Metis. We observe that, overall, the maximum metric outperforms the total cost metric in BGP, worm local, and distributed client/server simulations. In the case of worm global, the maximum metric shows marginally better memory balancing performance at bigger problem sizes. The memory balancing gaps between the maximum and total cost metrics in these applications are consistent with the results in Figure 5.6.

Figure 5.8 (top) shows CPU balancing performance with respect to computation time—`usr` and `sys` time of the slowest (i.e., highest processing load) machine—for the same benchmark runs. As expected, the total cost metric, overall, outperforms the maximum cost metric with respect to CPU balancing. The biggest gaps occur in the cases of worm global and BGP. Figure 5.8 (bottom) shows CPU balancing performance with respect to completion time which includes synchronization penalty



(a) BGP

(b) Worm local

(c) Worm global

(d) Distributed client/server

Figure 5.7. Memory balancing performance of M_i (max) and C_i (total) cost metrics as a function of problem size for different benchmark applications.

among PCs in distributed simulation stemming from per-epoch barriers. An across-the-board upward shift is accompanied by a decrease in the maximum vs. total cost metric performance gap. Communication cost—processing time expended for sending and receiving of messages across network partitions—is dominated by MPI packing and unpacking operations which are accounted for in both computation and completion time.

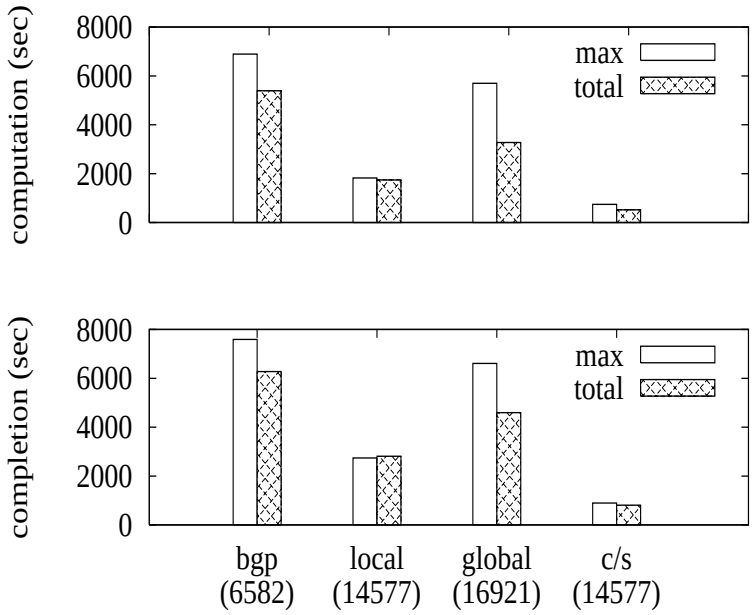
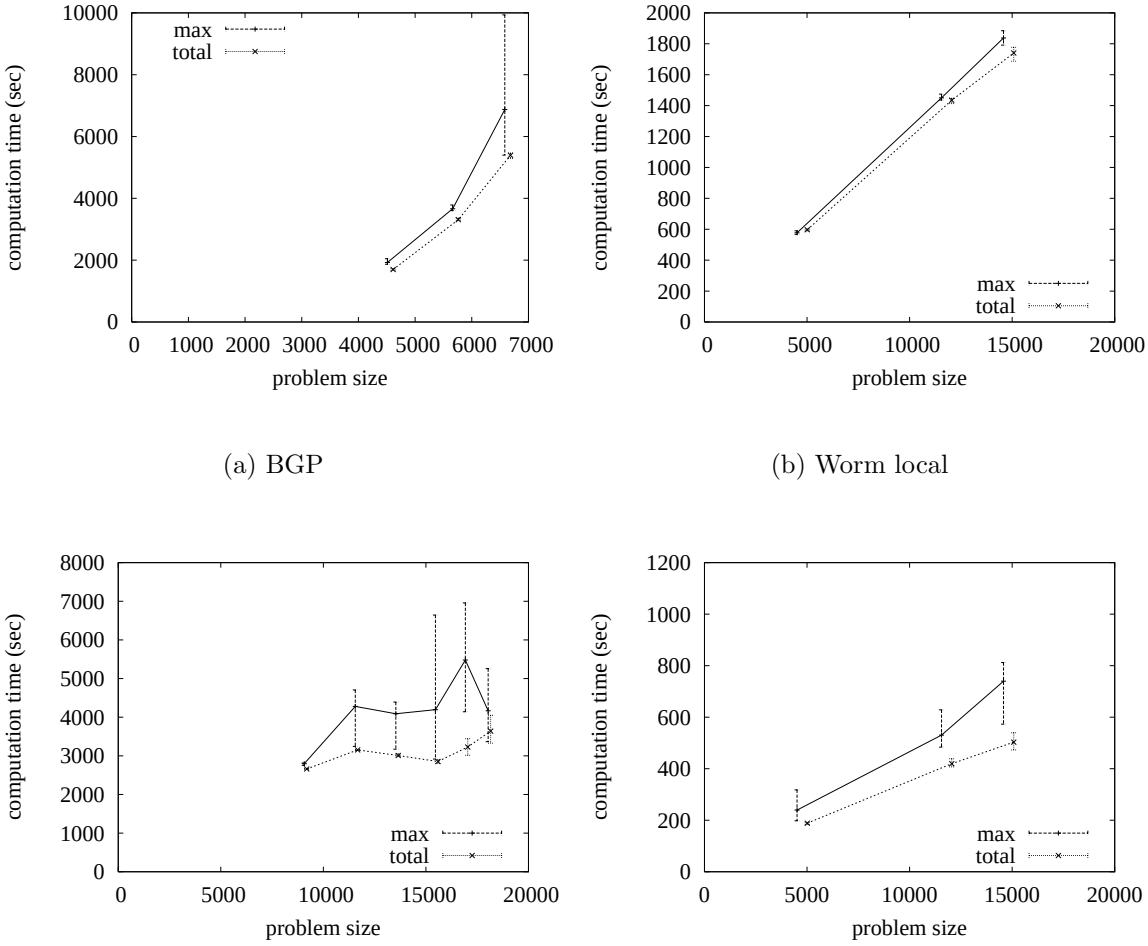


Figure 5.8. CPU balancing: computation time (top) and completion time (bottom).

Figure 5.9 shows CPU balancing performance of the maximum and total cost metrics with respect to computation time as a function of problem size for different benchmark applications. For each data point, the minimum and maximum values are shown along with the average value of 5 runs under different random seeds in Metis. We observe that the total cost metric outperforms the maximum cost metric with respect to CPU balancing. The CPU balancing gaps between the maximum and total cost metrics are consistent with the results in Figure 5.8. One noticeable trend is that CPU balancing performance of the maximum cost metric is highly variable unlike the case of memory balancing performance. Furthermore, the minimum computation time of the maximum cost metric is close to the range of the total's computation time. Based on these observations, we further investigate an enhancement of the maximum cost metric's CPU balancing performance in Section 5.5.



(a) BGP

(b) Worm local

(c) Worm global

(d) Distributed client/server

Figure 5.9. CPU balancing performance of M_i (max) and C_i (total) cost metrics with respect to computation time as a function of problem size for different benchmark applications.

5.3.2 Effect of Topology and Application Type

Influence of Power-law Connectivity

We compare BGP and worm local benchmark applications for which the memory balancing performance gaps between the maximum and total cost metrics are small

and large, respectively. Figure 5.10 (top) shows per-node M_i and C_i load distribution as a function of node rank—nodes are ranked by their degree with rank 1 indicating a node with the largest number of neighbors—for the BGP benchmark application. The abscissa is shown in log-scale to highlight the load values of high-degree nodes. We observe a pronounced skew in the load distribution—both for the maximum and total cost metrics—which stems from traffic concentration at high-degree nodes. Traffic skewness, in turn, is induced by power-law tendencies characteristic of Internet measurement topologies [69][2]. The total cost metric C_i exhibits a greater skew than the maximum metric M_i since the former is the sum over time whereas the latter is the maximum. Figure 5.10 (bottom) shows the corresponding plots for worm local which exhibit similar power-law skews.

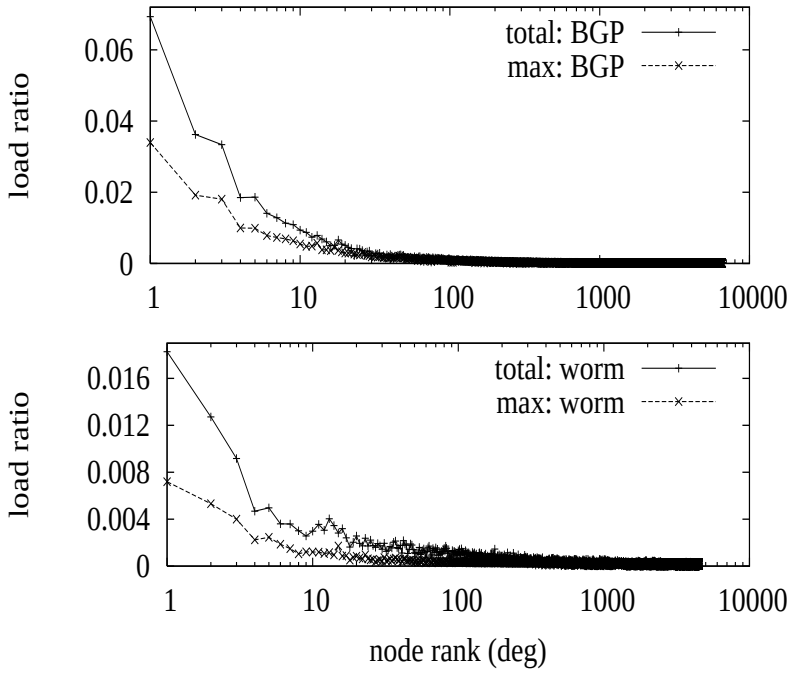


Figure 5.10. Node load distribution as a function of node rank: BGP (top) and worm local (bottom).

[2]Power-law tendencies also exist in router-level topologies [69, 70] although their detailed structure and causality differ.

Influence of Application Behavior

The key difference between BGP and worm local is shown in Figure 5.11 which plots the cumulative node load distribution corresponding to Figure 5.10. In BGP, we observe that both total and max increase rapidly initially with C_i climbing a bit higher than M_i. In worm local the initial rate of increase of M_i is significantly slower than that of C_i, almost resembling a linear curve. The sharp increase in cumulative M_i in BGP is caused by a sharp rise in message memory which dominates table memory (i.e., BGP and IP routing tables). In both BGP and worm local, cumulative table memory increases gradually.

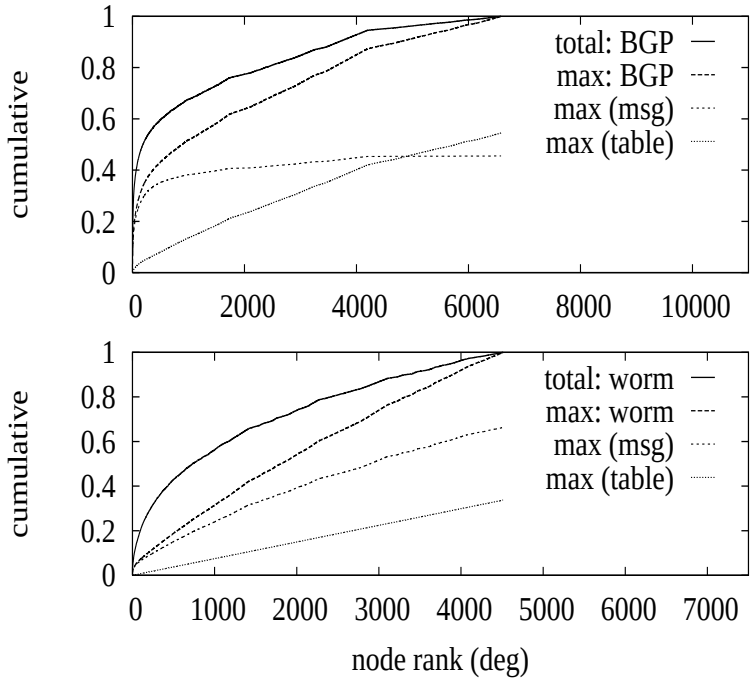


Figure 5.11. Cumulative node load distribution: BGP (top) and worm local (bottom).

All else being equal, the more skewed a node load distribution, the harder it is to balance due to diminished interchangeability. In both Figure 5.10 and 5.11, total cost metric has a higher node load skew than maximum cost metric. The key difference is that worm local has a significantly bigger skew gap between total and maximum

cost metrics as shown in Figure 5.11 (bottom) which leads to a commensurately large memory imbalance under total vs. maximum cost metric seen in Figure 5.6. The difference in initial ramp-up in message memory can be explained by differences in application behavior of BGP and worm local.

5.3.3 Robustness of Memory vs. CPU Balancing Trade-off

Section 5.3.1 has established a trade-off relation between memory vs. CPU balancing. In this section, we show that the trade-off relation is robust with respect to network partitioning by evaluating memory-CPU balancing using Chaco [46], a popular network partitioning that implements multilevel recursive k-way partitioning. Figure 5.12 shows memory and CPU balancing performance of worm local. The same M_i and C_i values used for worm local in Figure 5.6 and Figure 5.8 are given as input to Chaco. We find that the maximum cost metric outperforms the total cost metric in terms of memory balancing, and there are only negligible gaps between the maximum and total cost metrics in terms of CPU balancing. This is consistent with the results shown in Figure 5.6 and Figure 5.8.

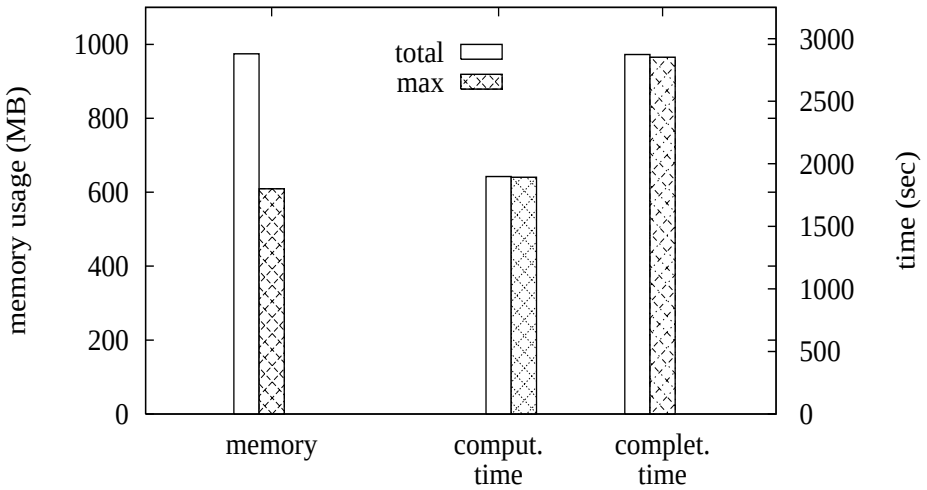


Figure 5.12. Memory and CPU balancing performance of worm local (14577) using Chaco.

5.3.4 Impact of Number of Partitions

In this section, we show the performance impact of the number of partitions on memory vs. CPU balancing trade-off using distributed client/server simulation. Figure 5.13 shows memory balancing performance of distributed client/server simulation as a function of the number of machines or partitions. The problem size of the example case is 14577. Since we have only 16 machines with comparable processor performance, we conduct experiments with 4, 8, and 16 machines. We observe that the maximum cost metric outperforms the total cost metric for all machine set-ups. The relative memory gap between the two cost metrics decreases as the number of partitions is increased.

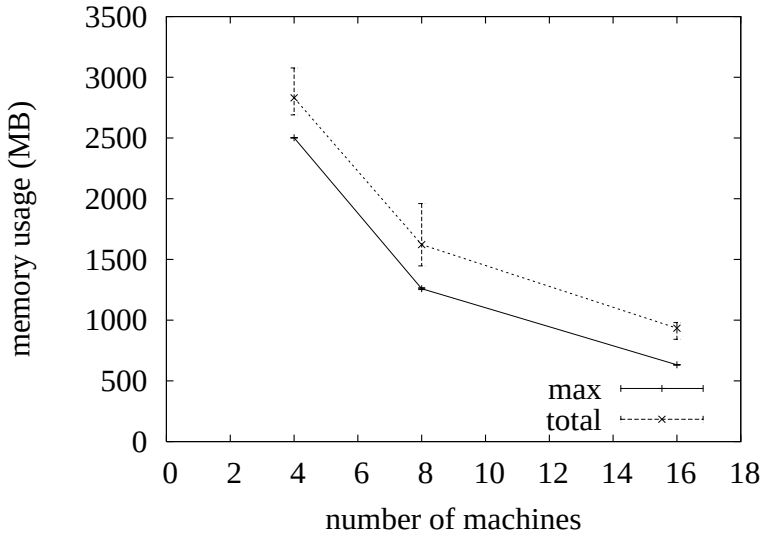


Figure 5.13. Memory balancing performance of distributed client/server simulation (14577) as a function of the number of machines.

Figure 5.14 shows CPU balancing performance of distributed client/server simulation as a function of the number of machines. We find that the CPU balancing performance of the total cost metric compared to the maximum cost metric improves as the number of machines is increased. The total cost metric outperforms the maximum cost metric with respect to CPU balancing only for the set-up with 16 machines.

In the cases of 4 and 8 machine cases, the maximum cost metric performs better or comparable compared to the total cost metric.

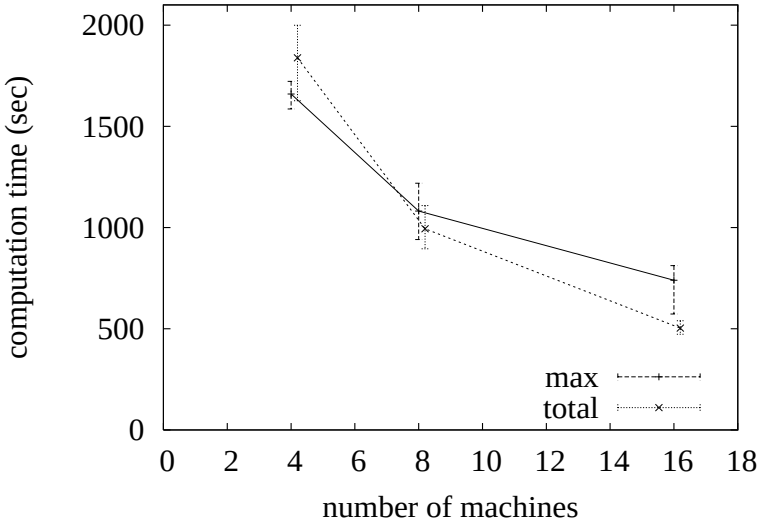


Figure 5.14. CPU balancing performance of distributed client/server simulation (14577) as a function of the number of machines.

5.4 Joint Memory-CPU Balancing

Sections 5.2 and 5.3 studied basic properties of memory balancing, established a trade-off between memory and CPU balancing, and studied their causes. In this section we examine joint memory-CPU balancing.

5.4.1 Multi-constraint Optimization

Multilevel recursive bisection methods including Metis [47] and Chaco [46] that implement k-way partitioning seek to find heuristic solutions to the NP-hard constrained optimization problem: minimize edge cut subject to node weight balancing. Multilevel recursive bisection heuristics may be extended to include a memory balance constraint in which case every node i has a 2-dimensional weight vector (M_i, C_i).

Metis has support for multi-dimensional node weights, and to the best of our knowledge Metis is the only tool supporting multi-dimensional node weights. We use this set-up for joint memory-CPU balancing. In general, a trade-off between two objectives implies that to improve one there has to be a sacrifice of the other. We show that in network simulation this need not be the case.

5.4.2 Performance Results

Figure 5.15 shows memory balancing performance for the benchmark set-up of Section 5.3 under joint memory-CPU balancing. We find that joint memory-CPU

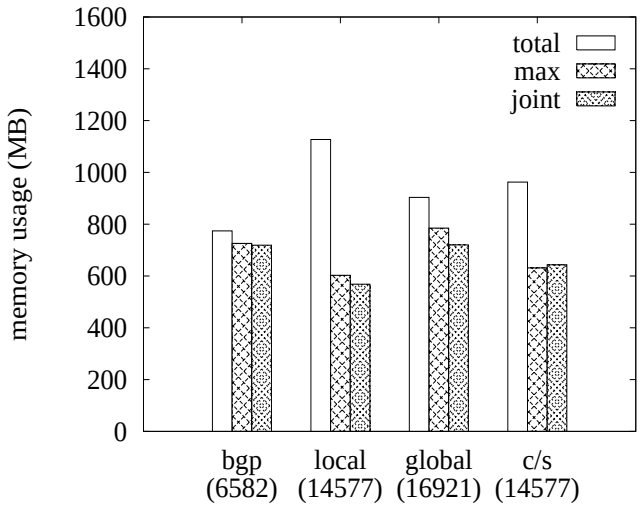


Figure 5.15. Memory balancing performance under joint max-total cost metric.

balancing performs as well as memory-centric load balancing which uses only node weight M_i. Figure 5.16 shows computation and completion time results under joint memory-CPU balancing. As with memory balancing, we observe that joint memory-CPU balancing performs as well as CPU-centric balancing that uses node weight C_i only. The results indicate that joint memory-CPU balancing finds solutions that match the individual performance of memory and CPU balancing, respectively. Given the trade-off relationship between the two, this implies that joint balancing outperforms both.

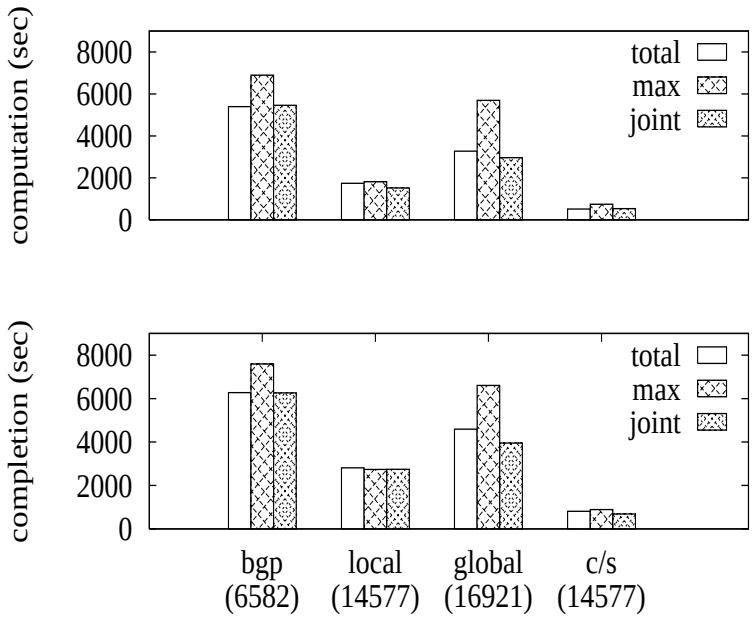


Figure 5.16. CPU balancing under joint max-total cost metric. computation time (top) and completion time (bottom).

