Splash-4
Improving Scalability with Lock-Free Constructs

Eduardo José Gómez-Hernández[1], Ruixiang Shao[1], Christos Sakalis[2], Stefanos Kaxiras[2] and Alberto Ros[1]

[1]Computer Engineering Department
University of Murcia, Spain

[2]Department of Information Technology
Uppsala University Sweden

Splash-2[1]

- First major parallel benchmark suite
- Many works based on its behavior
- Still relevant and useful
- Quite old with outdated programming techniques and bugs

Splash-3[2]

- Fixes many bugs of the previous version
- Focused in synchronization
- Not focused in performance only correctness

Splash-4

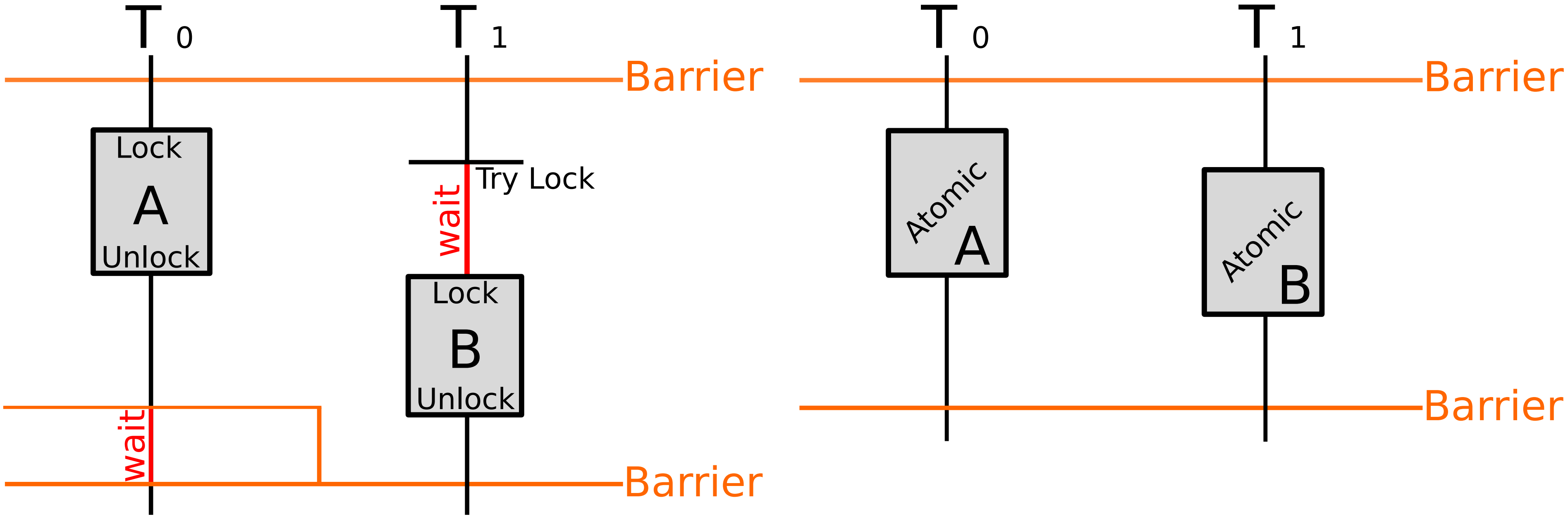
- Focused on atomic operations
- Better scalability in current hardware

[1] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “TheSPLASH-2 programs: Characterization and methodological considera-tions,” in22nd Int’l Symp. on Computer Architecture (ISCA), Jun. 1995,pp. 24–36.

