Spring 2013

APPLICATION OF SECRETARY ALGORITHM TO DYNAMIC LOAD BALANCING IN USER-SPACE ON MULTICORE SYSTEMS

Kyoung-Hwan Yun

APPLICATION OF SECRETARY ALGORITHM TO DYNAMIC LOAD BALANCING IN USER-SPACE ON MULTICORE SYSTEMS

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Kyoung-Hwan Yun

May 2013

The Designated Committee Approves the Writing Project Titled

APPLICATION OF SECRETARY ALGORITHM TO DYNAMIC LOAD BALANCING IN USER-SPACE ON MULTICORE SYSTEMS

by

Kyoung-Hwan Yun

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 6, 2013

Dr. Teng Moh Department of Computer Science

Dr. Robert Chun Department of Computer Science

Dr. Melody Moh Department of Computer Science

ABSTRACT

APPLICATION OF SECRETARY ALGORITHM TO DYNAMIC LOAD BALANCING IN USER-SPACE ON MULTICORE SYSTEMS

by Kyoung-Hwan Yun

In recent years, multicore processors have been so prevalent in many types of systems and are now widely used even in commodities for a wide range of applications. Although multicore processors are clearly a popular hardware solution to problems that were not possible with traditional single-core processors, taking advantage of them are inevitably met by software challenges. As Amdahl's law puts it, the performance gain is limited by the percentage of the software that cannot be run in parallel on multiple cores. Even when an application is "embarrassingly" parallelized by a careful design of algorithm and implementation, load balancing of tasks across different cores is a very important and critical aspect in utilizing a multicore system as close to its fullest potential as possible.

In this paper, we investigate how a solution to a cardinal payoff variant of the secretary problem can be applied to a proactive, decentralized, dynamic load balancing technique in user-space to assist single program, multiple data (SPMD) applications in multiprogrammed environment so that all tasks can make roughly equal progress distributed over all cores. We examine how this method compares with the default Linux load balancer in terms of scalability and predictability. Our experiments show promising results that show our technique outperforms the default Linux scheduler by an average 40%

speedup in multiprogrammed environment with less time variance among multiple executions.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and indebtedness to my advisor, Dr. Teng Moh, for his invaluable inspirations and guidance throughout the course of study and this project. He has instilled in me much interest in the areas of distributed computing, randomized algorithms, and machine learning.

My gratitude goes out as well to the rest of my master's committee members, Dr. Melody Moh and Dr. Robert Chun, for their gracious offer of time and advice.

I would also like to thank my current employer Cisco Systems for their financial support, and all my unnamed co-workers for their invaluable help and encouragement.

I must not forget the late Im Young Kim, as I'm deeply moved by the remembrance of her love and influence in my life. As I come close to reaching this milestone in my life, I can't help but remember her even more with profound gratitude.

My thanks also go to my middle school math teacher, Dong-Ok Choi, who opened up an invaluable opportunity to get introduced to computers and come this far.

I thank God, who is the "beginning of knowledge" (Proverbs 1:7), for His love and grace that can't be matched by anything else in the world. I thank those who supported me with their prayers.

Finally, I dedicate this work to my wife Jung-Min Choi, and my children Ethan and Serena, who have sacrificed so much for me and whose encouragement kept me going. I look forward to returning the time with me to them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

In recent years, multicore processors have been so prevalent that they are no longer considered to be novelty, and parallel computing systems are now less often narrowly equated to high performance computing (HPC) systems specialized for scientific research, engineering, and such. In fact, multicore processors are now widely used in commodities. Most desktop computers, laptops, and even smartphones are parallel computing machines powered by multicores and are constantly evolving towards many-core systems; Intel has already produced an 80-core "Teraflops Research Chip" and said that they could easily have processors with hundreds of cores in next 5-10 years [14]. They are used in a wide range of applications including general purpose, machine vision, virtualization, graphics, networking, medical imaging, digital signal processing (DSP), and gaming. Ever increasing constraints on Moore's Law like heat/power wall and diminishing return from instruction-level parallelism (ILP) have forced the industry to turn to multicore processors.

Although multicore processors are clearly a popular hardware solution to problems that were not possible with traditional single-core processors, taking advantage of them are inevitably met by software challenges. Parallelization of software is the central aspect on performance improvement by multicore processors in light of the effect of what is known as Amdahl's law. The performance gain is limited by the percentage of the software that cannot be run in parallel on multiple cores. Given that an application is "embarrassingly" parallelized by a careful design of algorithm and implementation, ideally, its tasks should be well distributed across different cores. As increased number

of cores is available in a system, it is even more important to utilize them as much as possible. The most undesirable scenario is all tasks of a parallelized application or a single-threaded application running on only one core, leaving the other cores idle and unutilized.

Particularly in single program, multiple data (SPMD) application (which is the most common parallel programming technique [18]), tasks are almost equally load balanced within it, but the actual runtime doesn't often reflect this internal balance due to many external reasons. Whatever happens outside the application such as OS jitter, other applications, heterogeneous cores, unequal number of tasks among cores, etc. impact the progress of each task. Consequently, some tasks that finished early block at the barrier while some others are still doing their computations, making the total execution time longer. All tasks within the SPMD application must make equal progress executing on all available cores in order to achieve good performance. Any parallel application is only as fast as its slowest task.

The load balancing is a fundamental problem in parallel computing, and there has been a great deal of research in this subject. A load balancing strategy can be as simple as static task assignment, or as sophisticated as using adaptively several heuristics (e.g., [10, 11]) to dynamically move around tasks among cores. Implementations of load balancing algorithms are done in different domains. For example, the programmer can do static balancing at the application level, whether arbitrarily or with some intelligent decision method. Modern operating systems implement schedulers that do certain load balancing operations. Moreover, programming languages designed for parallel

programming paradigm (e.g., OpenMP, MPI, Cilk, X10, UPC, etc.) implement load balancing algorithms, assisted by compiler and runtime library. They all have different advantages and disadvantages, and there is none that performs well in all aspects.

In this paper, we briefly survey a few basic load-balancing strategies, including also an interesting concept of user-space application assisted dynamic load balancing that can coexist with other load balancers. Then, we investigate how a solution to a cardinal payoff variant of the secretary problem can be applied to the latter concept of user level dynamic load balancing to assist SPMD applications in multiprogrammed environment so that all tasks can make roughly equal progress distributed over all cores. We examine how this method compares with the default Linux load balancer in terms of performance and execution time variation.

The results from our experiments of running NAS Parallel Benchmarks (NPB) [15] in OpenMP in both dedicated and non-dedicated environments show promising results where our online ("on the fly") selection load balancer (OSLB) achieves good speedups overall. In a dedicated environment with most other system processes disabled, we see an 8% average speedup of all the benchmarks when they are run individually. In the same environment, when there is an induced imbalance by a CPU-intensive process, the average speedup rises to 14.8%. Furthermore, we see a 40% average speedup when all the benchmarks run together to simulate a non-dedicated environment. Our load balancer is able to correct imbalances very well by making balancing decisions based on partial information of the parallel subtasks. It demonstrates the applicability and potential of optimal stopping strategies to load balancing for scalability.