5.4.3 Overcoming Memory-CPU Balancing Trade-off

How is it possible for joint memory-CPU using the max-total metric to circumvent the balancing trade-off? The answer lies in network simulation having a tendency to induce positive correlation between M_i and C_i. For high-degree nodes, this can be gleaned from Figure 5.10 for both BGP and worm local. For worm local the correlation coefficient, $\text{corr}(M_i, C_i)$, is 0.94. Individually they face balancing trade-off difficulties as discussed in Section 5.3. When combined, their individual feasible regions in the combined solution space have non-empty intersection facilitated by the correlation such that a partitioning is found that can balance both memory and CPU well. This property is not available to discrete-event simulation in general which can be surmised from Figure 5.17 which shows memory and CPU balancing performance across 16 machines participating in distributed simulation when $\text{corr}(M_i, C_i)$ is "strong"—original worm local benchmark—or comparatively "weak". In the lat-

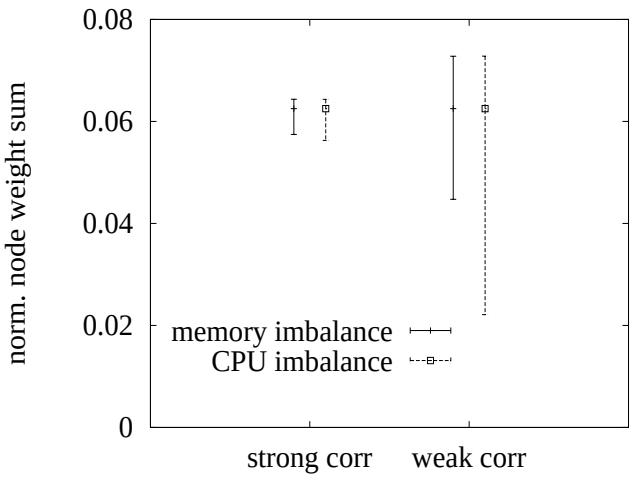


Figure 5.17. Joint memory-CPU balancing when corr(M_i, C_i) is strong and weak.

ter, node load skew of M_i in Figure 5.10 (bottom) is severely flattened such that corr(M_i, C_i) = 0.01. The ordinate of Figure 5.17 shows the average and spread of aggregate node weight sum—both memory (M_i) and CPU (C_i)—across 16 partitions. For comparison, we normalize node weight values by their total sum. We find that when M_i and C_i are weakly correlated memory and CPU balancing trade-off is more difficult to overcome.

5.5 Memory-CPU Balancing with Optimization of Communication Cost

In this section, we investigate an enhancement of the maximum cost metric's CPU balancing performance, focusing on the observations in Figure 5.9. Recall that CPU balancing performance of the maximum cost metric is variable unlike memory balancing performance where the minimum computation time of the maximum cost metric is close to the range of the total's computation time. Given the small variance associated with memory balancing performance under the maximum cost metric, we aim to achieve commensurately low variance CPU balancing performance if it is possible.

5.5.1 Impact of Communication Cost

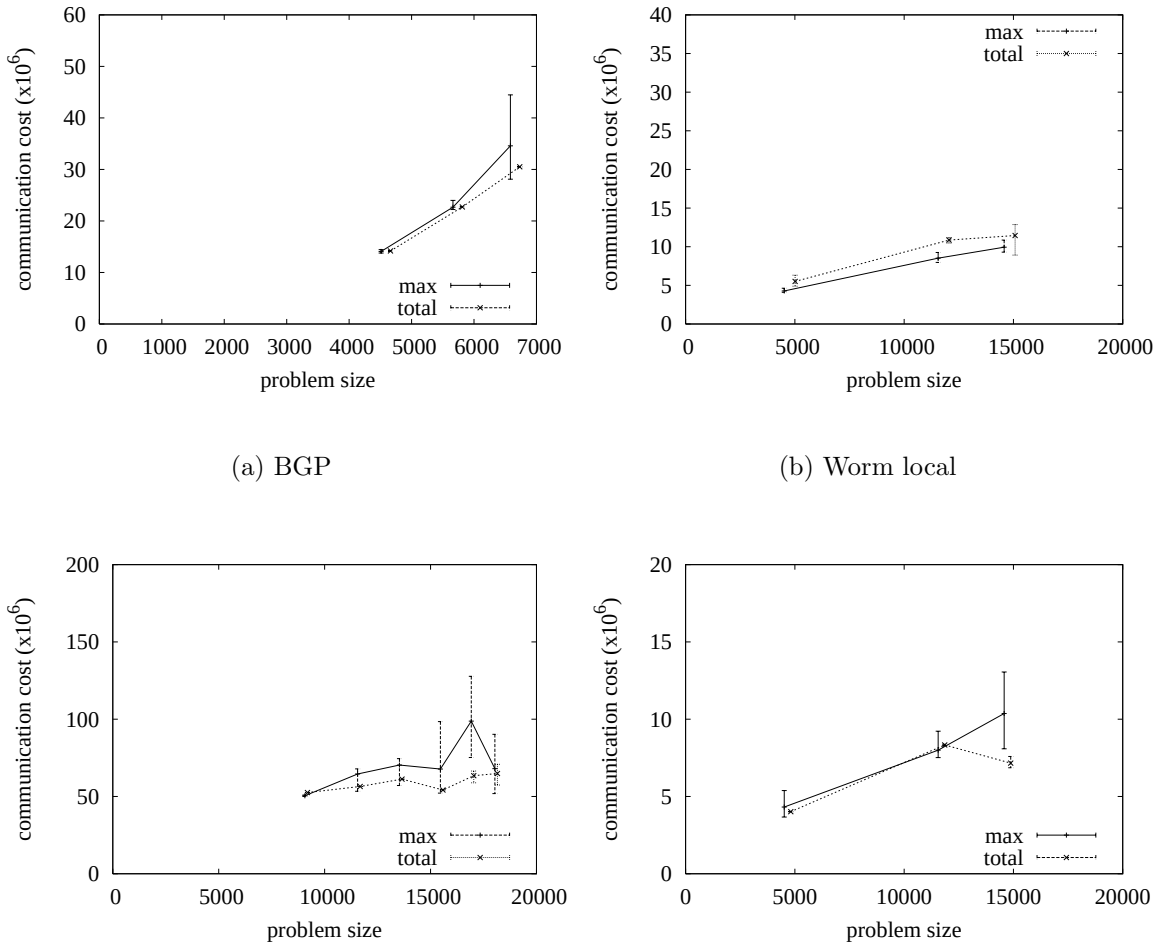
In this section, we will show that communication cost has an impact on balancing performance variability. As we mentioned when explaining message event evolution in Section 3.2.4, kernel event KEVT_OUTCH is mapped to an intermediate channel event in the case when the receiving node is on a different machine. The channel event is sent via MPI to the machine where the receiving node is located. At the receiving machine the channel event is mapped to kernel event KEVT_INCH. Reflecting the resultant network I/O related overhead, we define communication cost at a machine as the total number of sent or received channel events over time.

Figure 5.18 compares communication cost of the maximum and total cost metrics as a function of problem size for different benchmark applications. Communication cost is with respect to the maximum across all machines. Comparing Figure 5.18 with Figure 5.9, we observe variability in communication cost that is similar to the variability in CPU balancing performance. One goes with the other because the time taken for MPI message packing and unpacking operations resulting from inter-machine communication cost is accounted for in computation time. The high processing cost associated with MPI message packing/unpacking causes variability in communication cost to induce variability in CPU cost and balancing.

5.5.2 Performance Results of Variability Reduction Heuristic

Variability Reduction Heuristic

The objective of the variability reduction heuristic is to find a network partitioning instance using the maximum cost metric, which achieves CPU balancing performance as good as that of the total cost metric with respect to variability. We first generate a set of network partitioning instances under different random seeds in Metis. We estimate communication cost of each partitioning instance and pick the one with the minimum communication cost as a solution. We expect the quality of a solution to



(a) BGP

(b) Worm local

(c) Worm global

(d) Distributed client/server

Figure 5.18. Comparison of communication cost of M_i (max) and C_i (total) cost metrics as a function of problem size for different benchmark applications.

improve as we increase the number of elements in the set of network partitioning instances.

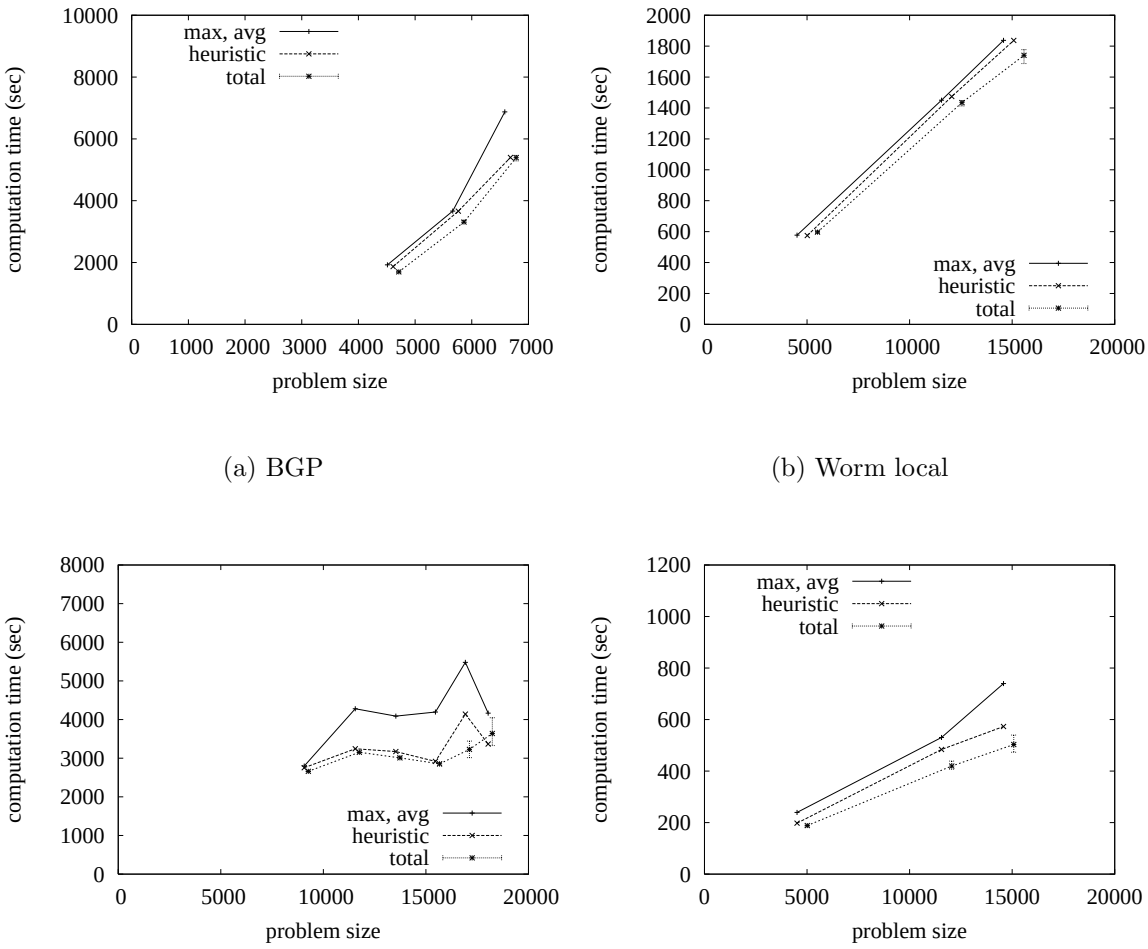
We estimate communication cost of a network partitioning instance as follows. We first obtain the logical communication cost of each link from the benchmark simulation run where we obtain M_i and C_i. Let us consider logical communication cost $T_{i,j}$ of a link between two nodes i and j which is defined as the total number of messages

transferred from i to j or from j to i over time. Note that the logical communication cost is independent of network partitioning. Given a network partitioning instance, communication cost T_K of a machine K is defined as $\sum T_{i,j}$ where $i \in K$ and $j \notin K$ for every pair i, j. We define communication cost of a network partitioning instance as the maximum T_K across all machines.

Performance Results

Figure 5.19 shows CPU balancing performance of the variability reduction heuristic compared against that of the maximum and total cost metrics as a function of problem size for different benchmark applications for a set of 5 network partitioning instances. Figure 5.20 shows a comparison of the variability reduction heuristic and the maximum cost metric relative to the total cost metric with respect to computation time difference (%) for the results in Figure 5.19. We observe that the variability reduction heuristic, overall, outperforms the average of the maximum cost metric.

Figure 5.21 shows a comparison of the variability reduction heuristic and the maximum cost metric relative to the total cost metric with respect to memory usage difference (%) as a function of problem size for different benchmark applications for 5 network partitioning instances. We observe that the variability reduction heuristic, overall, shows comparable memory balancing performance to the average of the maximum cost metric.



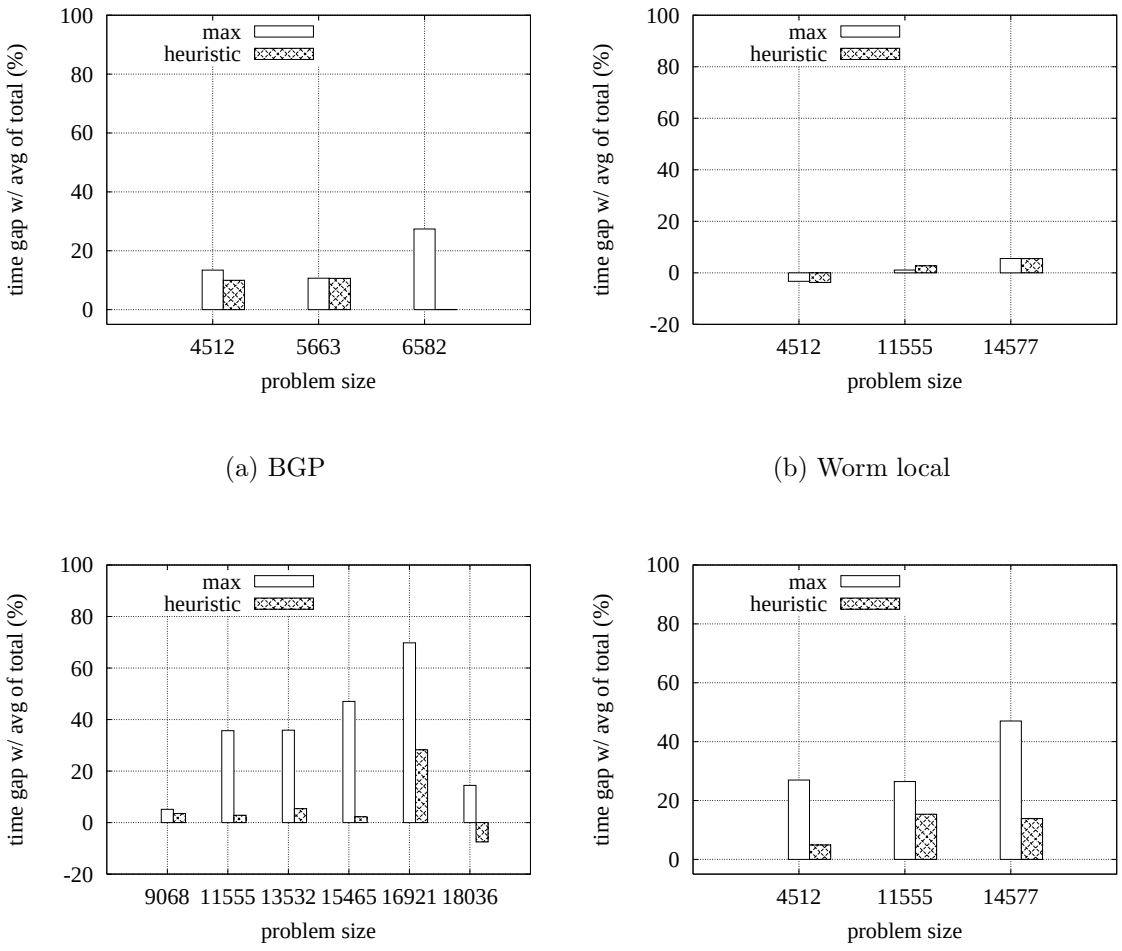
(a) BGP

(b) Worm local

(c) Worm global

(d) Distributed client/server

Figure 5.19. CPU balancing performance of the variability reduction heuristic compared to M_i (max) and C_i (total) cost metrics as a function of problem size for different benchmark applications for 5 network partitioning instances.



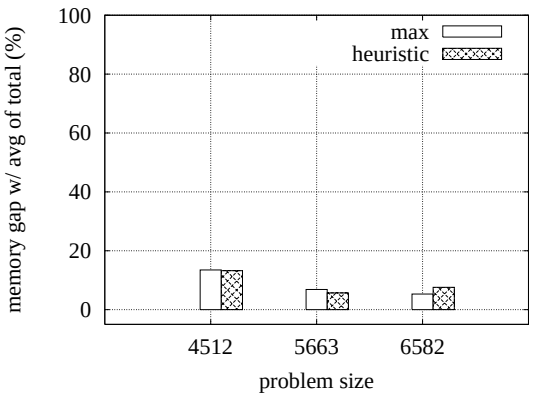
(a) BGP

(b) Worm local

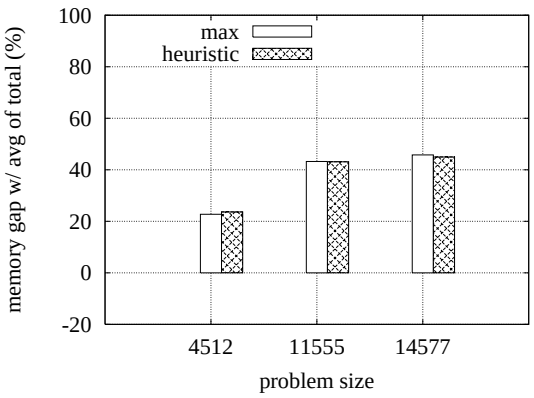
(c) Worm global

(d) Distributed client/server

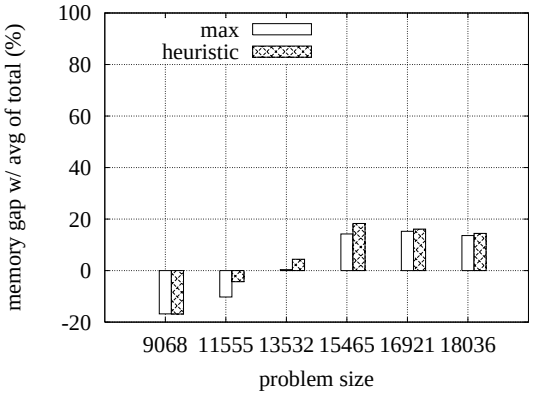
Figure 5.20. Comparison of the variability reduction heuristic and M_i (max) cost metric against C_i (total) with respect to computation time difference (%) as a function of problem size for different benchmark applications for 5 network partitioning instances.



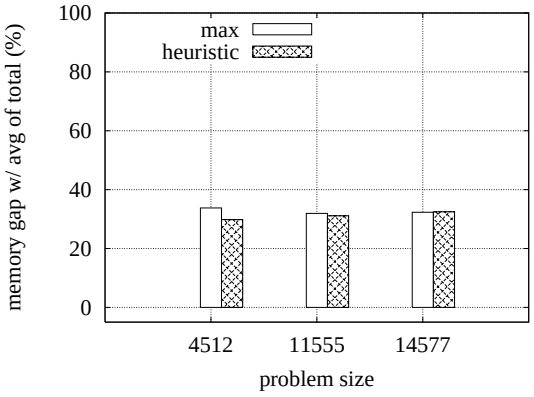
(a) BGP



(b) Worm local



(c) Worm global

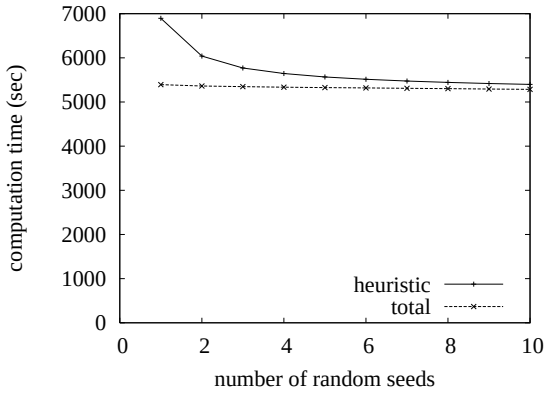


(d) Distributed client/server

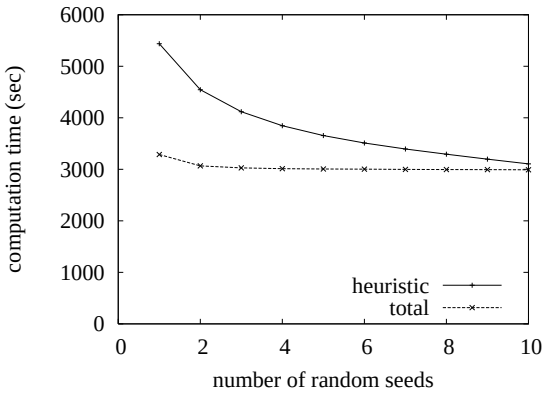
Figure 5.21. Comparison of the variability reduction heuristic and M_i (max) cost metric against C_i (total) with respect to memory usage difference (%) as a function of problem size for different benchmark applications for 5 network partitioning instances.