[2] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, “Splash-3: Aproperly synchronized benchmark suite for contemporary research,” inInt’l Symp. on Performance Analysis of Systems and Software (ISPASS),Apr. 2016, pp. 101–111

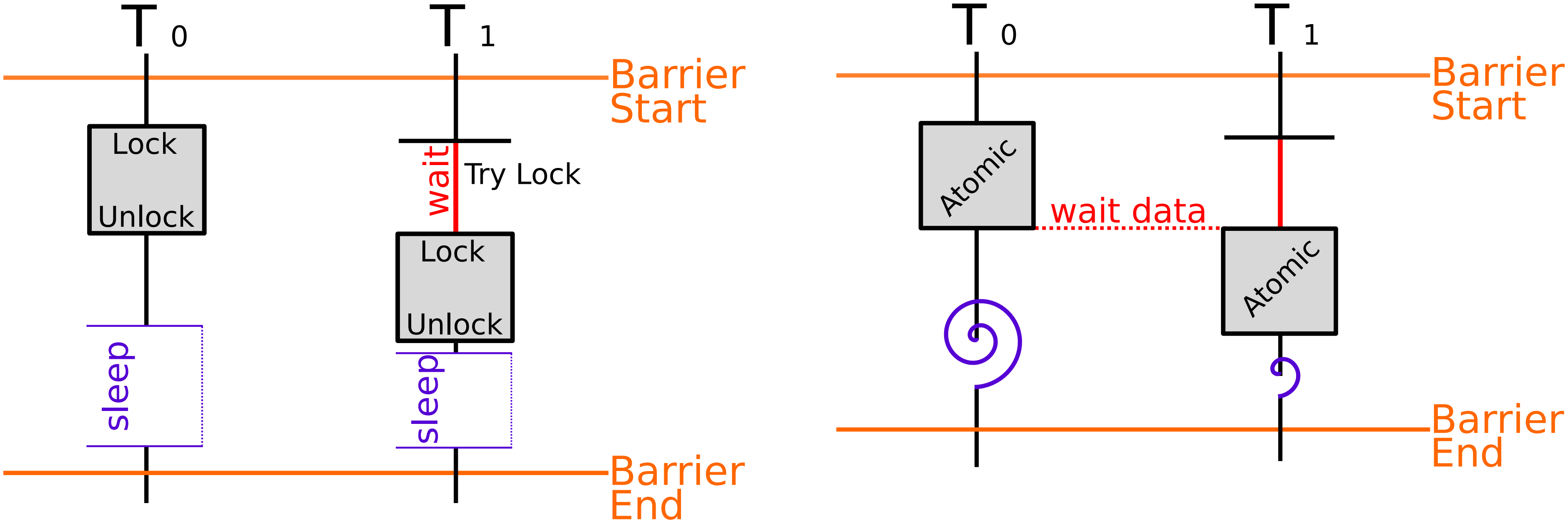
Locks (Mutexes)

Critical sections guarded by the same lock mutex, even if there is no data conflict, cannot be run in parallel, unless they are converted to atomics.