2. BACKGROUND

There are various and vastly many techniques applied to multicore load balancing, all with one goal in mind: To improve performance by distributing the total load of all tasks evenly among all cores. They can be broadly categorized into *static load balancing* and *dynamic load balancing*. In this section, we lay down some fundamental understanding of load balancing, and then discuss how modern operating system schedulers support the load balancing. In addition, we describe a different approach taken by others' prior work, which explored the idea of user-space application performing proactively dynamic load balancing. From the discussions of these topics and this latter technique we develop our work.

2.1. Static Balancing

In static balancing, load balancing is done before the application is run. Tasks are statically distributed to cores before execution of the parallel application. One simple and classic but not effective or useful (other than being a popular benchmark for comparison) algorithm is to assign tasks randomly to cores. Whereas this algorithm does not require any prior knowledge about the application's behavior, there are other methods that make improvements by obtaining and utilizing such information. Round robin assignment of tasks to processors is a very simple but not much effective approach. There are also partitioning methods where tasks are represented by a graph. One particular approach of such kind is to use mapping heuristics using an *Expected Time to Compute* (ETC) matrix. By using *genetic algorithm* (GA), search for an acceptable task mapping from a large

search space in a reasonable amount of time is attempted [1]. In a practical sense, calculating an ETC matrix with reasonable accuracy could be challenging, especially in a multiprogrammed environment where other programs change the system state. Also, the fact that many factors have to be considered when calculating an ETC matrix adds complexity to the designing of algorithm. Although advantages of static load balancing are simplicity to implement and minimal runtime load balancing overhead, the example of GA method highlights the inherent problem all static load balancing methods face, namely, the complexity in predicting execution times with plausible degree of accuracy when computation pattern is not predictable. Static load balancing works well if computation pattern is predictable. However, SPMD applications in a multiprogrammed environment do not benefit much from static load balancing and need a balancing method that reacts dynamically and timely to constantly evolving task progress status.

Our work saw this very limitation of the static load balancing and thus naturally explored the utility of dynamic load balancing for SPMD applications which run, more likely than not, on systems that have many elements of imbalance.

2.2. Dynamic Balancing

If an SPMD application runs on a non-dedicated system (as it is often the case) with other applications or system processes, all of its tasks will make different progresses because of different and constantly changing load on each core. The progress of each task is generally unpredictable and uneven just as the system state is often not predictable. In such an environment, it is better that load balancing decisions be made based on the

current state of the system and that task assignments to cores be changed during runtime. Unlike static load balancing, dynamic load balancing incurs extra overhead for monitoring and rebalancing, but performance gain generally offsets the overhead with a proper decision on monitoring and rebalancing policies.

Depending on the location where decisions on load balancing are made, a load balancing technique can be classified as either *centralized* or *decentralized* (*distributed*). In the centralized technique, a single process makes decisions on distributing tasks. The single process can gather necessary information on demand, so the reduced amount of exchanged data is an advantage. On the other hand, in the decentralized technique, multiple processes (e.g., one balancing process per core) individually make decisions on distributing tasks. This necessitates some form of information exchange among one another, resulting in an increased amount of data communication, significantly more than the centralized technique as the number of the decision-making processes increases. Also, since a centralized decision maker could be a bottleneck when there are a lot of tasks to balance, the decentralized technique is more scalable. Many researchers believe the decentralized load balancing is more scalable. Each process makes decisions based on its local information or global information, and its optimal policy [8].

Our dynamic load balancer has per-core balancing threads that make balancing decisions in decentralized manner without enforcing lock-step execution of the threads, and this approach will be more beneficial as we see increasing number of cores and tasks running on them.

The following subsections describe centralized and decentralized algorithms that are frequently discussed and contrasted.

2.2.1. Work-Sharing

In *work-sharing*, whenever a core creates new tasks, the scheduler attempts to distribute some of them to other underutilized cores. The tasks are usually pooled at the global centralized location. This algorithm causes frequent migrations of tasks by the scheduler, and there are scalability problems because of the contention on centralized task pool. The scalability issue has been observed on the parallel programming language X10, and there has been work to implement work-stealing scheduling as an improvement over its existing work-sharing runtime system [10].

2.2.2. Work-Stealing

One area that has generated a great deal of research is *work-stealing* (or task-stealing). There have been many variants since the introduction of its idea as far back as the work by Burton and Sleep [5]. Work-stealing is used in many task-parallel runtime implementations. It allows dynamic load balancing with low overheads incurred to critical paths because idle cores steal from busy cores. Fundamentally, tasks are kept on each core's queue and other idle cores steal from busy cores; and there are fewer task migrations in work-stealing than in work-sharing. Because of its distributed implementation, work-stealing is more scalable than work-sharing. In many work-stealing variant implementations, steal overheads, cache sharing, and NUMA characteristics are considered when designing work-stealing algorithm. All variant

approaches overcome certain shortcomings of the pure work-stealing policy. Issues they address are mostly salient on multi-socket multicore architectures. A hierarchical policy that combines shared LIFO queue among local cores and work-stealing between chips has been shown to maintain good load balance and cache performance with limited overhead [16]. Randomized work-stealing can incur severe cache misses for memory-bound tasks that share data, and a cache aware work-stealing based on online profiling for programs whose tasks can be represented as tree-shaped DAGs (e.g., divide-and-conquer programs) has been shown to perform well [6]. To address locality-obliviousness due to randomized stealing and inflexibility of fixed task scheduling policy (*work-first* or *help-first* [10]), a scalable locality-aware adaptive work-stealing scheduler was designed so that the programmer can provide locality hints to the runtime or compiler. Work-first policy and help-first policy have different stack and memory requirements, and context switch patterns, so this work-stealing algorithm adaptively switches its scheduling policy during runtime [11]. It is commonly important among all work-stealing approaches to limit randomness of stealing which negatively affect cache locality and load balancing, not to mention other overheads. The parallel programming language Cilk is an example that has compiler and runtime implementations to support work-stealing.

Both work-sharing and work-stealing require support by compiler and runtime (if not by OS or by application modification). In this aspect, our load balancer offers flexibility and portability by running as an independent user-space application without requiring changes to the parallel application or compiler and runtime support.