5.5.3 Effect of the Number of Random Seeds

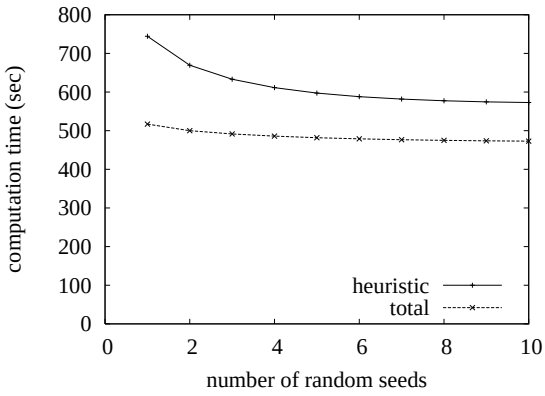
We study the improvement of the quality of a solution as we increase the number of elements in the set of network partitioning instances. Figure 5.22 shows a comparison of CPU balancing performance of the variability reduction heuristic and the total cost metric as a function of the number of random seeds varying from 1 to 10 for different



(a) BGP



(b) Worm global



(c) Distributed client/server

Figure 5.22. Comparison of CPU balancing performance of the variability reduction heuristic and C_i (total) cost metrics as a function of the number of random seeds for different benchmark applications.

benchmark applications. CPU balancing performance is with respect to computation time. As we increase the number of random seeds—i.e., increasing the number of elements in the set of network partitioning instances—the computation time gap between the variability reduction heuristic and the total cost metric decreases. In the cases of BGP and worm global simulation, the variability reduction heuristic achieves CPU balancing performance comparable to that of the total cost metric as the number of random seeds is increased.

5.6 Optimizing the Overhead of Benchmark-based Cost Estimation

In previous sections, we have considered scenarios where we obtain per-node memory cost—the maximum cost metric—from a complete run of a benchmark simulation. Then, we used the maximum cost metric to improve memory balancing. We define that a benchmark simulation is complete if the simulation duration of a benchmark simulation is the same as the the simulation duration of a memory-balanced simulation using the maximum cost metric. As discussed in Section 4.2.2, the overhead of our benchmark-based cost estimation, i.e., the completion time of a benchmark simulation, can be significant. In this section, we investigate how to optimize the overhead of the benchmark-based cost estimation.

We investigate the utility of per-node memory cost estimation obtained from $x\%$ of simulation execution in terms of memory balancing performance, varying x. Figure 5.23 shows per-node memory cost distributions estimated at 5%, 30%, 50%, and 100% execution of a worm global propagation simulation. Figure 5.24 shows memory balancing performance as a function of simulation progress ($x\%$ of wall clock time). We observe that, in BGP, memory cost estimation from 70% of the simulation execution shows comparable memory balancing performance to memory cost obtained from a complete execution. In the worm global and distributed client/server simulations, memory cost estimation from 30% of the simulation execution shows significant memory balancing performance. This result is expected from the per-node memory cost

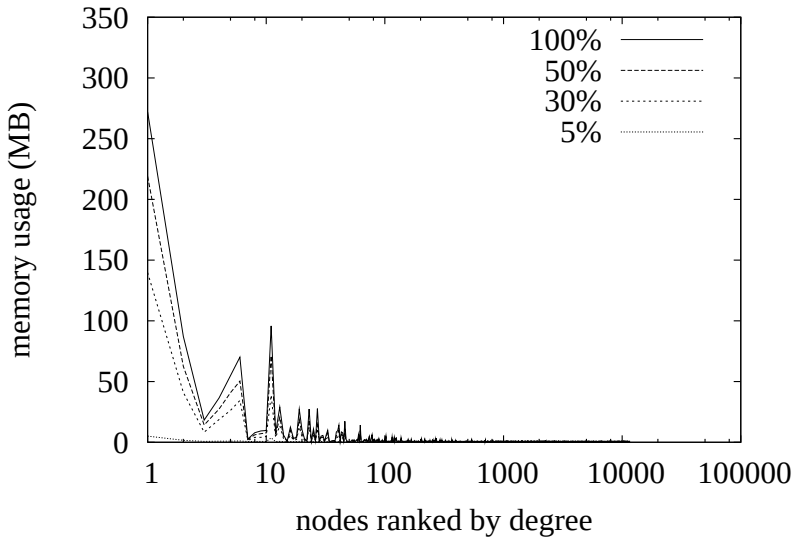
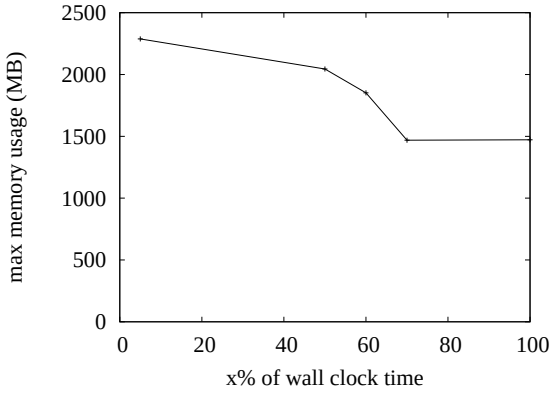
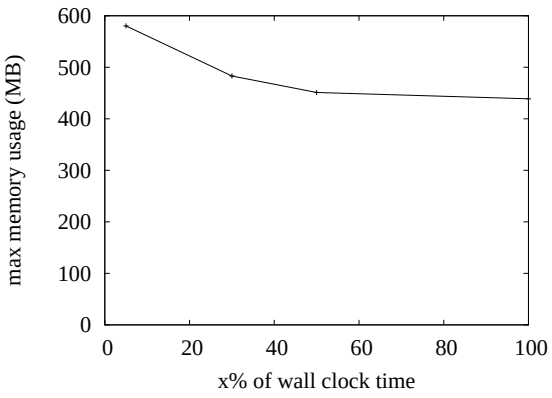


Figure 5.23. Per-node memory cost estimated at 5%, 30%, 50%, and 100% of simulation execution: worm global.

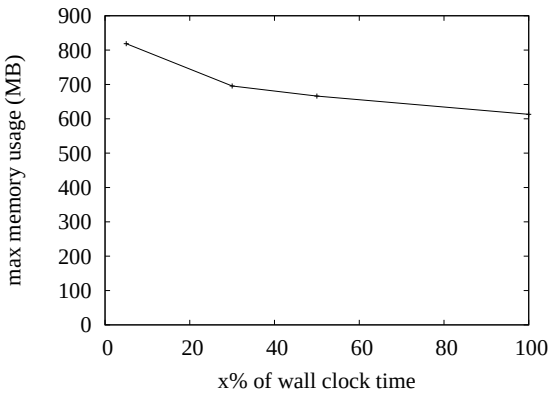
distributions shown in Figure 5.23. We observe that the per-node memory cost estimation at 30% of simulation execution is similar to that of 100% simulation execution, especially in high degree nodes.



(a) BGP



(b) Worm global



(c) Distributed client/server

Figure 5.24. Memory balancing performance as a function of simulation progress (x% of wall clock time).

6 MEMORY BALANCING PERFORMANCE WITH VIRTUAL MEMORY PAGING

6.1 Performance Evaluation Framework

6.1.1 Overview of Linux's Virtual Memory Paging

In this section, we provide an overview of Linux's virtual memory paging based on the Linux 2.4.25 kernel [71]. When a page fault exception occurs for a page, the page can be still in the memory—specifically, in the swap cache—or on disk. The first case is called minor page fault and it does not involve disk I/O. The latter is called major page fault and involves disk I/O to read the missing page in.

There are three main operations which need to be distinguished. The first operation is *swapping in*. When a page fault exception occurs and its handler detects that the corresponding page is not in memory, the handler calls a function that swaps the missing page in from disk. When reading in the missing page from disk, up to 8 pages located close to the missing page are read in together. The second operation is *swapping out (or page frame releasing)*. A page frame is released when it is removed from the page tables of all processes that share it. Released page frames are put into the swap cache—implemented as a part of the page cache with different indexing mechanism. Note that page frames in the swap cache still contain previous page information. The third operation is *page frame reclaiming*. The goal of page frame reclaiming is to maintain a minimal pool of free page frames. It is triggered every time the kernel fails in allocating memory—for its own use or for servicing a user's memory requests via `sbrk()` or `mmap()`—or the kernel swap daemon (kswapd) discovers that the number of free page frames in memory is less than a certain limit. To implement the least recently used (LRU) policy, pages in the page cache are maintained with

two lists—active list and inactive list—reflecting accesses to each page. Once page frame reclaiming is triggered, it frees page frames by writing inactive pages in the page cache to disk.

6.1.2 Operating System Monitoring

In this section, we describe an operating system (OS) monitoring module which we included for performance evaluation under the virtual memory paging. The OS monitoring module reads system variables and dumps readings into a log file periodically. A main feature of this measurement module is that OS monitoring is done at the granularity of an epoch. Initially we designed and implemented this module as a separate monitoring process outside DaSSFNet which reads system variables and dumps readings into a log file periodically with 1 second time interval. It turned out that, under the virtual memory paging, the monitoring process's periodic activity can slow down simulation progress significantly. Monitoring at every epoch is sufficient for our performance evaluation, but epoch is a time interval defined only within the simulated world. To minimize the number of readings and logs and their impact on overhead, we ported the monitoring module into DaSSFNet.

Inside the DaSSF kernel, the OS monitoring module reads system variables from the /proc file system and writes them into a file after simulating each epoch. Table 6.1 shows the main system variables that are monitored by the measurement subsystem. Linux keeps track of the number of major page faults for every process. However, other statistics related to paging and disk I/O are provided for the whole system, not per process. As mentioned in Section 6.1.1, a major page fault can cause up to 8 pages to be read in from disk. Hence, the number of major page faults is less than or equal to the number of swap pages brought in. Each disk read/write operation represents a block device request. Since a block device request can handle multiple pages at the same time, the number of swap pages brought in may not be the same as the number of disk read operations. The same is true for the number of swap pages

brought out and the number of disk write operations. We monitor the memory usage of each process using `vsize`, and we confirm low CPU utilization during thrashing by monitoring the CPU idle time measured as the time spent by the idle process.

Table 6.1
System variables read from Linux /proc file system.

system variables	description
majflt, /proc/[pid]/stat	the number of major page faults the process has made which have required loading a memory page from disk.
swap a b, /proc/stat	the number of swap pages brought in (a) and the number of swap pages brought out (b).
disk_io: (a,b):(c,d,e,f,g), /proc/stat	the number of disk read operations (d), the number of disk write operations (f), the number of disk blocks read (e), and the number of disk blocks written (g).
vsize, /proc/[pid]/stat	virtual memory size in bytes
/proc/uptime	the amount of time spent by the idle process (in seconds)

6.1.3 Application Memory Referencing Behavior

When a machine's memory load exceeds its physical memory limit with the support of VM paging, an application's memory referencing behavior is a key factor determining when trashing occurs. In this section, we demonstrate and compare memory referencing behavior of benchmark applications with respect to memory usage and page fault rate over simulation time. Simulations are conducted on a single machine with 1GB memory. Figure 6.1 shows a BGP simulation of a 4512-node AS topology. Figure 6.1(a) and Figure 6.1(c) show memory usage and page fault rate as

a function of simulation time. Page fault rate is calculated for each 1-msec simulation time duration, i.e., epoch. Figure 6.1(b) and Figure 6.1(d) show blow-ups of the plots after simulation time 60 second where thrashing occurs. We observe that memory

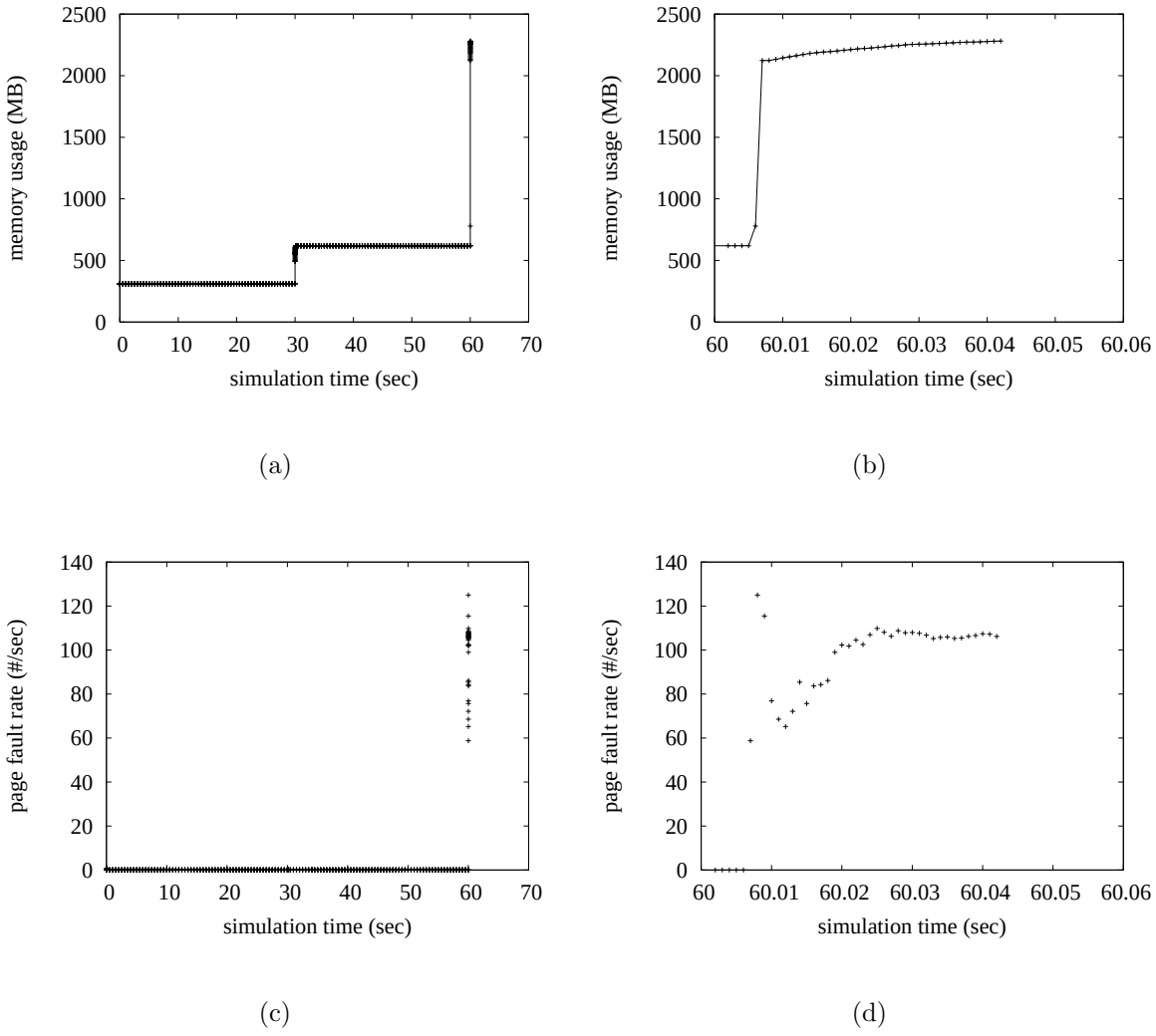


Figure 6.1. BGP simulation of 4512-node AS topology on a single machine with 1GB memory. (a) Memory usage as a function of simulation time. (b) Blow-up of (a). (c) Page fault rate as a function of simulation time. (d) Blow-up of (c).

usage exceeds 1GB physical limit at 60.007 second simulation time exceeding 2GB

eventually. Page fault rate jumps at 60.007 second simulation time and converges to around 110 page faults per second.

Figure 6.2 shows a worm global propagation simulation of a 6582-node AS topology. Figure 6.2(a) and Figure 6.2(c) show memory usage and page fault rate as a function of simulation time. Figure 6.2(b) and Figure 6.2(d) show their blow-up after

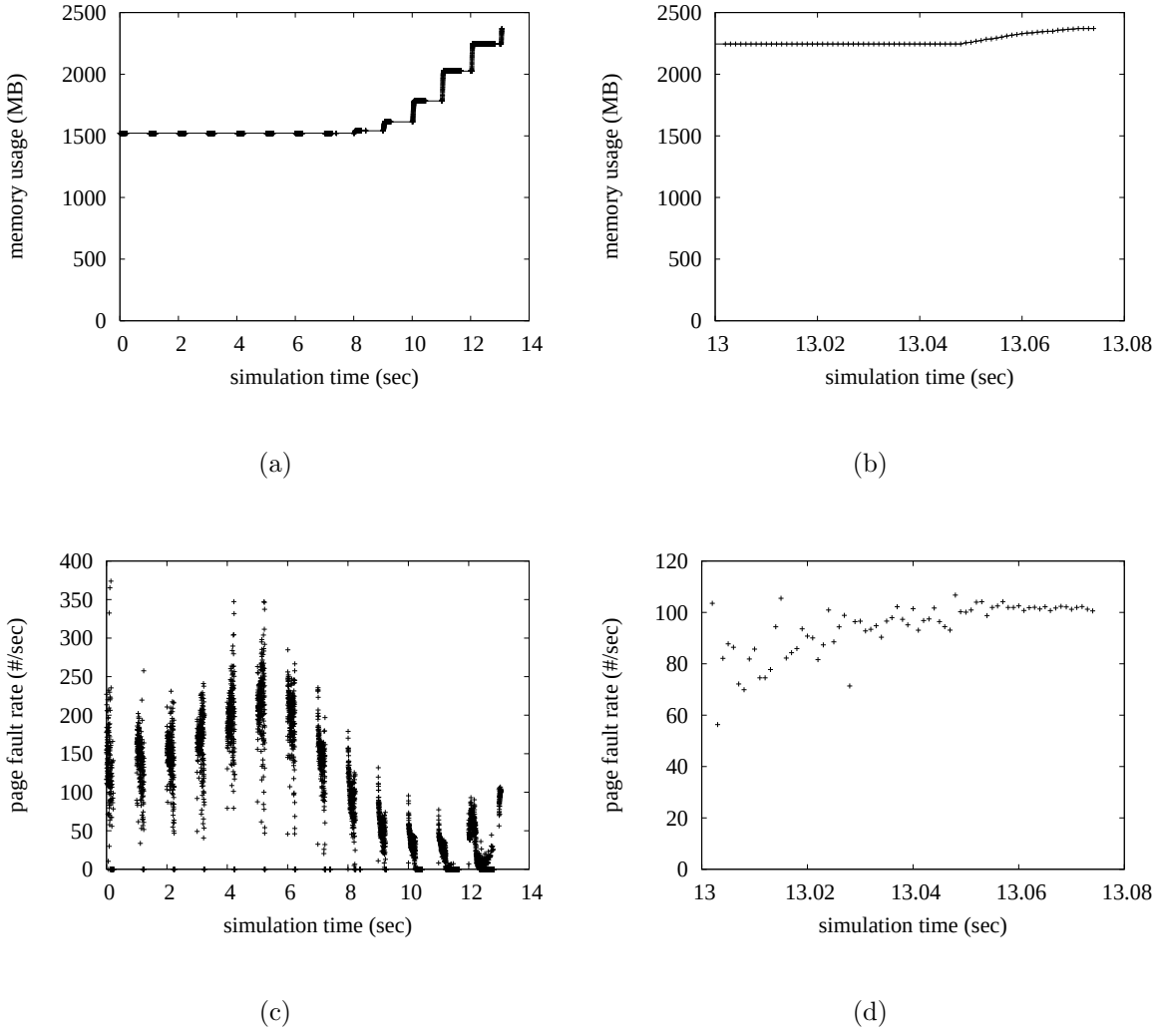


Figure 6.2. Worm global propagation of 6582-node AS topology on a single machine with 1GB memory. (a) Memory usage as a function of simulation time. (b) Blow-up of (a). (c) Page fault rate as a function of simulation time. (d) Blow-up of (c).

13 second simulation time where thrashing occurs. We observe that memory usage exceeds the 1GB physical limit from the start of the simulation reaching around 2.5GB eventually. The initial memory consumption of around 1.5GB is due to IP routing tables loaded at each simulation node before simulation starts. A noticeable increase of memory usage happens from 9 second simulation time at 1 second intervals. In terms

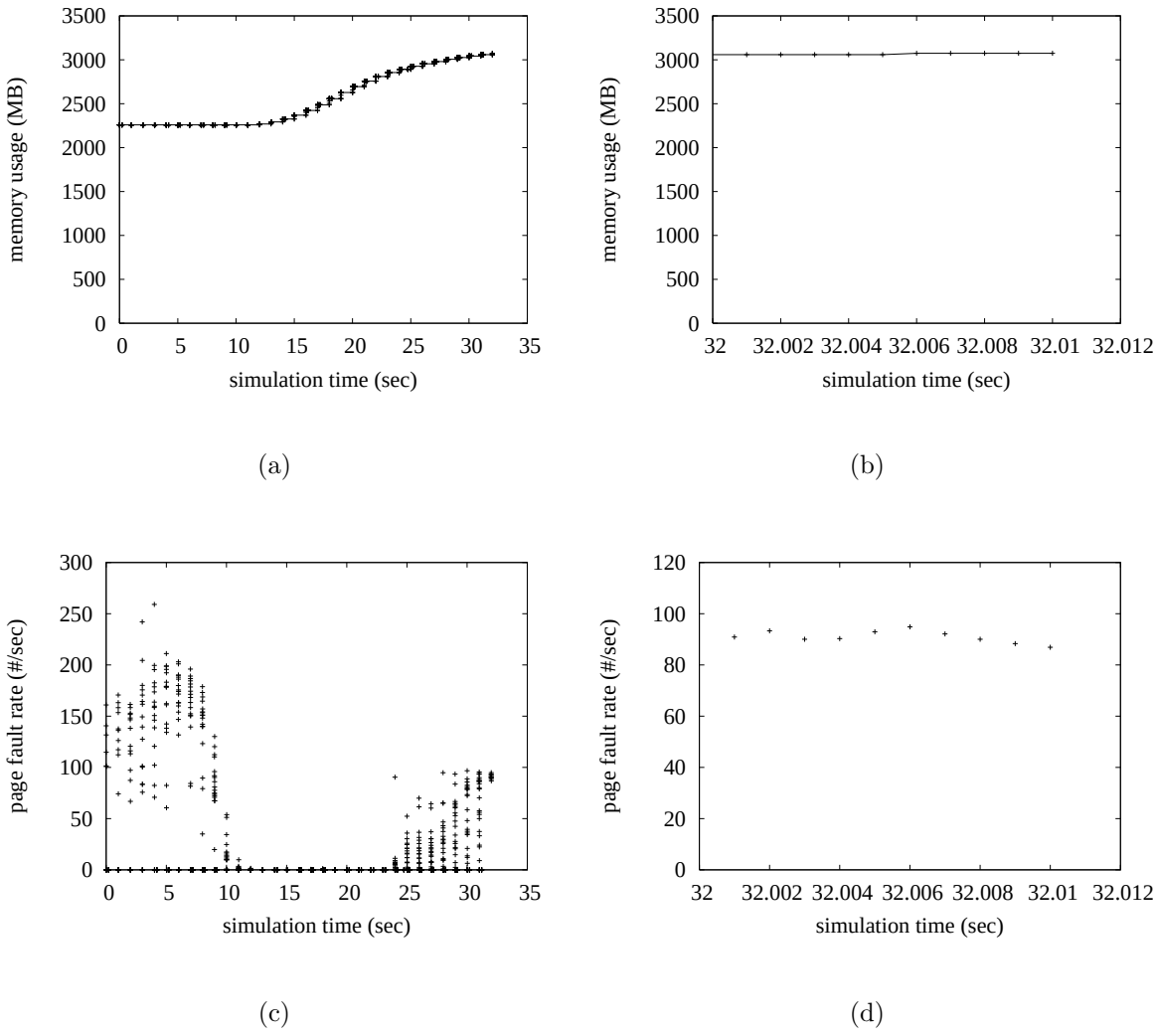
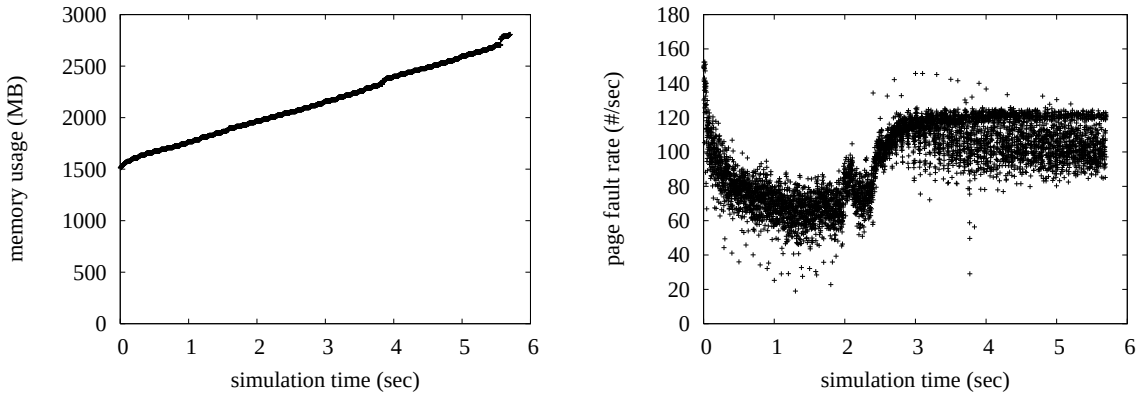


Figure 6.3. Worm local propagation of 8063-node AS topology on a single machine with 1GB memory. (a) Memory usage as a function of simulation time. (b) Blow-up of (a). (c) Page fault rate as a function of simulation time. (d) Blow-up of (c).

of page fault rate, we observe fluctuation of page fault rate at 1 second intervals. It finally converges to around 100 page faults per second. The increase of memory and page fault rate at 1 second intervals is due to the worm application parameter set-up, where each worm application sends probing packets at 1 second intervals.