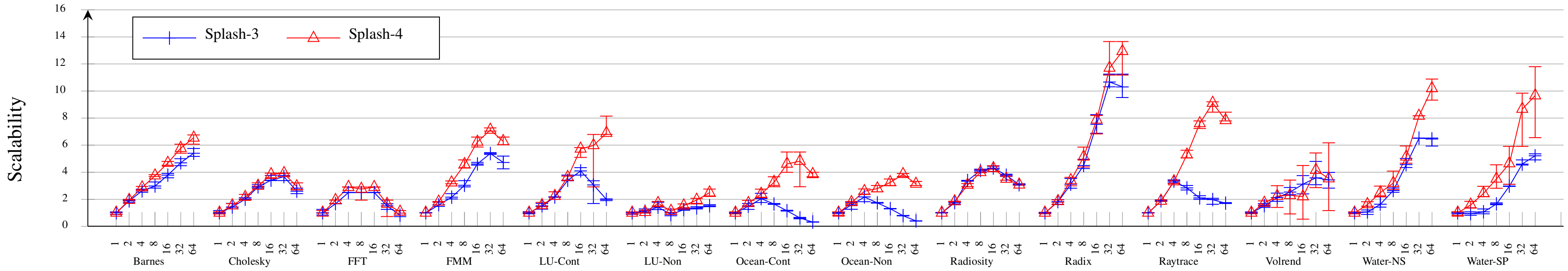


Barrier

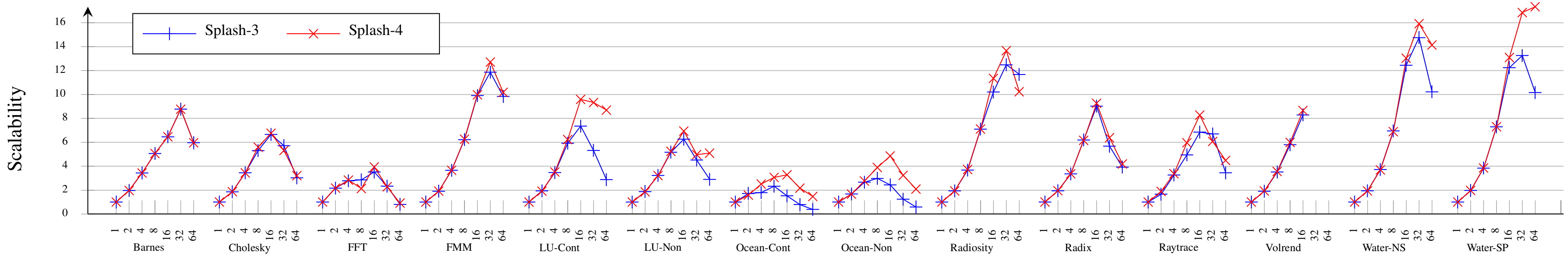
Barriers are often implemented using mutexes and a thread sleep.
When the time spent between barriers is high, this overhead is irrevelant.
For contended barriers, a spinlock allow for a faster wakeup to continue the execution.



Results



Splash-3 vs Splash-4 Scalability on an **actual processor**



Splash-3 vs Splash-4 Scalability in **simulation**

Splash-4

Improving Scalability with Lock-Free Constructs

Eduardo José Gómez-Hernández[1], Ruixiang Shao[1], Christos Sakalis[2], Stefanos Kaxiras[2] and Alberto Ros[1]

eduardojose.gomez@um.es

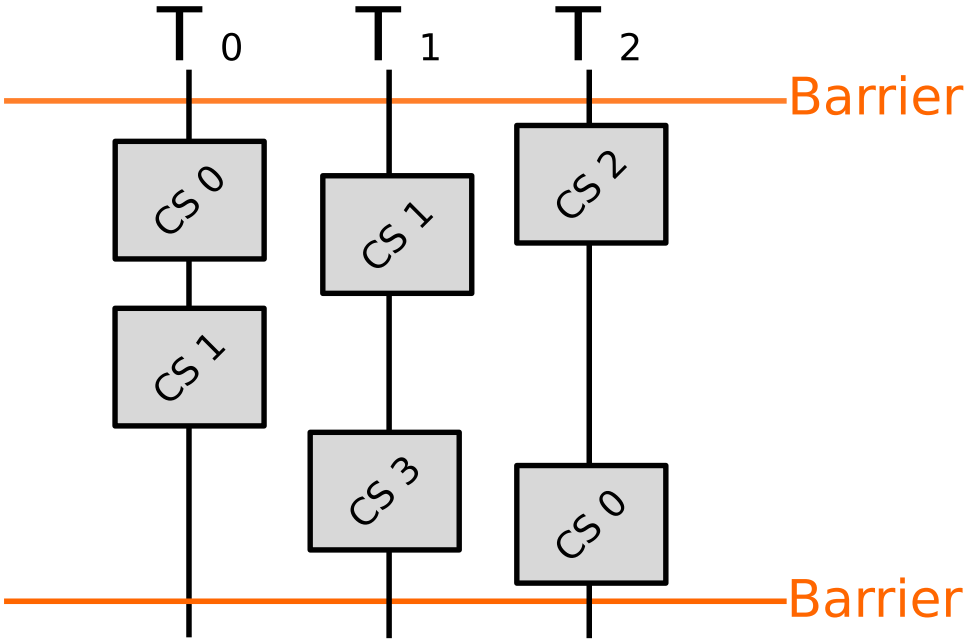
Thank you for your attention!

European Research Council
Established by the European Commission

ECHO, ERC Consolidator Grant (No 819134)

Barrier Groups

We define a barrier group as the set of critical sections that can be executed between two consecutive barriers (for all threads).