2.3. Load Balancing by Operating System Scheduler

Modern operating systems assume symmetric speed of cores (thus, SMC). However, many systems have asymmetric multicores (AMC) with different capabilities and specialized usages. We have particularly seen AMCs being used in many smartphones in recent years; different speed and power consumptions of each core enable efficient usage of power while delivering appropriate performance for tasks that require different computation intensities. The Exynos 5 Octa processor in the recently announced Samsung Galaxy S4 has low clock speed cores (thus, low power consumption) for lighter tasks like web browsing and e-mail checking, and high clock speed cores for computationally intensive tasks like gaming [7]. Most OS schedulers treat each core as equal when distributing tasks, but tasks make different progress because of different core speeds and other external factors like OS jitter. This assumption for load balancing is problematic because tasks do end up making different progress on different cores due to such reasons.

Moreover, operating systems keep an independent runqueue on each core for scalability reason as number of cores increases. Each runqueue has certain number of tasks assigned to it. Besides the local scheduling of tasks from each queue, the scheduler periodically checks the lengths of all queues and tries to balance the queues by redistributing tasks across cores so that the queue length differences are as small as possible. With external factors like core asymmetry, OS jitter, and other applications

running, some tasks run more frequently than others, and balancing based on runqueue length do not help tasks make equal progress.

The task independence assumption is often wrong with parallel applications like OpenMP, MPI, UPC, Cilk, X10, etc. These applications have their subtasks dependent on each other. Consequently, the OS scheduler would do better in load balancing if it would incorporate into its balancing policy the relationships among tasks. Our load balancer runs along with a particular application it needs to balance, so the application's tasks are grouped in a scope of the balancing decisions. In this way, the relationships among tasks are taken account. Another characteristic of our load balancer is that it proactively watches for imbalance and tries to migrate tasks to correct it whereas the OS scheduler migrates only when they block.

2.4. Proactive User-Space Application Assisted Dynamic Load Balancing

Proactive user-space dynamic load balancing is not a novel idea. Clavis is an open-source project that uses various user-level CPU and memory statistics, hardware performance counters, and hardware-supported instruction sampling to enforce scheduling decisions to support Linux operating system scheduler [4]. One drawback is that relying on hardware-specific features makes it less portable. Juggle (similarly, its previous work) is another load balancer that explores the proactive user-space dynamic load balancing concept, yet it takes a simpler approach to measuring the task progress by using the concept of *load balancing on speed*; and task progress is measured by gathering only a few statistics—mainly user time, system time, and total elapsed time [12, 13].

Focused on SPMD programs, speed balancing assists OS scheduler by proactively and periodically migrating tasks so that all tasks within the program make equal progress. Whereas the OS scheduler balances in terms of runqueue length, the speed balancer balances the time each task spends on "fast" and "slow" cores. The speed is defined as the task's execution time (both user and system times) divided by the elapsed time. It is a reasonable and simple measurement of how much share of CPU time the task received without being bogged down with complex formulation of various system measurements, hardware-dependent performance counters, etc., and it also works well in asymmetric systems. Whatever happened for a period of time outside the task that influenced its share of execution time is well captured by the particular definition of speed. No assumption is made about application behavior, as it just observes task speeds for migrations. The balancer runs in user-level and no kernel, runtime system, or program modification is necessary, as it runs independent from the OS scheduler. Architecture independence is another advantage of the speed balancer.

1 Determine progress of threads (all balancers)
2 Determine fast and slow processors (all balancers)
3 [Barrier]
4 Classify threads as ahead and behind (single balancer)
5 Redistribute threads (single balancer)
6 [Barrier]
7 Migrate threads (all balancers)
8 [Barrier]

Figure 1. Pseudocode for Juggle [12].

Figure 1 shows in high level how Juggle works. An instance of balancer thread runs the algorithm independently on each core. Lines 3, 6, and 8 are global synchronizations needed for coordinating with one another to achieve as optimal balance as possible. By having those barriers, Juggle is able to calculate the provably optimal task distributions and migrate accordingly. Synchronizing at barriers, however, can be a bottleneck, especially with a large number of cores.

Our work investigates further on this interesting concept of proactive user-space dynamic load balancer. With scalability in mind, our decentralized load balancer is aimed to minimize the data communication complexity while achieving a good level of load balancing. As we are clearly heading to the direction of many cores becoming commonplace, we anticipate such scalability issue of load balancing so many tasks on so many cores. Moreover, having many load balancing threads synchronizing at a barrier to calculate redistribution plan of many tasks is another scalability issue. To address these, our work is described in more detail in the next section, which will be followed by experimental results.

3. OUR APPROACH

With increasing number of cores and proportionally many parallel tasks running on them, we are concerned with the scalability of such user-space dynamic load balancers with barrier synchronization. Synchronization among many balancer threads in order to calculate the optimal redistribution plan may cause some bottleneck. Especially, if there is some significant amount of latency and data communication complexity involved in gathering all task speeds, the scalability issue may be proportionally significant. If we were to be free from global synchronization, we are to deal with unsynchronized arrival of speed measurements at a given window of time. As a solution taken in this project, each balancer on each core makes online observations of task speeds from other cores in random order, and makes independent task migration decisions, as they are available at the time of observation. The strategy that our load balancer uses to choose the likely optimal candidate tasks to be migrated is a solution to a cardinal payoff variant of the secretary problem [9, 17]. The optimal policy for selecting the maximum value with the highest probability when making online observations of n candidates one at a time is to skip the first $\sqrt{n} - 1$ candidates and select the next candidate that has a higher value than all previously observed candidates [3]. By using the optimal stopping solution, we limit the number of tasks to be observed and allow each load balancing thread make online decisions without having to synchronize with the other threads.

3.1. Optimal Stopping: Cardinal Payoff Variant of the Secretary Problem

The classical secretary problem is a famous optimal stopping problem, and is also known as the marriage problem. It can be described as the following:

1. There is a single secretary position for hire.
2. The number of applicants is n, and the value of n is known.
3. Each candidate arrives sequentially in random order, where each order is equally likely.
4. The interviewer has to either accept or reject the candidate immediately after an interview. The decision is irrevocable and is based on the relative ranks of the applicants seen so far.
5. The expected payoff of selecting the best candidate is 1; otherwise, 0. The goal is to maximize the probability of selecting the best candidate.

The solution to this problem is remarkably elegant and simple. The optimal stopping policy is to reject initially about n/e candidates and accept afterwards the first candidate who is better than those interviewed so far. Also, for any value of n, the probability of making the best choice is at least $1/e \approx .368$ [9, 17]. Obviously, the same theory can be applied for making the worst choice.

In many situations, it may be more natural to relax the strict rule of accepting only the best. The problem where the employer would rather hire a higher-valued candidate than a lower-valued one, instead of only the best, is known as the cardinal payoff variant of the secretary problem. If the candidates are drawn independently and identically from a uniform distribution on [0, 1], the optimal cutoff for the cardinal payoff variant problem

is $\sqrt{n} - 1$ for the cardinal payoff variant, which is considerably smaller than the n/e optimal cutoff of the classical secretary problem.