Figure 6.3 shows worm local propagation simulation of a 8063-node AS topology. Figure 6.3(a) and Figure 6.3(c) show memory usage and page fault rate as a function of simulation time. Figure 6.3(b) and Figure 6.3(d) show their blow-up of the range after 32 second simulation time. We observe that memory usage exceeds the 1GB physical limit from the start reaching around 3GB eventually. The initial memory consumption of around 2.2GB is due to IP routing tables loaded at each simulation node before simulation starts. A noticeable increase of memory usage happens from 15 second simulation time at 1 second intervals. In terms of page fault rate, we observe fluctuation of page fault rate at 1 second intervals. It finally converges to around 90 page faults per second. The increase of memory and page fault rate at 1 second intervals is due to the worm application parameter set-up.



(a) (b)

Figure 6.4. Distributed client/server simulation of 6582-node topology on a single machine with 1GB memory. (a) Memory usage as a function of simulation time. (b) Page fault rate as a function of simulation time.

Figure 6.4 shows distributed client/server simulation of a 6582-node AS topology. Figure 6.4(a) and Figure 6.4(b) show memory usage and page fault rate as a function of simulation time. Thrashing occurs at around 2.7 second simulation time. We observe that memory usage exceeds the 1GB physical limit from the start reaching around 2.7GB eventually. The initial memory consumption of around 1.5GB is due to IP routing tables loaded at each simulation node before simulation starts. Memory usage increases linearly over time. Page fault rate is over around 50 page faults per second over the course of the simulation. It finally settles between 80 and 120 page faults per second.

6.1.4 Quantification of Performance Dilation

Quantification of Dilation in Single-machine Setting

We define metrics to quantify the degree of performance dilation due to VM paging. In a single-machine setting, one can simply use the completion time of an entire simulation measured as wall clock time divided by the computation time of an entire simulation measured as CPU user time plus system time to quantify performance dilation due to VM paging. But, this definition is too coarse to identify temporal regions affected by VM paging. Under the conservative synchronization mechanism which DaSSFNet employs, simulation progress across distributed simulators is synchronized at a fixed periodic simulation time interval—i.e., epoch. With the intention of extending metrics defined for a single-machine setting under distributed settings later, we measure the degree of dilation in the unit of a fixed simulation time interval, which corresponds to an epoch in distributed simulation. In a single-machine setting, dilation $d(i)$ of a given fixed simulation time interval i is defined as $l(i)/c(i)$, where $c(i)$ represents computation time of interval i measured as CPU user time plus system time and $l(i)$ represents completion time of interval i measured as wall clock time. If the CPU is fully utilized during interval i, $d(i) = 1$. The value of $d(i)$ indicates the degree of performance dilation relative to the case with 100% CPU utilization.

We quantify the performance dilation of an entire simulation duration by taking a weighted sum of dilations of all fixed simulation time intervals. For given fixed simulation time intervals $[1, t]$ for an entire simulation duration, we define dilation of an entire simulation duration as $\sum_{i=1}^{t} w(i) \cdot d(i)$, where $w(i) = l(i)/\sum_{j=1}^{t} l(j)$. The rationale behind this definition is that we focus on intervals of longer completion time, since they have a greater impact on simulation performance under VM.

In Figure 6.5, we demonstrate the performance of our metrics in the BGP simulation of a 4512-node AS topology. Figure 6.5 zooms in on the time period following 60 second simulation time where thrashing occurs. We set a fixed simulation time interval to 1 millisecond. Figure 6.5(a) and Figure 6.5(b) show the completion time and computation time as a function of simulation time. Figure 6.5(c) shows the dilation as a function of simulation time. Figure 6.5(d) shows the weighted sum of dilations over all intervals as a function of simulation time. We observe a significant increase of the dilation and weighted sum of all dilations near 60.02 second simulation time. We observe that the computation time approaches 0 at around 60.025 simulation time. Note that, when the computation time approaches 0, computation time alone is not sufficient for evaluating the degree of dilation.

Impact of Resource Availability across Applications

In this section, we compare the impact of resource availability on benchmark applications by varying the physical memory of a single machine in the range 512MB, 768MB, 1GB, and 2GB. This also serves as a performance of our dilation metrics. Simulation duration is fixed for each application across different physical memory configurations. Figure 6.6(a) shows the maximum memory usage as a function of physical memory limit. Memory usages across physical memory configurations are the same for each application due to fixed simulation duration. Figure 6.6(b) shows dilation as a function of physical memory limit for all benchmark applications. As physical memory increases, each application experiences less dilation. A comparison

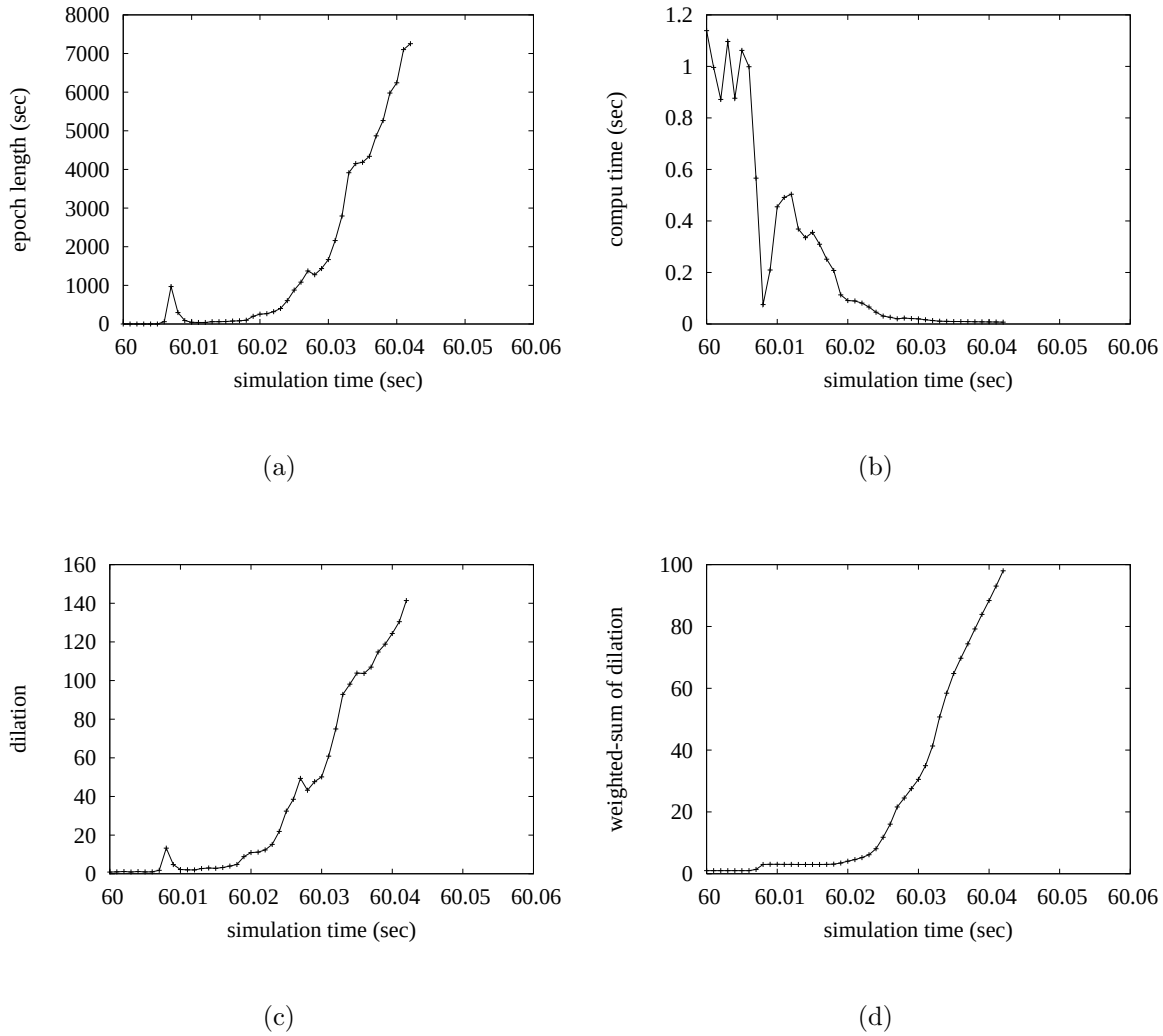
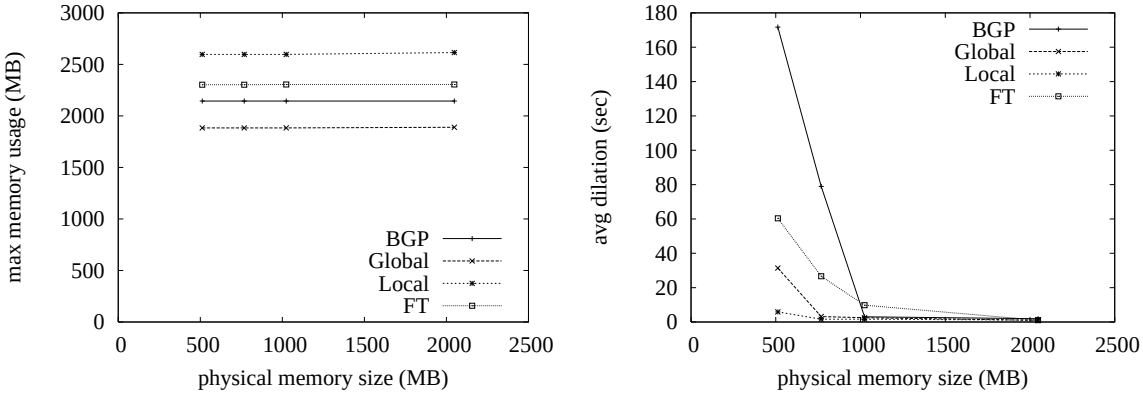


Figure 6.5. BGP simulation of a 4512-node AS topology on a single machine with 1GB memory. (a) Completion time as a function of simulation time. (b) Computation time as a function of simulation time. (c) Dilation as a function of simulation time. (d) Weighted-sum of dilations over all intervals as a function of simulation time.

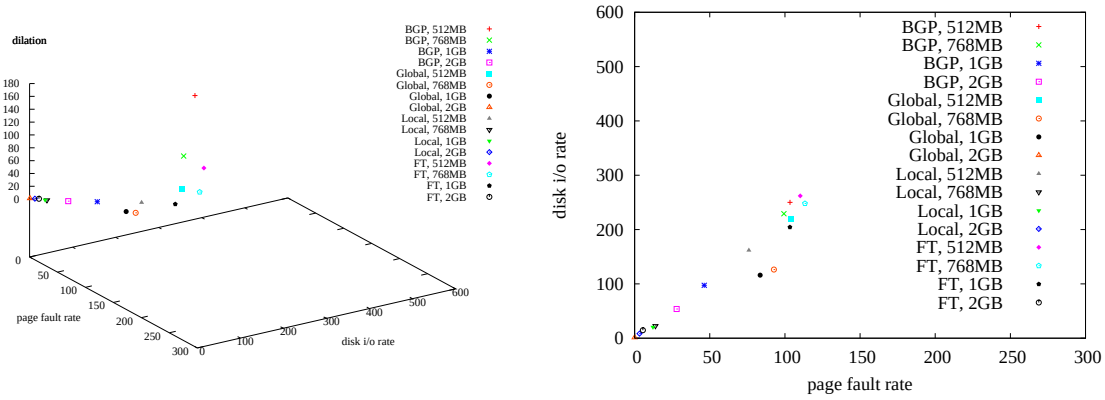
of BGP and worm local ("Local") applications in Figure 6.6(a) and Figure 6.6(b) shows that more memory usage does not imply more severe dilation.

Figure 6.7(a) shows dilation as a function of average page fault rate and average disk I/O rate for all benchmark applications with different physical memory configurations. Average page fault rate is calculated as a weighted sum of page fault rates for



(a) (b)

Figure 6.6. (a) Maximum memory usage as a function of physical memory limit for different benchmark applications. (b) Dilation as a function of physical memory limit for different benchmark applications.



(a) (b)

Figure 6.7. (a) Dilation as a function of average page fault rate and average disk I/O rate for all benchmark applications with different physical memory configurations. (b) Average page fault rate vs. average disk I/O rate for all benchmark applications with different physical memory configurations.

all fixed simulation time intervals, using the same weighting scheme as the weighted sum of dilations for all intervals. Average disk I/O rate is calculated in the same way. Its unit is the number of disk I/O operations per second. Figure 6.7(b) shows a scatter plot of average page fault rate vs. average disk I/O rate. In Figure 6.7(b), we observe that disk I/O rate is proportional to page fault rate which is approximately invariant across applications and physical memory configurations. This is due to the way that Linux handles page faults mapping them to disk I/O operations. As seen in Figure 6.7(a), when thrashing occurs, applications exhibit a drastic increase in dilation, encountering around 110 page faults per second and around 220 disk I/O operations per second.

Quantification of Dilation in Distributed Network Simulation

In this section, we describe our extension of the dilation metrics in distributed network simulation. The main issue is given by the fact that dilation should reflect the impact of bottleneck machines that slow down the entire simulation due to heavy VM paging. As we noted when defining dilation metrics in a single-machine setting, the accounting of dilation at per epoch granularity is adequate to accurately incorporate the impact of bottleneck machines and distinguish delay due to VM paging at bottleneck machines from delay due to distributed synchronization. In distributed simulation, dilation $d(i)$ of epoch i is defined as $l(i)/c^m(i)$. Here $c^m(i)$ represents the maximum computation time of epoch i across all the machines, which is measured as CPU user time plus system time, and $l(i)$ represents the completion time of epoch i measured as wall clock time. Note that $l(i)$ is defined for an epoch i independently from individual machines. We summarize per-epoch dilation across all epochs, by taking a weighted sum of per-epoch dilations. For given epochs $[1, t]$ for an entire simulation duration, we define dilation of an entire simulation duration as $\sum_{i=1}^{t} w(i)d(i)$, where $w(i) = l(i)/\sum_{j=1}^{t} l(j)$.

We demonstrate the performance of the dilation metrics in a BGP simulation of a 6582-node AS topology using 16 PCs with 1GB memory each. Figure 6.8 shows the memory usage and memory referencing behavior of distributed BGP simulation. Figure 6.8(a) shows the maximum memory usage at each of the 16 machines. We observe that only machine 1 consumes more memory than its 1GB physical memory. Figure 6.8(b) shows the average page fault rate at each machine. The average page fault rate is calculated as a weighted sum of per-epoch page fault rate at each machine, applying the same weighting scheme as the weighted sum of per-epoch dilation. We observe that only machine 1 exhibits significant page fault rate. Figure 6.8(c) and Figure 6.8(d) show memory usage and average page fault rate as a function of simulation time at machine 1. We show only the period after 90 simulation second where thrashing occurs.

In Figure 6.9, we also focus on the period after 90 simulation second where thrashing occurs. Figure 6.9(a) shows per-epoch completion time as a function of simulation time. Figure 6.9(b) shows per-epoch maximum computation time across all machines as a function of simulation time. Figure 6.9(c) shows per-epoch dilation as a function of simulation time. Figure 6.9(d) shows the weighted sum of per-epoch dilations over all epochs as a function of simulation time. We observe a significant increase of per-epoch completion time and a decrease of per-epoch maximum computation time at around 90.1 second simulation time. This translate to the increase of per-epoch dilation at around 90.1 simulation second and, hence, the increase of the weighted sum of per-epoch dilations at around 90.1 simulation second. The jump of per-epoch dilation and the weighted sum of per-epoch dilations at around 90.1 simulation second occurs synchronously with average page fault rate at the bottleneck machine—machine 1—reaching a plateau as seen in Figure 6.8(d).

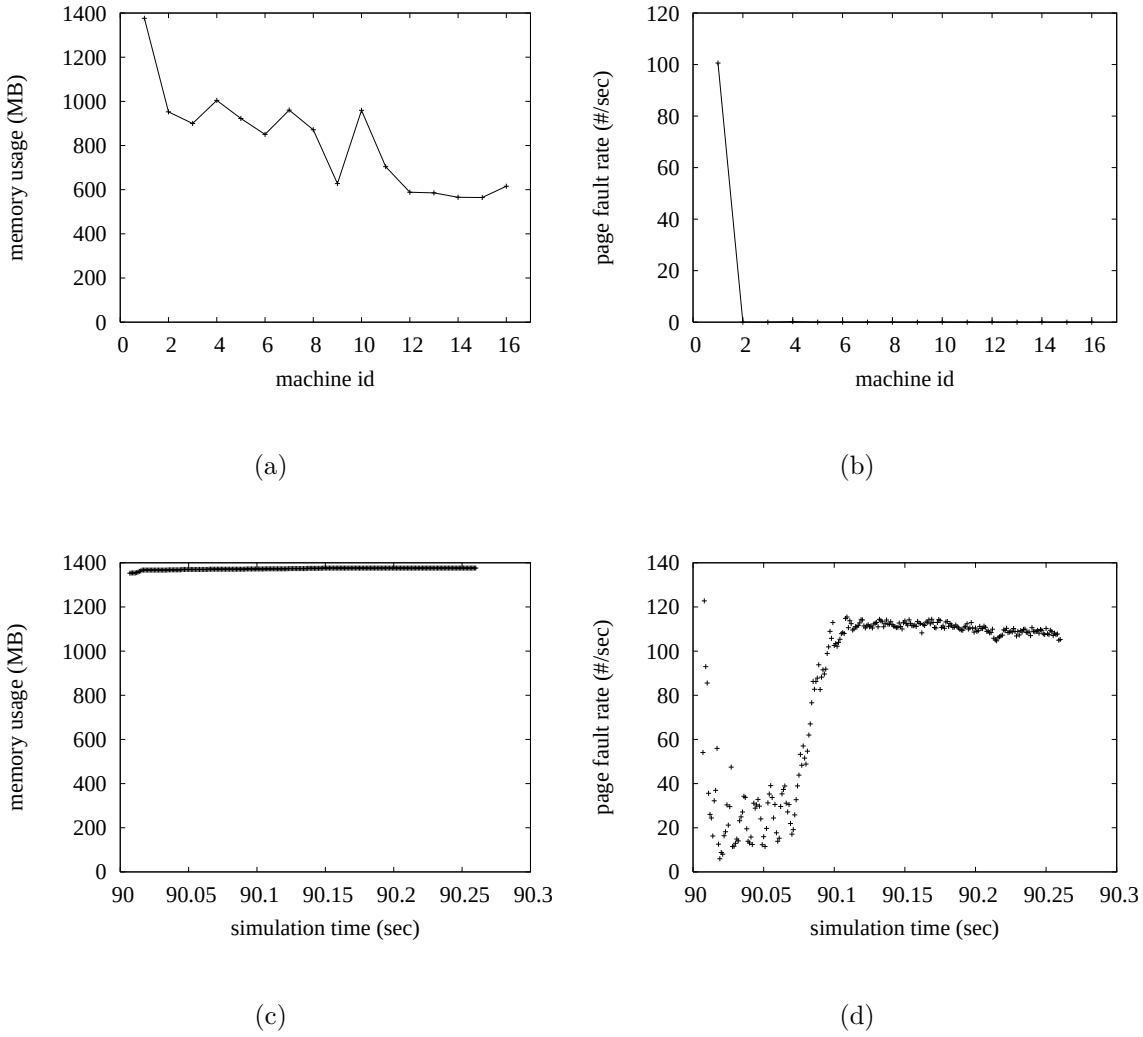


Figure 6.8. BGP simulation of 6582-node AS topology using 16 PCs with 1GB memory each. (a) Maximum memory usage at each machine. (b) Average page fault rate at each machine. (c) Memory usage as a function of simulation time at machine 1. Blow-up of the period after 90 second simulation time. (d) Average page fault rate as a function of simulation time at machine 1.

