Multicore Locks: The Case is not Closed Yet
(Technical report – Extended and updated version of the USENIX ATC'16 article)
Version: June 20th, 2016

Hugo Guiroux[†⋆] Renaud Lachaize[†⋆] Vivien Quéma[†‡⋆]
[†]*Université Grenoble Alpes* [‡]*Grenoble INP*
[⋆]*LIG (CNRS UMR 5217)*

Abstract

NUMA multicore machines are pervasive and many multithreaded applications are suffering from lock contention. To mitigate this issue, application and library developers can choose from the plethora of optimized mutex lock algorithms that have been designed over the past 25 years. Unfortunately, there is currently no broad study of the behavior of these optimized lock algorithms on realistic applications. In this paper, we attempt to fill this gap. We perform a performance study of 27 state-of-the-art mutex lock algorithms on 35 applications. Our study shows that the case is not yet closed regarding locking on multicore machines. Indeed, our conclusions include the following findings: (i) at its optimized contention level, no single lock is the best for more than 48% of the studied workloads; (ii) every lock is harmful for several applications, even if the application parallelism is properly tuned; (iii) for several applications, the best lock changes when varying the number of threads. These findings call for further research on optimized lock algorithms and dynamic adaptation of contention management.

1 Introduction

Today, multicore machines are pervasive and many multithreaded applications are suffering from bottlenecks related to critical sections and their corresponding locks. To mitigate these issues, application and library developers can choose from the plethora of optimized mutex lock algorithms that have been designed over the past 25 years but there is currently no clear study to guide this puzzling choice for realistic applications. In particular, the most recent and comprehensive empirical performance evaluation on multicore synchronization [9], due to its breadth (from hardware protocols to high-level data structures), only provides a partial coverage of locking algorithms. Indeed, the aforementioned study only considers 9 algorithms, does not consider hybrid spinning/blocking waiting policies, omits emerging approaches (e.g., load-control algorithms described in §2) and provides a modest coverage of hierarchical locks [14, 5, 6], a recent and efficient approach. Furthermore, most of the observations are based on microbenchmarks. Besides, in the case of papers that present a new lock algorithm, the empirical observations are often focused on the specific workload characteristics for which the lock was designed [21, 26], or mostly based on microbenchmarks [14, 12].

The present paper provides a broad performance study on Linux/x86 of 27 state-of-the-art mutex lock algorithms on a set of 35 realistic and diverse applications (the PARSEC, Phoenix, SPLASH2 suites, MySQL and an SSL proxy). We make a number of observations, several of which have not been previously mentioned: (*i*) about 60% of the studied applications are significantly impacted by lock performance; (*ii*) no single lock is systematically the best, even for a fixed number of contending cores; (*iii*) worse, at their optimized contention level (individually tuned for each application), the best locks never dominate for more than 48% of the lock-sensitive applications; (*iv*) any of the locks is harmful (i.e., significantly inefficient compared to the best one) for at least several workloads; (*v*) across all the lock-sensitive applications, there is no clear performance hierarchy among the locks, even at a fixed number of contending cores; (*vi*) for a given application, the best lock varies according to both the number of contending cores and the machine; (*vii*) unlike previous recommendations [9] advocating that standard Pthread mutex locks should be avoided for workloads using no more than one thread per core, we find that, with our studied workloads, the current Linux implementation of these locks actually yields good performance for many applications with this pattern. Moreover, we show that all these results hold even when each configuration, i.e., each *(application, lock)* pair, is tuned to its optimal degree of parallelism. From our performance study, we draw two main conclusions. First, specific lock algorithms should not be hardwired into the

code of applications. Second, the observed trends call for further research both regarding lock algorithms and runtime support for parallel performance and contention management.

To conduct our study, manually modifying all the applications in order to retrofit the studied lock algorithms would have been a daunting task. Moreover, using a meta-library that allows plugging different lock algorithms under a common API (such as liblock [26] or libslock [9]) would not have solved the problem, as this would still have required a substantial re-engineering effort for each application. In addition, such meta-libraries provide no or limited support for important features like Pthread condition variables, used within many applications. Therefore, we implemented LiTL[1], a low-overhead library that allows transparent interposition of Pthread mutex lock operations and support for mainstream features like condition variables, without any restriction on the application-level locking discipline.

The remainder of the paper is organized as follows: §2 presents a taxonomy of existing lock designs and the list of algorithms covered by our study. §3 describes our experimental setup and the studied applications. §4 describes the LiTL library. §5 exposes the main results from our empirical observations. §6 discusses related works and §7 concludes the paper.

2 Lock algorithms

2.1 Background

The body of existing works on optimized lock algorithms for multicore architectures is rich and diverse and can be split into the following five categories:

1) Flat approaches correspond to simple algorithms (typically based on one or a few shared variables accessed by atomic instructions) such as: simple spinlock [33], backoff spinlock [2, 30], test and test-and-set (TTAS) lock [2], ticket lock [30], partitioned ticket lock [11], and standard Pthread mutex lock.
2) Queue-based approaches correspond to locks based on a waiting queue in order to improve fairness as well as the memory traffic, such as: MCS [30, 33] and CLH [7, 29, 33].
3) Hierarchical approaches are specifically aimed at providing scalable performance on large-scale NUMA machines, by attempting to reduce the rate of lock migrations among NUMA nodes. This category includes HBO [32], HCLH [28], FC-MCS [13], HMCS [5], AHMCS [6] and the algorithms that stem from the *lock cohorting* framework [14]. A cohort lock is based on a combination of two lock algorithms (similar or different): one used for the global lock and one used for the local locks (there is one local lock per NUMA node); in the usual C-L_A-L_B notation, L_A and L_B respectively correspond to the global and the node-level lock algorithms. The list includes C-BO-MCS, C-PTL-TKT and C-TKT-TKT (also known as Hticket [9]). The *BO*, *PTL* and *TKT* acronyms respectively correspond to backoff lock, partitioned ticket lock, and standard ticket lock.
4) Load-control approaches correspond to algorithms that aim at limiting the number of threads that concurrently attempt to acquire a lock, in order to prevent a performance collapse. These algorithms are derived from queue-based locks. This category includes MCS-TimePub[2] [19] and so-called *Malthusian algorithms* like Malth_Spin and Malth_STP[3] [12].
5) Delegation-based approaches correspond to algorithms in which it is (sometimes or always) necessary for a thread to delegate the execution of a critical section to another thread. The typical benefits expected from such approaches are improved cache locality and better resilience under high lock contention. This category includes Oyama [31], Hendler [20], RCL [26], CC-Synch [15] and DSM-Synch [15].

Another important design dimension is the *waiting policy* used when a thread cannot immediately obtain a requested lock [12]. There are three main approaches: (i) spinning on a memory address, (ii) immediate parking (i.e., blocking the thread) either for a fixed amount of time or until the thread gets a chance to obtain the lock, and (iii) spinning-then-parking (STP), a hybrid strategy using a fixed or adaptive threshold [22]. The choice of the waiting policy is mostly orthogonal to the lock design but, in practice, policies other than pure spinning are only considered for certain types of locks: the queue-based (from categories 2–4 above) and the standard Pthread mutex locks. Besides, note that the GNU C library for Linux provides two versions of Pthread mutex locks: the default one uses parking (via the `futex` syscall) and the second one uses an adaptive spin-then-park strategy. The latter version can be enabled with the `PTHREAD_MUTEX_ADAPTIVE_NP` option [23].

2.2 Studied algorithms

Our choice of studied locks is guided by the decision to focus on *portable* lock algorithms. We therefore exclude the following locks that require strong assumptions on

[1] LiTL: Library for Transparent Lock interposition.

[2] MCS-TimePub is mostly known as MCS-TP but we use MC-TimePub to avoid confusion with MCS_STP.

[3] Malth_Spin and Malth_STP correspond to MCSCR-S and MCSCR-STP, respectively, but we do not use the latter names to avoid confusion with other MCS locks.

Name Total #cores	**A-64** 64	**A-48** 48	**I-48** 48 (no hyperthreading)
Server model	Dell PE R815	Dell PE R815	SuperMicro SS 4048B-TR4FT
Processors Microarchitecture Core clock Last-level cache (per node)	4× AMD Opteron 6272 Bulldozer / Interlagos 2.1 GHz 8 MB	4× AMD Opteron 6344 Piledriver / Abu Dhabi 2.6 GHz 8 MB	4× Intel Xeon E7-4830 v3 Haswell-EX 2.1 GHz 30 MB
Interconnect	HT3 - 6.4 GT/s per link	HT3 - 6.4 GT/s per link	QPI - 8 GT/s per link
Memory #NUMA nodes (#cores/node)	256 GB DDR3 1600 MHz 8 (8)	64 GB DDR3 1600 MHz 8 (6)	256 GB DDR4 2133 MHz 4 (12)
Network interfaces (10 GbE)	2× 2-port Intel 82599	2× 2-port Intel 82599	2-port Intel X540-AT2

Table 1: Hardware characteristics of the testbed platforms.

the application/OS behavior, code modifications, or fragile performance tuning: HCLH, HBO, FC-MCS, and all the delegation-based locks (see Dice et al. [14] for detailed arguments).

Our study considers 27 mutex lock algorithms that are representative of both well-established and state-of-the-art approaches. We use the *_Spin* and *_STP* suffixes to differentiate variants of the same algorithm that only differ in their waiting policy. The *-LS* tag corresponds to optimized algorithms borrowed from libslock [9]. Our set includes ten flat locks (Backoff, Partitioned ticket, Phtread, Pthread adaptive, Spinlock, Spinlock-LS, Ticket, Ticket-LS, TTAS, TTAS-LS), seven queue-based locks (Alock-LS, CLH-LS, CLH_Spin, CLH_STP, MCS-LS, MCS_Spin, MCS_STP), seven hierarchical locks (C-BO-MCS_Spin, C-BO-MCS_STP, C-PTL-TKT, C-TKT-TKT, Hticket-LS, HMCS, AHMCS), and three load-control locks (Malth_Spin, Malth_STP, MCS-TimePub).

3 Experimental setup and methodology

3.1 Testbed and studied applications

Our experimental testbed consists of three Linux-based servers whose main characteristics are summarized in Table 1. All the machines run the Ubuntu 12.04 OS with a 3.17.6 Linux kernel (CFS scheduler), glibc 2.15 and gcc 4.6.3. For our comparative study of lock performance, we consider (i) the applications from the PARSEC benchmark suite (emerging workloads), (ii) the applications from the Phoenix 2 MapReduce benchmark suite, (iii) the applications from the SPLASH2 high-performance computing benchmark suite[4], (iv) the MySQL database running the Cloudstone workload, and (v) SSL proxy, an event-driven SSL endpoint that processes small messages. In order to evaluate the impact of workload changes on locking performance, we also consider so called "long-lived" variants of four of the above workloads denoted with a "_ll" suffix. Note that six of the applications cannot be evaluated on the two 48-core machines because, by design, they only accept a number of threads that correspond to a power of two: facesim, fluidanimate (from PARSEC), fft, ocean_cp, ocean_ncp, radix (from SPLASH2).

Most of these applications use a number of threads equal to the number of cores, except the three following ones: dedup (3× threads), ferret (4× threads) and MySQL (hundreds of threads). Two thirds of the applications use Pthread condition variables.

3.2 Tuning and experimental methodology

For the lock algorithms that rely on static thresholds, we use the recommended values from the original papers and implementations. The algorithms based on a spin-then-park waiting policy (e.g., Malth_STP [12]) rely on a fixed threshold for the spinning time that corresponds to the duration of a round-trip context switch [22] — in this case, we calibrate the duration using a microbenchmark on the testbed platform.

All the applications are run with memory interleaving (via the `numactl` utility) in order to avoid NUMA memory bottlenecks. Generally, in the experiments presented in this paper, we study the performance impact of a lock for a given contention level, i.e., the number of threads of the application. We vary the contention level at the granularity of a NUMA node (i.e., 8 cores for the A-64 machine, 6 cores for the A-48 machine, and 12 cores for the I-48 machine). For most of the experiments detailed in the paper, the application threads are not pinned to specific cores. The impact of pinning is nonetheless discussed in §5.3.

Finally, each experiment is run at least five times and we compute the average value. Overall, we observe little variability for most configurations. For all experiments, the considered application-level performance metric is the throughput (operations per time unit).

[4]We excluded the Cholesky application because of extremely short completion times.

4 The LiTL lock interposition library

In order to carry out the lock comparison study, we have developed LiTL, an interposition library for Linux/x86 allowing transparently replacing the lock algorithm used for Pthread mutexes. We describe its design, implementation, and assess its performance.

4.1 Design

The design of LiTL does not impose any restriction on the level of nested locking and is compatible with arbitrary locking disciplines (e.g., hand-over-hand locking [33]). The pseudo-code of the main wrapper functions of the LiTL library is depicted in Figure 1.

```
// return values and error checks
// omitted for simplification

pthread_mutex_lock(pthread_mutex_t *m) {
    optimized_mutex_t *om = get_optimized_mutex(m);
    if (om == null) {
        om = create_and_store_optimized_mutex(m);
    }
    optimized_mutex_lock(om);
    real_pthread_mutex_lock(m);
}

pthread_mutex_unlock(pthread_mutex_t *m) {
    optimized_mutex_t *om = get_optmized_mutex(m);
    optimized_mutex_unlock(om);
    real_pthread_mutex_unlock(m);
}

pthread_cond_wait(pthread_cond_t *c,
                  pthread_mutex_t *m) {
    optimized_mutex_t *om = get_optimized_mutex(m);
    optimized_mutex_unlock(om);
    real_pthread_cond_wait(c, m);
    real_pthread_mutex_unlock(m);
    optimized_mutex_lock(om);
    real_pthread_mutex_lock(m);
}

// Note that the pthread_cond_signal and
// pthread_cond_broadcast primitives
// do not need to be interposed
```

Figure 1: Overview of the pseudocode for the main wrapper functions of LiTL.

General principles The primary role of LiTL is to maintain a mapping structure between an instance of the standard Pthread lock (`pthread_mutex_t`) and an instance of the chosen optimized lock type (e.g., MCS_Spin). This implies that LiTL must keep track of the lifecycle of all the application's locks through interposition of the calls to `pthread_mutex_init()` and `pthread_mutex_destroy()`, and that each interposed call to `pthread_mutex_lock()` must trigger a lookup for the instance of the optimized lock. In addition, lock instances that are statically initialized can only be discovered and tracked upon the first invocation of `pthread_mutex_lock()` on them (i.e., a failed lookup leads to the creation of a new mapping).

The `lock`/`unlock` API of several lock algorithms requires an additional parameter (called "struct" hereafter) in addition to the lock pointer. For example, in the case of an MCS lock, this parameter corresponds to the record to be inserted in (or removed from) the lock's waiting queue. In the general case, a struct cannot be reused nor freed before the corresponding lock has been released. For instance, an application may rely on nested critical sections (i.e., a thread T must acquire a lock L_2 while holding another lock L_1). In this case, T must use a distinct struct for L_2 in order to preserve the integrity of L_1's struct. In order to gracefully support the most general cases, LiTL systematically allocates exactly one struct per lock instance and per thread.

Supporting condition variables Dealing with condition variables inside each optimized lock algorithm would be complex and tedious as most locks have not been designed with condition variables in mind. We therefore use the following strategy: our wrapper for `pthread_cond_wait()` internally calls the true `pthread_cond_wait()` function. To issue this call, we need to hold a real Pthread mutex lock (of type `pthread_mutex_t`). This strategy (depicted in the pseudocode of Figure 1) does not introduce high contention on the internal Pthread lock. Indeed, for workloads that do not use condition variables, the Pthread lock is only requested by the holder of the optimized lock associated with the critical section. Furthermore, workloads that use condition variables are unlikely to have more than two threads competing for the Pthread lock: the holder of the optimized lock and a notified thread. Note that the latter claim also holds for workloads that rely on `pthread_cond_broadcast()` because the Linux implementation of this call only wakes up a single thread from the wait queue of the condition variable and directly transfers the remaining threads to the wait queue of the Pthread lock.

Support for specific lock semantics The design of LiTL is compatible with specific lock semantics when the underlying lock algorithms offer the corresponding properties. For example, LiTL supports non-blocking lock requests (`pthread_mutex_trylock()`) for all the currently implemented locks except CLH-based locks and Hticket-LS, which are not compatible with such semantics. Although not yet implemented, LiTL could easily support blocking requests with timeouts for the so-called "abortable" locks (e.g., MCS-Try [34] and MCS-TimePub [19]). Moreover, support for optional Pthread

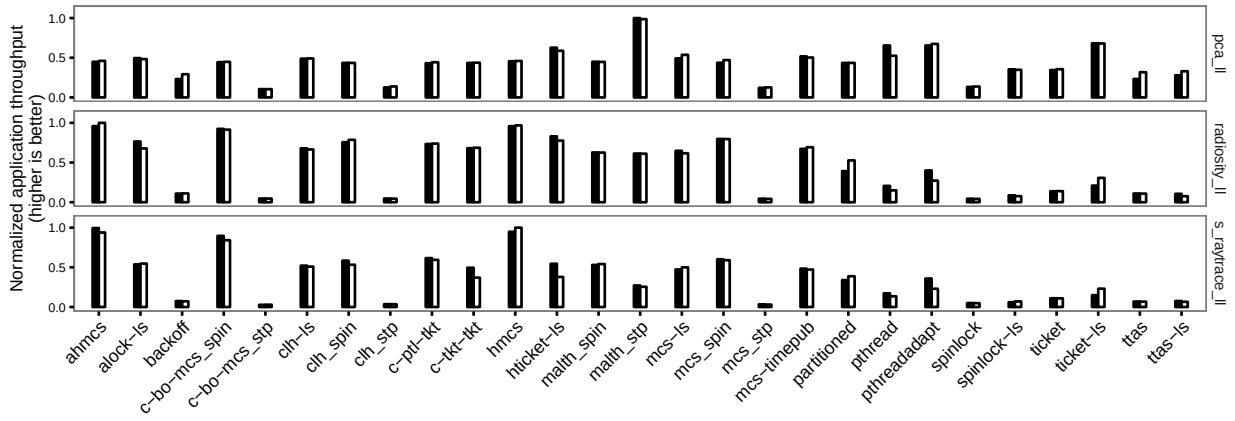

Figure 2: Performance comparison (throughput) of manually implemented locks (black bars) vs. transparently interposed locks using LiTL (white bars). The throughput is normalized with respect to the best performing configuration for a given application (**A-64 machine**).

mutex behavior like reentrance and error checks[5] could be easily integrated in the generic wrapper code by managing fields for the current owner and the lock acquisition counter.