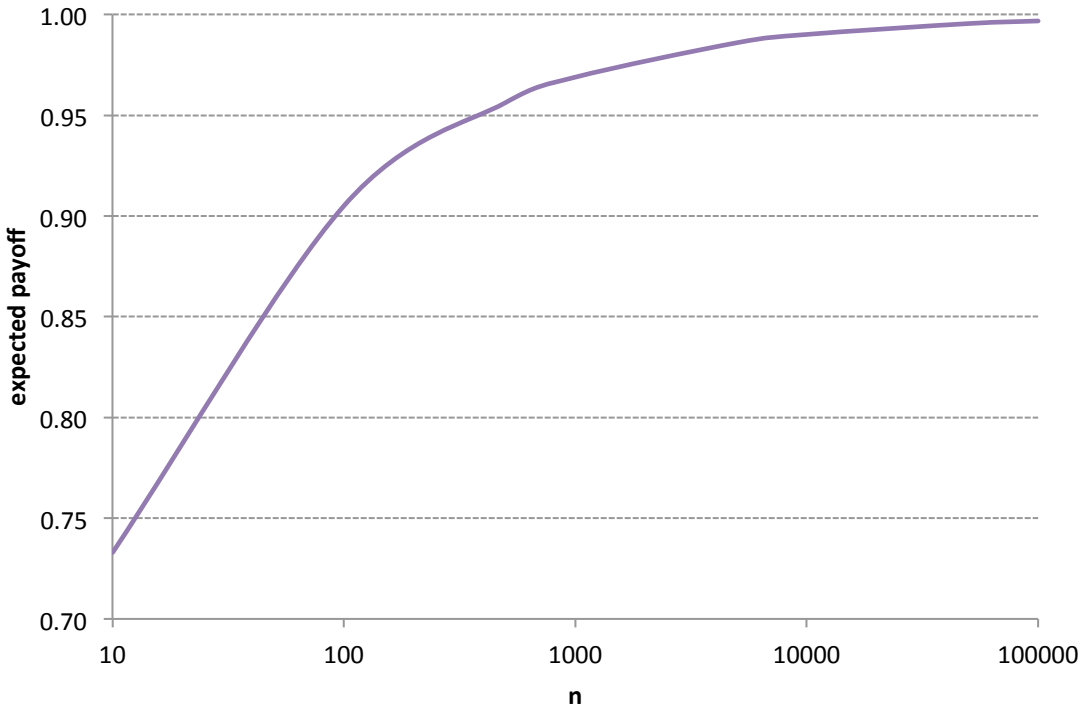

Figure 2. Expected payoff of the selected candidate (*n* in logarithmic scale)

As can be seen from Figure 2, the expected payoff of the selected candidate is well above average even at small values of *n*, and surpasses .73 fairly quickly at the small number $n = 10$, .90 at $n = 100$, and .95 at $n = 500$.

3.2. Algorithm

Our dynamic load balancer runs as a user-space program that lives during the lifetime of a parallel application that it needs to load balance. It can either launch the application it needs to load balance, or the process IDs (PIDs) of the already-running

application's subtasks can be specified when it is started. Upon termination of the parallel application, the load balancer also terminates since there remains no more balancing work to be done. Its goal is to watch the progress of each task of the parallel application, compare it against the overall average progress of all tasks, and appropriately migrate the task (if necessary) to a remote core. The load balancer does not require any modification of the parallel application and the balancing operations are transparent to the application. The balancer consists of per-core balancer threads, each of which runs independently on a core without global synchronization with the other balancer threads. The load balancing decision-making process is decentralized. Each balancer thread wakes up at every interval τ, checks for imbalance, migrates tasks, and goes back to sleep. With a shorter τ, migrations occur more frequently and thus more overheads are incurred; and with a longer τ, the opposite is true. The section 4.2 will discuss about the load-balancing interval.

Before discussing the algorithm further, the following definition of speed needs to be established [12]:

$$\text{speed} := \frac{time\ \ process\ executed}{elapsed\ time} \in [0,1]$$

At every interval τ, the balancer thread calculates the speeds of all tasks running on the local core and also their average. The average is considered as the *speed of the local core*. Also, the *global core speed* is calculated by taking an average of all core speeds. Every time a balancer thread updates its local core speed, it also updates the global core speed.

In order to prevent race condition, the global core speed is protected with a lock. The balancer thread considers its local core as *fast* if its speed is greater than the global speed, and as *slow* if less. We define a task to be *ahead* if its speed is above the global core speed, and *behind* if below.

Figure 3 outlines general action taken by each balancer thread at every interval. The general strategy is that each balancer thread on a fast core tries to swap the fastest local task with a slow task on a remote core.

```
1  Compute the local core task speeds
2  Compute the local core speed and the global core speed
3  If local core speed = fast:
4      Let proc_ahead := the fastest process in the local core
5      Let proc_behind := the process as "slow" as proc_ahead is "fast" relative to
       the global average speed
6      Migrate proc_ahead to the core proc_behind is running on; and migrate
       proc_behind to the local core
7  Else:
8      // Do nothing
```

Figure 3. Pseudocode for the balancer threads using the solution to the cardinal payoff variant.

In line 5, as an attempt to make as evenly balanced distributions of behind tasks on fast cores as possible, the balancer tries to make an online decision to select the higher (not necessarily the best, as in the classical secretary problem) fit behind task. For example, if $proc_{ahead}$ is 25% (or any other value p in general) faster than the global average speed,

then an online selection is done on a process that is as close to 25% slower than the global average speed as possible; the selection is done by using the solution to the cardinal payoff variant. Also, in order to prevent a particular task from being migrated too frequently before being given enough time to run on a core, only a task that hasn't been migrated recently is considered as an eligible candidate. Using such optimal stopping strategy, online observations are made and task migration decisions are made independently without observing all speeds and synchronizing with the other balancer threads. On average, only a small proportion of all process speeds are observed before making a selection.

Migrating memory-bound tasks across nodes particularly on NUMA systems can be expensive because memory pages are not migrated and accessing non-local memory is slower. Performance degradation due to a high rate of migrations can be a problem for memory-bound tasks, unlike CPU-bound tasks. As a general strategy to mitigate this, we have disabled inter-node migrations whenever the migration rate is high.

3.3. Software Implementation

The load balancer is a multithreaded user-space application written in C (although it can very well be another programming language). It takes as an argument the statement to execute the parallel application. Then, it forks a child process that starts the parallel application. Alternatively, the load balancer can be launched with a list of PIDs of the parallel application that has been already running. After the parallel application is started, the load balancer creates per-core balancer threads, each of which executes the

algorithm in Figure 3 at every interval τ. Initially, the tasks of the parallel application are evenly distributed by pinning them to all cores in a round-robin fashion.

Throughout the program, pinning a task to a core is achieved by using the `sched_setaffinity()` system call. The system call sets a process's CPU affinity mask and the effect is immediate. Once a task is migrated to a core using the system call, the pinning remains constant throughout the execution and the Linux scheduler does not change it. This behavior is important and beneficial to our load balancer because we don't want the Linux scheduler interfere with the load balancing and move around tasks to cores against our intention.