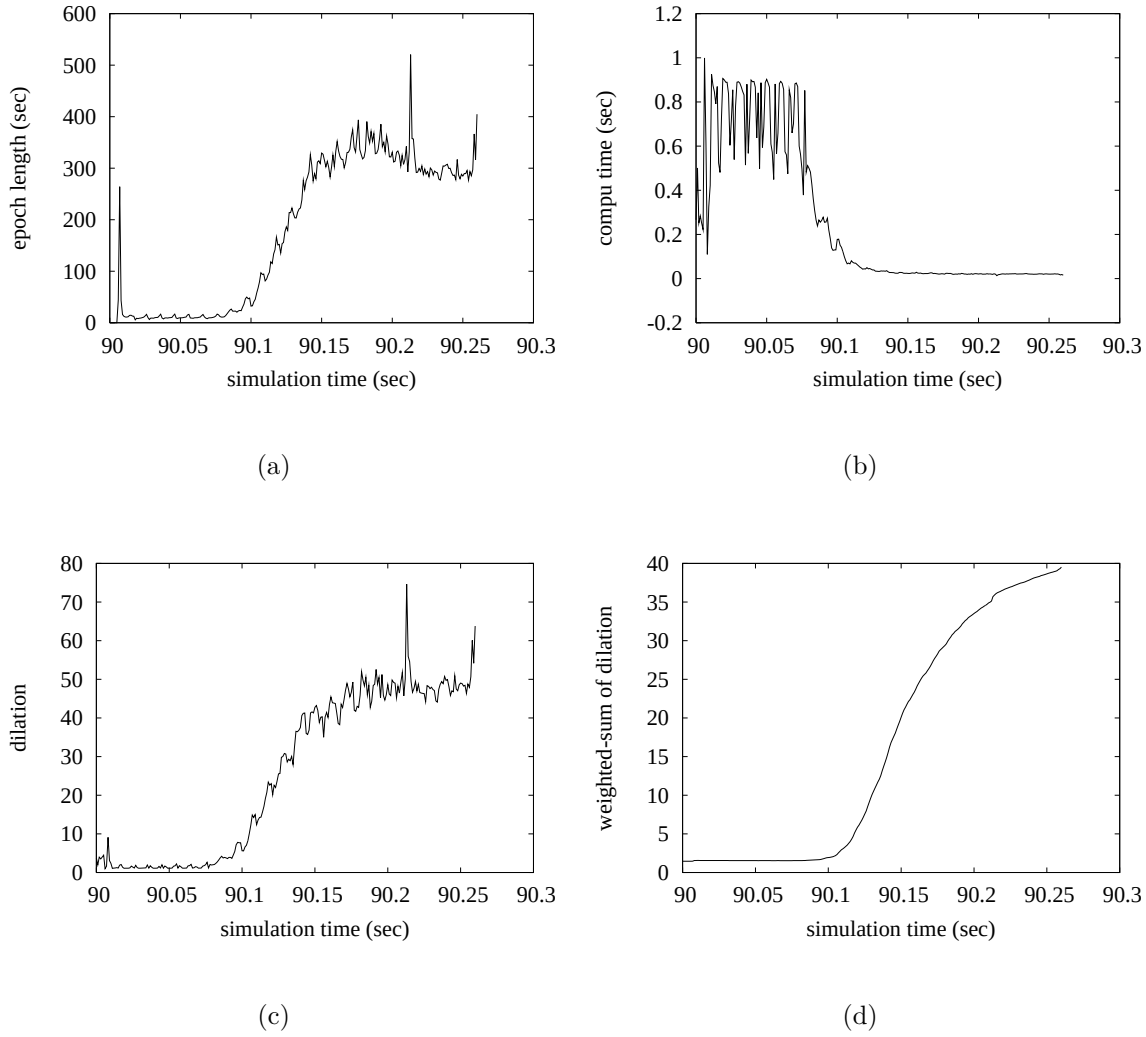


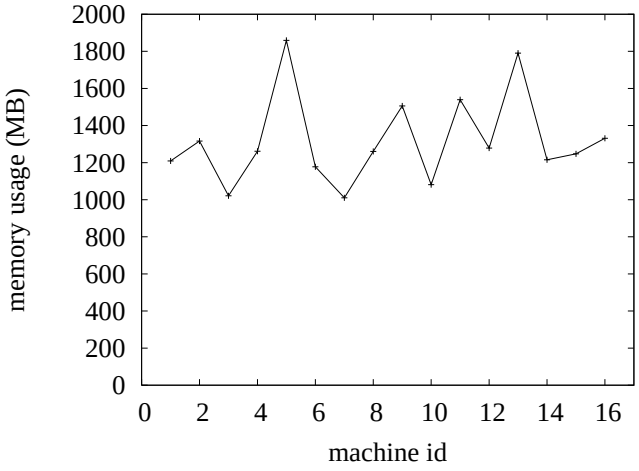
Figure 6.9. BGP simulation of 6582-node AS topology using 16 PCs with 1GB memory each. Blow-up of the period after 90 simulation second. (a) Per-epoch completion time as a function of simulation time. (b) Per-epoch maximum computation time across all machines as a function of simulation time. (c) Per-epoch dilation as a function of simulation time. (d) Weighted-sum of per-epoch dilations over all epochs as a function of simulation time.

6.1.5 Impact of Message Memory Imbalance to the Thrashing

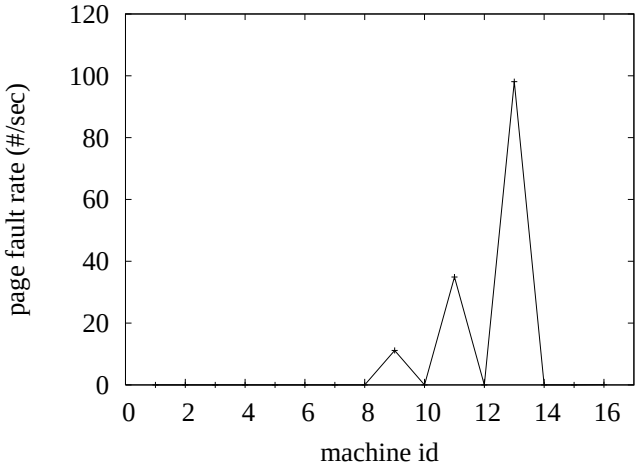
Memory Usage vs. Page Fault Rate

High memory usage does not always mean high page fault rate. In this section, we demonstrate this with worm local simulation of a 14577-node AS topology using 16 PCs with 1GB memory each. We used the C_i (total) cost metric for partitioning. Let t_t be the simulation time when thrashing occurs in C_i (total) cost metric case, and let t_m be the simulation time when thrashing occurs in M_i (max) cost metric case. Memory usage and average page fault rate are measured at t_m simulation second where $t_t < t_m$. It means that measurement is made while the simulation is experiencing thrashing. Figure 6.10 shows the maximum memory usage at each machine and the average page fault rate at each machine. In Figure 6.10(a), we observe that machine 5 and machine 13 consume the most amount of memory, and machine 9 and machine 11 come next. In terms of the average page fault rate shown in Figure 6.10(b), machine 13 experiences the most page fault rate, and machine 11 and machine 9 come next in order. However, machine 5 does not experience a noticeable page fault rate.

In order to find the causes underlying the difference in page fault rate, we focus on the two memory-overloaded machines—machine 5 and machine 13. Figure 6.11 shows the memory usage and page fault rate at the two memory-overloaded machines as a function of simulation time. In Figure 6.11(a), we observe that machine 5 consumes significant memory (around 1500MB out of around 1900MB), which exceeds the 1GB physical memory limit, at the start of the simulation. Figure 6.11(b) shows that machine 5 undergoes high page fault rate during the initial phase of the simulation, but not afterwards. On the other hand, in Figure 6.11(c), we observe that machine 13 consumes around 500MB of memory at the start of the simulation and increases its memory consumption gradually over the course of the simulation, exceeding the 1GB physical memory limit at around 0.225 simulation second. Figure 6.11(d) shows that machine 13 experiences an increase in page fault rate from around 0.27 simulation



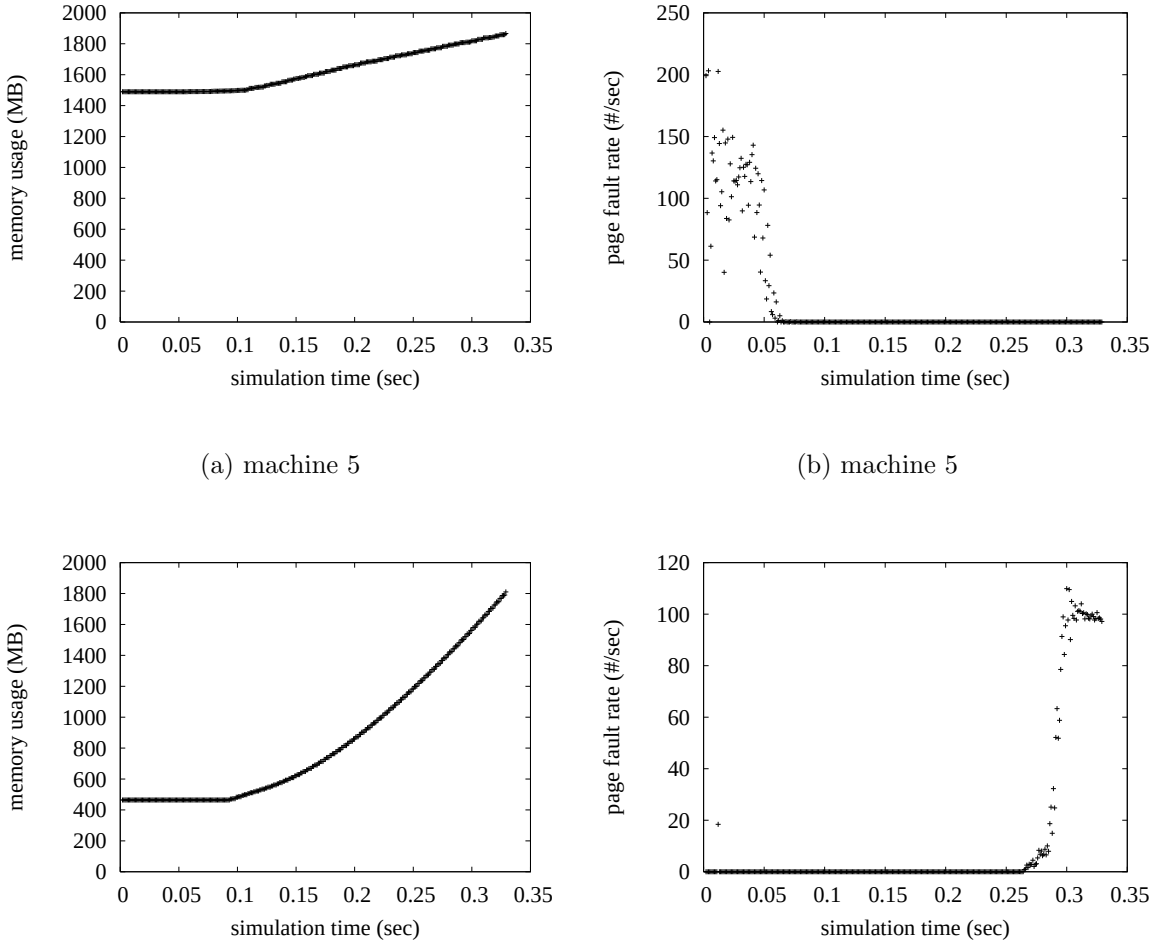
(a)



(b)

Figure 6.10. Worm local simulation over 16 PCs with 1GB memory each. Measurement is made while the simulation is experiencing thrashing. (a) Maximum memory usage at each machine. (b) Average page fault rate at each machine.

second, reaching a plateau of around 100 page faults per second at around 0.3 simulation second. The results show that the main difference of the two memory-overloaded



(a) machine 5

(b) machine 5

(c) machine 13

(d) machine 13

Figure 6.11. Memory usage and page fault rate at the two memory-overloaded machines as a function of simulation time. (a) Memory usage as a function of simulation time at machine 5. (b) Page fault rate as a function of simulation time at machine 5. (c) Memory usage as a function of simulation time at machine 13. (d) Page fault rate as a function of simulation time at machine 13.

machines in their page fault rate trajectories is due to the difference in their memory usage trajectories over time. One important fact in their memory usage trajectories is that, although their maximum memory consumptions are comparable (1859MB vs. 1790MB), their initial memory consumptions are significantly different (1489MB

vs. 464MB). The initial memory consumption is due to loading per-node IP routing tables.

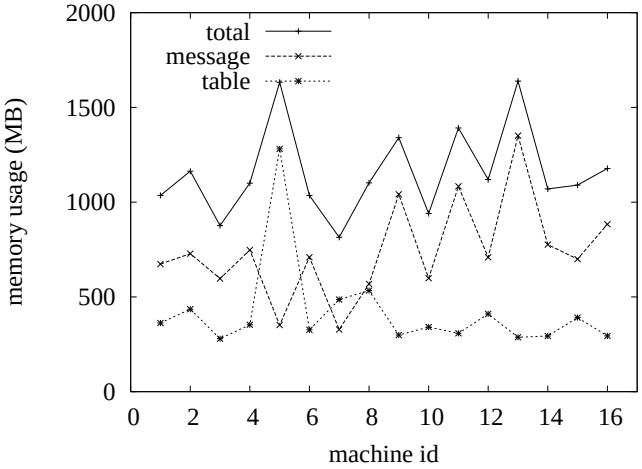


Figure 6.12. Per-machine maximum memory usage showing the memory allocated for message events and memory allocated for tables.

Figure 6.12 shows per-machine maximum memory usage with respect to memory allocated for message events and memory allocated for tables. We observe that table memory is dominant in machine 5's memory usage, whereas message memory is dominant in machine 13's memory usage. Moreover, we observe that the trajectory of per-machine message memory and per-machine average page fault rate shown in Figure 6.10(b) correlate to a high degree.

Discussion

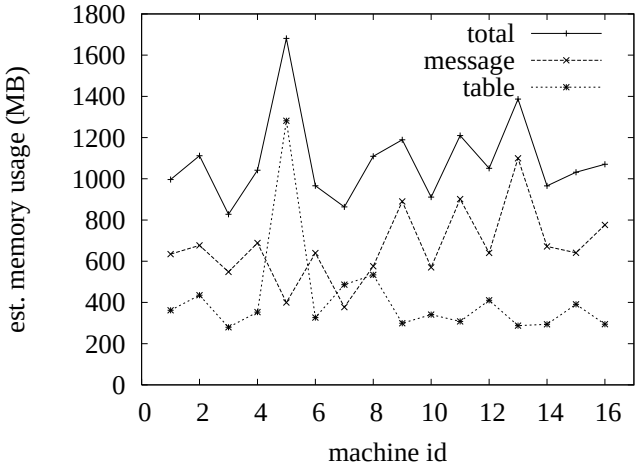
What happens to message memory at machine 13 in Figure 6.12, which is more than 1GB? In general, message memory—memory allocated for instantiating message events—lacks locality of reference for two reasons. First, message memory is dynamic and short-lived compared to table memory for instantiating protocol tables such as IP routing tables and BGP tables. In DaSSFNet, in particular, IP routing table entries—calculated statically—are loaded (i.e., instantiated) during initializa-

tion before simulation starts and table memory is released after simulation completes. Second, messages are stored in queues at various protocol layers. The first-in-first-out (FIFO)-based access pattern of message queue conflicts with least-recently-used (LRU) page replacement policy. For example, let us consider a TCP send buffer. Application messages are stored at the end of the TCP send buffer. When memory consumption by the TCP send buffer exceeds a given physical memory limit, least recently used pages containing messages stored at the start of TCP send buffer are swapped out to disk following LRU page replacement. However, since a TCP segment is generated from the start of the TCP send buffer upon receipt of TCP acknowledgement, the evicted pages are going to be accessed resulting in page faults. In contrast, the memory access pattern of table memory obeys a locality of reference. For example, let us consider a worm local simulation of a 14577-node AS topology. For each node, DaSSFNet instantiates an IP routing table with 14576 entries whose the total size is 469KB. Due to the worm local propagation mechanism, scan packets are sent to a randomly chosen neighboring node. In the case of a node with three adjacent nodes, at most 3 out of 14576 entries are accessed (i.e., locality of reference) over the course of simulation. Hence, the rest of the 14573 entries need not to be present in main memory when memory usage exceeds the physical memory limit.

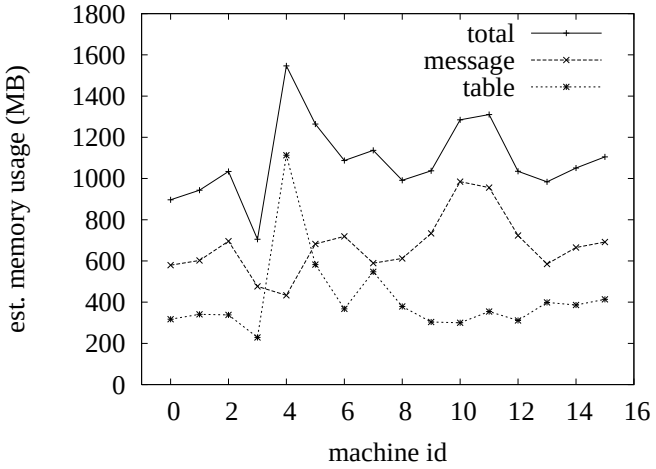
Message Memory Balance for Speed-up of Completion Time

In the previous section, we showed that we can use message memory imbalance across partitions as an indicator of average page fault rate distribution across partitions. In this section, we investigate the idea that, if we can estimate message memory imbalance across partitions, we can delay the start of thrashing for speed-up of completion time by finding a network partitioning with more balanced message memory distribution. First, we discuss how to estimate message memory imbalance for a given network partitioning. Given a network partitioning instance, we estimate per-partition memory usage by summing up M_i (max) of all nodes in a partition.

Table-related memory allocation is not as dynamic as message-related memory allocation. In particular, if IP routing table is the main memory-consuming component



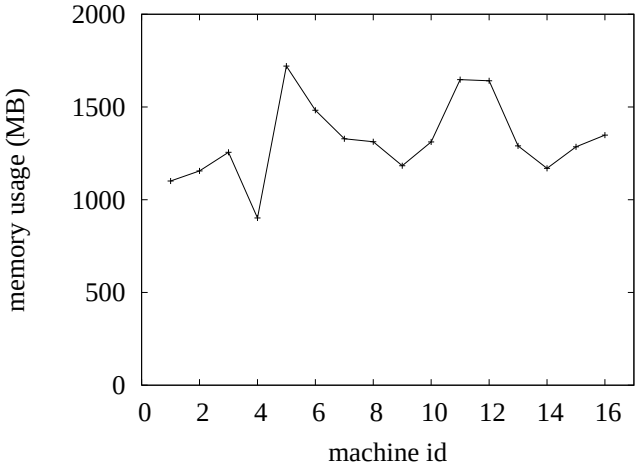
(a)



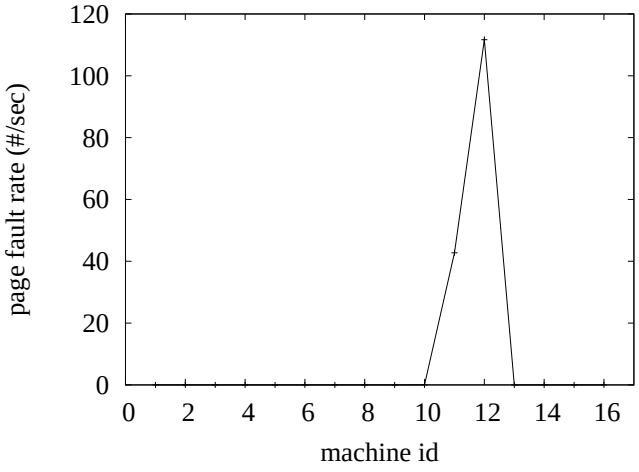
(b)

Figure 6.13. Estimation of per-partition total memory requirement, message memory requirement, and table memory requirement for two different network partitioning instances.

related to tables, a constant amount of memory is required for each node. We esti-



(a)



(b)

Figure 6.14. Worm local simulation using 16 PCs with 1GB memory each using the network partitioning in Figure 6.13(b). Measurement is made while the simulation is experiencing thrashing. (a) Maximum memory usage at each machine. (b) Average page fault rate at each machine.

mate per-partition message memory requirement by subtracting the sum of per-node table memory requirement from the per-partition memory usage.

Next, we consider two network partitioning instances with different random seeds in Metis. Figure 6.13 shows estimation of per-partition memory requirement, message memory requirement, and table memory requirement for two different network partitioning instances. Figure 6.13(a) shows estimation of message memory distribution for the case used in Figure 6.10. We observe that estimation of message memory imbalance in Figure 6.13(a) is strongly correlated with run-time measurement of message memory distribution in Figure 6.12 and average page fault rate distribution in Figure 6.10(b). Figure 6.13(b) shows another network partitioning instance for the same simulation scenario. Figure 6.14 shows the run-time measurement—maximum memory usage at each machine and average page fault rate at each machine—for the network partitioning instance. We observe that message memory distribution in Figure 6.13(b) is correlated with average page fault rate distribution in Figure 6.14(b).

In the case of the network partitioning in Figure 6.13(a), the peak message memory estimate is 1099MB, thrashing occurred at 0.291 second simulation time, and the wall clock time taken to complete t_m simulation seconds is 24105 seconds. In the case of the network partitioning in Figure 6.13(b), the peak message memory estimate is 985MB, thrashing occurred at 0.303 simulation second, and the wall clock time taken to complete t_m simulation seconds is 13640 seconds. We can delay the start of thrashing for speed-up of completion time by finding a network partitioning with more balanced message memory distribution.

6.2 Experimental Set-up

We used the same experimental set-up as Section 5.1 except for a few configuration changes with respect to hardware and OS set-up. In this section we focus on the configuration changes. We used the 16 2.4GHz machines that were used for evaluating Memory-CPU balancing. For experiments with virtual memory paging enabled, we had to take care of the following. Since Linux's virtual memory mechanism has evolved significantly over time, we reconfigured the testbed so that all participating

machines run the same version, Linux 2.4.25. We set each machine to have 1GB physical memory as a default configuration. To do so, we set Linux kernel parameter '`mem=1024M`' during bootstrapping which sets the memory recognized by the OS to be 1GB. The size of swap space is set to 7821MB for all machines. Table 6.2 shows the default memory configuration of 16 participating machines for experiments with virtual memory paging.