4.2 Implementation

The library relies on a scalable concurrent hash table (CLHT [10]) in order to store, for each Pthread mutex instance used in the application, the corresponding optimized lock instance, and the associated per-thread structs. For well-established locking algorithms like MCS, the code of LiTL borrows from other libraries [9, 1, 26]. Other algorithms are implemented from scratch based on the description of the original papers. For algorithms that are based on a parking or on a spinning-then-parking waiting policy, our implementation directly relies on the `futex` Linux system call.

Finally, the source code of LiTL relies on preprocessor macros rather than function pointers. Indeed, we have observed that the use of function pointers in the critical path introduced a surprisingly high overhead. Moreover, all data structures are cache-aligned in order to mitigate the impact of false sharing.

4.3 Experimental validation

In this section, we assess the performance of LiTL using the A-64 machine. To that end, we compare the performance (throughput) of each lock on a set of applications running in two distinct configurations: manually modified applications and unmodified applications using interposition with LiTL. Clearly, one cannot expect to obtain exactly the same results in both configurations, as the setups differ in several ways, e.g., with respect to the exercised code paths, the process memory layout and the allocation of the locks (e.g., stack- vs. heap-based). However, we show that between both configurations: (i) the achieved performance is close and (ii) the general trends for the different locks remain stable.

We selected three applications: pca_ll, radiosity_ll and s_raytrace_ll. These three applications are particularly lock-intensive and the last two use Pthread condition variables. Therefore, all three represent an unfavorable case for LiTL. Moreover, we focus the discussion on the results under the highest contention level (i.e., when the application uses all the cores of the target machine), as this again represents an unfavorable case for LiTL.

Figure 2 shows the normalized performance (throughput) of both configurations (manual/interposed) for each *(application, lock)* pair: black bars correspond to manually implemented locks, whereas white bars correspond to transparently interposed locks using LiTL. In addition, Table 2 summarizes the performance differences for each application: number of locks for which each version performs better and, in each case, the average gain and the relative standard deviation.

We observe that, for all of the three applications, the results achieved by the two versions of the same lock are very close: the average performance difference is below 5%. Besides, Figure 2 highlights that the general trends observed with the manual versions are preserved with the interposed versions. We thus conclude that using LiTL to study the behavior of lock algorithms in an application yields only very modest differences with respect to the performance behavior of a manually modified version.

[5]Using respectively the `PTHREAD_MUTEX_RECURSIVE` and `PTHREAD_MUTEX_ERRORCHECK` attributes.

		pca_ll	radiosity_ll	s_raytrace_ll
Manual	Winners	10	17	19
	Average Gain	2%	3%	4%
	Rel. Dev.	4%	4%	5%
LiTL	Winners	17	10	8
	Average Gain	2%	3%	3%
	Rel. Dev.	2%	5%	3%

Table 2: Detailed statistics for the performance comparison of manually implemented locks vs. transparently interposed locks using LiTL (**A-64 machine**).

5 Performance study of lock algorithms

In this section, we use LiTL to compare the behavior of the different lock algorithms on different workloads and at different levels of contention. In the interest of space, we do not systematically report the observed standard deviations. However, in order to mitigate the impact of variability, when comparing the performance of two locks, we consider a margin of 5%: lock A is considered better than lock B if B's achieved performance is below 95% of A's. Besides, in order to make fair comparisons, the results presented for the Pthread locks are obtained using the same library interposition mechanism as with the other locks.

Note that some configurations are not tested because of specific restrictions. First, streamcluster, streamcluster_ll, and vips cannot use CLH-based locks or Hticket-LS as they do not support trylocks semantics. Second, we omit the results for most locks with MySQL: given the extremely large ratio of threads to cores, most locks yield performance close to zero. Third, some applications, e.g., dedup and fluidanimate, run out of memory for some configurations.

Finally, for the sake of space, we do not report all the results for the three studied machines. We rather focus on the A-64 machine and provide summaries of the results for the A-48 and I-48 machines. Nevertheless, the entire set of results can be found in the Appendices.

The section is structured as follows. §5.1 provides preliminary observations that drive the study. §5.2 answers the main questions of the study regarding the observed lock behavior. §5.3 discusses additional observations.

5.1 Preliminary observations

Before proceeding with the detailed study, we highlight some important characteristics of the applications.

5.1.1 Selection of lock-sensitive applications

Table 3 shows two metrics for each application and for different numbers of nodes on the A-64 machine: the performance gain of the best lock over the worst one, as well as the relative standard deviation for the performance of the different locks. For the moment, we only focus on the relative standard deviations at the maximum number of nodes (*max nodes*—highest contention) given in the 5th column (the detailed results from this table are discussed in §5.2.1).

We consider that an application is *lock-sensitive* if the relative standard deviation for the performance of the different locks at max nodes is higher than 10% (highlighted in bold font). We observe that about 60% of the applications are impacted by locks. We observe similar trends on the three studied machines (Tables 4 , 15, 16).

In the remainder of this study, we focus on lock-sensitive applications.

	Gain 1 node	R.Dev. 1 node	Gain max nodes	R.Dev. max nodes	Gain opt nodes	R.Dev. opt nodes
barnes	10%	2%	36%	8%	31%	7%
blackscholes	11%	2%	2%	1%	2%	1%
bodytrack	1%	0%	9%	2%	4%	1%
canneal	5%	1%	7%	2%	7%	2%
dedup	**984%**	**59%**	**944%**	**55%**	**984%**	**59%**
facesim	**9%**	**2%**	**771%**	**69%**	**13%**	**3%**
ferret	**1%**	**0%**	**349%**	**56%**	**93%**	**25%**
fft	8%	2%	11%	3%	9%	2%
fluidanimate	**60%**	**13%**	**277%**	**27%**	**173%**	**22%**
fmm	**20%**	**5%**	**42%**	**11%**	**42%**	**10%**
freqmine	7%	2%	6%	1%	6%	1%
histogram	9%	3%	19%	6%	20%	4%
kmeans	9%	3%	12%	2%	12%	2%
linear_regression	**16%**	**3%**	**313%**	**26%**	**49%**	**10%**
lu_cb	11%	2%	5%	1%	5%	1%
lu_ncb	17%	5%	8%	2%	8%	2%
matrix_multiply	**6%**	**1%**	**639%**	**25%**	**357%**	**18%**
mysqld	**30%**	**9%**	**174%**	**42%**	**122%**	**36%**
ocean_cp	**23%**	**5%**	**129%**	**17%**	**21%**	**5%**
ocean_ncp	**23%**	**6%**	**116%**	**15%**	**18%**	**4%**
pca	**10%**	**3%**	**339%**	**34%**	**47%**	**9%**
pca_ll	**17%**	**4%**	**680%**	**49%**	**118%**	**21%**
p_raytrace	2%	0%	1%	0%	2%	0%
radiosity	**3%**	**1%**	**91%**	**17%**	**13%**	**4%**
radiosity_ll	**11%**	**3%**	**2299%**	**68%**	**180%**	**28%**
radix	2%	1%	8%	2%	8%	2%
s_raytrace	**5%**	**1%**	**1869%**	**60%**	**124%**	**28%**
s_raytrace_ll	**4%**	**1%**	**3163%**	**75%**	**157%**	**26%**
ssl_proxy	**40%**	**6%**	**1312%**	**61%**	**59%**	**11%**
streamcluster	**11%**	**3%**	**1143%**	**63%**	**11%**	**3%**
streamcluster_ll	**20%**	**4%**	**2968%**	**63%**	**35%**	**7%**
string_match	5%	2%	11%	2%	11%	2%
swaptions	8%	2%	10%	2%	10%	2%
vips	**3%**	**1%**	**177%**	**22%**	**6%**	**2%**
volrend	**7%**	**2%**	**169%**	**23%**	**24%**	**6%**
water_nsquared	**10%**	**2%**	**94%**	**14%**	**94%**	**14%**
water_spatial	**23%**	**5%**	**98%**	**15%**	**96%**	**15%**
word_count	4%	1%	19%	4%	13%	2%
x264	4%	1%	6%	2%	5%	2%

Table 3: For each application, performance gain of the best vs. worst lock and relative standard deviation (**A-64 machine**).

	A-64	A-48	I-48
# tested applications	39	33	33
# lock-sensitive applications	23	19	17

Table 4: Number of tested applications and number of lock-sensitive applications (**all machines**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	248	126	89	86	228	200	204	110	114	75	94	119	119	106	110	113	80	136	120	126	147	125	141	130	125	199
facesim	412	898	464	185	200	949	895	78	292	344	324	229	711	71	1k	948	87	26	895	91	67	726	632	919	456	138	232
ferret	134	127		7		116	174		56	76	100	109	57		194	192			173					182	29		
fluidanimate	-	69		7	28	-	-	-				-	7	53	8	12	54	7				16	9	13	9		64
fmm							15	12																			
histogram	95	91	78	75	99	92	92	92	90	79	94	82	90	88	89	85	109	84	89	125	88	107	83	105	84	90	95
linear_regression	44	115		97	315	80	45	34	17	7	44	125	25	8	51	47	24		50	10	8	38	21	21	9	30	24
matrix_multiply											287				62				7								
mysqld	-	-	-	-	18	-	-	-	-	-	-	-	-		-	-		25	-			-	-	-	-	-	-
ocean_cp	107	98	122	123	153	83	124	121	84	85	96	74	87	75	111	114	82	45	103	72	73	234	255	136	75	78	71
ocean_ncp	93	73	76	91	138	84	93	79	72	80	81	58	84	85	73	92	95	61	98	97	85	206	196	89	59	59	74
pca	77	60		22	244	74	44	148	47	58	68	48	37		49	55	134	19	50	97	36	229	134	116	35	239	143
pca_ll	91	72		6	421	94	41	321	97	85	88	30	7	21	58	41	403		21	195	114	513	427	108	54	480	306
radiosity					57												69						53		10	58	38
radiosity_ll		12	7		630	12	10	699	28	16			7		13	11	792	18	48	157	71	987	577	296	115	848	466
s_raytrace				24	905	30	66	460	25	13	13	8		7			436		100	88	14	269	150	134	52	282	233
s_raytrace_ll	19	103	15	34	1k	109	107	1k	143	145	15	105	68	161	108	88	1k	118	178	371	185	1k	781	495	278	1k	649
ssl_proxy	44	69	42	38	858	65	61	1k	68	72	608	59	36	52	95	99	1k	73	87	268	195	2k	1k	360	169	879	522
streamcluster	2k	1k	3k	2k	16k	-	-	-	1k	1k	1k	-	4k	16k	4k	3k	16k	1k	1k	2k	3k	9k	7k	5k	4k	4k	3k
streamcluster_ll	421	275	577	483	8k	-	-	-	233	239	250	-	816	4k	774	590	4k	301	275	446	450	2k	2k	1k	727	1k	510
vips	64	62	28	143	35	-	-	-	61	59	131	-	229	18	46	51	18	21	60	20	21	20	31	37	33	32	37
volrend	52	80	77	87	163	70	82	123	46	53	52	49	69	128	79	86	109	82	83	131	162	222	182	74	67	80	86
water_nsquared																											
water_spatial																											

Table 5: For each *(application, lock)* pair, performance gain (in %) of the optimized configuration over the max-node configuration. The background color of a cell indicates the number of nodes (1, 2, 4, 6, or 8 nodes) for the optimized configuration: 1 | 2 | 4 | 6 | 8. Dashes correspond to untested cases. (**A-64 machine**).

5.1.2 Selection of the number of nodes

In multicore applications, optimal performance is not always achieved at the maximum number of available nodes (abbreviated as *max nodes*) due to various kinds of scalability bottlenecks. Therefore, for each *(application, lock)* pair, we empirically determine the *optimized configuration* (abbreviated as *opt nodes*), i.e., the number of nodes that yields the best performance. For the A-64 and A-48 machines, we consider 1, 2, 4, 6, and 8 nodes. For the I-48 machines, we consider 1, 2, 3, and 4 nodes. Note that 6 nodes on A-64 and A-48 correspond to 3 nodes on I-48, i.e., 75% of the available cores.

The results are displayed in Tables 5, 18, 19 (resp. for the A-64, A-48 and I-48 machines). For each *(application, lock)* pair, the corresponding cell indicates the performance gain of the optimized configuration with respect to the max-node configuration. The background color of a cell indicates the number of nodes for the optimized configuration. In addition, Table 6 provides a breakdown of the *(application, lock)* pairs according to their optimized number of nodes for all machines.

We observe that, for many applications, the optimized number of nodes is lower than the max number of nodes. Moreover, we observe (Table 5) that the performance gain of the optimized configuration is often extremely large. This confirms that tuning the degree of parallelism has frequently a very strong impact on performance. We also notice that, for some applications, the optimized number of nodes varies according to the chosen lock.

	A-64	A-48
1 Node	12%	8%
2 Nodes	28%	24%
4 Nodes	26%	21%
6 Nodes	7%	10%
8 Nodes	27%	37%

	I-48
1 Node	31%
2 Nodes	15%
3 Nodes	10%
4 Nodes	44%

Table 6: Breakdown of the *(application, lock)* pairs according to their optimized number of nodes (**all machines**).

In light of the above observations, the main questions investigated in the study (§5.2) will be considered from two complementary angles: (i) comparing locks at a fixed number of nodes, and (ii) comparing locks at their optimized configurations (i.e., with possibly a different number of nodes for each). The first angle offers insight for situations in which the degree of parallelism cannot be adjusted, while the second is useful for scenarios in which more advanced application tuning is possible.

5.2 Main questions

5.2.1 How much do locks impact applications?

Table 3 shows, for each application, the performance gain of the best lock over the worst one at 1 node, max nodes, and opt nodes for the A-64 machine. The table also shows the relative standard deviation for the performance of the different locks.

We observe that the impact of locks on the performance of applications depends on the number of nodes. **At 1 node, the impact of locks on lock-sensitive applications is moderate**. More precisely, most applications exhibit a gain of the best lock over the worst one that is lower than 30%. In contrast, **at max nodes, the impact of locks is very high for all lock-sensitive applications**. More precisely, the gain brought by the best lock over the worst lock ranges from 42% to 3163%. Finally, **at the optimized number of nodes, the impact of locks is high, but noticeably lower than at max nodes.** We explain this difference by the fact that, at max nodes, some of the locks trigger a performance collapse for certain applications (as shown in Table 5), which considerably increases the observed performance gaps between locks. We observe the same trends on the A-48 and I-48 machines (Tables 15 and 16 in the Appendices).

5.2.2 Are some locks always among the best?

Table 7 shows the *coverage* of each lock, i.e., how often it stands as the best one (or is within 5% of the best) over all the studied applications for the A-64 machine. The results are shown for three configurations: 1 node, max nodes, and opt nodes. Besides, Table 8 displays, for each machine (at 1 node, max nodes and opt nodes) the following metrics aggregated over the different locks: the min and max coverage, the average coverage, and the relative standard deviation of the coverage.

	Number of nodes		
Locks	1	Max	Opt
ahmcs	61%	22%	48%
alock-ls	65%	4%	26%
backoff	61%	39%	35%
cbomcs_spin	39%	4%	4%
cbomcs_stp	42%	8%	12%
clh-ls	68%	0%	26%
clh_spin	68%	5%	32%
clh_stp	58%	16%	21%
c-ptl-tkt	61%	17%	35%
c-tkt-tkt	65%	9%	30%
hmcs	61%	17%	39%
hticket-ls	74%	26%	42%
malth_spin	65%	9%	22%
malth_stp	54%	29%	25%
mcs-ls	65%	9%	22%
mcs_spin	65%	22%	48%
mcs_stp	75%	21%	29%
mcs-timepub	62%	38%	33%
partitioned	70%	22%	35%
pthread	46%	29%	29%
pthreadadapt	54%	33%	29%
spinlock	74%	26%	30%
spinlock-ls	78%	9%	26%
ticket	74%	22%	39%
ticket-ls	78%	22%	39%
ttas	70%	17%	30%
ttas-ls	61%	4%	26%

Table 7: For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: 1 node, max nodes, and opt nodes (**A-64 machine**).