The `/proc` file system is queried to identify the task PIDs of the parallel application. The PIDs belong the parallel application are listed under `/proc/<PID>/task` where `<PID>` is the PID of the parallel application. Gathering the execution time and elapsed time statistics of tasks is done through the netlink-based `taskstats` interface. By opening a unicast netlink socket (NETLINK_GENERIC family) from user-space, commands specifying a PID can be sent to the kernel to get the response containing the statistics for the task. The `taskstats` interface is an efficient way to obtain task statistics, and is more accurate than gathering statistics in user-space alone. From the interface encapsulated in `/usr/include/linux/taskstats.h`, three basic accounting fields are of particular interest in our load balancer.

```
struct taskstats {

     ...

     __u64   ac_etime;   /* Elapsed time [usec]    */
     __u64   ac_utime;   /* User CPU time [usec]   */
     __u64   ac_stime;   /* System CPU time [usec] */

     ...

};
```

The speed is calculated by $(ac_{utime} + ac_{stime})/ac_{etime}$. Any calculated speed that is above 1.0 due to measurement inaccuracy and floating point precision is adjusted to 1.0.

4. EXPERIMENTS

In this section, we describe the experiments we have performed to demonstrate the effectiveness of our cardinal payoff online selection load balancer in dedicated and non-dedicated environments.

4.1. Setup

The evaluation of our online selection load balancer (OSLB) was done on a 12-core system. The specification of the system is shown in Table 1.

Table 1. Specification of the test system

Item	Details
Processor	Intel® Xeon® X5675 @ 3.07GHz
Cores	12 (6 cores per chip)
L1 Cache	32 KB (I-cache) + 32 KB (D-cache) per core
L2 Cache	256 KB per core
L3 Cache	12 MB per chip
Memory	80 GB
OS	Ubuntu 12.10 (GNU/Linux kernel 3.5.0-21)
NUMA Nodes	2
Hyper-Threading®	Yes

We have disabled Hyper-Threading® so that we test only on physical cores and avoid possible idiosyncrasies from having logical cores. Also, we have disabled unnecessary processes that are not related to our load balancer and parallel applications.

We ran various benchmarks from the NAS Parallel Benchmarks (NPB) [15] in OpenMP on dedicated and non-dedicated environments. They range in small and large memory footprints, and short and long execution times. One study found that the

methods of these benchmarks would be used in various commercial software domains [2], so experimenting with these benchmarks is reasonable. The selected benchmarks from the NPB are ep.C, cg.B, ft.B, is.C, lu.A, sp.A, mg.B, bt.A, and ua.A. The benchmarks were compiled with `gcc` 4.7.2, which has `-fopenmp` option support for OpenMP API version 3.0. They were run on dedicated environment and non-dedicated environment, and were repeated ten times. Each benchmark was run with 12 threads, one thread per core. We compared test results from load balancing by the Linux scheduler and by OSLB. For simulating an imbalance greater than the one caused by OS jitter, a highly CPU-bound application that sleeps periodically was pinned to one of the cores while executing. We simulated the non-dedicated environment by executing all the benchmarks concurrently, and the times for all of them to complete under the Linux scheduler and OSLB were compared.

Each benchmark has a varying degree of memory usage, some greater and some others less. Benchmarks with more memory usage are impacted more by task migrations. Migrating memory-bound tasks across nodes on NUMA systems is expensive because memory pages are not migrated and accessing non-local memory is slower. The problem is exacerbated if the migration rate is high. For a highly CPU-bound benchmark like ep.C, this is not an issue. As a general strategy to mitigate this, we have disabled inter-node migrations whenever the migration rate is high.

4.2. Balancing Interval

In order to determine what value of τ works well for our balancer, we repeated experiments by executing all the benchmarks at the same time with OSLB balancing at different values of τ. Figure 4 shows speedup values we measured at various points of balancing interval τ.

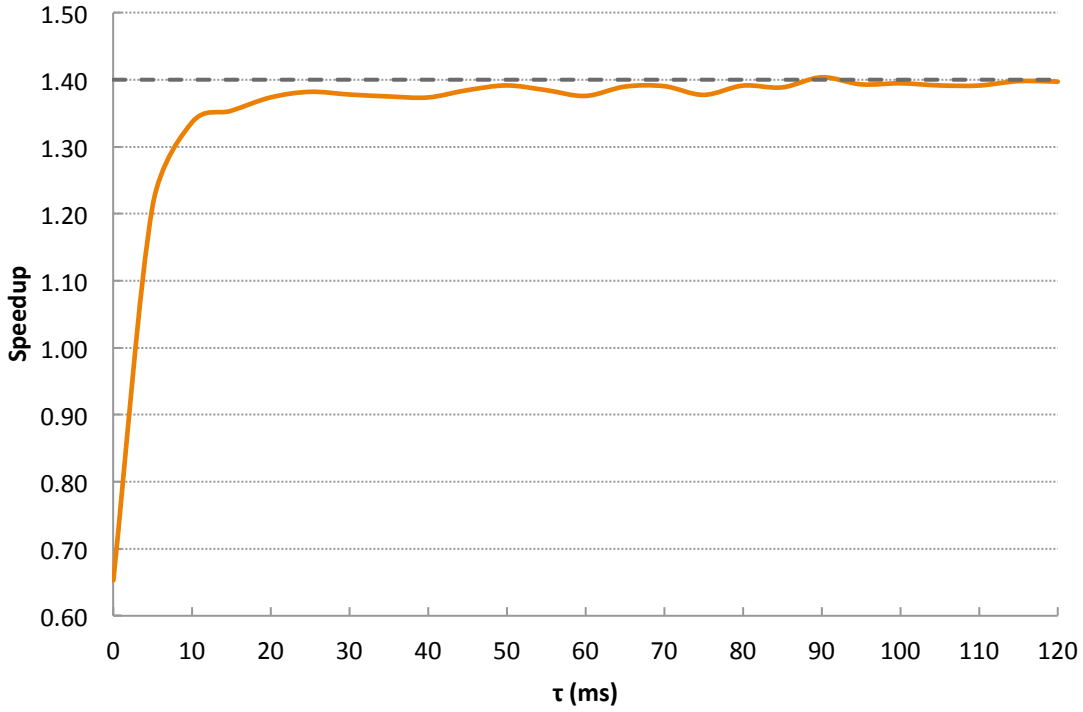

Figure 4. Speedup at different balancing interval τ.

Particularly for $\tau < 10$ (in milliseconds), the benefit from OSLB is drastically degraded. This is attributed to the overhead from frequent balancing activities. The overhead has taken a toll on the performance improvement. Furthermore, the speedup actually falls

steeply below 1.00 at $\tau < 4$ and reaches .65 at $\tau = 0$. It can be seen that a very short balancing interval is not beneficial.