Table 6.2
Default memory configuration of 16 participating machines for experiments with virtual memory paging.

host name	RAM size	VM paging	swap space size
5 4GB-mem greeks	1GB	on	7821MB
5 2GB-mem greeks	1GB	on	7821MB
6 others	1GB	on	7821MB

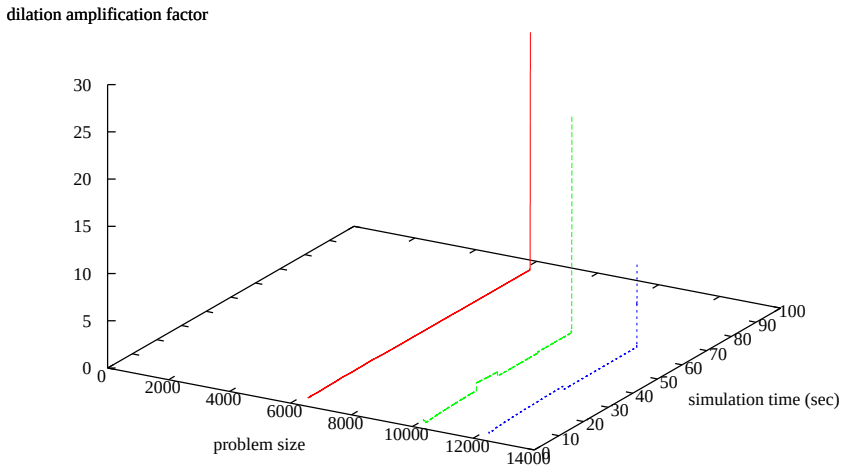
6.3 Impact of Thrashing: Comparison of Uniform vs. Max Cost Metrics

In this section, we first show the impact of thrashing comparing dilation of network partitionings under uniform and M_i cost metrics. In the remaining section, we compare their memory balancing performance and performance gain in terms of speed-up. Note that, when VM paging is turned on, the penalty of memory load imbalance exceeding a given physical memory limit is translated to delay in simulation completion. Hence, our main focus is evaluating performance gain in terms of speed-up by memory load balancing.

6.3.1 Impact of Thrashing

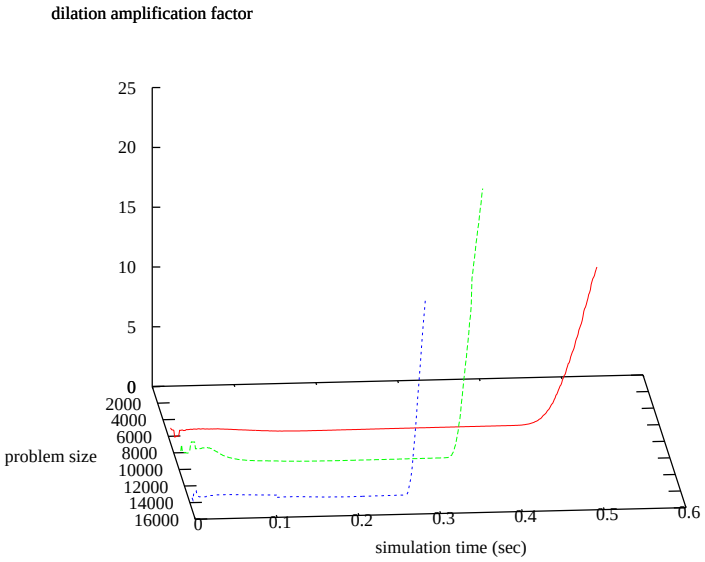
We evaluate the impact of thrashing comparing dilation of network partitionings with the uniform and M_i cost metrics. For this comparison, we define a metric—

dilation amplification factor—as dilation of the uniform cost metric divided by dilation of the M_i cost metric. Figure 6.15 and Figure 6.16 show dilation amplification factor as a function of problem size and simulation duration for various benchmarks applications using 16 PCs with 1GB memory each. As problem size or simulation duration is increased, the scale of a network simulation expands. We observe that the uniform cost metric delays simulation progress up to 26 times compared to the performance of the M_i cost metric in the case of BGP simulations; up to about 23 times, in worm local simulations; up to about 23 times, in worm global simulations. This is due to the impact of thrashing which started earlier in the case of the uniform cost metric stemming from memory load imbalance.

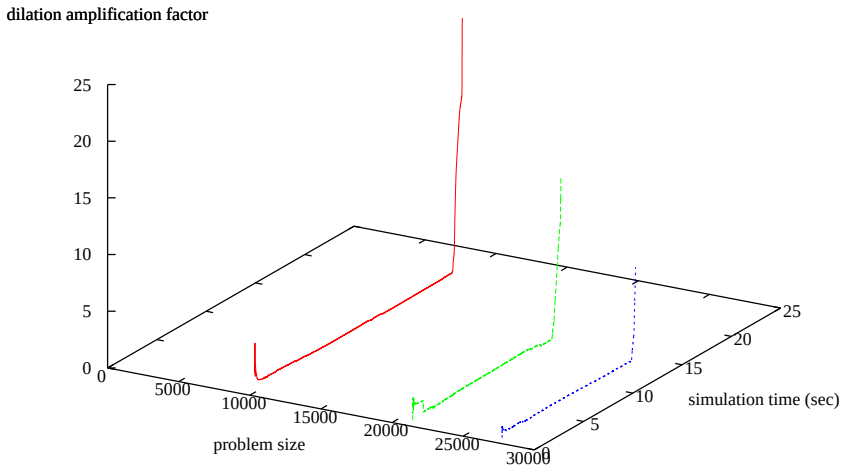


(a) BGP

Figure 6.15. Dilation amplification factor as a function of problem size and simulation duration for various benchmark applications using 16 PCs with 1GB memory each: BGP.



(a) Worm local



(b) Worm global

Figure 6.16. Dilation amplification factor as a function of problem size and simulation duration for various benchmark applications using 16 PCs with 1GB memory each: worm local and worm global.

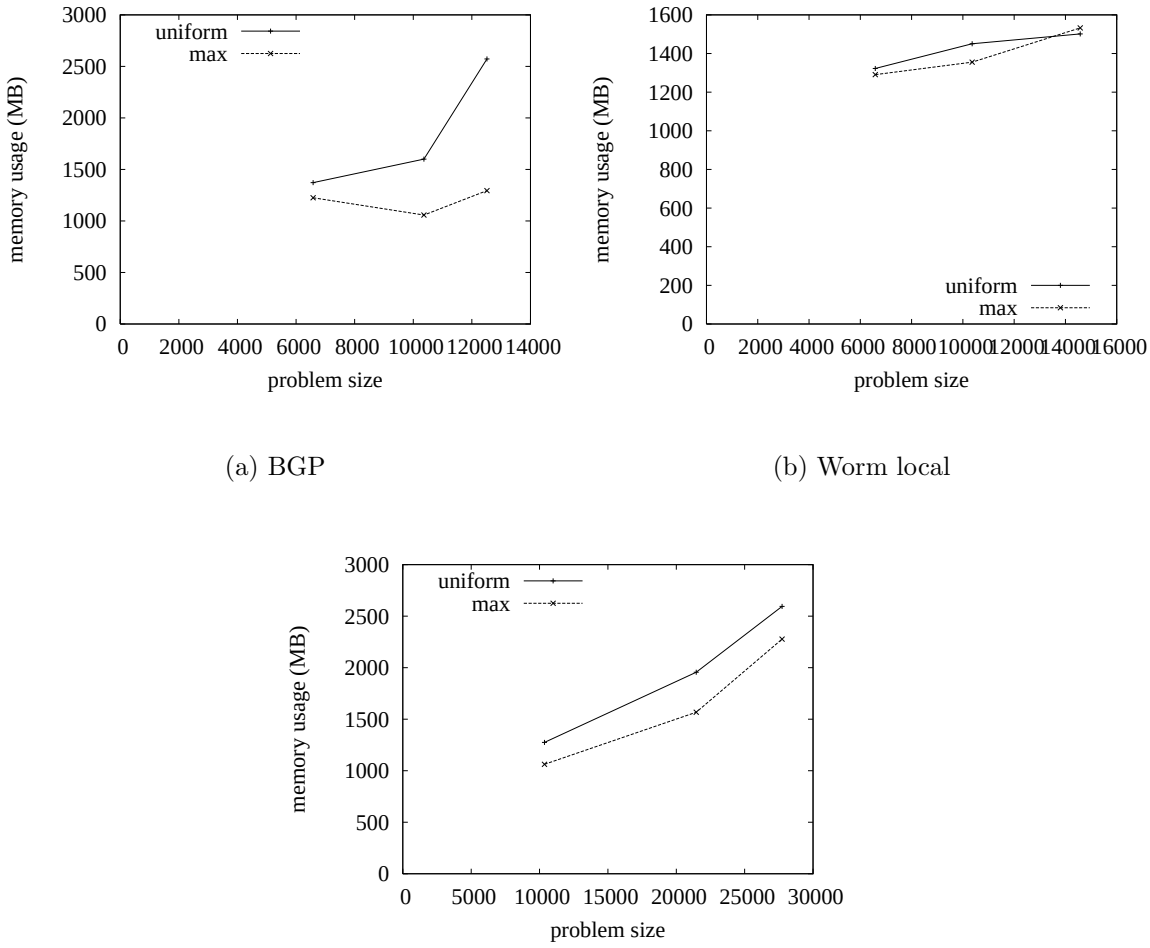
6.3.2 Memory Balancing Performance

Figure 6.17 compares the memory balancing performance of the uniform cost metric and the M_i cost metric as a function of problem size for various benchmark applications. Memory usage is with respect to the maximum across all machines. Let t_u be the simulation time when thrashing occurs in the case of the uniform cost metric, and let t_m be the simulation time when thrashing occurs in the case of M_i. We measured the memory usage at t_u and t_m simulation second in the uniform and M_i partitioning cases, respectively. Since $t_u < t_m$ for all cases, our comparison of memory balancing performance is conservative. We observe that the M_i cost metric outperforms the uniform cost metric in BGP and worm global propagation simulations, but in worm local propagation simulations both of the uniform and M_i cost metrics show comparable memory balancing performance. This is consistent with the memory balancing performance of the uniform and M_i (max) cost metrics shown in Figure 5.2 without involvement of VM paging.

We evaluate distributed memory utilization comparing the uniform and M_i (max) cost metrics. Memory utilization is defined as the sum of memory usage at each machine divided by the total physical memory size, i.e., 16×1GB. Figure 6.18 shows the performance results, where memory utilization of the uniform and M_i (max) cost metrics is measured at t_u and t_m simulation time, respectively. In worm local simulations, the M_i (max) cost metric achieves significant increase of memory utilization, consuming more than 100% with Linux VM paging support. In BGP simulations, memory utilization under the M_i (max) cost metric compared to that of the uniform cost metric depends on problem size. In worm global simulations, the M_i (max) cost metric achieves negligible increase of memory utilization.

6.3.3 Performance Gain: Speed-up

Ideally, speed-up in completion time should be evaluated by comparing completion time of two simulations—one with M_i cost metric and another with the uniform cost



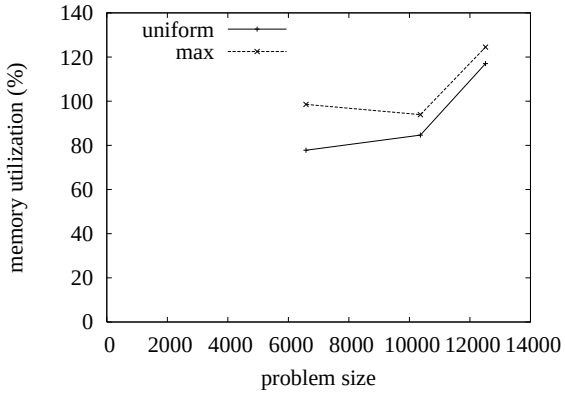
(a) BGP

(b) Worm local

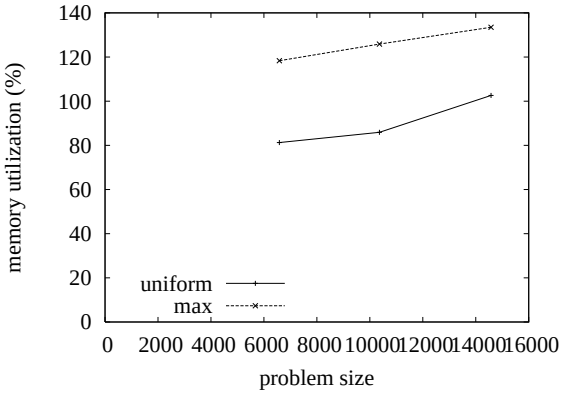
(c) Worm global

Figure 6.17. Memory balancing performance of uniform and M_i (max) cost metrics using 16 PCs with 1GB memory each.

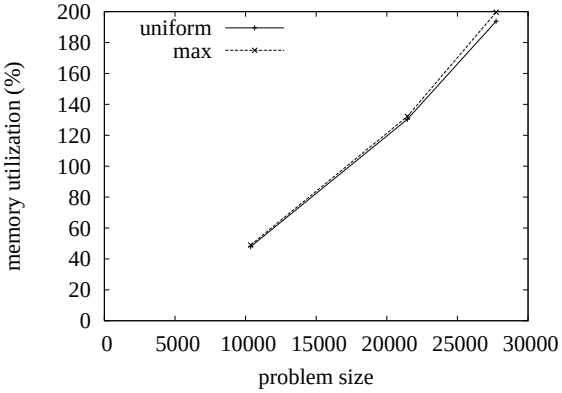
metric—where both of them run till t_m simulation time, i.e., the time instance when both experience thrashing. Recall that generally $t_u < t_m$ due to higher memory load imbalance under the uniform cost metric. Simulation progress under the uniform cost metric case slows down severely once thrashing occurs at t_u simulation second. We stopped a simulation with the uniform cost metric once it has run around 10 hours



(a) BGP



(b) Worm local



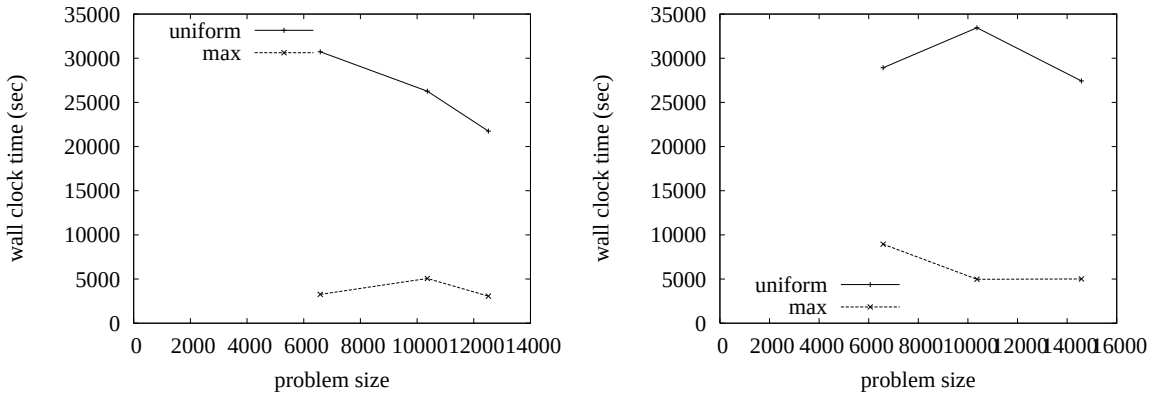
(c) Worm global

Figure 6.18. Memory utilization of the uniform and M_i maximum cost metrics as a function of problem size for various benchmarks applications. The memory utilization is calculated at t_u simulation time in the case of the uniform; at t_m, in the case of M_i.

of wall clock time even though it has not yet reached t_m simulation second. Let t_e be the simulation time at which a simulation under the uniform cost metric is stopped.

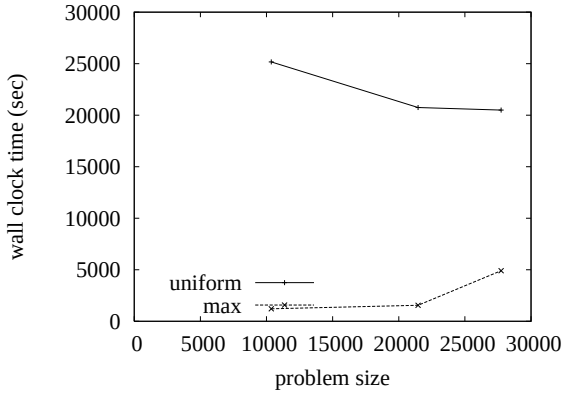
In this section, we compare the wall clock time taken to simulate $[t_u, t_e]$ in the uniform case and the wall clock time taken to simulate $[t_u, t_m]$ in the M_i (max) case. Figure 6.19 shows completion time as a function of problem size for various benchmark

applications. We observe that the M_i cost metric outperforms the uniform cost metric in all cases, the speed-up ratio ranging from 3.2 times (worm local, 6582) to 20.7 times (worm global, 10363). As with the comparison of memory balancing performance, the



(a) BGP

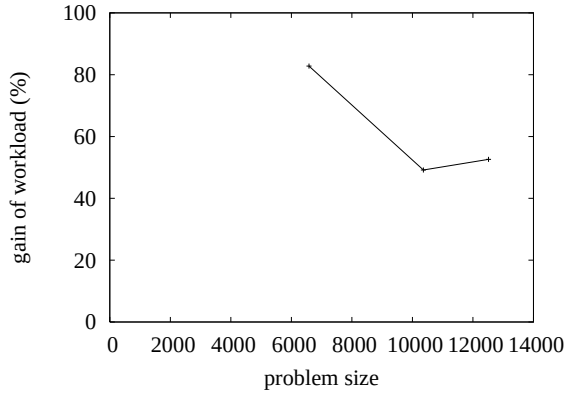
(b) Worm local



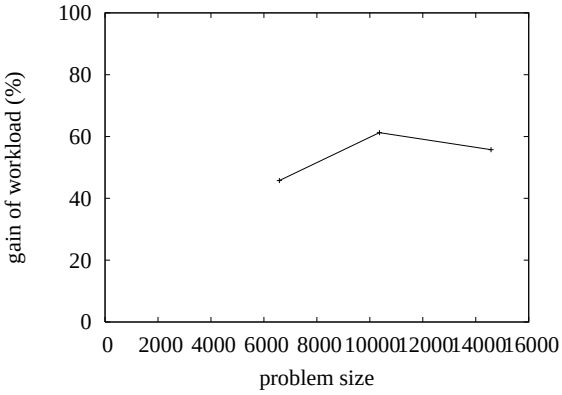
(c) Worm global

Figure 6.19. Completion time taken to simulate $[t_u, t_e]$ by uniform vs. completion time taken to simulate $[t_u, t_m]$ by M_i (max) as a function of problem size for various benchmark applications.

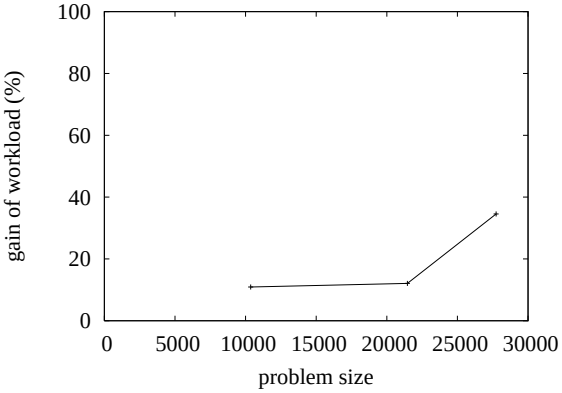
comparison is done conservatively since the wall clock time taken to simulate $[t_u, t_m]$ by the uniform cost metric is expected to be much longer than the time for $[t_u, t_e]$.



(a) BGP



(b) Worm local



(c) Worm global

Figure 6.20. Total message events processed in percentage during $[t_u, t_m]$ by M_i without experiencing thrashing. Results are shown as a function of problem size for various benchmark applications.

We consider performance gain of the M_i cost metric compared to that of the uniform cost metric in terms of the progress of simulation execution. Note that the completed simulation duration is not an appropriate metric to quantify the simulation progress due to the non-uniform workload distribution over simulation time. Since workload in network simulations is dominated by message events, we normalize a given simulation duration into the number of message events processed during

the simulation duration. Figure 6.20 shows the total message events processed—as a percentage—during $[t_u, t_m]$ by M_i without experiencing thrashing. The results are shown as a function of problem size for various benchmark applications. We observe significant performance gain for BGP and worm local propagation simulations, ranging from 49.2% to 82.8% in BGP and from 45.8% to 61.3% in worm local. In the case of worm global propagation simulation, we observe comparatively less performance gain, ranging from 10.9% to 34.5%.