# nodes	Coverage	A-64	A-48	I-48
1	[min; max]	[39%; 78%]	[27%; 89%]	[44%; 89%]
	Avg.	64%	67%	62%
	Rel. Dev.	10%	15%	10%
Max	[min; max]	[0%; 39%]	[0%; 47%]	[12%; 42%]
	Avg.	18%	17%	26%
	Rel. Dev.	11%	13%	10%
Opt	[min; max]	[4%; 48%]	[0%; 47%]	[6%; 47%]
	Avg.	30%	22%	27%
	Rel. Dev.	10%	12%	11%

Table 8: Statistics on the coverage of locks for three configurations: 1 node, max nodes, and opt nodes (**all machines**).

We make the following observations (Table 8). **No lock is among the best for more than 89% of the applications at 1 node and for more than 48% of the applications both at max nodes and at the optimal number of nodes**. We also observe that the average coverage is much higher at 1 node than at max nodes, and slightly higher at the optimized number of nodes than at max nodes. This is directly explained by the observations made in §5.2.1. First, at 1 node, locks have a much lower impact on applications than in other configurations and thus yield closer results, which increases their likelihood to be among the best ones. Second, at max nodes, all of the different locks cause, in turn, a performance collapse, which reduces their likelihood to be among the best locks. This latter phenomenon is not observed at the optimized number of nodes. We observe the same trends on the A-48 and I-48 machines (Tables 21 and 22 in the Appendices).

5.2.3 Is there a clear hierarchy between locks?

Table 9 shows pairwise comparisons for all locks, at max nodes on the A-64 machine. In each table, cell $(rowA, colB)$ contains the score of lock A vs. lock B, i.e., the percentage of applications for which lock A is at least 5% better than lock B. For example, Table 9 shows that for 35% of the applications, AHMCS performs at least 5% better than Backoff at the optimized number of nodes. Similarly, the table shows that Backoff is at least 5% better than AHMCS for 30% of the applications. From these two values, we can conclude that the two above mentioned locks perform very closely for 35% of the applications. At the end of each line (resp. column), the table also shows the mean of the fraction of applications for which a lock is better (resp. worse) than others. Besides, the latter two metrics are summarized for the three machines in Table 10.

We observe that **there is no clear global performance hierarchy between locks**. More precisely, for most pairs of locks (A, B), there are some applications for which A is better than B, and vice-versa (Table 9). The only marginal exceptions are the cells having 0% for value. This corresponds to pairs of locks (A, B) for which A

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		9	35	39	52	16	16	58	26	26	4	21	30	39	35	35	43	48	22	35	39	52	35	30	22	30	30	32
alock-ls	22		30	26	43	16	21	58	13	22	9	11	35	35	22	30	43	30	13	30	43	48	35	26	13	22	22	28
backoff	30	35		43	52	32	37	63	17	17	35	16	43	35	17	39	48	30	22	39	43	43	26	30	9	26	39	33
cbomcs_spin	26	26	9		35	32	26	63	9	13	17	5	17	30	22	22	39	17	13	35	35	43	26	26	4	17	35	25
cbomcs_stp	30	35	9	4		32	26	53	13	9	22	5	9	4	13	13	29	8	9	12	21	22	13	13	4	13	22	17
clh-ls	26	11	21	37	53		11	47	21	16	21	11	42	32	16	32	47	32	11	42	53	53	47	37	21	42	37	31
clh_spin	26	26	32	37	58	32		53	26	37	21	26	32	26	32	21	47	32	11	37	37	47	42	32	26	37	37	33
clh_stp	37	26	16	16	21	32	16		21	21	26	16	16	11	21	16	11	5	11	11	11	21	26	11	16	16	16	18
c-ptl-tkt	30	39	26	43	52	32	32	68		13	30	21	39	43	30	35	65	39	22	39	48	52	48	35	22	39	52	38
c-tkt-tkt	26	26	26	48	61	21	37	68	13		26	16	48	52	30	30	57	35	17	39	43	52	39	35	17	30	43	36
hmcs	22	35	39	39	43	42	32	74	17	26		26	39	35	35	26	52	39	26	39	39	48	43	30	30	30	43	37
hticket-ls	21	26	32	47	63	21	32	68	11	11	11		32	42	21	32	58	32	16	47	47	58	42	42	21	42	47	35
malth_spin	22	26	17	30	52	26	16	63	22	17	22	16		22	22	13	39	17	4	35	35	35	39	17	13	26	39	26
malth_stp	30	30	17	30	50	37	37	58	26	22	26	21	4		22	17	33	25	9	33	29	35	30	17	17	22	35	27
mcs-ls	30	26	13	39	48	16	26	63	9	9	17	16	35	26		17	39	17	4	39	43	43	35	30	4	35	43	28
mcs_spin	35	43	30	52	65	42	32	68	39	35	39	42	39	43	43		43	22	22	35	39	35	43	39	22	39	48	40
mcs_stp	35	30	17	30	33	32	32	42	30	22	30	21	17	17	26	9		12	17	21	25	17	17	13	13	17	26	23
mcs-timepub	39	43	22	48	50	42	37	68	22	26	30	16	39	29	22	9	38		13	29	33	30	43	30	22	35	48	33
partitioned	30	35	30	48	74	32	32	68	26	39	39	26	52	43	35	35	61	35		43	48	48	48	26	26	39	52	41
pthread	35	30	17	26	42	37	32	58	26	26	39	32	30	25	35	26	46	25	13		21	39	22	17	13	22	30	29
pthreadadapt	35	35	22	39	33	42	37	53	30	35	35	26	26	25	35	30	42	25	17	21		22	22	17	17	22	35	30
spinlock	35	35	22	30	39	32	32	53	35	35	39	26	43	35	35	22	39	17	22	26	30		22	13	26	26	26	31
spinlock-ls	26	26	13	30	39	26	32	47	17	17	30	11	39	30	22	39	43	30	26	26	30	39		17	4	0	17	26
ticket	35	26	22	26	52	32	32	63	30	30	35	26	30	26	30	26	48	22	13	26	39	30	30		17	13	26	30
ticket-ls	26	30	22	39	65	26	32	63	17	17	26	16	39	48	22	39	52	30	26	39	48	48	35	35		30	39	35
ttas	30	26	9	35	43	32	37	58	17	22	26	21	35	35	26	43	48	30	26	30	35	39	22	17	9		17	30
ttas-ls	35	17	13	13	35	26	21	53	13	13	9	5	22	17	13	26	39	22	13	30	35	39	30	22	9	13		22
average	30	29	22	35	48	30	29	60	21	22	26	19	32	31	26	26	44	26	16	32	37	40	33	25	16	26	35	

Table 9: For each pair of locks *(rowA, colB)* at the optimized number of nodes, score of lock A vs lock B: percentage of applications for which lock A performs at least 5% better than B (**A-64 machine**).

	Better			Worse		
Lock	A-64	A-48	I-48	A-64	A-48	I-48
ahmcs	32%	37%	51%	30%	28%	27%
alock-ls	28%	39%	38%	29%	26%	32%
backoff	33%	48%	42%	22%	13%	22%
cbomcs_spin	25%	40%	44%	35%	25%	24%
cbomcs_stp	17%	23%	32%	48%	48%	38%
clh-ls	31%	42%	27%	30%	34%	38%
clh_spin	33%	37%	32%	29%	36%	36%
clh_stp	18%	11%	9%	60%	73%	70%
c-ptl-tkt	38%	47%	48%	21%	18%	14%
c-tkt-tkt	36%	44%	51%	22%	26%	16%
hmcs	37%	48%	53%	26%	20%	17%
hticket-ls	35%	40%	40%	19%	25%	21%
malth_spin	26%	36%	30%	32%	39%	35%
malth_stp	27%	20%	28%	31%	54%	36%
mcs-ls	28%	39%	35%	26%	20%	25%
mcs_spin	40%	39%	37%	26%	34%	23%
mcs_stp	23%	23%	21%	44%	58%	52%
mcs-timepub	33%	38%	34%	26%	36%	30%
partitioned	41%	40%	40%	16%	31%	23%
pthread	29%	33%	35%	32%	43%	34%
pthreadadapt	30%	34%	33%	37%	38%	37%
spinlock	31%	34%	22%	40%	45%	51%
spinlock-ls	26%	29%	27%	33%	33%	44%
ticket	30%	24%	16%	25%	46%	55%
ticket-ls	35%	35%	25%	16%	24%	38%
ttas	30%	38%	35%	26%	27%	29%
ttas-ls	22%	25%	27%	35%	41%	42%

Table 10: For each lock, at the optimized number of nodes, mean of the fraction of applications for which the lock is better (resp. worse) than other locks (**all machines**).

never yields better performance than *B*. The results at max nodes (Table 24 in the Appendices) exhibit similar trends as the ones at opt nodes. Besides, we make the same observations (both at opt nodes and max nodes) on the A-48 and I-48 machines (Tables 25, 26, 27 and 28 in the Appendices).

5.2.4 Are all locks potentially harmful?

Our goal is to determine, for each lock, if there are applications for which it yields substantially lower performance than other locks and to quantify the magnitude of such performance gaps. Table 11 displays, for the A-64 machine, the performance gain brought by the best lock with respect to each of the other locks for each application at max nodes (top part) and at the optimized number of nodes for each lock (bottom part). For example, the top part of the table shows that for the dedup application, the best lock (0%, here Spinlock-LS) is 579% better than the Alock-LS lock. The gray cells highlight values greater than 15%. Thus, for each lock in a column, the number of grey cells corresponds to the number of applications for which the lock is beaten by a gap of 15% or more by the best lock(s) for this application. In addition, Table 12 displays, for each machine, the fraction of applications that are significantly hurt by a given lock.

On the three machines, we observe that, **both at max**

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	
dedup	-	579	0	135	132	944	562	564	23	8	141	123	126	125	119	123	124	101	11	4	0	0	0	2	2	0	564	
facesim	298	687	349	130	146	722	687	52	202	241	234	151	531	40	771	710	52	0	685	56	44	572	507	719	333	87	161	
ferret	329	234	7	41	0	232	312	0	188	222	255	286	196	0	349	317	0	4	314	0	1	10	9	331	76	7	6	
fluidanimate	-	276	2	54	83	-	-	-	29	14	63	-	29	84	42	33	83	42	8	0	6	19	7	11	8	2	182	
fmm	41	38	27	30	28	37	39	33	28	31	35	31	16	14	32	2	0	0	14	25	23	2	28	15	28	28	31	
histogram	1	3	1	0	19	8	3	11	9	2	1	2	0	0	4	0	13	0	4	19	1	18	14	10	4	14	7	
linear_regression	31	130	3	93	312	68	30	61	17	7	27	110	16	0	48	45	55	2	38	14	0	82	45	31	8	46	33	
matrix_multiply	9	6	1	7	2	9	9	3	17	12	608	4	5	3	639	27	2	0	33	3	3	5	5	3	4	3	3	
mysqld	-	-	-	-	99	-	-	-	-	-	-	-	-	0	-	-	7	173	-	97	102	-	-	-	-	-	-	
ocean_cp	31	26	39	44	74	12	38	38	17	18	29	11	23	27	27	43	32	0	24	11	19	129	122	55	10	13	11	
ocean_ncp	26	9	19	23	52	19	25	27	9	8	15	0	18	20	13	34	35	10	28	30	26	116	90	24	1	5	9	
pca	71	53	0	27	338	71	54	232	44	52	66	43	43	3	48	64	226	28	51	118	44	265	123	118	27	224	135	
pca_ll	67	48	0	8	675	46	72	597	57	58	61	13	35	55	34	70	614	31	45	249	91	679	367	160	36	412	253	
radiosity	13	12	9	9	73	13	9	0	8	9	7	8	8	12	9	0	90	0	0	0	0	1	72	0	18	74	51	
radiosity_ll	0	46	48	25	1k	49	16	2k	40	50	3	23	59	63	62	12	2k	44	76	567	267	2k	1k	614	212	1k	802	
s_raytrace	2	13	13	68	2k	44	75	1k	35	27	18	25	26	64	16	0	1k	13	122	230	122	714	462	412	96	726	565	
s_raytrace_ll	6	87	22	59	2k	96	87	3k	129	138	0	96	84	291	99	69	3k	111	157	639	335	2k	1k	813	258	2k	1k	
ssl_proxy	0	16	0	15	718	16	17	881	13	18	380	8	16	36	43	48	902	29	36	294	154	1k	858	271	94	662	400	
streamcluster	52	19	112	82	1k	-	-	-	0	9	8	-	225	1k	197	128	1k	12	4	99	144	534	449	309	195	232	121	
streamcluster_ll	77	27	159	98	3k	-	-	-	0	21	17	-	268	1k	232	156	1k	47	24	94	122	580	510	299	191	303	101	
vips	41	40	8	102	11	-	-	-	40	35	101	-	177	0	28	28	1	3	37	0	2	3	12	16	12	12	18	
volrend	4	28	33	38	117	19	29	62	0	6	2	1	20	67	25	30	51	27	28	82	109	168	119	28	17	33	34	
water_nsquared	94	49	6	12	11	59	35	35	9	6	14	12	7	6	9	3	2	7	4	6	7	0	6	4	5	6	36	
water_spatial	98	47	0	5	6	64	40	39	2	1	8	5	8	5	5	9	9	10	1	0	0	2	0	1	0	0	40	
dedup	983	378	9	205	206	680	443	436	43	24	237	183	153	152	161	160	158	174	16	16	9	0	9	3	9	9	445	
facesim	1	3	4	5	7	2	3	11	0	0	3	0	1	7	2	1	6	3	3	6	13	6	8	5	2	2	2	
ferret	88	51	6	35	0	57	53	0	89	87	82	89	93	0	56	46	0	3	55	0	0	7	7	56	40	5	5	
fluidanimate	172	128	0	47	46	-	-	-	33	16	59	-	24	23	34	21	22	36	11	2	8	6	0	1	1	0	76	
fmm	41	36	27	30	28	37	21	19	28	31	33	31	16	14	32	2	0	0	14	25	23	1	28	15	28	28	31	
histogram	0	4	9	10	15	9	3	11	11	10	1	7	1	3	6	4	4	5	6	2	3	9	19	3	9	15	5	
linear_regression	2	20	10	10	11	5	1	34	13	12	0	5	4	4	10	11	39	14	4	16	4	48	35	22	11	26	20	
matrix_multiply	9	6	1	7	2	9	9	3	17	12	83	4	5	3	357	23	2	0	24	3	3	5	5	3	4	3	3	
mysqld	-	-	-	-	69	-	-	-	-	-	-	-	-	0	-	-	8	121	-	96	96	-	-	-	-	-	-	
ocean_cp	5	5	3	6	14	1	1	3	5	6	9	5	8	20	0	10	19	13	1	6	14	13	3	8	4	5	7	
ocean_ncp	8	5	12	7	6	7	8	18	5	0	5	5	7	8	8	16	15	14	7	9	13	17	6	9	6	10	5	
pca	2	1	4	10	34	3	12	41	3	1	4	2	10	6	4	12	46	13	6	16	12	16	0	6	0	0	2	
pca_ll	15	13	26	34	96	0	61	118	5	13	12	15	66	69	12	59	87	65	58	56	18	67	16	65	16	16	14	
radiosity	9	9	8	8	10	9	8	0	6	7	6	8	6	9	8	0	13	0	0	0	0	1	12	0	8	9	10	
radiosity_ll	0	30	38	25	89	33	5	180	10	29	3	23	49	59	42	1	165	22	19	159	114	120	90	80	44	58	59	
s_raytrace	2	7	13	35	95	10	5	123	8	12	5	15	26	53	14	0	117	12	10	75	94	120	124	119	29	116	99	
s_raytrace_ll	2	6	22	37	83	7	4	157	8	11	0	10	25	72	9	3	150	11	6	79	74	75	79	75	8	76	80	
ssl_proxy	3	2	4	23	26	5	7	31	0	2	0	0	27	32	9	10	24	11	8	58	28	21	17	20	7	15	19	
streamcluster	9	8	8	8	8	-	-	-	5	5	5	-	8	8	7	0	0	3	5	10	5	0	4	6	5	5	4	
streamcluster_ll	13	13	27	13	26	-	-	-	0	19	11	-	34	28	26	24	21	23	10	18	34	27	18	16	17	10	9	
vips	4	5	3	1	0	-	-	-	5	3	6	-	2	3	6	3	3	3	4	0	2	4	3	2	2	2	4	
volrend	2	5	11	9	22	3	4	8	1	2	0	1	5	8	4	3	7	4	3	17	18	23	15	8	3	9	7	
water_nsquared	94	49	6	12	11	59	35	35	9	6	14	12	7	6	9	3	2	7	4	6	7	0	6	4	5	6	36	
water_spatial	95	47	0	5	6	64	40	39	2	1	8	5	8	5	5	9	9	10	1	0	0	2	0	1	0	0	40	

Max nodes

Opt nodes

Table 11: For each application, at max nodes (top part) and at the optimized number of nodes (bottom part), performance gain (in %) obtained by the best lock(s) with respect to each of the other locks. The grey background highlights cells for which the performance gains are greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**A-64 machine**).

nodes and at the optimal number of nodes, all locks are potentially harmful, yielding sub-optimal performance for a significant number of applications (Table 12). We also notice that locks are significantly less harmful at the optimized number of nodes than at max nodes. This is explained by the fact that several of the locks create performance collapses at max nodes, which does not occur at the optimized number of nodes. Moreover, we observe that, for each lock, the performance gap to the best lock can be significant (Tables 11, 31 and 32).