As the balancing interval was increased from $\tau \approx 20$, the speedup increased very quickly and stayed very close to the asymptotic value 1.40 measured from our experiment. From our empirical results, we decided $\tau = 100$ is a reasonable value for the balancing interval of OSLB.

4.3. Dedicated Environment

Figure 5 shows the speedup of OSLB against the Linux scheduler on a system where there are no other applications but OS jitter to cause significant (if indeed) imbalance.

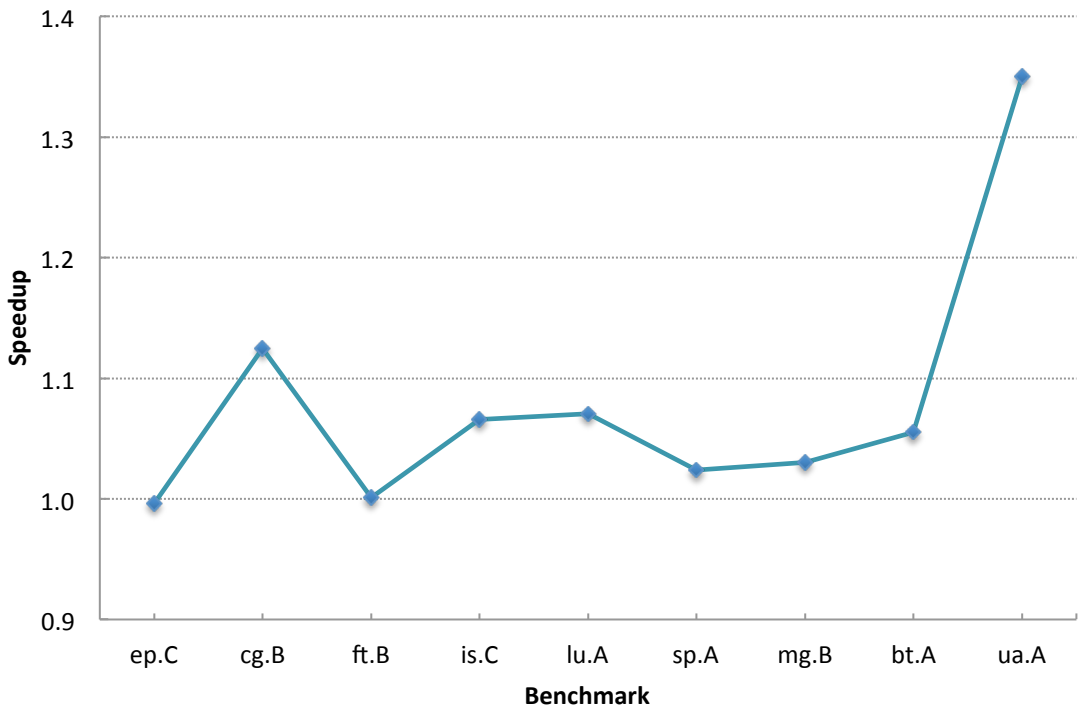

Figure 5. Speedup of OSLB against the Linux scheduler on a system with only OS jitter.

In this particular test environment, the overall average speedup was 8%. The ua.A saw the most benefit from OSLB by getting 35% speedup. The ua.A apparently got much imbalance and the Linux scheduler did very poorly in correcting imbalance in this benchmark. Aside the extraordinary case, most benchmarks got speedup not far from the average speedup. The ep.C did not see any speedup at all and the speedup fell just below 1.0 by an insignificant amount, and one explanation is that the embarrassingly parallel benchmark (thus, the name) doesn't exchange much data among the subtasks and the amount of imbalance caused by the OS jitter is not much. The speedup for ft.B was insignificant, too, as the dedicated environment didn't cause much imbalance.

Figure 6 shows the speedup of OSLB against the Linux scheduler when additional imbalance was caused by a CPU-bound application.

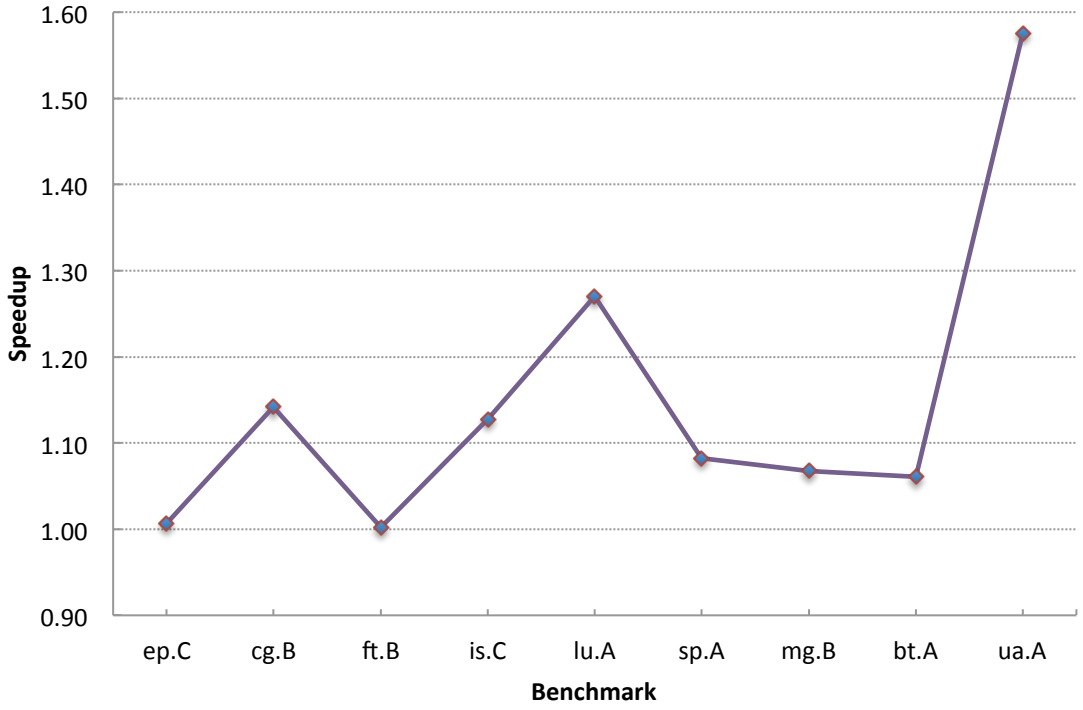

Figure 6. Speedup of OSLB against the Linux scheduler on a system with a CPU-bound application.

The average speedup was 14.8% and the speedup of all benchmarks rose overall compared to the results from figure 5. Again, the ep.C and ft.B saw an almost nonexistent amount of speedup. Furthermore, the ua.A saw a remarkable 58% speedup.