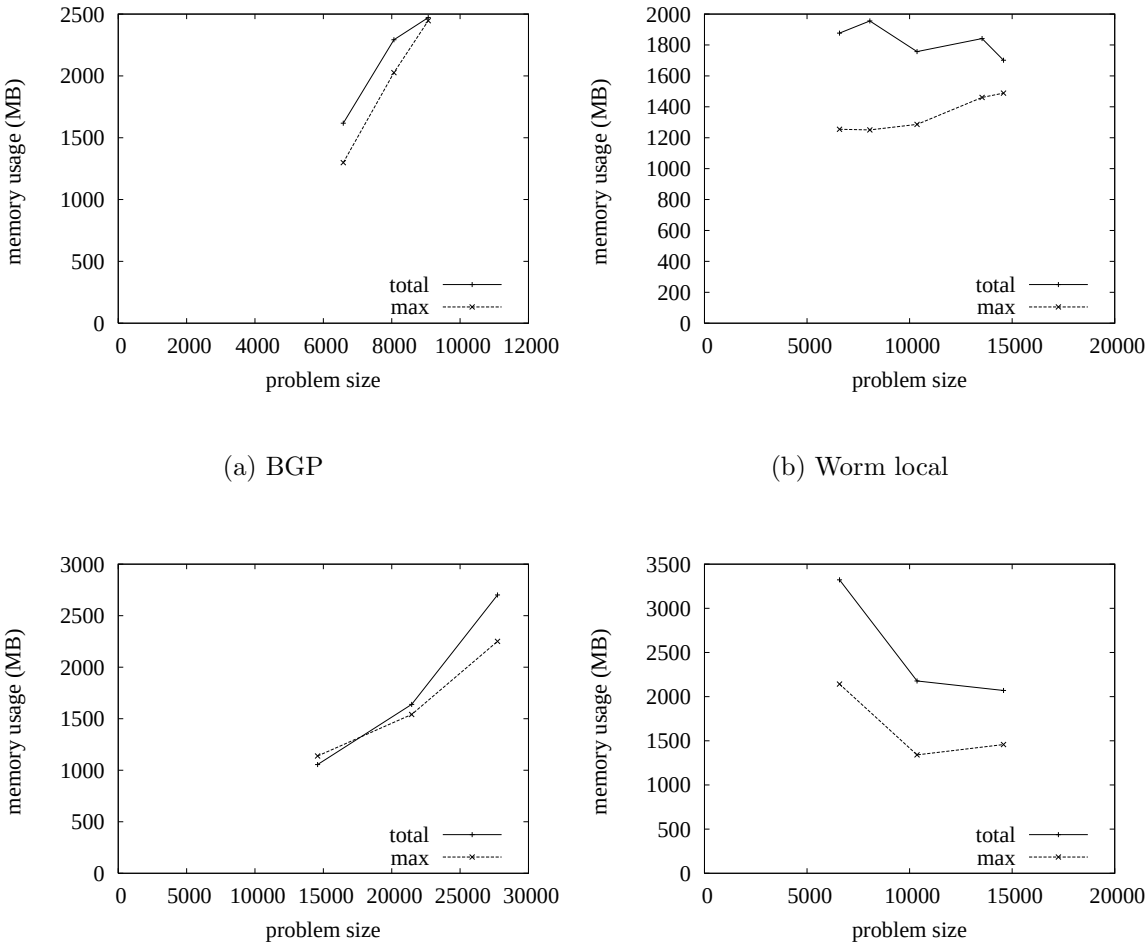
6.4 Comparison of Max vs. Total Cost Metrics

In this section, we compare the performance of M_i (max) and C_i (total) cost metrics with respect to memory balancing performance and performance gain in terms of speed-up for various benchmark applications. Note that, when VM paging is turned on, the penalty of memory load imbalance exceeding a given physical memory limit is translated to delay in simulation completion. Hence, our main focus is evaluating performance gain in terms of speed-up by memory load balancing.

6.4.1 Memory Balancing Performance

Figure 6.21 compares memory balancing performance of M_i (max) and C_i (total) cost metrics using 16 PCs with 1GB memory each. Let t_t be the simulation time when thrashing occurs in C_i (total) cost metric case, and let t_m be the simulation time when thrashing occurs in M_i (max) cost metric case. Memory balancing performance is measured with respect to the maximum memory usage across all machine at t_m simulation second except in the worm global 21460-node topology case. In the latter, memory balancing performance is measured at t_t simulation second since $t_m < t_t$. The time instance for measuring performance is chosen based on the assumption that, once thrashing occurs in both M_i and C_i cost metric cases, the speed of simulation execution degenerates in both cases and comparing their performance results is meaningless. The performance results are plotted as a function of problem

size for all benchmark applications. We observe that M_i (max) cost metric outperforms C_i (total) cost metric overall. In particular, we observe significant performance gaps in the worm local and distributed client/server simulations. This is consistent



(a) BGP

(b) Worm local

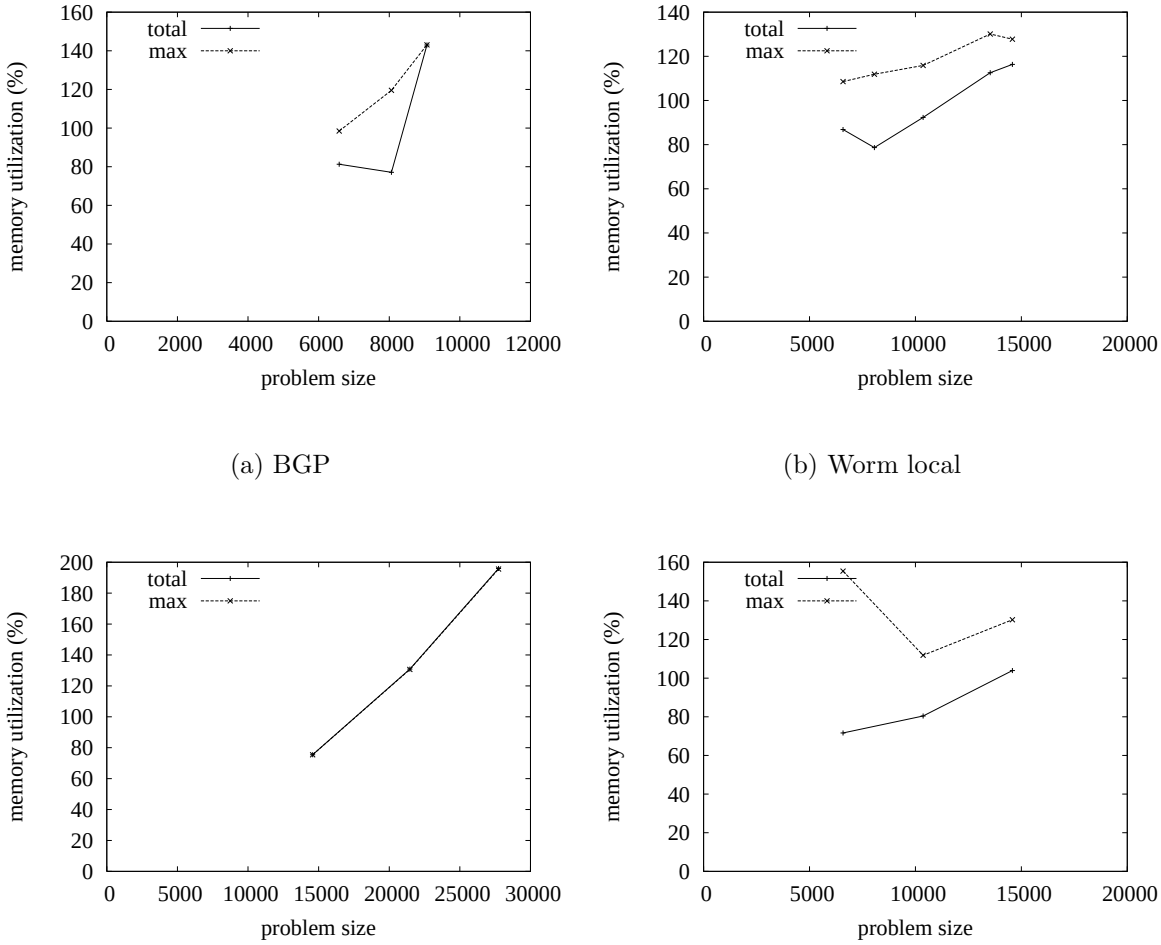
(c) Worm global

(d) Distributed client/server

Figure 6.21. Memory balancing performance of M_i (max) and C_i (total) cost metrics using 16 PCs with 1GB memory each.

with the memory balancing performance of M_i (max) and C_i (total) cost metrics shown in Figure 5.7 without VM paging.

Figure 6.22 shows memory utilization of M_i (max) and C_i (total) cost metrics using 16 PCs with 1GB memory each. Memory utilization is defined as the sum of memory usage at each machine divided by the total physical memory size, 16 ×



(a) BGP

(b) Worm local

(c) Worm global

(d) Distributed client/server

Figure 6.22. Memory utilization of M_i (max) and C_i (total) cost metrics using 16 PCs with 1GB memory each.

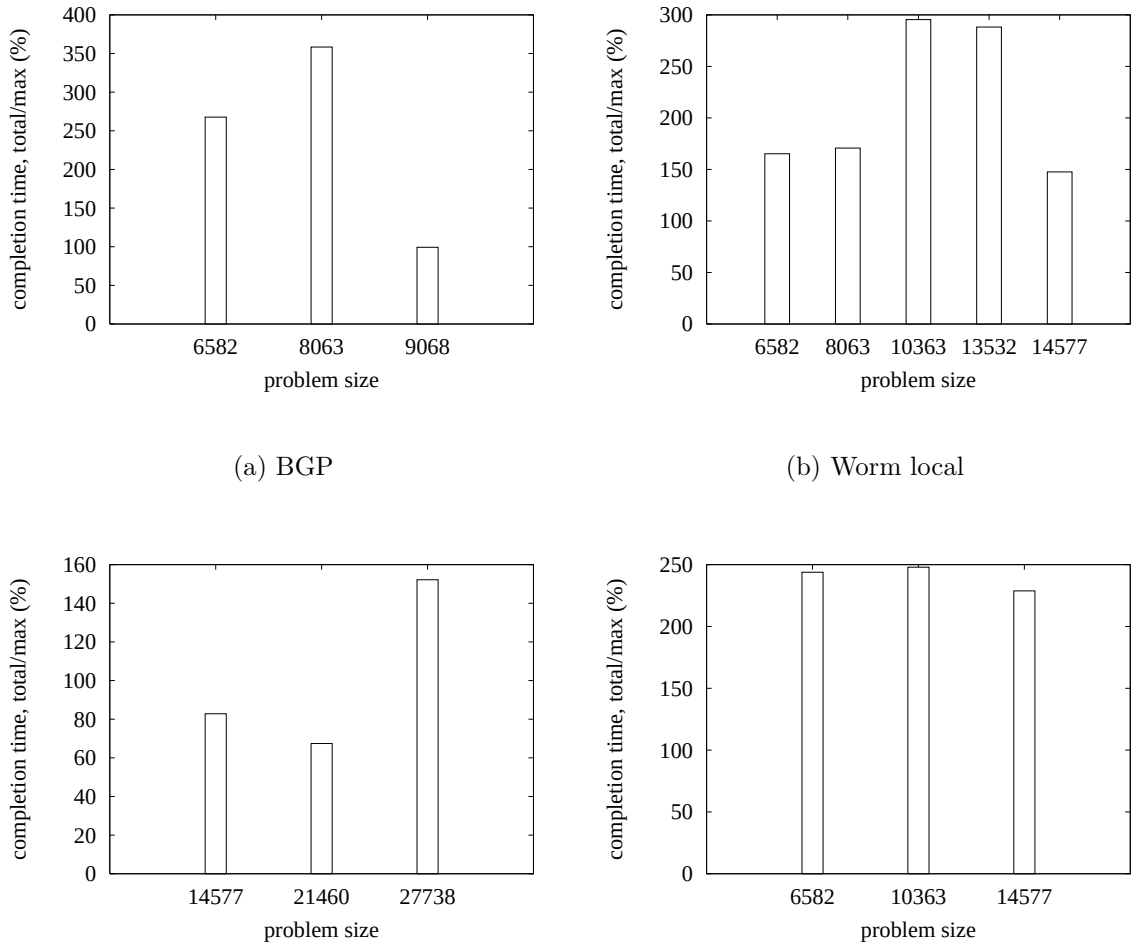
1GB. It is measured at t_m simulation time in the case of the M_i; at t_t, in the case of C_i. The memory utilization of M_i (max) and C_i (total) is shown as a function of problem size for all benchmarks applications. We observe that, overall, M_i achieves

increased memory utilization with the magnitude of the gap depending on benchmark application and problem size. In addition, M_i achieves more than 100% of memory utilization in most cases.

6.4.2 Performance Gain: Speed-up

Figure 6.23 compares M_i and C_i with respect to completion time. Completion time is measured as the wall clock time taken to simulate t_m simulation seconds (t_t if $t_m < t_t$). The ratio of C_i's completion time divided by M_i's completion time is plotted as a function of problem size for all benchmark applications. We observe that M_i outperforms C_i, overall, even though C_i is more advantageous for CPU load balancing than M_i as shown in Chapter 5. In BGP simulations, C_i takes up to 3.5 times longer than M_i. In worm local simulations, C_i takes up to around 3 times longer than M_i. In distributed client/server simulations, C_i takes more than 2 times longer for all problem sizes. In worm global simulations, C_i outperforms M_i for 14577-node and 21460-node topologies, but M_i outperforms C_i for the 27738-node topology, C_i taking 1.5 times longer than M_i.

Let $n(t)$ be the number of message events that are processed from 0 second simulation time until t second simulation time. We define performance gain $\gamma(M_i, C_i)$ of M_i over C_i with respect to simulation progress as $(n(t_m) - n(t_t))/n(t_t)$. $\gamma(M_i, C_i)$ denotes the number of message events that the M_i case processes additionally without experiencing thrashing compared to the C_i case. A negative value indicates that $t_m < t_t$. Figure 6.24 shows performance gain $\gamma(M_i, C_i)$ of M_i over C_i with respect to simulation progress as a function of problem size for all benchmark applications. $\gamma(M_i, C_i)$ is shown in percentage. We observe significant performance gain for the worm local and distributed client/server simulations. In the case of the worm local simulation, the performance gain ranges from 15.8% to 48.3%. In the case of the distributed client/server simulation, the performance gain ranges from 58.3% to 151.9%. In the case of BGP, the performance gains are 18.8% and 29.3% in the 6582-node



(a) BGP

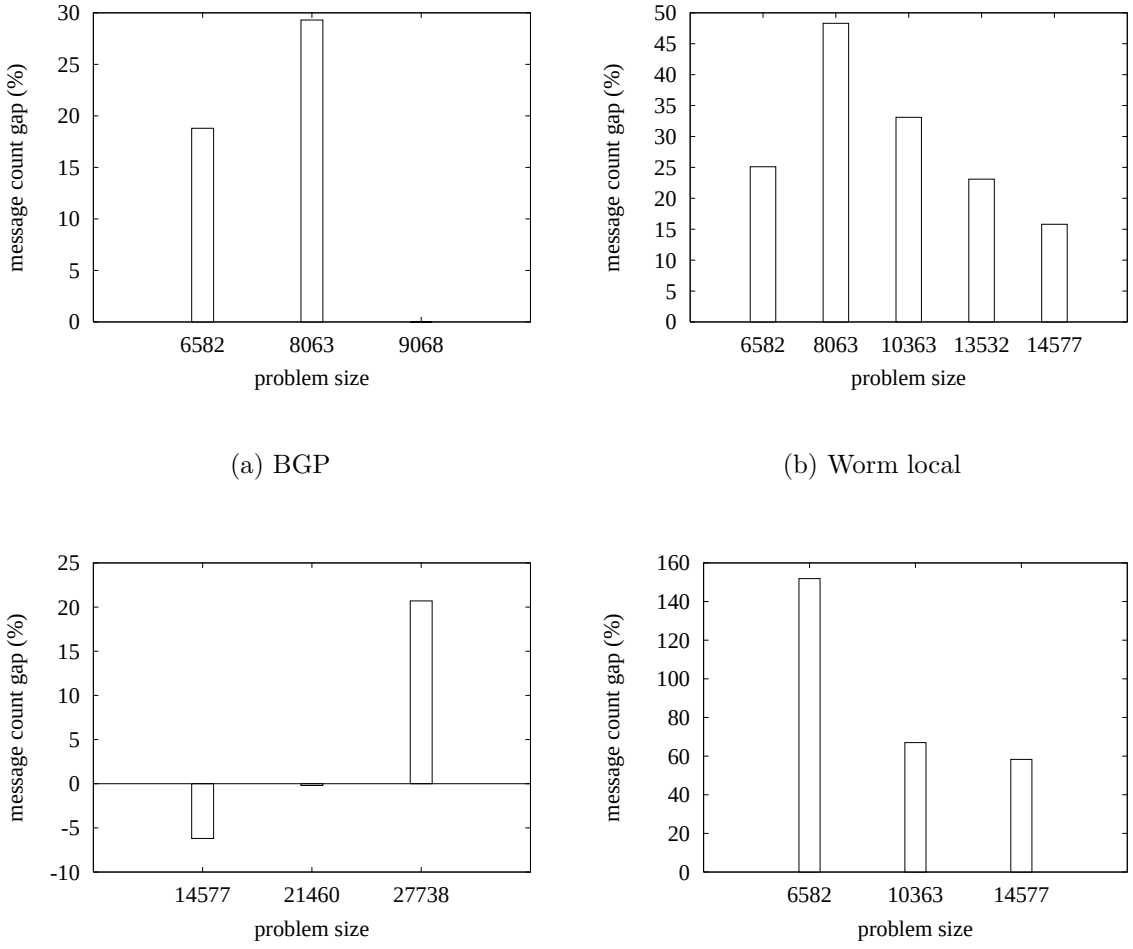
(b) Worm local

(c) Worm global

(d) Distributed client/server

Figure 6.23. Comparison of M_i and C_i with respect to completion time. Completion time is measured as the wall clock time taken to simulate t_m simulation seconds. The ratio of C_i's completion time divided by M_i's completion time is plotted as a function of problem size for all benchmark applications.

and 8063-node AS topologies, respectively; the performance gain in the 9068-node AS topology is 0%. In the case of the worm global simulation, we observe performance gain ranging from -6.2% to 20.7%.



(a) BGP

(b) Worm local

(c) Worm global

(d) Distributed client/server

Figure 6.24. Performance gain $\gamma(M_i, C_i)$ of M_i over C_i with respect to simulation progress as a function of problem size for all benchmark applications. $\gamma(M_i, C_i)$ is shown in percentage.

6.5 Joint Memory-CPU Balancing

In general, the impact of CPU and communication cost imbalance can cause the M_i (max) cost metric to underperform against the C_i (total) with respect to completion time. This holds true even under the impact of VM paging, which decreases the completion time gap between the M_i (max) and C_i (total) cost metrics. As we have seen in Section 5.4, in network simulations, joint memory-CPU balancing using both the M_i and C_i cost metrics can overcome the memory-CPU balancing trade-off. In this section, we show analogous results using the worm local application, and we evaluate joint memory-CPU balancing performance with respect to memory balancing and performance gain in terms of speed-up.

We first show a worm local 6582-node simulation using 16 PCs with 1GB memory each. Here, the impact of CPU and communication cost imbalance causes the M_i (max) cost metric to underperform against the C_i (total) cost metric with respect to completion time. Figure 6.25 shows the wall clock time taken to simulate n message events vs. the total number of message events processed (n), comparing M_i, C_i, and joint memory-CPU balancing. The data points correspond to the total number of message events processed at t_u, t_t, t_m, and t_j from the left, respectively. Here, t_u, t_t, t_m, and t_j represent the simulation time instances where the uniform, M_i (max), C_i (total), and the joint memory-CPU balancing start to experience thrashing. We observe that C_i (total) outperforms M_i (max) until t_t—i.e., before thrashing sets in in any of them, although M_i performs better than C_i eventually. Our detailed analysis showed that M_i's slower progress compared to C_i without thrashing is due to CPU and communication cost imbalance. In Figure 6.25, we also observe that joint memory-CPU balancing achieves reduced completion time matching that of C_i. We have observed similar patterns in other worm-local simulations with various problem sizes ranging 8063-node, 10363-node, 13532-node, and 14577-node.

Figure 6.26 shows memory balancing performance of M_i, C_i, and joint memory-CPU balancing for worm local simulations varying the problem size. We confirm that

M_i outperforms C_i in terms of memory balancing and joint memory-CPU balancing achieves comparable memory balancing performance to that of M_i.

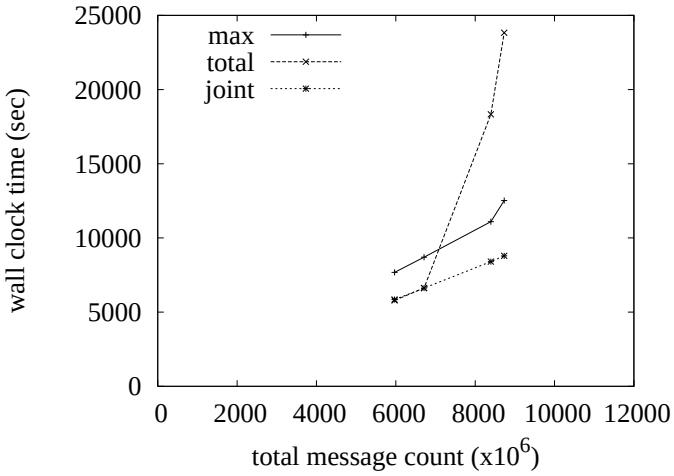


Figure 6.25. Completion time taken to simulate n message events vs. the total number of message events processed, n, in the worm local 6582-node simulation, comparing the M_i, C_i, and joint memory-CPU balancing. The data points correspond to the total number of message events processed at t_u, t_t, t_m, and t_j from the left, respectively.

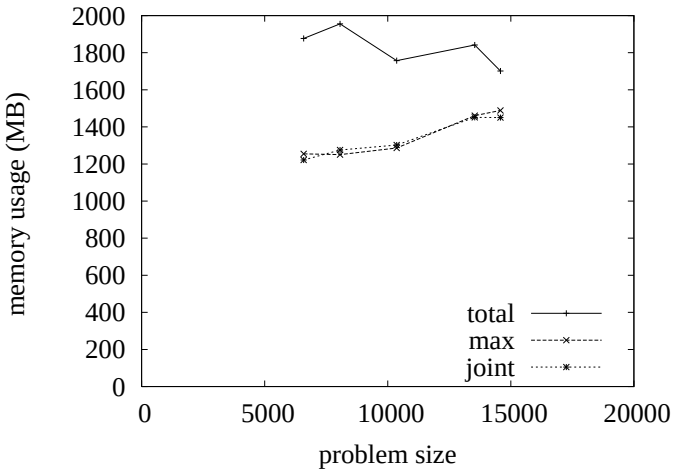


Figure 6.26. Memory balancing performance of M_i, C_i, and joint memory-CPU balancing for worm local propagation simulations.

Figure 6.27 compares joint memory-CPU balancing and C_i with respect to completion time. Completion time is measured as wall clock time taken to simulate t_j simulation seconds. The ratio of C_i's completion time divided by the completion time of joint balancing is plotted as a function of problem size for the worm local application. For comparison, we plot the ratio of completion times between M_i and C_i shown in Figure 6.23. We observe that joint memory-CPU balancing outperforms C_i significantly, achieving increased speed-up compared to M_i.