5.3 Additional observations

Impact of the number of nodes. Table 13 shows, for each application on the A-64 machine, the number of pairwise changes in the lock performance hierarchy when the number of nodes is modified. For example, in the case of the facesim application, there are 19% of the pairwise performance comparisons between locks that change when moving from a 1-node configuration to a 2-node configuration. Similarly, there are 96% of pairwise comparisons that change at least once when considering

	A-64		A-48		I-48	
Lock	Max	Opt	Max	Opt	Max	Opt
ahmcs	62%	30%	56%	39%	42%	37%
alock-ls	78%	35%	67%	39%	68%	58%
backoff	39%	22%	32%	26%	42%	37%
cbomcs_spin	74%	39%	74%	58%	42%	37%
cbomcs_stp	79%	54%	80%	70%	70%	55%
clh-ls	79%	32%	73%	47%	75%	69%
clh_spin	84%	32%	60%	53%	62%	56%
clh_stp	79%	63%	87%	87%	81%	75%
c-ptl-tkt	61%	22%	47%	26%	47%	37%
c-tkt-tkt	52%	26%	74%	42%	47%	32%
hmcs	65%	22%	53%	32%	37%	21%
hticket-ls	47%	32%	62%	38%	44%	50%
malth_spin	78%	43%	68%	53%	53%	53%
malth_stp	54%	38%	65%	60%	55%	55%
mcs-ls	78%	30%	84%	47%	63%	58%
mcs_spin	70%	30%	63%	53%	63%	58%
mcs_stp	67%	50%	70%	65%	65%	60%
mcs-timepub	42%	25%	65%	55%	50%	50%
partitioned	65%	22%	74%	53%	63%	47%
pthread	58%	50%	60%	60%	60%	50%
pthreadadapt	58%	38%	55%	55%	55%	50%
spinlock	65%	43%	68%	58%	63%	53%
spinlock-ls	65%	43%	63%	53%	58%	47%
ticket	70%	39%	79%	63%	74%	68%
ticket-ls	57%	26%	58%	42%	68%	47%
ttas	57%	35%	68%	53%	58%	42%
ttas-ls	78%	43%	72%	61%	68%	47%

Table 12: For each lock, at max nodes and at the optimized number of nodes, fraction of the applications for which the lock is harmful (**all machines**).

the 1-node, 2-node, 4-node and 8-node configurations.

We observe that, **for all applications, the lock performance hierarchy changes significantly according to the chosen number of nodes**. Moreover, we observe the same trends on the A-48 and I-48 machines (Tables 34 and 35 in the Appendices).

	% of pairwise changes between configurations			
Applications	1/2	2/4	4/8	1/2/4/8
dedup	12%	5%	11%	17%
facesim	19%	48%	81%	96%
ferret	0%	73%	27%	88%
fluidanimate	4%	6%	22%	25%
fmm	28%	6%	18%	40%
histogram	36%	44%	31%	71%
linear_regression	54%	40%	54%	91%
matrix_multiply	8%	14%	40%	45%
mysqld	33%	27%	7%	40%
ocean_cp	47%	50%	64%	91%
ocean_ncp	55%	51%	51%	85%
pca	42%	53%	30%	86%
pca_ll	32%	39%	27%	77%
radiosity	11%	41%	53%	76%
radiosity_ll	60%	32%	16%	89%
s_raytrace	2%	71%	32%	97%
s_raytrace_ll	16%	64%	30%	98%
ssl_proxy	65%	14%	24%	80%
streamcluster	70%	32%	33%	98%
streamcluster_ll	60%	37%	32%	90%
vips	3%	6%	83%	84%
volrend	22%	23%	39%	81%
water_nsquared	24%	22%	18%	56%
water_spatial	7%	8%	12%	26%

Table 13: For each application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**A-64 machine**).

Impact of the machine. Table 14 shows the number of pairwise lock inversions observed between the machines (both at max nodes and at the optimized number of nodes). More precisely, for a given application at a given node configuration, we check whether two locks are in the same order or not on the target machines.

We observe that **the lock performance hierarchy changes significantly according to the chosen machine**. Interestingly, we observe that there is approximately the same number of inversions between each pair of machines.

	A-64 vs. A-48	A-48 vs. I-48	A-64 vs. I-48
# nodes			
Max	35%	36%	38%
Opt	28%	29%	29%

Table 14: For each pair of machines, at max nodes and at opt nodes, percentage of pairwise changes in the lock performance hierarchy (**all machines**).

A note on Phtread locks. The various results presented in this paper show that the current Linux **Pthread locks perform well (i.e., are among the best locks) for a significant share of the studied applications**, thus providing a different insight than recent results, which were mostly based on synthetic workloads [9]. Beyond the changes of workloads, these differences may also be explained by the continuous refinement of the Linux Pthread implementation. It is nevertheless important to note that on each machine, some locks stand out as the best ones for a higher fraction of the applications than Pthread locks. Finally, we note that Pthread adaptive locks perform slightly better than standard Pthread locks.

Impact of thread pinning. As explained in §3.2, all the above-described experiments were run without any restriction on the placement of threads, leaving the corresponding decisions to the Linux scheduler. However, in order to better control CPU allocation and improve locality, some developers and system administrators use pinning to explicitly restrict the placement of each thread to one or several core(s). The impact of thread pinning may vary greatly according to workloads and can yield both positive and negative effects [9, 27]. In order to assess the generality of our observations, we also performed the complete set of experiments with an alternative configuration in which each thread is pinned to a given node, leaving the scheduler free to place the thread among the cores of the node. Note that for an experiment with a N-node configuration, the complete application runs on exactly first N nodes of the machine. We chose thread-to-node pinning rather than thread-to-core pinning because we observed that the former generally provided better

performance for our studied applications, especially the ones using more threads than cores. The detailed results of our experiments with thread-to-node pinning are available in the companion technical report [18]. Overall, we observe that **all the conclusions presented in the paper still hold with per-node thread pinning**.

6 Related work

The design and implementation of the LiTL lock library borrows code and ideas from previous open-source toolkits that provide application developers with a set of optimized implementations for some of the most-established lock algorithms: Concurrency Kit [1], liblock [25, 24, 26], and libslock [9]. All of these toolkits require potentially tedious source code modifications in the target applications, even in the case of algorithms that have been specifically designed to lower this burden [3, 33, 36]. Moreover, among the above works, none of them provides a simple and generic solution for supporting Pthread condition variables. The authors of liblock [26] have proposed an approach but we discovered that it suffers from liveness hazards due to a race condition. Indeed, when a thread T calls `pthread_cond_wait()`, it is not guaranteed that the two steps (releasing the lock and blocking the thread) are always executed atomically. Thus, a wake-up notification issued by another thread may get interleaved between the two steps and T may remain indefinitely blocked.

Several research works have leveraged library interposition to compare different locking algorithms on legacy applications (e.g., Johnson et al. [21] and Dice et al. [14]) but, to the best of our knowledge, they have not publicly documented the design challenges to support arbitrary application patterns, nor disclosed the corresponding source code and the overhead of their interposition library has not been discussed.

Several studies have compared the performance of different multicore lock algorithms, either from a theoretical angle or based on experimental results [4, 33, 9, 24, 14]. In comparison, our study encompasses significantly more lock algorithms and waiting policies. Moreover, the bulk of these studies is mainly focused on characterization microbenchmarks while we focus instead on workloads designed to mimic real applications. Two noticeable exceptions are the work from Boyd-Wickizer et al. [4] and Lozi et al. [26] but they do not consider the same context as our study. The former is focused on kernel-level locking bottlenecks, and the latter is focused on applications in which only one or a few heavily contended critical sections have been optimized (after a profiling phase). For all these reasons, we make observations that are significantly different from the ones based on all the above-mentioned studies. Other synchronization-related studies like the one from Gramoli [16] have a different scope and focus on concurrent data structures, possibly based on other facilities than locks.

Finally, some tools have been proposed to facilitate the identification of locking bottlenecks in applications [35, 8, 26]. These publications are orthogonal to our work. We note that, among them, the profilers based on library interposition can be stacked on top of LiTL.

7 Conclusion and future work

Optimized lock algorithms for multicore machines are abundant. However, there are currently no clear guidelines and methodologies helping developers to select the right lock for their workloads. In this paper, we have presented a broad study of 27 locks algorithms with 35 applications on Linux/x86. To perform that study, we have implemented LiTL, an interposition library allowing the transparent replacement of lock algorithms used for Pthread mutex locks. From our study, we draw several conclusions, including the following ones: at its optimized contention level, no single lock dominates for more than 48% of the lock-sensitive applications; any of the locks is harmful for at least several applications; for a given application, the best lock varies according to both the number of contending cores and the machine that executes the application. These observations call for further research on optimized lock algorithms, as well as tools and dynamic approaches to better understand and control their behavior.

The source code of LiTL and the data sets of our experimental results are available online [17].

Acknowledgments

We thank the anonymous reviewers and our shepherd, Tim Harris, for their insightful comments on ealier drafts of this paper. Dave Dice provided detailed answers for our questions on Malthusian locks. Baptiste Lepers provided valuable insights for some of the case studies. Pierre Neyron provided his help to set up experiments on the I-48 machine. Finally, this work has been partially supported by: LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01), EmSoc Replicanos and AGIR CAEC projects of Université Grenoble-Alpes and GrenobleINP, and the INRIA/LIG Digitalis project.

References

[1] AL BAHRA, S. Concurrency Kit, 2015. `http://concurrencykit.org`.

[2] ANDERSON, T. E. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transaction on Parallel and Distributed Systems* (Jan. 1990), 6–16.

[3] AUSLANDER, M., EDELSOHN, D., KRIEGER, O., ROSENBURG, B., AND WISNIEWSKI, R. Enhancement to the MCS Lock for Increased Functionality and Improved Programmability. U.S. Patent Application Number 20030200457 (abandoned), October 2003.

[4] BOYD-WICKIZER, S., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. Non-scalable Locks are Dangerous. In *Proceedings of the Linux Symposium* (Ottawa, Canada, July 2012).

[5] CHABBI, M., FAGAN, M., AND MELLOR-CRUMMEY, J. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)* (2015), ACM.

[6] CHABBI, M., AND MELLOR-CRUMMEY, J. Contention-conscious, Locality-preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)* (2016), ACM.

[7] CRAIG, T. S. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Tech. Rep. TR 93-02-02, University of Washington, 1993.

[8] DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Continuously Measuring Critical Section Pressure with the Free-lunch Profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (2014), OOPSLA '14, ACM.

[9] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)* (2013), ACM.

[10] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)* (2015), ACM.

[11] DICE, D. Brief Announcement: A Partitioned Ticket Lock. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)* (2011), ACM.

[12] DICE, D. Malthusian Locks, november 2015. `http://arxiv.org/abs/1511.06035`.

[13] DICE, D., MARATHE, V. J., AND SHAVIT, N. Flat-Combining NUMA Locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)* (2011), ACM.

[14] DICE, D., MARATHE, V. J., AND SHAVIT, N. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Transactions on Parallel Computing 1*, 2 (Feb. 2015), 13:1–13:42.

[15] FATOUROU, P., AND KALLIMANIS, N. D. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)* (2012), ACM.

[16] GRAMOLI, V. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)* (2015), ACM.

[17] GUIROUX, H., LACHAIZE, R., AND QUÉMA, V. LiTL source code and data sets, 2016. `https://github.com/multicore-locks`.

[18] GUIROUX, H., LACHAIZE, R., AND QUÉMA, V. Multicore Locks: the Case is not Closed Yet. Technical report, 2016. Available from `https://github.com/multicore-locks`.

[19] HE, B., SCHERER, W. N., AND SCOTT, M. L. Preemption Adaptivity in Time-published Queue-based Spin Locks. In *Proceedings of the 12th International Conference on High Performance Computing (HiPC'05)* (2005), Springer-Verlag.

[20] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)* (2010), ACM.

[21] JOHNSON, F. R., STOICA, R., AILAMAKI, A., AND MOWRY, T. C. Decoupling Contention Management from Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)* (2010), ACM.

[22] KARLIN, A. R., LI, K., MANASSE, M. S., AND OWICKI, S. Empirical Studies of Competitve Spinning for a Shared-memory Multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP'91)* (1991), ACM.

[23] KYLHEKU, K. What is PTHREAD_MUTEX_ADAPTIVE_NP?, 2014. `http://stackoverflow.com/a/25168942`.

[24] LOZI, J.-P. *Towards More Scalable Mutual Exclusion for Multicore Architectures*. PhD thesis, UPMC, Paris, July 2014. `http://www.i3s.unice.fr/~jplozi/documents/lozi-phd-thesis.pdf`.

[25] LOZI, J.-P., DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), USENIX Association.

[26] LOZI, J.-P., DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Fast and Portable Locking for Multicore Architectures. *ACM Transactions on Computer Systems 33*, 4 (Jan. 2016), 13:1–13:62.

[27] LOZI, J.-P., LEPERS, B., FUNSTON, J., GAUD, F., QUÉMA, V., AND FEDOROVA, A. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)* (2016), ACM.

[28] LUCHANGCO, V., NUSSBAUM, D., AND SHAVIT, N. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing (Euro-Par'06)* (2006), Springer-Verlag.

[29] MAGNUSSON, P. S., LANDIN, A., AND HAGERSTEN, E. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing* (1994), IEEE Computer Society.

[30] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (Feb. 1991), 21–65.

[31] OYAMA, Y., TAURA, K., AND YONEZAWA, A. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing For Symbolic And Irregular Applications (PDSIA'99)* (1999), World Scientific.

[32] RADOVIC, Z., AND HAGERSTEN, E. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)* (2003), IEEE Computer Society.

[33] SCOTT, M. L. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013.

[34] SCOTT, M. L., AND SCHERER, W. N. Scalable Queue-based Spin Locks with Timeout. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP'01)* (2001), ACM.

[35] TALLENT, N. R., MELLOR-CRUMMEY, J. M., AND PORTERFIELD, A. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)* (2010), ACM.

[36] WANG, T., CHABBI, M., AND KIMURA, H. Be My Guest — MCS Lock Now Welcomes Guests. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)* (2016), ACM.

A Selection of the lock-sensitive applications

	Gain 1 node	R.Dev. 1 node	Gain max nodes	R.Dev. max nodes	Gain opt nodes	R.Dev. opt nodes
barnes	7%	2%	18%	5%	18%	5%
blackscholes	5%	1%	5%	1%	5%	1%
bodytrack	2%	1%	26%	6%	19%	4%
canneal	7%	1%	9%	2%	8%	2%
dedup	**156%**	**38%**	**224%**	**46%**	**156%**	**38%**
ferret	**1%**	**0%**	**490%**	**72%**	**137%**	**29%**
fmm	**16%**	**4%**	**53%**	**13%**	**48%**	**12%**
freqmine	12%	2%	5%	1%	5%	1%
histogram	**22%**	**5%**	**54%**	**11%**	**46%**	**9%**
kmeans	4%	1%	14%	3%	14%	3%
linear_regression	**38%**	**7%**	**266%**	**24%**	**109%**	**15%**
lu_cb	4%	1%	3%	1%	3%	1%
lu_ncb	19%	4%	37%	9%	33%	8%
matrix_multiply	8%	2%	30%	6%	29%	5%
mysqld	**62%**	**18%**	**54%**	**16%**	**53%**	**16%**
pca	**44%**	**7%**	**298%**	**30%**	**120%**	**16%**
pca_ll	**273%**	**16%**	**819%**	**49%**	**778%**	**33%**
p_raytrace	3%	0%	3%	0%	3%	0%
radiosity	**33%**	**9%**	**828%**	**34%**	**42%**	**9%**
radiosity_ll	**53%**	**7%**	**3064%**	**70%**	**349%**	**31%**
s_raytrace	**4%**	**1%**	**1448%**	**57%**	**344%**	**30%**
s_raytrace_ll	**6%**	**1%**	**3192%**	**70%**	**383%**	**39%**
ssl_proxy	**876%**	**31%**	**1485%**	**63%**	**1319%**	**35%**
streamcluster	**1342%**	**29%**	**2075%**	**57%**	**955%**	**28%**
streamcluster_ll	**16%**	**3%**	**2285%**	**58%**	**45%**	**11%**
string_match	6%	1%	18%	5%	18%	5%
swaptions	1%	0%	6%	1%	6%	1%
vips	**2%**	**0%**	**820%**	**40%**	**19%**	**6%**
volrend	**9%**	**2%**	**176%**	**27%**	**33%**	**7%**
water_nsquared	**13%**	**3%**	**79%**	**11%**	**79%**	**11%**
water_spatial	**18%**	**4%**	**70%**	**13%**	**70%**	**13%**
word_count	8%	2%	36%	9%	25%	7%
x264	3%	1%	5%	1%	4%	1%

Table 15: For each application, performance gain of the best vs. worst lock and relative standard deviation (**A-48 machine**).