4.4. Non-Dedicated Environment

Figure 7 shows the comparison of the time for all the benchmarks to complete under each load balancer in a non-dedicated environment. Both from figure 7 and the

actual measurement values shown in table 2, it is clear that OSLB outperformed the Linux scheduler in load balancing. Compared to the mean time of completion under the Linux scheduler, the benchmarks completed about 40% faster under OSLB.

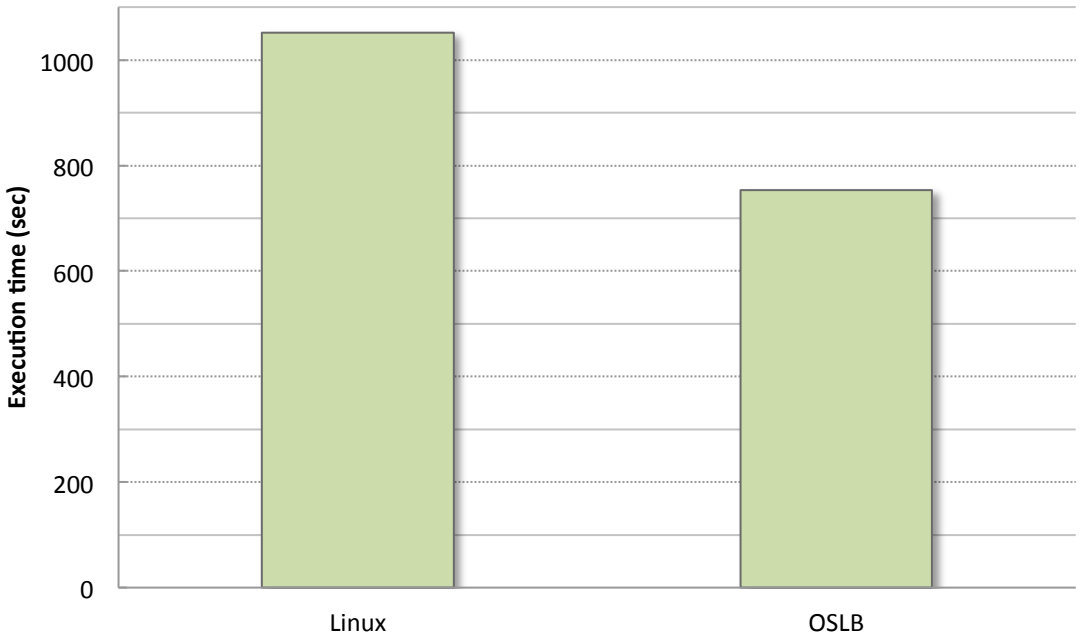

Figure 7. Total execution time in a non-dedicated environment.

The variance among the repeated results under OSLB was 38.34, which was much lower than the variance 550.78 under the Linux scheduler. Also, the maximum-to-minimum ratio of the run times under OSLB was 1.03, which is lower than 1.07 under the Linux scheduler. This implies that we can expect OSLB to achieve a more consistent load balancing speedup under a competitive environment.

Table 2. Total execution time (sec) and variance in a non-dedicated environment.

	Linux	OSLB
Mean:	1051.54	753.76
Variance:	550.78	38.34
Max/Min Ratio:	1.07	1.03

Figure 8 shows the speedup for each benchmark under OSLB from our non-dedicated environment tests.

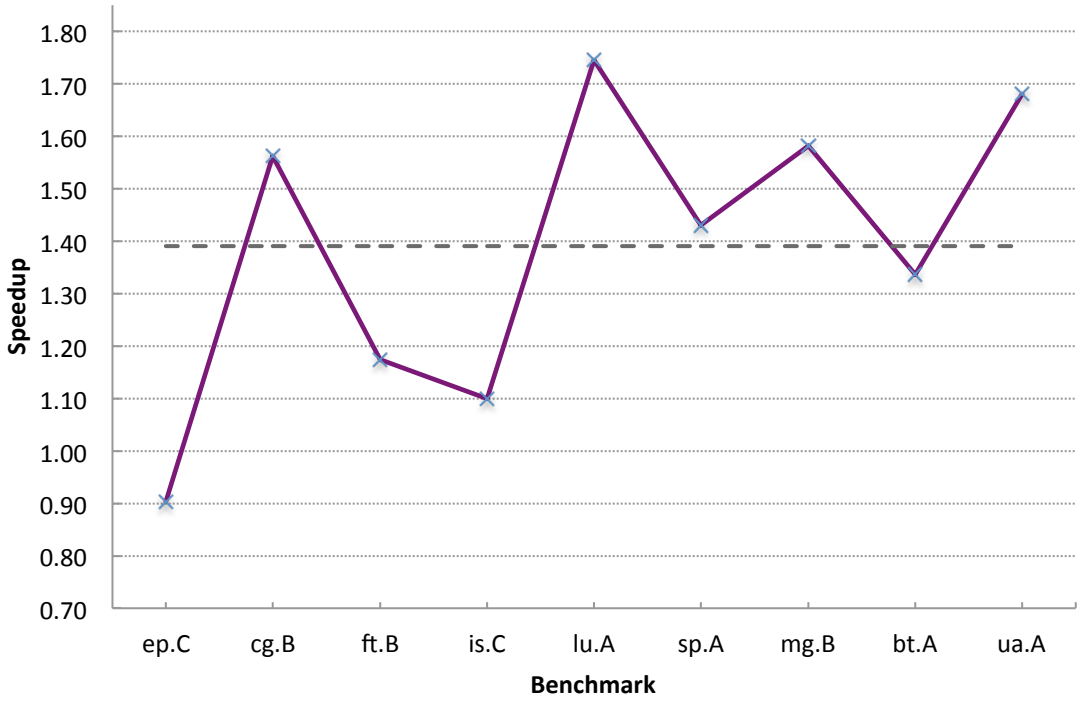

Figure 8. Speedup for each benchmark under OSLB in a non-dedicated environment.

All benchmarks except the ep.C saw good speedups ranging from 10% for is.C to 75% for lu.A, and the average speedup was close to 40%. We attribute the slow down of the

ep.C to a combination of its well-parallelized characteristics and the balancing overhead that exceeded the mediocre benefit from balancing. Also, the Linux scheduler apparently did a good job in load balancing a benchmark of such kind.

5. CONCLUSIONS

In this paper, we have investigated how a solution to a cardinal payoff variant of the secretary problem can be applied to a proactive, decentralized, dynamic load balancing technique in user-space to assist SPMD applications so that all tasks can make roughly equal progress distributed over all cores. Our load balancer OSLB used the optimal stopping strategy for a cardinal payoff variant of the secretary problem to examine only a small subset of all subtasks to make online balancing decisions without global synchronization. We examined how this method compares with the runqueue length based load balancing by the current default Linux scheduler (Completely Fair Scheduler) in terms of scalability and predictability.