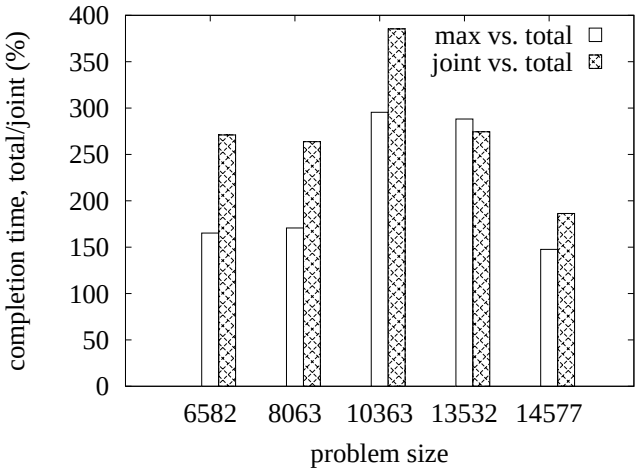


Figure 6.27. Comparison of joint memory-CPU balancing and C_i with respect to completion time. Completion time is measured as wall clock time taken to simulate t_j simulation seconds. The ratio of C_i's completion time divided by the completion time of joint balancing is plotted as a function of problem size for the worm local application.

We define performance gain $\gamma(joint, C_i)$ of the joint memory-CPU balancing over C_i with respect to simulation progress as $(n(t_j) - n(t_t))/n(t_t)$. $\gamma(joint, C_i)$ denotes the number of message events that the joint balancing case processes additionally without experiencing thrashing compared to the C_i case. A negative value indicates that $t_j < t_t$. Figure 6.28 shows performance gain $\gamma(joint, C_i)$ of the joint memory-CPU balancing over C_i with respect to simulation progress as a function of problem size for the worm local application. For the comparison with the M_i case, we plot $\gamma(M_i, C_i)$ as well. $\gamma(joint, C_i)$ and $\gamma(M_i, C_i)$ are shown in percentage. We observe

that joint memory-CPU balancing achieves comparable performance gain compared to the M_i case in terms of simulation progress with the magnitude of gap depending on problem size.

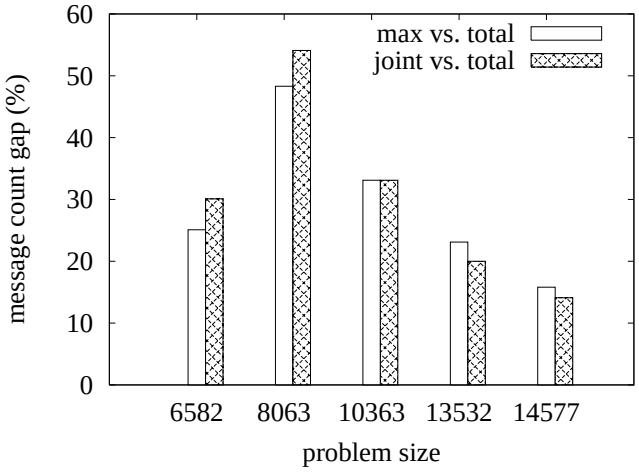


Figure 6.28. Performance gain $\gamma(joint, C_i)$ of the joint memory-CPU balancing over C_i with respect to simulation progress as a function of problem size for the worm local application. For the comparison with the M_i case, we plot $\gamma(M_i, C_i)$ as well. $\gamma(joint, C_i)$ and $\gamma(M_i, C_i)$ are shown in percentage.

6.6 Optimizing the Overhead of Benchmark-based Cost Estimation

As we discussed in Section 4.2.2, from a user's point of view, reducing the overhead of our benchmark-based cost estimation, i.e., the completion time of a benchmark simulation, is important. In particular, if trashing sets in during a benchmark simulation run, simulation progress becomes excessively slow and the overhead becomes prohibitively high. In this section, we compare two M_i cost metrics—one obtained right after t_u and another obtained after running for around 10 hours wall clock time. Let the first be M_i^s and the latter be M_i^l. Similarly, we compare two C_i cost metrics—one C_i^s obtained right after t_u and another C_i^l obtained after running for around 10 hours. The comparison is done in terms of gain and cost. Let t_m^s be the simula-

tion time instance where a simulation with M_i^s starts to experience thrashing. Let t_m^l be the simulation time instance where a simulation with M_i^l starts to experience thrashing. t_t^s and t_t^l are defined in the same way for the C_i^s and C_i^l cases. Recall that we define $n(t)$ as the number of message events that are processed from 0 second simulation time until t second simulation time. We define gain $\delta(M)$ of the M_i^l case over the M_i^s case as $(n(t_m^l) - n(t_m^s))/n(t_m^s)$. $\delta(M)$ denotes the number of additional message events processed using M_i obtained after a 10-hour run compared to the total message events using M_i obtained right after t_u. Similarly, we define gain $\delta(C)$ of the C_i^l case over the C_i^s case as $(n(t_t^l) - n(t_t^s))/n(t_t^s)$. We define cost as the wall clock time taken for a benchmark simulation. Table 6.3 summarizes three performance results:

Table 6.3
Summary of gain and cost: per-node memory and CPU costs from the 10-hour-long benchmark simulation vs. those obtained right after t_u.

case	$\delta(M)$	$\delta(C)$	cost of t_u	cost of t_m^l in M_i^l case
bgp, 6582	5.4%	8.8%	1hr 20min	2hr 00min
worm-local, 6582	3.5%	1.2%	1hr 43min	2hr 20min
worm-local, 10363	8.4%	7.4%	1hr 30min	2hr 15min
worm-local, 14577	2.1%	-2.8%	1hr 30min	2hr 04min
worm-global, 21460	11.1%	10.6%	2hr 30min	3hr 21min

one, gain $\delta(M)$ of M_i^l from the 10-hour-long benchmark simulation over M_i^s obtained right after t_u, two, gain $\delta(C)$ of C_i^l from the 10-hour-long benchmark simulation over C_i^s obtained right after t_u, and three, costs. We observe that, in terms of both M_i and C_i cost metrics, using the cost estimation from 10-hour-long benchmark simulation achieves marginal gain in terms of total message events processed—i.e., simulation progress. However, the cost of 10 hours—i.e., the wall clock time taken to complete a benchmark simulation—is substantial compared to the completion time of t_u. In

addition, the completion time of t_m^l in the actual memory-balanced simulation runs using M_i^l is much faster than 10 hours. Hence, a user may stop a benchmark simulation when thrashing occurs reducing the overhead of benchmark simulation. Not much simulation progress occurs once thrashing starts.

7 CONCLUSION AND FUTURE WORK

7.1 Conclusion

This dissertation studied the memory balancing problem for large-scale network simulation in power-law networks over PC clusters, taking a memory-centric approach to network simulation partitioning.

First, we designed and implemented a measurement subsystem for dynamically tracking memory consumption in DaSSFNet, a distributed network simulator. We showed that the measurement subsystem achieves accurate monitoring of memory consumption at per-node granularity. Although the specific implementation details are DaSSFNet dependent, the underlying measurement methodology is applicable to other distributed network simulation environments.

Second, we achieved efficient memory cost monitoring by showing that the maximum cost metric, which has constant per-node space complexity, enables effective memory balancing during network partitioning. We showed that although the max-of-sum vs. sum-of-max problem cannot be solved because peak memory consumption across simulated nodes is, in general, not synchronized, the maximum cost metric suffices to achieve relative memory balancing. Relative memory balancing using the maximum cost metric is achievable in large-scale network simulation for various benchmark application types—BGP, Internet worm with global and local scanning, and distributed file client/server—with diverse communication requirements.

Third, we evaluated the impact of power-law connectivity of large-scale measurement networks on memory and CPU balancing. We showed that multilevel recursive partitioning [21] implemented by popular graph partitioning tools such as Metis [47] and Chaco [46] achieves relative memory balancing in power-law network partitioning.

Fourth, we identified and established a memory vs. CPU balancing trade-off. We explained the difference in trade-off across application types by showing that BGP and worm global exhibit a more pronounced memory cost skew induced by power-law connectivity that makes balancing more difficult. We showed that the performance trade-off can be overcome through joint memory-CPU balancing which is, in general, not feasible due to constraint conflicts. This is facilitated by network simulation having a tendency to induce correlation between memory and CPU costs. As shown with the benchmark application types, the memory and CPU balancing performance gap between the maximum and total cost metrics varies depending on application type. However, the memory vs. CPU balancing trade-off and the performance of joint memory-CPU balancing due to network simulation's tendency to induce correlation between memory and CPU costs are expected in large scale network simulation over PC clusters without depending on application types.

Fifth, we advanced a performance evaluation framework for evaluating memory balancing under VM. We defined metrics for quantifying distributed simulation slow down under thrashing and the performance gain achieved by one network partitioning method over another. We showed that onset of thrashing in network simulation is effected not only by the amount of excess memory demand (beyond physical memory) but also its composition with respect to message vs. routing table memory. We found that an idiosyncrasy of network simulation is that messages are not conducive to locality of reference. The metrics for quantifying distributed simulation slow down under thrashing and the performance gain achieved by one network partitioning over another is applicable to distributed simulation environments implementing a barrier-style global synchronization scheme. Based on our finding of the idiosyncrasy of network simulation that messages are not conducive to locality of reference, one can devise a new memory management mechanism for large-scale network simulation where memory allocated for messages is handled differently from memory for tables so that onset of thrashing is delayed further.

Sixth, we showed that improved memory balancing under the maximum cost metric in the presence of VM manifests as faster completion time compared to the total cost metric despite the CPU balancing advantage of the latter. We showed that in the cases where the CPU balancing advantage of the total cost metric is strong, joint memory-CPU balancing can achieve the best of both worlds.

7.2 Future Work

In this section we present a number of open problems and potentially promising avenues for extending this research. They include:

Non-uniform memory balancing In this dissertation, we showed that the maximum cost metric suffices to achieve relative memory balancing in large-scale network simulation. Our performance evaluation focused on PC clusters with uniform memory. In reality, it is common to have PC clusters with non-uniform memory. The maximum cost metric approach is extensible to non-uniform memory cases. One can conduct the same performance evaluation under non-uniform memory set-up to evaluate the utility of our accurate and efficient memory estimation and memory balancing mechanism.

Built-in adaptive dynamic load balancer Our methodology for detecting the onset of thrashing based on the temporal page fault rate reaching a plateau at memory-overloaded machines can be automated and embedded in distributed simulators as built-in system support. We expect that one immediate utility is to stretch simulation execution until the onset of thrashing with VM paging support. Dynamic load balancing has been studied in the context of conservative parallel simulation on a multicomputer for parallel speed-up based on a process migration mechanism [72]. We anticipate that one can extend our memory estimation/balancing mechanism with the automatic thrashing detection support into a built-in dynamic load balancer, which accurately estimates memory cost at run-time and dynamically relocates nodes from memory-overloaded machines into other machines with more available memory.

Communication and synchronization cost Our performance evaluation in this dissertation focuses on memory and CPU balancing. We considered communication cost in terms of CPU cost for processing MPI messages and implicitly through edge cut reduction during network partitioning. We also limited the influence of lookahead to synchronization cost by setting uniform link latency. Explicit incorporation of communication and synchronization cost (lookahead) as edge weight during network partitioning can be explored as part of future work.

LIST OF REFERENCES

LIST OF REFERENCES

[1] R. Fujimoto, K. Perumalla, A. Park, H. Wu, M. Ammar, and G. Riley. Large-scale network simulation: How big? How fast? In *Proc. IEEE MASCOTS '03*, pages 116–123, 2003.

[2] E. Page, D. Nicol, O. Balci, R. Fujimoto, P. Fishwick, P. L'Ecuyer, and R. Smith. Panel: Strategic directions in simulation research. In *Proc. 1999 Winter Simulation Conference*, pages 1509–1520, 1999.

[3] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed Internet routing convergence. *IEEE/ACM Trans. on Networking (TON)*, 9(3):293–306, June 2001.

[4] Craig Labovitz, Abha Ahuja, Roger Wattenhofer, and Venkatachary Srinivasan. The impact of Internet policy and topology on delayed routing convergence. In *Proc. IEEE INFOCOM '01*, pages 537–546, 2001.

[5] Vern Paxson. End-to-end routing behavior in the Internet. In *Proc. ACM SIGCOMM '96*, pages 25–38, 1996.

[6] Craig Labovitz, G. Robert Malan, and Farnam Jahanian. Internet routing instability. *IEEE/ACM Trans. on Networking (TON)*, 6(5):515–528, October 1998.

[7] Xenofontas A. Dimitropoulos and George F. Riley. Large-scale simulation models of bgp. In *Proc. IEEE MASCOTS '04*, pages 287–294, 2004.

[8] Songjie Wei, Jelena Mirkovic, and Martin Swany. Distributed worm simulation with a realistic internet model. In *Proc. IEEE PADS '05*, pages 71–79, 2005.

[9] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distribut ed dos attack prevention in power-law internets. In *In Proc. ACM SIGCOMM '01*, pages pp. 15–26, 2001.

[10] Wei-Min Yao and Sonia Fahmy. Downscaling network scenarios with denial of service (dos) attacks. In *Proc. IEEE Sarnoff Symposium, 2008*, 2008.

[11] Xin Zhang and George F. Riley. Performance of routing protocols in very large-scale mobile wireless ad hoc networks. In *Proc. IEEE MASCOTS '05*, pages 115–124, 2005.

[12] Q. He, M. Ammar, G. Riley, H. Raj, and R. Fujimoto. Mapping peer behavior to packet-level details: a framework for packet-level simulation of peer-to-peer systems. In *Proc. MASCOTS '03*, 2003.

[13] D. Xu, G. Riley, M. Ammar, and R. Fujimoto. Enabling large-scale multicast simulation by reducing memory requirements. In *Proc. IEEE PADS '03*, pages 69–76, 2003.

[14] Peter Denning. Working sets past and present. *IEEE Trans. on Software Engineering*, 6(1):64–84, 1980.

[15] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *Proc. ACM SIGMETRICS '97*, pages 115–126, 1997.

[16] SSFNet. http://www.ssfnet.org/ last accessed 17 April 2005.

[17] Dartmouth SSF. Dassf 3.1.5. http://www.cs.dartmouth.edu/∼jasonliu/projects/ssf/, August 2001.

[18] Georgia tech parallel/distributed NS. pdns 2.1b7a. http://www.cc.gatech.edu/computing/compass/pdns/, February 2001.

[19] J-Sim. http://www.j-sim.org.

[20] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[21] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[22] National Laboratory for Applied Network Research. Routing data, 2000. Supported by NSF, http://moat.nlanr.net/Routing/rawdata/.

[23] University of Oregon. Oregon route views. http://www.routeviews.org/ and http://archive.routeviews.org/.

[24] University of Michigan. AS graph data sets, 2002. http://topology.eecs.umich.edu/data.html.

[25] Mercator Internet AS map. Courtesy of Ramesh Govindan, USC/ISI, 2002.

[26] RIPE. Routing information service raw data, 2002. http://data.ris.ripe.net.

[27] CAIDA. Skitter, 2002. http://www.caida.org/tools/measurement/skitter.

[28] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the Internet topology. In *Proc. ACM SIGCOMM*, pages 251–262, 1999.

[29] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, and R. Stata. Graph structure in the Web. *Computer Networks*, 33:309–320, 2000. Proc. 9th WWW Conference.

[30] H. Jeong, B. Tomber, R. Albert, Z. Oltvai, and A. L. Barabasi. The large-scale organization of metabolic networks. *Nature*, pages 378–382, 2000.

[31] A. J. Lotka. The frequency distribution of scientific productivity. *The Journal of the Washington Academy of the Sciences*, page 317, 1926.

[32] M. Newman. The structure of scientific collaboration networks. *Proc. Natl. Acad. Sci. USA 98*, 4:404–409, 2001.

[33] S. Redner. How popular is your paper? *Euro. Phys. J. B*, 4:131–134, 1998.

[34] P. Erdős and A. Rényi. On random graphs. *Publ. Math. Debrecen*, 6:290–291, 1959.

[35] X. Dimitropoulos and G. Riley. Efficient large-scale BGP simulations. *Computer Networks*, 50(12):2013–2027, 2006.

[36] A. Hiromori, H. Yamaguchi, K. Yasumoto, T. Higashino, and K. Taniguchi. Reducing the size of routing tables for large-scale network simulation. In *Proc. IEEE PADS '03*, 2003.

[37] Polly Huang and John Heidemann. Minimizing routing state for light-weight network simulation. In *Proc. IEEE MASCOTS '01*, page 108, 2001.

[38] V. Krishnamurthy, M. Faloutsos, M. Chrobak, L. Lao, J.H. Cui, and A. G. Percus. Reducing large internet topologies for faster simulations. In *Proc. IFIP Networking 2005*, 2005.

[39] G. Carl, S. Phoha, G. Kesidis, and B. Madan. Path preserving scale down for validation of internet inter-domain routing protocols. In *Proc. the Winter Simulation Conference '06*, pages 2210–2218, 2006.

[40] F. Papadopoulos, K. Psounis, and R. Govindan. Performance preserving network downscaling. In *Proc. Annual Simulation Symposium '05*, pages 285–294, 2005.

[41] SSF. Ssfnet 1.5. http://www.ssfnet.org/homePage.html, May 2003.

[42] The network simulator – ns-2. http://www.isi.edu/nsnam/ns/.

[43] George F. Riley. The Georgia tech network simulator. In *Proc. ACM MoMeTools '03*, pages 5–12, 2003.

[44] Y. Liu, B. Szymanski, and A. Saifee. Genesis: A scalable distributed system for large-scale parallel network simulation. *Computer Networks*, 50(12):2028–2053, 2006.

[45] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proc. IEEE IPDPS 2006*, 2006.

[46] B. Hendrickson and R. Leland. The Chaco user's guide, version 2.0. Technical Report, SAND94-2692, Sandia National Laboratories, 1994.

[47] G. Karypis and V. Kumar. METIS 4.0: Unstructured graph partitioning and sparse matrix ordering system. Technical Report, Department of Computer Science, University of Minnesota. Available at http://www.cs.umn.edu/∼metis, 1998.

[48] Robert Preis and Ralf Diekmann. Party - a software library for graph partitioning. In *Advances in Computational Mechanics with Parallel and Distributed Processing*, pages 63–71. Civil-Comp Press, 1997.

[49] C. Walshaw, M. Cross, M. G. Everett, and S. Johnson. Jostle: Partitioning of unstructured meshes for massively parallel machines. In *Parallel Computational Fluid Dynamics: New Algorithms and Applications*. Elsevier, 1994.

[50] Universite Bordeaux I. Scotch 3.1 user's guide, 1997.

[51] D. Xu and M. Ammar. BencHMAP: Benchmark-based, hardware and model-aware partitioning for parallel and distributed network simulation. In *Proc. IEEE MASCOTS '04*, pages 455–463, 2004.

[52] K. Yokum, E. Eade, J. Degesys, D. Becker, J. Chase, and A. Vahdat. Toward scaling network emulation using topology partitioning. In *Proc. IEEE MASCOTS '03*, pages 242–245, 2003.

[53] S. Lee, J. Leaney, T. O'Neill, and M. Hunter. Performance benchmark of a parallel and distributed network simulator. In *Proc. IEEE PADS '05*, 2005.

[54] H. Ohsaki, G. Oscar, and M. Imase. Quasi-dynamic network model partition method for accelerating parallel network simulation. In *Proc. IEEE MASCOTS '06*, pages 255–264, 2006.

[55] B. Gan, Y. Low, S. Jain, S. Turner, W. Cai, W. Hsu, and S. Huang. Load balancing for conservative simulation on shared memory multiprocessor systems. In *Proc. IEEE PADS '00*, pages 139–146, 2000.

[56] Richard Mills. *Dynamic adaptation to CPU and memory load in scientific applications*. PhD thesis, The College of William and Mary, 2004.

[57] L. Xiao, S. Chen, and X. Zhang. Dynamic cluster resource allocations for jobs with known and unknown memory demands. *IEEE Trans. on Parallel and Distributed Systems*, 13(3):223–240, 2002.

[58] X. Zhang, Y. Qu, and L. Xiao. Improving distributed workload performance by sharing both cpu and memory resources. In *Proc. IEEE ICDCS '00*, pages 233–241, 2000.

[59] L. Iftode, K. Li, and K. Petersen. Memory servers for multicomputers. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 534–547, 1993.

[60] Douglas Comer and Jim Griffioen. A new design for distributed systems: The remote memory model. In *USENIX Summer*, pages 127–136, 1990.

[61] R. Leland and B. Hendrickson. An empirical study of static load balancing algorithms. *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 682–685, May 1994.

[62] F. Ercal and J. Ramanujam. Cluster partitioning approaches to mapping parallel programs onto a hypercube. *Parallel Computing*, 13:1–16, 1990.

[63] M. Ashraf Iqbal and Shahid H. Bokhari. Efficient algorithms for a class of partitioning problems. *IEEE Trans. on Parallel and Distributed Systems*, 06(2):170–175, 1995.

[64] Jason Liu and David Nicol. Learning not to share. In *Proc. IEEE PADS '01*, pages 46–55, 2001.

[65] D. Nicol and J. Liu. Composite synchronization in parallel discrete-event simulation. *IEEE Trans. Parallal and Distributed Systems*, 13(5):433–446, 2002.

[66] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGARCH Comput. Archit. News*, 28(5):117–128, 2000.

[67] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. In *Proc. ACM SIGMETRICS '96*, pages 160–169, 1996.

[68] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. In *Proc. ACM SIGCOMM '93*, pages 183–193, 1993.

[69] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the Internet topology. In *Proc. ACM SIGCOMM '99*, pages 251–262, 1999.

[70] L. Li, D. Alderson, W. Willinger, and J. Doyle. A first-principles approach to understanding the Internet's router-level topology. In *Proc. ACM SIGCOMM '04*, pages 3–14, 2004.

[71] Daniel Bovet and Marco Cesati. *Understanding the Linux kernel, second edition.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[72] A. Boukerche and S.K. Das. A dynamic load balancing algorithm for conservative parallelsimulations. In *Proc. IEEE MASCOTS '97*, 1997.

VITA

VITA

HyoJeong Kim received her B.S. degree in Computer Science and Engineering from Korea University, Seoul, Republic of Korea in 1999. After staying on as a graduate student for one year at Korea University, she started her graduate studies at Purdue University in 2001. She received her M.S. degree in Computer Science from Purdue University in 2003. She received her Ph.D. degree in Computer Science from Purdue University in 2008. HyoJeong Kim received a state scholarship for oversears study from the Ministry of Information & Communication, Republic of Korea. She received teaching assistantships and research assistantships (DARPA) at Purdue University. She previously worked as a research intern at Simulex Inc.