	Gain 1 node	R.Dev. 1 node	Gain max nodes	R.Dev. max nodes	Gain opt nodes	R.Dev. opt nodes
barnes	4%	1%	16%	4%	16%	4%
blackscholes	0%	0%	1%	0%	1%	0%
bodytrack	3%	1%	5%	1%	5%	1%
canneal	1%	0%	1%	0%	1%	0%
dedup	**839%**	**59%**	**1126%**	**63%**	**839%**	**59%**
ferret	**0%**	**0%**	**689%**	**80%**	**81%**	**19%**
fmm	6%	1%	21%	5%	19%	4%
freqmine	20%	4%	2%	0%	2%	0%
histogram	16%	4%	22%	6%	16%	4%
kmeans	6%	2%	41%	9%	41%	9%
linear_regression	**11%**	**3%**	**124%**	**23%**	**90%**	**14%**
lu_cb	0%	0%	2%	1%	2%	1%
lu_ncb	7%	2%	31%	7%	31%	7%
matrix_multiply	7%	2%	9%	2%	9%	2%
mysqld	**166%**	**34%**	**115%**	**22%**	**166%**	**32%**
pca	**265%**	**20%**	**293%**	**30%**	**266%**	**20%**
pca_ll	**611%**	**26%**	**1193%**	**50%**	**1025%**	**35%**
p_raytrace	3%	1%	5%	1%	2%	1%
radiosity	**82%**	**9%**	**160%**	**22%**	**91%**	**10%**
radiosity_ll	**989%**	**31%**	**2181%**	**66%**	**1898%**	**48%**
s_raytrace	**3%**	**1%**	**1373%**	**52%**	**203%**	**31%**
s_raytrace_ll	**5%**	**1%**	**2387%**	**65%**	**238%**	**39%**
ssl_proxy	**1651%**	**42%**	**1550%**	**52%**	**1610%**	**46%**
streamcluster	**39%**	**9%**	**628%**	**71%**	**39%**	**9%**
streamcluster_ll	**48%**	**12%**	**886%**	**80%**	**186%**	**33%**
string_match	1%	0%	19%	4%	19%	4%
swaptions	0%	0%	3%	1%	3%	1%
vips	**1%**	**0%**	**1132%**	**50%**	**26%**	**9%**
volrend	**8%**	**2%**	**168%**	**15%**	**44%**	**8%**
water_nsquared	**13%**	**3%**	**93%**	**15%**	**93%**	**15%**
water_spatial	**24%**	**5%**	**98%**	**16%**	**91%**	**16%**
word_count	3%	1%	10%	2%	3%	1%
x264	1%	0%	3%	0%	2%	0%

Table 16: For each application, performance gain of the best vs. worst lock and relative standard deviation (**I-48 machine**).

	Gain 1 node	R.Dev. 1 node	Gain max nodes	R.Dev. max nodes	Gain opt nodes	R.Dev. opt nodes
barnes	3%	1%	23%	5%	23%	5%
blackscholes	1%	0%	2%	0%	2%	0%
bodytrack	0%	0%	11%	3%	5%	2%
canneal	2%	0%	4%	1%	4%	1%
dedup	**522%**	**49%**	**946%**	**55%**	**522%**	**52%**
facesim	**1%**	**0%**	**301%**	**27%**	**21%**	**6%**
ferret	**8%**	**3%**	**389%**	**66%**	**356%**	**64%**
fft	8%	2%	10%	2%	10%	2%
fluidanimate	**42%**	**10%**	**302%**	**26%**	**183%**	**23%**
fmm	4%	1%	12%	3%	12%	3%
freqmine	4%	1%	3%	1%	3%	1%
histogram	8%	2%	20%	5%	19%	4%
kmeans	7%	2%	5%	1%	5%	1%
linear_regression	**4%**	**1%**	**96%**	**18%**	**73%**	**14%**
lu_cb	0%	0%	4%	1%	4%	1%
lu_ncb	6%	1%	5%	1%	5%	1%
matrix_multiply	4%	2%	5%	1%	5%	1%
mysqld	**125%**	**34%**	**174%**	**38%**	**122%**	**34%**
ocean_cp	**4%**	**1%**	**135%**	**21%**	**15%**	**4%**
ocean_ncp	**4%**	**1%**	**115%**	**17%**	**11%**	**3%**
pca	**2%**	**1%**	**350%**	**33%**	**64%**	**10%**
pca_ll	**2%**	**1%**	**843%**	**49%**	**191%**	**25%**
p_raytrace	1%	0%	2%	0%	1%	0%
radiosity	**3%**	**1%**	**115%**	**19%**	**7%**	**2%**
radiosity_ll	**10%**	**2%**	**2261%**	**65%**	**267%**	**28%**
radix	0%	0%	15%	3%	15%	3%
s_raytrace	**4%**	**1%**	**1219%**	**57%**	**211%**	**28%**
s_raytrace_ll	**2%**	**0%**	**2894%**	**75%**	**150%**	**25%**
ssl_proxy	**29%**	**5%**	**1271%**	**54%**	**68%**	**14%**
streamcluster	**35%**	**6%**	**871%**	**61%**	**45%**	**10%**
streamcluster_ll	**37%**	**7%**	**860%**	**60%**	**89%**	**21%**
string_match	8%	3%	9%	2%	9%	2%
swaptions	0%	0%	1%	0%	1%	0%
vips	**131%**	**23%**	**199%**	**27%**	**196%**	**30%**
volrend	**5%**	**1%**	**110%**	**16%**	**29%**	**6%**
water_nsquared	**7%**	**2%**	**92%**	**16%**	**92%**	**16%**
water_spatial	**16%**	**4%**	**90%**	**16%**	**90%**	**16%**
word_count	2%	1%	7%	2%	3%	1%
x264	0%	0%	1%	0%	1%	0%

Table 17: For each application, performance gain of the best vs. worst lock and relative standard deviation (**A-64 machine with thread-to-node pinning**).

B Selection of the number of nodes

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	-	53	84	77	-	-	-	65	47	72	84	93	83	86	83	84	86	50	48	45	47	53	51	53	53	-
ferret	172	214		237		205	216		161	137	132	160	193		209	192			207					187	152		
fmm																											
histogram	27	38	37	40	51	45	36	48	36	45	39	33	42	42	25	40	46	35	30	41	26	53	33	52	33	39	40
linear_regression	11	95		10	177		10	27	13	74	46	33	43	51	80		15		24	12		26	18			15	13
matrix_multiply																			10								
mysqld	-	-	-	-		-	-	-	-	-	-	-	-		-	-			-			-	-	-	-	-	-
pca	45	48		25	131	63	50	70	25	32	14	19	9	41	19	57	85	28	39	54	29	97	107	65	28	119	83
pca_ll	58	99	8	19	296	69		86	41	77	18	36		83	27	27	31	12		91	42	292	232	74	76	240	155
radiosity					92		18	593				7	12	36			523		17	56	13	172	61	41	11	55	53
radiosity_ll			7		622			438						451		6	605	7	32	245	91	888	364	238	77	357	298
s_raytrace		10		12	688	16		118		11		23	64	126	21		201		7	77	24	226	136	150	42	151	203
s_raytrace_ll					731		8	332	8			39	9	300			622	24	51	156	62	280	200	153	184	259	270
ssl_proxy	39	49	31	35	638	47	85	104	72	37	42	36	48		45	78	46	52	85	247	114	813	537	369	171	448	361
streamcluster	619	722	772	1k	2k	-	-	-	467	601	495	-	2k	7k	2k	2k	12k	1k	797	71	97	1k	1k	2k	1k	1k	1k
streamcluster_ll	113	111	246	315	797	-	-	-	87	144	141	-	459	3k	462	389	3k	351	183	300	389	385	336	403	298	191	158
vips	68	90		204	24	-	-	-	160	106	132	-	701		24	52			64					31			
volrend	30	45	31	49	102	43	30	100	33	36	32	30	26	155	49	89	178	87	32	117	115	162	67	41	39	49	38
water_nsquared																											
water_spatial																											

Table 18: For each *(application, lock)* pair, performance gain (in %) of the optimized configuration over the max-node configuration. The background color of a cell indicates the number of nodes (1, 2, 4, 6, or 8 nodes) for the optimized configuration: 1 | 2 | 4 | 6 | 8. Dashes correspond to untested cases. (**A-48 machine**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	124	123	66	83	78	131	122	120	79	73	68	86	87	82	78	81	80	87	80	71	72	73	72	71	73	71	114
ferret	420	341	6	328		353	353		382	372	400	300	399		331	333			349					314	240		
fmm																											
histogram	61	59	60	53	88	55	45	65	49	58	56	54	56	50	56	65	59	47	67	62	59	70	65	75	53	63	52
linear_regression		78		57	59	19			15	36		16		32	59	30		7				12	15	13			
matrix_multiply																											
mysqld	-	-	-	-	9	-	-	-	-	-	-	-	-	10	-	-	17	15	-	82	72	-	-	-	-	-	-
pca	17	30		11	111	24	25	19	13	11	28	20	14		24	21	23	13	26	19	18	172	197	47	31	50	50
pca_ll					155	7	9	12	56	7		20		29		10	16		22	14	26	370	347	45	29	56	63
radiosity					31			36									136		9			85	65	10		19	20
radiosity_ll					522			12									28		34			374	386	48	30	110	99
s_raytrace					138		11	379				11	8				395		32		10	346	216	31	28	76	91
s_raytrace_ll		9			233	19	23	643				35	7	16	6	7	640	15	57	29	19	576	374	60	63	92	110
ssl_proxy	60	54	33	44	529	40	37	22	58	67	58	39	27	34	52	45	52	54	84	37	25	390	294	34	31	42	73
streamcluster	311	127	1k	627	1k	-	-	-	473	424	383	-	951	1k	710	882	1k	828	370	1k	1k	1k	903	1k	1k	816	777
streamcluster_ll	93		303	166	462	-	-	-	69	94	137	-	154	196	109	143	173	118	77	268	165	168	191	210	182	151	144
vips	84	179		887		-	-	-	239	161	182	-	316		99	69			80					76			
volrend			94	8	12			7		7		6				6			8		11	18	12	7	10	8	8
water_nsquared																											
water_spatial																											

Table 19: For each *(application, lock)* pair, performance gain (in %) of the optimized configuration over the max-node configuration. The background color of a cell indicates the number of nodes (1, 2, 3, or 4 nodes) for the optimized configuration: 1 | 2 | 3 | 4. Dashes correspond to untested cases. (**I-48 machine**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	174	53	41	47	164	166	174	49	46	26	47	45	48	43	43	54	51	50	60	61	55	54	52	55	54	135
facesim	18	20	45	57	155	20	20	60	18	18	18	17	14	56	19	20	61	22	19	42	59	282	240	44	17	44	41
ferret										14	10		9														
fluidanimate	-	42				-	-	-				-															33
fmm																											
histogram	45	40	34	33	34	34	45	54	41	49	33	34	43	44	46	46	54	41	42	44	41	50	45	45	33	50	43
linear_regression		14			23			21									22					13	11	8		9	
matrix_multiply																											
mysqld	-	-	-	-		-	-	-	-	-	-	-	-		-	-		25	-			-	-	-	-	-	-
ocean_cp	22	14	45	45	106	18	19	61	19	18	18	21	21	58	24	17	60	33	19	38	56	147	139	29	17	41	39
ocean_ncp	20		28	31	89	14	14	44	13	14	14	16	13	38	11	11	41	17	12	27	36	114	85	23	14	31	23
pca	29	20	9			24	28	140	26	26	26	27	25		21	27	143	23	28	88	26	279	121	65		208	115
pca_ll	33	14			26	24	20	193	30	21	32		34		40	27	197	28	12	92	16	408	233	74	12	226	125
radiosity								66									68			25	15	106	58	28	11	60	39
radiosity_ll					35		8	485	9	17					9	6	538	14	34	157	114	990	647	301	87	804	430
s_raytrace					12			323									333			83	29	411	269	192	25	441	305
s_raytrace_ll	7	80		41	218	86	74	1k	101	96	7	92	36	137	86	30	1k	92	152	328	153	1k	940	431	239	1k	568
ssl_proxy	67	59	39	29	72	67	79	1k	54	44	55	59	52	57	86	73	1k	72	95	278	199	1k	1k	336	198	789	540
streamcluster	1k	984	2k	1k	2k	-	-	-	638	998	813	-	3k	6k	2k	2k	5k	2k	1k	2k	3k	5k	4k	3k	2k	3k	2k
streamcluster_ll	203	198	465	422	577	-	-	-	234	224	197	-	466	2k	378	371	1k	233	229	445	492	1k	1k	879	620	667	392
vips	23	7	22			-	-	-	22			-		19		24	20	20		21	21	18	18		9	21	23
volrend	14	10	21	23	34	11	11	20	6	7	8	9	14	19	12	13	19	15	12	27	34	74	62	30	17	32	29
water_nsquared																											
water_spatial																											

Table 20: For each *(application, lock)* pair, performance gain (in %) of the optimized configuration over the max-node configuration. The background color of a cell indicates the number of nodes (1, 2, 4, 6, or 8 nodes) for the optimized configuration: 1 | 2 | 4 | 6 | 8. Dashes correspond to untested cases. (**A-64 machine with thread-to-node pinning**).

C Are some locks always among the best?

Locks	Number of nodes		
	1	Max	Opt
ahmcs	67%	28%	33%
alock-ls	61%	11%	33%
backoff	79%	47%	37%
cbomcs_spin	74%	16%	26%
cbomcs_stp	75%	10%	20%
clh-ls	67%	0%	27%
clh_spin	47%	13%	7%
clh_stp	27%	7%	7%
c-ptl-tkt	63%	32%	32%
c-tkt-tkt	79%	11%	32%
hmcs	79%	37%	47%
hticket-ls	69%	19%	25%
malth_spin	74%	5%	0%
malth_stp	30%	10%	10%
mcs-ls	79%	16%	21%
mcs_spin	63%	26%	37%
mcs_stp	50%	10%	10%
mcs-timepub	60%	25%	30%
partitioned	68%	0%	5%
pthread	55%	30%	30%
pthreadadapt	75%	40%	30%
spinlock	68%	21%	26%
spinlock-ls	84%	16%	21%
ticket	68%	0%	5%
ticket-ls	79%	11%	16%
ttas	89%	11%	16%
ttas-ls	72%	6%	11%

Table 21: For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: 1 node, max nodes, and opt nodes (**A-48 machine**).

Locks	Number of nodes		
	1	Max	Opt
ahmcs	63%	42%	47%
alock-ls	74%	21%	26%
backoff	68%	42%	32%
cbomcs_spin	63%	32%	21%
cbomcs_stp	70%	20%	25%
clh-ls	56%	19%	12%
clh_spin	56%	12%	6%
clh_stp	44%	12%	12%
c-ptl-tkt	58%	26%	26%
c-tkt-tkt	74%	32%	37%
hmcs	89%	37%	47%
hticket-ls	62%	31%	31%
malth_spin	68%	16%	16%
malth_stp	50%	40%	30%
mcs-ls	74%	21%	32%
mcs_spin	74%	16%	26%
mcs_stp	50%	20%	20%
mcs-timepub	60%	20%	15%
partitioned	68%	16%	26%
pthread	55%	35%	45%
pthreadadapt	60%	40%	45%
spinlock	47%	21%	21%
spinlock-ls	58%	21%	21%
ticket	53%	21%	21%
ticket-ls	63%	26%	21%
ttas	68%	37%	42%
ttas-ls	47%	16%	16%

Table 22: For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: 1 node, max nodes, and opt nodes (**I-48 machine**).

Locks	Number of nodes		
	1	Max	Opt
ahmcs	71%	29%	48%
alock-ls	74%	35%	30%
backoff	87%	48%	57%
cbomcs_spin	70%	22%	26%
cbomcs_stp	75%	25%	21%
clh-ls	84%	11%	32%
clh_spin	84%	32%	47%
clh_stp	74%	11%	16%
c-ptl-tkt	70%	48%	43%
c-tkt-tkt	70%	43%	52%
hmcs	74%	52%	52%
hticket-ls	89%	58%	53%
malth_spin	74%	43%	48%
malth_stp	75%	38%	42%
mcs-ls	74%	30%	39%
mcs_spin	74%	43%	57%
mcs_stp	79%	21%	29%
mcs-timepub	75%	21%	42%
partitioned	78%	35%	43%
pthread	83%	25%	29%
pthreadadapt	83%	29%	33%
spinlock	83%	17%	22%
spinlock-ls	91%	22%	30%
ticket	83%	22%	35%
ticket-ls	87%	43%	52%
ttas	91%	30%	57%
ttas-ls	83%	13%	39%

Table 23: For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: 1 node, max nodes, and opt nodes (**A-64 machine with thread-to-node pinning**).

D Is there a clear hierarchy between locks?