The results from our experiments of running NPB-OpenMP in both dedicated and non-dedicated environments showed promising results where OSLB achieved good speedups overall. In a dedicated environment with most other system processes disabled, we saw an 8% average speedup of all the benchmarks when they were run individually. In the same environment, when there was an induced imbalance by a CPU-intensive process, the average speedup rose to 14.8%. Furthermore, we saw a 40% average speedup when all the benchmarks ran together to simulate a non-dedicated environment. Our experiments showed that OSLB was able to correct imbalances very well by making balancing decisions based on partial information of the parallel subtasks, while the default Linux scheduler didn't do well.

Our user-space, decentralized, proactive load balancer (OSLB) demonstrated the applicability of optimal stopping strategies to load balancing for scalability. As the number of cores grows, scalability and effectiveness of load balancing will be a growing challenge; and our approach has potential.

6. FUTURE WORK

There are a plenty of opportunities to further explore our load balancing technique. As the number of cores continues to grow, it will be interesting to apply our optimal stopping load balancing method to even a larger number of cores. As a 100+-core system may be available in the future, we can again evaluate for scalability and predictability on many-core systems as such.

Besides experimenting on tightly coupled multiprocessor systems like multicore and even many-core systems, there's yet another possibility with loosely coupled multiprocessor systems. The data communication complexity becomes more visible on loosely coupled systems because of higher inter-processor communication latency, and the benefit from load balancing decision-making based on a smaller subset of information may be well demonstrated on these systems.

Although we have disabled inter-node migrations in the experiments on our NUMA system whenever the migration rate is arbitrarily high, further investigations and optimizations can be done on applying our load balancing technique to NUMA systems. There will be additional measurements of cache and memory usage pattern to enable smarter decision making on migrations. Also, being aware of the node configuration will help with initial distribution of parallel subtasks to cores in different nodes.

In our study, although the optimal stopping policy based on the uniform distribution model worked well, the policy based on a more accurate probability distribution model (e.g., Gaussian distribution) can be investigated.

Finally, another area of investigation will be load balancing on heterogeneous multicore systems. More thoughts can be given to whether the current definition of task speed might be adapted so that our load balancer can work better in such systems. Additionally, along the same line of thought, capabilities to adjust dynamically different parameters—e.g., balancing interval and enabling/disabling inter-node migrations on NUMA systems—for different balancing environments and target applications will further improve our load balancer.

REFERENCES

[1] ALEXANDRESCU, A., AGAVRILOAEI, I., AND CRAUS, M. A Genetic Algorithm for Mapping Tasks in Heterogeneous Computing Systems. In the *Proceedings of 15th International Conference on System Theory, Control, and Computing* (ICSTCC '11). IEEE Computer Society, Washington, DC, USA, 2011, pp. 1-6.

[2] ASANOVIC, K., BODIK, R., DEMMEL, J., KEAVENY, T., KEUTZER, K., KUBIATOWICZ, J., MORGAN, N., PATTERSON, D., SEN, K., WAWRZYNEK, J., WESSEL, D., AND YELICK, K. 2009. A view of the parallel computing landscape. Commun. ACM 52, 10 (October 2009), pp. 56-67. DOI=10.1145/1562764.1562783 http://doi.acm.org/10.1145/1562764.1562783

[3] BEARDEN, J. N. A new secretary problem with rank-based selection and cardinal payoffs. In *Journal of Mathematical Psychology*, Vol. 50, 2006, pp. 58-59. DOI=10.1016/j.jmp.2005.11.003 http://dx.doi.org/10.1016/j.jmp.2005.11.003

[4] BLAGODUROV, S. AND FEDOROVA, A. User-level scheduling on NUMA Multicore systems under Linux. In *Proceedings of the 13th Annual Linux Symposium*. 2011.

[5] BURTON, F. W. AND SLEEP, M. R. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture* (FPCA '81). ACM, New York, NY, USA, 1981, 187-194. DOI=10.1145/800223.806778 http://doi.acm.org/10.1145/800223.806778

[6] CHEN, Q., GUO, M., AND HUANG, Z. CATS: Cache Aware Task-Stealing Based on Online Profiling in Multi-socket Multi-core Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS 2012)*. ACM, New York, NY, USA, 2012, pp. 163-172. DOI=10.1145/2304576.2304599 http://doi.acm.org/10.1145/2304576.2304599

[7] CUNNINGHAM, A. Samsung Galaxy S4 processor: how the eight-core CPU works. Available from `http://www.wired.co.uk/news/archive/2013-03/19/exynos-5-octa`.

[8] EL-ZOGHDY, S. F. A Load Balancing Policy for Heterogeneous Computational Grids. In *International Journal of Advanced Computer Science and Applications*, 2(5), 2011, pp. 93-100.

[9] GILBERT, J. P. AND MOSTELLER, F. Recognizing the maximum of a sequence. In *Journal of the American Statistical Association*, *61*, 1966, pp. 35-73. DOI=10.1080/01621459.1966.10502008 http://dx.doi.org/10.1080/01621459.1966.10502008

[10] GUO, Y., BARIK, R., RAMAN, R., AND SARKAR, V. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (IPDPS '09). IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–12. DOI=10.1109/IPDPS.2009.5161079 http://dx.doi.org/10.1109/IPDPS.2009.5161079

[11] GUO, Y., ZHAO, J., CAVE, V., AND SARKAR, V. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '10). ACM, New York, NY, USA, 2010, pp. 341-342. DOI=10.1145/1693453.1693504 http://doi.acm.org/10.1145/1693453.1693504

[12] HOFMEYR, S., COLMENARES, J. A., IANCU, C., AND KUBIATOWICZ, J. Juggle: Addressing extrinsic load imbalances in SPMD applications on multicore computers. In the *Cluster Computing journal* (Springer). 2012. DOI=10.1007/s10586-012-0204-0 http://dx.doi.org/10.1007/s10586-012-0204-0

[13] HOFMEYR, S., IANCU, C., AND BLAGOJEVIĆ, F.. Load balancing on speed. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '10). ACM, New York, NY, USA, 2010, pp. 147-158. DOI=10.1145/1693453.1693475 http://doi.acm.org/10.1145/1693453.1693475

[14] INTEL. Intel's Teraflops Research Chip. Available from `http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf`.

[15] NASA. NAS Parallel Benchmarks. Available from `http://www.nas.nasa.gov/publications/npb.html`.

[16] OLIVIER, S. L., PORTERFIELD, A. K., WHEELER, K. B., AND PRINS, J. F. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers* (ROSS '11). ACM, New York, NY, USA, 2011, pp. 49-56. DOI=10.1145/1988796.1988804 http://doi.acm.org/10.1145/1988796.1988804

[17] WIKIPEDIA. Secretary Problem. Available from http://en.wikipedia.org/wiki/Secretary_problem.

[18] WIKIPEDIA. SPMD (Single Program, Multiple Data). Available from http://en.wikipedia.org/wiki/SPMD.