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		33	38	48	67	44	33	56	29	19	19	28	43	33	48	52	62	24	38	48	38	67	71	57	38	57	52	44
alock-ls	38		17	30	61	32	26	53	22	17	26	11	26	35	35	39	57	13	30	48	39	52	57	52	30	43	39	36
backoff	48	61		52	74	74	74	74	52	43	52	47	65	43	61	61	70	30	61	43	39	65	65	65	48	48	61	57
cbomcs_spin	33	52	17		70	58	47	58	35	30	30	26	39	35	43	43	57	17	39	43	39	57	61	52	43	43	61	44
cbomcs_stp	33	35	9	9		37	37	32	17	17	22	5	13	0	17	17	17	4	17	0	0	30	22	9	9	9	30	17
clh-ls	33	16	11	21	63		32	47	16	16	26	11	11	37	37	37	53	16	47	47	42	58	53	47	21	47	42	34
clh_spin	17	21	5	16	63	37		47	21	11	16	11	5	21	37	26	58	16	32	42	32	58	63	42	26	47	37	31
clh_stp	39	32	16	21	53	32	21		21	21	21	21	16	5	21	21	21	0	16	11	5	58	32	16	16	16	16	22
c-ptl-tkt	62	57	30	43	70	63	68	68		30	70	16	43	48	52	61	70	30	52	48	39	61	61	61	39	48	65	52
c-tkt-tkt	57	48	26	48	74	53	63	74	22		52	21	48	48	52	65	74	26	48	52	39	61	65	65	39	52	65	51
hmcs	43	48	30	39	65	58	63	74	17	13		26	35	30	35	48	61	17	39	43	35	61	65	57	35	52	61	44
hticket-ls	50	63	26	32	63	63	68	68	42	47	42		37	42	47	47	58	32	63	47	42	58	63	63	42	53	68	51
malth_spin	43	48	13	26	70	63	68	79	26	26	43	21		35	35	43	65	17	52	43	35	61	65	61	26	52	65	46
malth_stp	48	43	22	35	71	53	63	79	35	30	43	32	22		35	43	50	8	39	54	38	57	70	52	43	57	74	46
mcs-ls	33	30	13	35	70	42	42	74	17	17	26	11	22	35		17	57	4	26	39	30	57	57	48	22	48	52	36
mcs_spin	24	48	17	43	74	53	37	68	35	26	39	37	35	39	52		57	17	22	39	35	57	61	48	35	48	57	42
mcs_stp	33	39	22	30	62	42	37	32	26	22	30	26	17	12	26	22		4	22	17	12	39	30	22	17	17	35	27
mcs-timepub	71	65	35	65	79	79	74	84	57	48	61	47	65	58	61	57	71		61	62	54	70	78	70	57	65	87	65
partitioned	43	35	22	48	83	37	37	74	26	17	48	26	30	35	48	39	74	22		48	35	61	61	43	30	52	61	44
pthread	48	48	30	39	75	42	53	63	35	30	57	37	43	25	52	48	58	17	35		21	65	70	48	30	57	70	46
pthreadadapt	52	52	26	43	75	58	58	74	39	43	61	37	43	29	57	52	71	17	26	38		70	70	61	30	61	70	50
spinlock	29	43	17	35	52	37	37	32	35	30	39	32	35	30	39	30	35	17	26	9	9		17	17	22	17	30	29
spinlock-ls	24	39	0	26	48	37	32	47	22	22	35	16	22	17	26	30	52	13	22	0	0	57		9	4	22	30	25
ticket	29	35	9	35	83	32	32	63	26	22	39	26	22	26	39	30	65	13	17	22	9	61	57		9	39	57	34
ticket-ls	48	61	13	39	78	53	63	74	48	39	61	26	52	39	52	57	74	22	52	43	39	65	65	65		52	65	52
ttas	43	43	13	43	65	47	47	63	26	26	48	32	35	35	39	39	65	13	35	9	13	57	30	26	13		26	36
ttas-ls	43	39	17	22	61	42	32	53	17	22	22	11	30	22	35	35	61	0	26	13	22	61	57	26	17	48		32
average	41	44	19	36	68	49	48	62	29	26	40	25	33	31	42	41	58	16	36	35	28	58	56	45	29	44	53	

Table 24: For each pair of locks *(rowA, colB)* at the maximum number of nodes, score of lock A vs lock B: percentage of applications for which lock A performs at least 5% better than B (**A-64 machine**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		11	22	28	44	13	40	60	22	33	17	33	50	50	11	33	50	50	44	50	61	50	44	56	22	33	44	37
alock-ls	22		17	28	44	27	40	67	22	33	17	33	50	50	22	44	50	44	44	56	50	50	44	61	17	39	44	39
backoff	39	28		42	58	33	47	80	26	37	37	38	63	58	32	47	58	47	47	53	58	58	42	58	47	47	67	48
cbomcs_spin	22	22	16		42	27	40	80	5	16	16	12	58	58	21	47	63	58	42	53	58	58	37	68	26	32	56	40
cbomcs_stp	33	28	11	16		33	33	67	5	11	11	12	21	35	16	26	55	25	16	20	20	26	16	16	16	16	28	23
clh-ls	20	0	27	27	60		33	67	13	33	20	33	53	60	20	40	60	53	47	60	53	60	53	67	20	47	53	42
clh_spin	27	13	20	33	60	33		67	20	47	13	33	40	60	13	20	60	40	27	60	47	53	40	47	33	27	40	37
clh_stp	20	13	0	7	7	33	7		7	13	7	7	13	13	7	7	53	7	13	0	0	13	7	7	7	0	7	11
c-ptl-tkt	33	22	26	42	58	27	53	80		26	26	31	58	68	32	47	68	63	42	58	53	63	47	58	37	47	61	47
c-tkt-tkt	28	28	11	21	53	40	47	80	5		21	19	63	74	26	47	63	53	37	58	58	58	47	63	26	47	67	44
hmcs	33	33	26	32	53	47	47	80	26	37		38	58	63	26	42	63	58	53	63	58	63	47	63	37	37	56	48
hticket-ls	20	20	6	19	56	33	53	80	0	6	19		50	62	12	50	69	50	31	56	56	62	56	50	19	44	60	40
malth_spin	28	22	5	26	63	33	33	87	16	21	21	19		53	16	32	58	32	16	58	58	47	47	32	21	26	56	36
malth_stp	28	28	11	21	20	33	33	47	21	16	16	12	16		16	26	40	20	21	5	5	21	11	21	11	11	22	20
mcs-ls	33	22	5	26	53	33	40	80	11	26	16	25	63	58		42	58	47	32	53	53	53	37	58	26	32	44	39
mcs_spin	28	33	26	32	58	40	27	80	37	42	21	50	42	53	26		53	16	32	47	42	37	37	47	32	32	44	39
mcs_stp	33	39	16	26	25	40	27	27	26	32	26	25	21	20	26	16		5	21	15	20	5	21	21	32	11	22	23
mcs-timepub	33	33	11	32	60	40	33	80	32	37	21	38	47	45	32	11	50		26	45	40	37	32	47	42	32	50	38
partitioned	28	28	16	32	63	33	27	80	26	26	26	44	37	63	21	37	58	42		58	47	53	42	53	37	32	44	40
pthread	28	28	16	26	50	33	40	80	26	21	26	25	26	60	26	32	70	25	32		10	32	32	42	16	26	33	33
pthreadadapt	28	28	16	26	55	33	40	80	26	21	26	19	26	60	21	42	70	35	32	25		42	21	37	16	32	28	34
spinlock	33	39	16	32	42	40	33	67	32	32	32	31	37	63	32	21	63	11	32	42	26		16	37	32	16	28	34
spinlock-ls	28	33	0	21	42	33	33	80	16	21	21	12	26	58	16	42	58	32	21	32	16	42		47	16	0	17	29
ticket	28	17	5	16	37	33	20	80	16	21	21	25	11	47	16	16	58	16	11	37	11	32	11		11	5	17	24
ticket-ls	22	28	5	21	42	33	53	73	16	21	16	19	37	58	16	47	58	42	32	47	37	53	37	47		26	33	35
ttas	33	44	5	26	58	40	33	80	16	26	26	19	37	63	16	47	58	42	42	42	37	53	26	53	21		39	38
ttas-ls	28	33	6	6	39	33	27	60	6	11	6	7	17	61	11	28	56	28	22	39	22	39	11	50	17	0		25
average	28	26	13	25	48	34	36	73	18	26	20	25	39	54	20	34	58	36	31	43	38	45	33	46	24	27	41	

Table 25: For each pair of locks *(rowA, colB)* at the optimized number of nodes, score of lock A vs lock B: percentage of applications for which lock A performs at least 5% better than B (**A-48 machine**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		56	22	44	67	53	47	67	28	50	11	40	56	67	44	50	67	50	56	67	50	67	67	72	56	67	67	53
alock-ls	22		17	39	67	47	40	60	28	33	11	33	56	56	28	44	67	39	44	56	50	61	56	67	39	61	50	45
backoff	50	56		74	74	73	67	80	42	53	47	56	84	74	63	63	74	58	74	63	53	74	63	79	58	63	78	65
cbomcs_spin	28	28	5		63	47	47	80	11	32	11	25	63	63	21	53	74	63	37	53	53	68	63	63	37	47	61	46
cbomcs_stp	28	28	11	16		33	27	40	11	16	11	6	21	25	5	26	60	25	16	0	0	37	5	16	5	5	17	19
clh-ls	20	13	7	40	60		13	67	27	27	13	40	40	60	33	40	67	47	33	60	47	67	60	67	33	60	53	42
clh_spin	20	27	7	27	60	33		67	27	40	13	33	40	60	27	33	67	40	40	60	47	67	60	67	40	60	60	43
clh_stp	20	20	7	7	33	33	13		7	13	7	7	13	20	7	7	60	7	13	0	0	27	7	7	7	7	7	14
c-ptl-tkt	39	50	21	37	68	67	60	80		47	21	38	58	74	37	53	74	58	68	58	53	68	63	63	47	63	78	55
c-tkt-tkt	33	44	16	37	68	40	47	73	21		16	38	47	63	21	47	74	58	53	53	47	63	53	63	42	63	67	48
hmcs	33	61	21	37	63	73	60	80	37	53		38	53	63	37	58	74	58	53	58	53	68	58	63	58	58	72	55
hticket-ls	33	40	0	31	69	53	53	80	19	25	12		38	69	31	44	75	44	44	50	50	69	56	56	50	56	73	47
malth_spin	28	39	5	21	68	40	40	87	26	21	11	19		63	26	37	74	32	32	47	47	58	47	53	42	53	61	41
malth_stp	28	33	11	21	45	33	27	47	21	16	11	12	16		16	26	70	25	16	5	0	32	11	26	11	11	28	23
mcs-ls	39	39	0	16	68	47	47	80	21	42	16	12	58	68		47	74	47	37	47	47	58	47	53	37	42	56	44
mcs_spin	33	39	21	32	68	47	27	80	37	53	26	50	53	63	37		63	11	42	47	42	53	53	47	37	47	56	45
mcs_stp	28	28	16	21	30	33	27	27	21	26	21	19	21	5	21	16		5	21	5	5	5	16	16	21	16	22	19
mcs-timepub	33	50	11	32	75	53	47	80	32	37	26	44	53	60	47	37	60		47	50	50	58	47	68	53	47	61	48
partitioned	28	33	11	42	74	47	33	87	26	32	26	31	37	68	42	42	74	37		58	53	68	68	63	53	63	61	48
pthread	28	33	16	26	75	33	40	87	26	26	32	31	42	70	37	37	85	30	26		10	74	58	58	16	47	50	42
pthreadadapt	39	39	16	32	80	47	47	87	32	26	32	25	42	80	26	37	85	35	42	45		68	58	68	26	68	72	48
spinlock	28	33	11	26	53	33	27	53	26	26	26	25	42	53	37	26	74	11	26	16	21		11	37	26	11	22	30
spinlock-ls	28	28	0	21	68	33	27	80	16	26	21	12	37	53	32	42	74	26	26	11	16	74		32	16	5	28	32
ticket	22	22	5	16	68	20	20	80	21	21	21	19	26	63	16	26	74	16	16	21	11	63	42		5	32	39	30
ticket-ls	39	44	0	32	74	47	40	80	21	26	21	19	53	74	37	58	74	42	37	47	47	68	53	74		53	56	47
ttas	28	33	5	26	68	33	27	80	16	26	21	12	42	58	32	42	74	32	26	21	16	74	37	32	21		39	35
ttas-ls	28	28	6	28	67	33	13	67	11	22	17	13	33	56	28	33	67	22	17	28	17	72	44	39	17	39		32
average	30	36	10	30	64	44	37	72	23	31	19	27	43	59	30	39	71	35	36	39	34	60	46	52	33	44	51	

Table 26: For each pair of locks *(rowA, colB)* at the maximum number of nodes, score of lock A vs lock B: percentage of applications for which lock A performs at least 5% better than B (**A-48 machine**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		37	47	42	53	44	50	62	42	42	11	56	58	63	47	53	63	53	37	53	53	63	63	68	63	42	53	51
alock-ls	21		32	16	47	50	38	56	5	16	5	19	42	47	16	26	58	32	37	53	47	63	58	63	47	42	42	38
backoff	32	42		42	47	44	44	75	26	21	26	38	53	42	37	42	53	32	37	42	42	47	42	63	53	26	42	42
cbomcs_spin	32	37	42		42	50	56	75	11	16	5	25	47	53	32	32	63	42	32	53	53	63	58	63	53	53	63	44
cbomcs_stp	26	37	26	21		31	38	69	21	11	11	12	37	30	26	26	50	25	21	30	30	47	47	47	37	32	47	32
clh-ls	19	0	19	6	38		6	56	0	6	0	12	31	44	6	0	56	12	19	44	44	62	56	62	38	38	38	27
clh_spin	25	25	19	19	31	25		56	6	19	6	25	38	44	6	12	56	6	19	56	50	56	50	62	38	44	44	32
clh_stp	31	31	12	12	6	31	6		6	12	6	6	6	0	6	6	6	6	6	0	0	6	6	6	6	6	6	9
c-ptl-tkt	26	47	42	21	47	62	56	75		5	16	38	58	58	47	32	68	58	37	47	53	63	58	63	53	47	63	48
c-tkt-tkt	26	53	42	32	58	62	62	81	21		11	44	58	63	42	47	68	63	42	47	58	63	58	58	47	53	63	51
hmcs	37	58	47	37	47	62	62	81	37	37		38	53	63	42	42	63	58	42	53	53	63	63	68	53	53	68	53
hticket-ls	25	44	25	19	56	44	44	81	6	0	12		44	44	31	12	62	31	31	50	50	62	50	56	44	44	62	40
malth_spin	21	21	11	11	37	19	31	81	0	0	11	6		47	11	11	42	26	11	42	47	53	53	58	42	37	42	30
malth_stp	32	32	16	26	30	31	31	69	11	16	26	12	16		26	16	40	20	16	15	25	42	37	63	26	16	32	28
mcs-ls	21	16	21	16	42	50	44	81	0	11	11	19	37	47		5	47	37	16	42	53	53	47	58	47	37	53	35
mcs_spin	21	26	21	21	42	50	50	81	5	5	11	19	37	42	11		47	32	21	42	53	53	47	68	53	37	58	37
mcs_stp	32	32	11	26	20	31	31	44	16	16	26	12	26	5	21	16		25	16	10	15	16	16	32	26	5	26	21
mcs-timepub	32	32	16	26	35	38	44	75	11	16	21	12	32	30	16	11	45		32	40	40	47	42	53	47	32	53	34
partitioned	16	21	21	32	47	25	31	81	11	11	16	25	47	53	21	21	68	26		53	53	63	58	63	53	47	68	40
pthread	32	32	21	32	35	31	38	81	21	26	32	25	32	40	37	37	65	45	26		20	53	32	47	32	16	37	35
pthreadadapt	32	37	11	37	30	31	38	81	26	26	32	19	26	20	32	21	55	40	26	20		47	47	53	26	16	37	33
spinlock	32	32	11	26	21	31	31	56	16	26	32	19	26	5	21	16	32	21	16	11	16		0	26	26	0	16	22
spinlock-ls	32	32	16	26	21	31	31	62	21	21	32	19	37	16	37	32	42	21	21	16	21	26		37	26	0	21	27
ticket	21	21	5	11	26	25	19	69	5	5	11	12	5	5	5	5	47	11	0	16	5	37	21		0	0	16	16
ticket-ls	26	21	11	16	37	19	19	75	11	16	21	12	16	26	21	11	47	11	11	21	26	53	42	53		16	26	25
ttas	32	37	16	37	42	38	38	75	26	26	32	25	32	26	37	37	53	26	16	16	32	58	42	68	32		26	35
ttas-ls	32	32	16	16	37	31	6	50	11	16	11	6	21	26	26	26	47	21	16	26	26	58	47	63	26	11		27
average	27	32	22	24	38	38	36	70	14	16	17	21	35	36	25	23	52	30	23	34	37	51	44	55	38	29	42	

Table 27: For each pair of locks *(rowA, colB)* at the optimized number of nodes, score of lock A vs lock B: percentage of applications for which lock A performs at least 5% better than B (**I-48 machine**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		47	42	47	68	44	50	62	53	42	26	50	58	58	53	58	68	53	53	58	58	63	68	63	63	58	58	55
alock-ls	32		26	32	58	50	25	56	26	21	16	19	53	42	16	21	58	32	42	53	47	58	58	58	53	53	53	41
backoff	32	47		42	63	62	56	75	37	37	37	56	47	37	37	42	58	32	58	42	42	47	53	68	63	42	58	49
cbomcs_spin	42	63	37		63	62	62	75	26	21	16	38	47	47	37	32	53	42	42	53	53	63	58	63	53	53	68	49
cbomcs_stp	32	32	21	16		31	31	62	16	16	16	12	16	5	16	16	40	10	16	5	10	42	42	21	16	5	21	22
clh-ls	25	6	19	12	62		6	62	6	12	6	19	31	44	6	6	62	6	38	50	44	62	62	56	44	50	50	33
clh_spin	31	38	19	12	62	25		62	12	19	12	31	25	44	6	19	62	12	44	62	50	62	62	62	56	56	56	39
clh_stp	31	38	12	12	19	31	6		6	12	12	6	6	0	12	12	6	6	6	0	6	25	12	6	12	6	6	12
c-ptl-tkt	32	58	37	47	68	56	56	81		16	21	25	53	58	47	42	68	42	47	53	53	63	63	63	53	53	68	51
c-tkt-tkt	26	63	37	47	74	62	62	81	32		21	38	63	53	58	47	68	53	42	53	58	63	63	58	53	53	68	54
hmcs	21	63	47	37	63	62	56	81	42	32		38	53	53	42	47	63	47	42	58	53	63	63	63	53	53	68	52
hticket-ls	31	56	25	19	69	44	50	81	12	19	12		31	50	19	19	62	31	44	50	50	62	62	62	50	50	69	44
malth_spin	21	37	21	21	68	38	31	81	11	5	11	12		53	11	11	47	21	37	53	53	58	53	63	53	47	58	37
malth_stp	32	47	21	32	65	38	38	75	16	16	32	19	16		26	21	45	25	26	30	30	58	58	63	47	42	58	37
mcs-ls	26	37	26	26	63	50	56	75	16	16	16	25	42	47		11	58	37	37	53	53	58	58	58	53	47	58	42
mcs_spin	26	37	26	26	68	50	50	81	16	11	21	19	42	47	11		63	37	42	53	53	63	58	58	53	42	58	43
mcs_stp	32	37	21	26	40	31	31	31	16	16	26	12	16	10	21	16		20	16	15	15	16	11	26	26	11	26	22
mcs-timepub	32	47	26	26	65	44	50	81	21	21	21	19	32	50	21	26	60		58	55	50	58	53	74	68	42	58	45
partitioned	26	32	16	37	74	25	19	81	26	32	32	25	32	47	26	26	68	21		47	32	63	63	63	42	53	68	41
pthread	32	37	21	37	65	38	31	81	26	32	32	31	21	25	26	32	55	35	26		20	68	58	68	32	32	53	39
pthreadadapt	32	42	16	37	60	38	31	81	26	32	32	25	26	15	32	26	55	35	26	25		63	58	68	42	42	63	40
spinlock	32	37	16	26	26	31	31	56	21	26	32	19	21	16	26	21	42	11	21	5	11		0	21	26	0	16	23
spinlock-ls	32	37	16	26	37	31	31	69	21	21	32	19	32	16	32	21	47	16	21	11	11	47		26	26	0	21	27
ticket	26	32	11	21	53	25	19	69	16	21	21	12	16	11	16	11	47	11	5	0	5	47	47		5	16	37	23
ticket-ls	26	32	11	26	68	31	25	75	21	21	26	19	16	21	26	21	53	16	21	21	11	53	53	63		37	53	32
ttas	32	42	16	37	68	38	31	75	26	32	32	31	32	21	26	42	63	26	26	11	16	63	63	42	26		26	36
ttas-ls	32	37	16	21	63	38	6	56	16	21	16	12	21	16	16	32	58	16	16	11	16	63	63	42	26	5		28
average	30	42	23	29	60	41	36	71	22	22	22	24	33	34	26	26	55	27	33	36	34	56	52	53	42	36	50	

Table 28: For each pair of locks *(rowA, colB)* at the maximum number of nodes, score of lock A vs lock B: percentage of applications for which lock A performs at least 5% better than B (**I-48 machine**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		24	38	33	57	28	22	61	24	19	10	28	29	43	24	14	57	24	24	52	48	62	52	33	14	38	29	34
alock-ls	33		26	35	48	26	21	68	9	13	9	5	30	39	9	9	52	22	9	48	43	57	39	39	13	35	30	30
backoff	33	35		35	57	37	37	68	26	30	30	21	43	43	30	30	57	26	30	39	48	57	43	43	22	26	48	38
cbomcs_spin	29	30	13		52	42	32	63	13	9	22	11	13	22	17	13	39	13	13	39	35	52	43	26	13	22	43	28
cbomcs_stp	24	26	0	9		26	26	42	9	9	13	5	13	4	13	9	12	4	9	17	8	13	9	13	9	4	22	13
clh-ls	17	11	16	26	58		5	58	5	11	0	5	21	47	0	0	58	5	11	47	47	53	53	32	11	32	32	25
clh_spin	17	21	32	37	63	16		63	16	16	5	16	21	47	11	0	63	16	16	53	47	58	53	37	16	37	37	31
clh_stp	17	5	0	5	16	11	5		5	5	5	5	5	0	5	5	0	0	5	11	5	26	21	5	5	0	0	7
c-ptl-tkt	24	35	26	48	61	32	26	84		13	17	11	35	52	22	13	65	26	17	52	48	61	52	39	17	39	52	37
c-tkt-tkt	33	35	26	48	70	32	26	84	22		22	16	35	57	22	17	70	30	13	61	48	65	48	39	26	30	48	39
hmcs	29	48	35	43	57	37	47	79	30	17		16	30	43	30	13	57	30	26	48	43	57	48	39	26	35	52	39
hticket-ls	33	32	21	37	63	32	32	79	16	21	16		16	47	5	0	58	16	16	58	53	58	58	42	16	37	47	35
malth_spin	24	26	13	30	65	37	32	79	9	13	22	11		39	0	4	48	4	9	43	35	48	48	26	13	22	43	29
malth_stp	38	39	13	26	50	42	37	63	22	17	30	21	17		22	17	33	17	17	42	38	43	30	30	22	17	43	30
mcs-ls	29	30	22	39	61	37	32	84	17	17	26	21	26	35		9	48	9	17	48	39	48	48	26	17	30	48	33
mcs_spin	38	48	30	43	65	37	32	79	35	26	26	21	26	48	26		57	22	22	52	48	57	48	39	22	30	48	39
mcs_stp	24	26	4	13	29	32	26	32	9	9	17	5	17	4	17	13		8	9	17	17	30	17	9	9	4	17	16
mcs-timepub	29	30	22	30	46	32	32	74	17	17	26	21	26	33	17	13	46		17	46	33	52	43	35	26	30	48	32
partitioned	33	22	30	43	65	26	21	84	17	9	22	11	35	48	17	17	61	30		52	48	65	52	39	22	30	48	37
pthread	24	26	0	17	42	32	26	53	17	17	22	11	22	12	22	22	33	21	13		8	30	17	9	9	4	17	20
pthreadadapt	24	26	0	17	46	32	26	58	17	13	22	11	17	8	22	17	33	17	13	17		35	13	13	9	9	22	21
spinlock	24	26	0	17	39	21	21	37	17	13	22	11	17	9	22	17	26	13	13	17	4		0	9	9	0	17	16
spinlock-ls	33	26	9	30	52	26	26	53	17	13	22	11	26	17	30	26	39	22	13	30	22	35		17	13	0	26	24
ticket	33	22	9	30	61	32	26	74	17	17	22	11	26	26	22	22	57	22	9	39	35	57	26		4	13	30	28
ticket-ls	29	35	22	43	61	32	32	79	22	26	26	16	35	43	26	26	61	22	22	61	61	61	39	39		17	39	37
ttas	33	35	13	43	70	26	26	68	22	17	26	16	30	43	30	26	52	26	22	48	43	48	35	26	17		30	34
ttas-ls	33	26	13	26	52	21	21	58	9	9	9	5	17	30	17	17	43	9	9	39	35	39	30	17	17	4		23
average	28	29	17	31	54	30	27	66	17	15	19	13	24	32	18	14	47	17	15	41	36	49	37	28	15	21	35	

Table 29: For each pair of locks *(rowA, colB)* at the optimized number of nodes, score of lock A vs lock B: percentage of applications for which lock A performs at least 5% better than B (**A-64 machine with thread-to-node pinning**).

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		24	43	52	62	22	22	72	24	19	5	17	33	48	29	24	71	33	19	67	62	76	71	62	29	71	62	43
alock-ls	52		30	35	57	32	32	84	9	17	26	16	26	43	22	13	65	39	30	65	57	65	65	61	35	65	61	42
backoff	33	48		39	61	47	47	84	35	35	30	37	52	57	43	30	70	35	43	48	65	70	70	57	35	39	52	49
cbomcs_spin	33	35	13		57	53	32	84	26	22	17	16	26	39	30	13	57	26	22	52	61	65	65	48	35	52	57	40
cbomcs_stp	24	26	4	9		26	26	68	9	9	9	5	13	12	13	9	50	8	9	38	25	74	65	48	22	48	57	27
clh-ls	28	5	26	26	58		5	68	11	11	11	5	11	47	11	0	68	21	21	63	63	68	68	63	21	68	63	35
clh_spin	22	11	32	26	58	16		68	11	21	5	11	11	47	16	0	68	37	21	68	58	68	68	68	32	68	68	38
clh_stp	17	5	0	5	21	5	5		5	5	5	5	5	0	5	5	0	0	5	5	5	58	26	11	5	5	0	8
c-ptl-tkt	43	39	35	39	65	32	32	89		17	26	11	30	52	35	17	74	43	30	61	57	65	65	61	35	65	83	46
c-tkt-tkt	48	26	26	43	65	26	32	89	17		22	11	35	57	30	22	74	43	30	61	57	70	65	61	39	65	78	46
hmcs	38	48	43	52	57	47	47	89	30	26		21	35	48	35	26	65	39	35	65	57	65	65	65	39	65	83	49
hticket-ls	39	32	26	37	63	42	37	89	21	21	16		26	47	16	11	68	37	26	68	58	68	68	63	42	68	79	45
malth_spin	29	35	17	35	61	32	37	89	17	13	13	5		48	13	9	70	30	22	57	52	65	65	65	22	57	74	40
malth_stp	38	39	9	22	54	42	37	79	17	17	22	16	17		17	17	46	17	17	46	42	65	61	52	39	43	61	36
mcs-ls	38	26	22	39	61	26	37	89	13	13	26	16	26	48		13	70	26	26	52	61	65	65	61	22	57	70	41
mcs_spin	48	35	35	43	61	42	37	84	30	22	22	11	30	52	35		70	35	17	61	57	65	65	65	30	61	74	46
mcs_stp	24	26	4	13	25	26	26	21	9	9	9	5	9	8	9	9		8	9	8	8	52	17	13	9	4	17	15
mcs-timepub	33	35	30	35	54	32	32	89	13	17	17	11	26	42	22	22	62		26	62	58	70	65	65	35	61	65	42
partitioned	43	22	26	43	65	26	26	89	13	4	26	5	22	48	17	17	74	30		65	57	74	65	65	30	65	83	42
pthread	24	26	0	22	33	26	26	84	17	17	22	11	26	29	26	17	67	17	13		25	74	70	30	13	39	48	31
pthreadadapt	24	30	0	17	46	26	26	79	17	9	22	11	17	21	17	17	62	17	13	33		74	65	52	9	39	52	31
spinlock	24	26	0	17	9	21	21	26	17	13	22	11	17	17	17	17	22	13	13	0	0		0	9	4	0	17	14
spinlock-ls	24	26	0	22	13	26	26	58	17	13	22	11	17	17	22	17	48	13	13	0	0	65		9	4	17	22	20
ticket	24	22	9	30	30	26	26	84	17	17	22	11	17	30	13	17	70	13	13	26	17	70	65		0	39	48	29
ticket-ls	48	39	17	35	52	42	42	89	26	26	35	16	35	35	30	26	74	30	22	57	65	70	65	74		57	74	45
ttas	24	26	4	30	39	26	26	79	17	17	26	16	30	30	30	17	65	17	17	9	30	61	35	17	9		17	28
ttas-ls	24	22	17	30	26	21	21	79	9	9	9	5	17	22	17	17	61	4	9	26	26	65	57	17	13	48		26
average	32	28	18	31	48	30	29	77	17	16	19	12	24	36	22	16	61	24	20	45	43	67	59	49	23	49	56	

Table 30: For each pair of locks *(rowA, colB)* at the maximum number of nodes, score of lock A vs lock B: percentage of applications for which lock A performs at least 5% better than B (**A-64 machine with thread-to-node pinning**).

E Are all locks potentially harmful?

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	-	4	185	176	-	-	-	42	12	188	161	179	175	155	215	210	223	17	3	0	13	3	7	6	4	
ferret	455	385	9	397	1	392	402	0	483	444	447	489	441	0	395	387	2	7	397	0	0	14	12	395	270	10	8
fmm	47	44	41	43	34	48	53	40	38	38	40	40	24	34	39	5	0	3	26	33	33	3	39	41	36	36	39
histogram	0	6	2	4	12	12	14	32	3	6	0	4	11	10	0	29	53	29	13	8	2	45	11	24	5	12	10
linear_regression	9	93	0	11	266	5	5	78	11	75	28	31	36	77	63	26	110	22	27	22	3	81	33	27	3	27	21
matrix_multiply	2	4	5	4	6	29	6	5	8	15	2	8	13	2	4	1	5	1	22	0	0	4	6	10	6	5	4
mysqld	-	-	-	-	37	-	-	-	-	-	-	-	-	13	-	-	9	54	-	1	0	-	-	-	-	-	-
pca	46	51	0	26	230	66	63	191	29	33	23	17	21	128	25	84	298	51	49	96	39	146	118	91	31	130	97
pca_ll	29	66	0	17	482	43	38	471	15	40	19	8	6	296	20	76	819	58	17	188	79	458	303	147	49	252	213
radiosity	29	24	20	32	150	24	34	827	24	25	28	35	29	84	29	0	608	0	31	109	56	179	101	62	42	82	95
radiosity_ll	0	7	34	12	1k	21	8	2k	14	26	1	34	59	2k	31	10	3k	40	64	658	227	2k	694	502	190	647	558
s_raytrace	2	12	4	27	1k	18	2	775	3	22	3	33	99	550	28	0	1k	0	12	183	93	392	272	278	82	269	366
s_raytrace_ll	6	0	15	22	2k	12	15	2k	21	27	4	75	49	2k	15	11	3k	48	73	390	196	872	607	547	262	608	756
ssl_proxy	14	20	0	24	767	17	71	873	48	23	18	17	61	983	23	72	1k	56	85	347	153	931	638	437	122	401	432
streamcluster	33	58	63	140	378	-	-	-	0	26	4	-	245	1k	228	261	2k	178	67	202	263	181	167	236	196	142	97
streamcluster_ll	44	56	154	182	430	-	-	-	0	37	28	-	307	2k	294	247	2k	226	90	184	277	239	205	276	194	103	81
vips	94	126	4	253	46	-	-	-	202	143	171	-	819	6	42	76	7	4	88	0	0	3	5	52	13	5	12
volrend	0	12	5	19	93	9	7	81	0	5	3	1	4	152	17	59	176	60	9	88	87	137	38	18	9	20	13
water_nsquared	78	42	11	18	16	47	29	29	14	11	13	13	8	9	16	5	6	8	6	8	8	0	12	5	12	17	33
water_spatial	69	35	0	3	3	44	34	33	1	0	6	4	9	7	4	19	21	19	6	0	1	15	0	5	2	1	28
dedup	-	-	0	128	130	-	-	-	27	13	147	109	113	121	102	153	149	156	15	3	1	13	0	4	1	0	
ferret	105	55	5	48	0	62	59	0	124	130	136	127	85	0	60	67	1	4	63	0	0	9	7	73	47	5	5
fmm	47	44	41	43	34	48	47	40	38	38	40	40	24	34	39	5	0	3	26	33	33	3	39	38	36	36	39
histogram	8	6	3	3	3	6	15	24	5	1	0	8	8	7	12	27	46	32	20	6	12	32	15	13	9	11	9
linear_regression	11	12	13	15	50	13	8	59	12	14	0	12	8	33	3	39	108	38	16	24	17	63	28	37	17	26	21
matrix_multiply	2	4	5	4	6	29	6	5	8	15	2	8	11	2	4	1	5	1	11	0	0	4	6	10	6	5	4
mysqld	-	-	-	-	37	-	-	-	-	-	-	-	-	13	-	-	9	52	-	0	0	-	-	-	-	-	-
pca	3	4	0	3	46	4	11	75	5	3	10	1	13	65	7	20	120	20	10	29	10	27	7	18	5	7	9
pca_ll	2	5	15	23	84	5	71	284	3	0	26	0	33	171	18	74	777	77	47	89	58	78	52	77	6	30	54
radiosity	32	27	23	28	33	26	16	36	22	28	28	29	18	38	25	0	16	1	14	37	41	4	28	18	31	20	30
radiosity_ll	0	5	26	12	71	17	6	238	12	26	1	34	56	248	26	4	348	31	24	119	71	71	71	78	64	63	65
s_raytrace	2	2	4	14	96	2	2	301	3	11	3	9	21	188	6	0	343	0	5	60	55	51	57	51	28	47	54
s_raytrace_ll	6	0	13	22	159	7	6	367	12	27	4	26	37	382	10	6	355	19	14	91	83	155	135	156	27	97	131
ssl_proxy	8	5	0	19	53	4	21	523	12	18	9	13	43	1k	11	26	1k	34	31	68	54	47	51	50	7	19	51
streamcluster	6	9	6	0	5	-	-	-	0	2	0	-	8	8	8	3	6	8	6	907	954	7	4	10	6	4	0
streamcluster_ll	27	39	37	27	11	-	-	-	0	5	0	-	37	39	31	33	31	36	26	33	45	31	31	40	39	31	32
vips	15	19	4	16	18	-	-	-	16	18	17	-	14	6	15	16	7	4	15	0	0	3	5	15	13	5	12
volrend	3	2	7	6	27	1	10	20	0	2	4	4	10	31	4	12	32	14	10	15	16	20	10	11	5	8	9
water_nsquared	78	42	11	18	16	47	29	29	14	11	13	13	8	9	16	5	6	8	6	8	8	0	12	5	12	17	33
water_spatial	69	35	0	3	3	44	34	33	1	0	6	4	9	7	4	19	21	19	6	0	1	15	0	5	2	1	28

Table 31: For each application, at max nodes (top part) and at the optimized number of nodes (bottom part), performance gain (in %) obtained by the best lock(s) with respect to each of the other locks. The grey background highlights cells for which the performance gains are greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**A-48 machine**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	
dedup	1k	607	2	188	185	869	541	543	14	11	178	166	165	162	173	162	161	178	13	0	1	4	1	2	3	3	546	
ferret	657	545	15	521	1	578	556	0	599	654	626	623	688	0	533	528	0	11	553	0	0	14	13	555	393	14	13	
fmm	19	20	4	11	4	15	12	8	12	11	14	10	10	4	11	14	2	7	13	0	0	5	3	9	16	3	8	
histogram	1	8	4	3	21	4	0	15	1	4	0	2	2	3	3	5	15	0	6	5	2	22	14	18	3	3	3	
linear_regression	0	115	8	82	123	38	6	85	33	54	18	34	13	58	110	56	89	32	26	15	15	93	81	44	18	20	14	
matrix_multiply	0	3	0	9	7	4	2	3	5	9	4	8	6	5	3	8	7	3	1	2	3	4	6	4	5	2	3	
mysqld	-	-	-	-	0	-	-	-	-	-	-	-	-	19	-	-	24	114	-	27	20	-	-	-	-	-	-	
pca	6	19	1	0	112	16	21	292	5	4	20	14	4	2	15	17	284	11	23	36	27	181	194	54	28	42	40	
pca_ll	7	34	82	0	552	50	56	1k	63	12	9	41	43	116	30	35	1k	50	80	164	135	696	668	159	105	167	173	
radiosity	7	11	8	1	38	12	11	160	2	0	1	1	6	12	5	4	155	7	15	20	14	120	95	29	18	33	34	
radiosity_ll	0	66	104	8	1k	88	94	2k	45	24	0	17	127	149	66	61	2k	86	179	264	169	2k	1k	380	237	441	438	
s_raytrace	0	15	19	40	390	26	36	1k	20	17	9	44	65	112	24	22	1k	17	65	77	117	1k	837	186	130	208	261	
s_raytrace_ll	2	52	48	42	1k	83	89	2k	30	15	0	99	97	210	48	47	2k	65	157	185	228	2k	2k	389	263	456	502	
ssl_proxy	3	40	7	0	552	40	46	1k	12	19	0	12	47	132	41	42	2k	54	76	116	73	1k	707	174	106	164	156	
streamcluster	89	0	586	225	508	-	-	-	160	137	104	-	456	533	317	398	474	407	110	513	619	494	382	628	621	337	322	
streamcluster_ll	175	0	885	326	752	-	-	-	195	182	181	-	549	631	389	480	559	520	148	674	657	545	534	773	699	420	397	
vips	129	248	4	1k	18	-	-	-	323	223	246	-	420	3	149	111	7	8	126	1	0	6	5	119	14	3	3	
volrend	0	2	167	8	34	4	7	20	2	2	3	3	7	17	4	5	13	5	8	21	30	39	30	18	17	19	18	
water_nsquared	92	48	3	13	8	60	36	39	5	0	10	5	5	4	4	5	3	11	4	4	5	2	2	4	0	3	35	
water_spatial	97	51	3	4	4	66	41	41	1	2	7	2	5	4	6	4	3	3	1	1	0	1	0	3	0	1	39	
dedup	839	443	6	170	174	620	394	400	10	9	183	146	143	147	163	148	149	156	7	0	1	3	1	2	2	4	418	
ferret	45	46	8	45	1	49	45	0	44	59	45	81	58	0	46	45	0	8	45	0	0	8	9	58	45	7	7	
fmm	18	15	4	11	4	15	12	8	9	11	13	10	10	4	11	10	2	7	13	0	0	5	3	9	11	3	8	
histogram	0	8	3	7	2	6	9	11	7	5	1	5	4	9	5	1	15	8	0	3	1	13	9	7	7	0	7	
linear_regression	0	21	8	16	40	16	6	85	16	13	12	16	13	19	32	20	89	23	24	15	15	73	57	27	18	20	14	
matrix_multiply	0	3	0	9	7	4	2	3	5	9	4	8	6	5	3	8	7	3	1	2	3	4	6	4	5	2	3	
mysqld	-	-	-	-	30	-	-	-	-	-	-	-	-	54	-	-	52	165	-	0	0	-	-	-	-	-	-	
pca	0	1	5	0	11	3	7	265	2	3	3	4	1	9	3	7	244	9	8	26	19	14	9	16	8	5	3	
pca_ll	6	29	82	0	155	40	44	1k	4	5	8	17	43	67	29	23	1k	43	47	131	87	69	72	78	59	71	67	
radiosity	6	7	7	1	5	10	7	91	1	0	0	1	6	10	3	2	8	6	5	15	11	19	18	18	11	11	12	
radiosity_ll	0	63	102	8	89	85	86	2k	38	22	0	17	127	149	61	61	2k	85	108	245	169	260	197	224	159	157	170	
s_raytrace	0	14	19	40	105	23	23	203	15	16	9	29	52	104	20	18	197	17	25	70	97	196	196	119	79	75	89	
s_raytrace_ll	2	39	48	42	232	54	53	222	30	15	0	48	84	167	39	37	236	43	63	120	175	235	237	206	122	190	187	
ssl_proxy	1	43	27	9	63	58	68	1k	12	13	0	28	84	173	46	55	2k	58	51	149	119	373	224	224	149	192	133	
streamcluster	8	3	12	5	4	-	-	-	6	6	0	-	24	24	21	19	17	28	5	11	39	20	13	33	32	12	13	
streamcluster_ll	42	0	144	60	51	-	-	-	74	45	19	-	155	147	134	138	141	184	40	110	186	140	117	182	183	107	103	
vips	24	24	4	24	18	-	-	-	24	23	22	-	25	3	25	25	7	8	25	1	0	6	5	24	14	3	3	
volrend	2	5	44	5	25	5	6	18	2	0	3	2	9	17	4	4	12	5	5	21	23	23	21	15	11	16	14	
water_nsquared	92	48	3	13	8	60	36	39	5	0	10	5	5	4	4	5	3	11	4	4	5	2	2	4	0	3	35	
water_spatial	91	51	3	4	4	66	41	41	1	2	7	2	5	4	6	4	3	3	1	1	0	1	0	3	0	1	39	

Max nodes

Opt nodes

Table 32: For each application, at max nodes (top part) and at the optimized number of nodes (bottom part), performance gain (in %) obtained by the best lock(s) with respect to each of the other locks. The grey background highlights cells for which the performance gains are greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**I-48 machine**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	
dedup	-	558	1	127	129	945	945	945	21	7	125	112	114	109	113	111	115	134	11	1	2	2	1	0	2	0	544	Max nodes
facesim	3	5	31	48	167	5	5	56	4	3	4	2	0	47	4	5	55	6	5	30	56	301	256	28	1	27	22	
ferret	382	328	2	313	0	353	327	0	343	389	386	310	340	0	254	314	0	1	328	0	0	6	5	232	189	3	3	
fluidanimate	-	301	0	51	56	-	-	-	27	11	60	-	36	57	42	32	55	51	6	6	10	20	7	6	4	1	198	
fmm	11	5	0	1	0	7	8	8	0	0	2	1	0	0	1	3	1	2	0	0	0	3	1	1	0	0	5	
histogram	6	3	3	0	2	6	1	15	5	5	1	3	1	0	4	0	14	0	1	8	4	19	19	8	0	18	8	
linear_regression	7	25	6	6	35	5	5	74	10	1	0	1	4	11	5	2	71	6	6	34	16	95	67	35	8	60	37	
matrix_multiply	0	2	1	2	1	2	0	3	1	2	0	2	0	0	0	0	3	0	0	1	0	5	4	0	2	4	3	
mysqld	-	-	-	-	30	-	-	-	-	-	-	-	-	0	-	-	7	173	-	97	102	-	-	-	-	-	-	
ocean_cp	6	0	27	29	101	2	5	55	1	1	5	2	4	47	6	1	52	16	2	28	41	134	120	16	0	21	19	
ocean_ncp	10	0	20	22	87	5	4	41	3	3	6	4	4	37	2	2	39	10	3	22	30	115	83	13	2	18	15	
pca	50	40	18	13	72	43	49	289	46	47	47	48	44	0	40	47	289	43	49	134	53	349	158	92	18	258	149	
pca_ll	90	72	0	18	246	69	94	753	79	79	89	31	64	12	74	72	755	51	74	257	111	843	504	200	53	498	300	
radiosity	6	4	2	1	4	5	4	71	1	1	0	1	2	3	3	1	70	3	3	32	18	115	59	29	10	61	41	
radiosity_ll	0	44	47	26	214	47	26	2k	36	50	1	20	69	77	55	20	2k	67	83	547	311	2k	1k	602	209	1k	787	
s_raytrace	4	10	23	47	208	16	11	1k	14	19	2	33	32	70	15	0	1k	17	33	270	144	789	524	403	105	804	633	
s_raytrace_ll	1	100	29	94	644	109	92	3k	122	112	0	114	63	260	108	22	3k	119	177	703	377	2k	1k	850	282	2k	1k	
ssl_proxy	3	11	0	11	62	14	13	985	12	16	1	7	15	35	29	15	995	28	30	294	157	1k	841	258	85	623	422	
streamcluster	76	45	226	134	213	-	-	-	0	42	17	-	323	870	259	228	680	152	57	217	339	724	485	373	244	233	118	
streamcluster_ll	44	21	186	144	273	-	-	-	7	14	0	-	237	860	192	154	739	85	23	174	238	597	389	316	198	176	82	
vips	81	38	13	167	0	-	-	-	198	49	103	-	168	8	32	46	9	11	47	10	9	10	10	37	15	11	14	
volrend	7	4	26	23	51	6	5	23	0	0	1	2	10	19	7	7	20	11	8	37	50	110	75	29	11	31	25	
water_nsquared	91	44	1	5	6	56	56	55	3	2	9	5	5	4	5	4	4	9	2	1	2	2	0	3	1	0	33	
water_spatial	89	43	1	3	5	58	59	60	2	1	6	3	4	4	4	4	4	6	2	2	1	1	1	2	2	0	36	
dedup	-	278	4	154	145	522	519	501	28	16	181	127	132	122	135	132	120	144	16	0	0	3	4	3	4	2	331	Opt nodes
facesim	1	1	4	9	21	1	2	13	2	0	2	1	0	9	1	1	11	0	2	5	13	21	21	2	0	2	0	
ferret	355	315	2	313	0	328	327	0	343	327	342	302	303	0	254	314	0	1	328	0	0	6	5	232	189	3	3	
fluidanimate	-	183	0	51	56	-	-	-	27	11	60	-	36	52	42	32	49	51	6	6	10	20	7	6	4	1	124	
fmm	11	5	0	1	0	7	8	8	0	0	2	1	0	0	1	3	1	2	0	0	0	3	1	1	0	0	5	
histogram	7	8	12	9	11	15	2	9	8	2	11	12	3	1	4	0	8	3	4	9	7	15	19	8	10	15	10	
linear_regression	7	10	6	6	10	5	5	44	7	1	0	1	4	10	5	2	40	6	6	34	16	73	50	25	8	46	33	
matrix_multiply	0	2	1	2	1	2	0	3	1	2	0	2	0	0	0	0	3	0	0	1	0	5	4	0	2	4	3	
mysqld	-	-	-	-	26	-	-	-	-	-	-	-	-	0	-	-	8	121	-	96	96	-	-	-	-	-	-	
ocean_cp	2	3	3	5	15	2	3	13	0	1	5	0	1	9	1	2	12	3	0	9	6	11	8	5	0	1	1	
ocean_ncp	2	4	4	3	9	2	1	8	2	0	2	0	2	9	2	2	9	4	2	6	5	11	9	2	0	0	3	
pca	16	17	9	9	64	15	16	62	16	16	17	17	16	0	16	16	60	16	16	24	22	18	17	16	13	16	16	
pca_ll	43	51	0	18	175	36	62	190	37	48	43	31	22	12	24	35	188	18	55	85	82	85	81	72	36	83	78	
radiosity	4	3	2	2	4	4	3	5	0	0	0	1	2	3	2	0	2	2	2	7	4	6	2	3	1	2	3	
radiosity_ll	0	39	47	26	132	42	17	267	24	28	1	20	61	77	42	13	237	46	36	151	92	116	71	75	65	65	67	
s_raytrace	4	10	23	47	174	16	11	211	14	19	2	26	32	70	15	0	205	17	31	102	89	73	68	72	64	67	81	
s_raytrace_ll	1	18	31	46	149	20	17	105	18	15	0	19	28	62	19	0	101	21	17	99	101	102	58	90	20	33	104	
ssl_proxy	0	12	15	38	52	10	2	47	17	30	5	9	22	38	11	7	37	21	8	68	38	41	32	32	0	31	31	
streamcluster	23	8	34	26	44	-	-	-	9	4	3	-	27	26	24	23	22	26	5	31	30	29	5	14	20	0	8	
streamcluster_ll	48	25	56	44	70	-	-	-	0	9	4	-	84	82	89	67	65	72	16	56	77	76	26	31	28	11	14	
vips	62	42	2	195	4	-	-	-	170	65	125	-	196	0	46	31	0	2	54	0	0	3	4	52	17	1	2	
volrend	0	0	11	7	20	2	1	9	0	0	0	1	3	7	2	1	7	4	2	15	19	28	15	6	1	6	4	
water_nsquared	91	44	1	5	6	56	56	55	3	2	9	5	5	4	5	4	4	9	2	1	2	2	0	3	1	0	33	
water_spatial	89	43	1	3	5	58	59	60	2	1	6	3	4	4	4	4	4	6	2	2	1	1	1	2	2	0	36	

Table 33: For each application, at max nodes (top part) and at the optimized number of nodes (bottom part), performance gain (in %) obtained by the best lock(s) with respect to each of the other locks. The grey background highlights cells for which the performance gains are greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**A-64 machine with thread-to-node pinning**).

F Impact of the number of nodes

Applications	% of pairwise changes between configurations			
	1/2	2/4	4/8	1/2/4/8
dedup	5%	1%	5%	9%
ferret	0%	75%	17%	87%
fmm	26%	23%	31%	49%
histogram	38%	39%	33%	64%
linear_regression	33%	34%	50%	70%
matrix_multiply	28%	34%	49%	67%
mysqld	27%	0%	7%	33%
pca	23%	31%	29%	65%
pca_ll	44%	30%	34%	74%
radiosity	46%	40%	20%	77%
radiosity_ll	19%	55%	12%	77%
s_raytrace	21%	43%	32%	92%
s_raytrace_ll	23%	47%	30%	93%
ssl_proxy	42%	21%	11%	54%
streamcluster	23%	52%	32%	88%
streamcluster_ll	44%	56%	40%	95%
vips	1%	4%	84%	86%
volrend	20%	18%	40%	76%
water_nsquared	35%	25%	21%	65%
water_spatial	14%	17%	11%	34%

Table 34: For each application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**A-48 machine**).

Applications	% of pairwise changes between configurations			
	1/2	2/3	3/4	1/2/3/4
dedup	8%	6%	5%	12%
ferret	27%	69%	14%	89%
fmm	9%	14%	32%	53%
histogram	36%	21%	22%	49%
linear_regression	22%	31%	52%	82%
matrix_multiply	11%	14%	17%	28%
mysqld	27%	27%	27%	60%
pca	44%	21%	14%	59%
pca_ll	47%	9%	8%	54%
radiosity	35%	16%	10%	53%
radiosity_ll	22%	9%	2%	28%
s_raytrace	70%	18%	12%	95%
s_raytrace_ll	75%	17%	12%	98%
ssl_proxy	15%	7%	10%	22%
streamcluster	19%	17%	18%	42%
streamcluster_ll	23%	13%	24%	41%
vips	0%	0%	82%	82%
volrend	25%	33%	6%	61%
water_nsquared	20%	0%	16%	36%
water_spatial	0%	1%	2%	4%

Table 35: For each application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**I-48 machine**).

Applications	% of pairwise changes between configurations			
	1/2	2/4	4/8	1/2/4/8
dedup	13%	8%	2%	17%
facesim	0%	48%	24%	72%
ferret	34%	16%	21%	51%
fluidanimate	9%	5%	21%	31%
fmm	7%	10%	5%	23%
histogram	23%	42%	46%	75%
linear_regression	60%	50%	30%	89%
matrix_multiply	0%	0%	0%	0%
mysqld	13%	20%	7%	27%
ocean_cp	14%	48%	39%	82%
ocean_ncp	10%	24%	45%	76%
pca	32%	45%	17%	80%
pca_ll	34%	67%	21%	97%
radiosity	0%	51%	10%	61%
radiosity_ll	43%	46%	14%	90%
s_raytrace	0%	53%	46%	93%
s_raytrace_ll	7%	69%	34%	95%
ssl_proxy	67%	17%	17%	79%
streamcluster	58%	22%	25%	80%
streamcluster_ll	55%	18%	26%	77%
vips	10%	9%	15%	24%
volrend	23%	21%	33%	77%
water_nsquared	22%	9%	9%	41%
water_spatial	2%	0%	1%	3%

Table 36: For each application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**A-64 machine with thread-to-node pinning**).