Barrier group 0
- CS 0
- CS 1
- CS 2
- CS 3

Critical sections 0,1,2,3 could be executed in parallel between two barriers.

This stablish a relation that limits the parallelism

While&CAS

```
var oldValue;
var newValue;
do {
   oldValue = *(ptr);
   newValue = new;
} while (!CAS(ptr, oldValue, newValue));
```

While&CAS structure

```
double oldValue;
double newValue;
do {
  oldValue = *ptr;
  newValue = oldValue + addition;
} while (!CAS(ptr, oldValue, newValue));
```

FETCH_AND_ADD_DOUBLE operation

Atomic operations in modern processors are limited.
But using the while&cas structure is possible to craft custom “atomic constructs”[1].

In this example we propose the FETCH_AND_ADD_DOUBLE atomic, that allows to add 64-bits floating point numbers atomically.

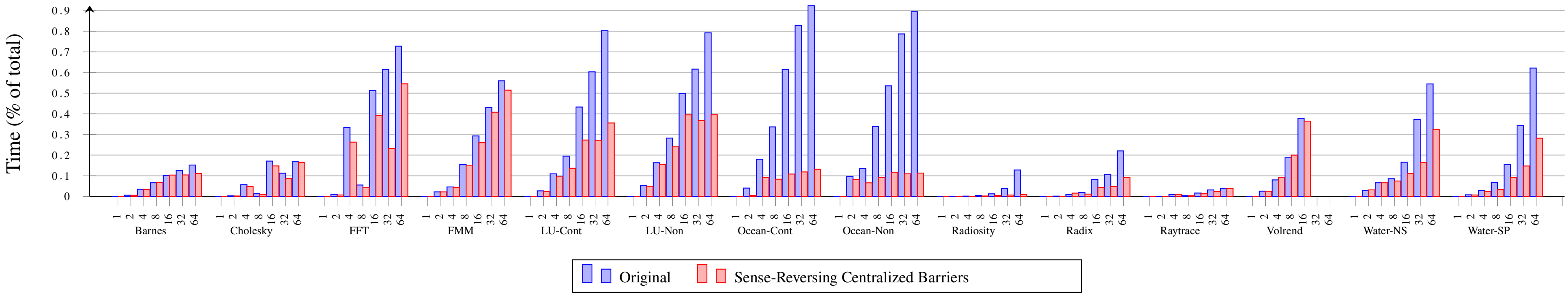
[1] H. Gao and W. Hesselink, “A general lock-free algorithm using compare-and-swap,”Information and Computation, vol. 205, no. 2, pp. 225–241,2007.

Sense Reversing Barrier

The barrier we used is called “sense reversing barrier”[1].

```
local_sense = !local_sense;
if (atomic_fetch_sub(&(count), 1) == 1) {
   count = cores;
   sense = local_sense;
} else {
   do {} while (sense != local_sense);
}
```

Sense-reversing barrier



Ratio of time spent waiting on barriers, Original vs Sense-Reversing Centralized

[1] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable syn-chronization on shared-memory multiprocessors,”ACM Trans. Comput.Syst., vol. 9, no. 1, pp. 21–65, Feb. 1991.

Lock Split

In certain situations is possible to break the critical section into multiple ones without changing the result (breaking the group atomicity).

These examples are from the "water-nsquare" benchmark.

Group atomicity is not needed and neither is assumed anywhere in the code.

We surmise that in the original Splash-2 such clustering with the purpose of amortizing the high cost of the lock and unlock over many operations.

```
/* Lock */
LOCK(gl->PotengSumLock);
*POTA = *POTA + LPOTA;
*POTR = *POTR + LPOTR;
*PTRF = *PTRF + LPTRF;
UNLOCK(gl->PotengSumLock);
```

```
/* Lock-free */
FETCH_AND_ADD_DOUBLE(POTA, LPOTA);
FETCH_AND_ADD_DOUBLE(POTR, LPOTR);
FETCH_AND_ADD_DOUBLE(PTRF, LPTRF);
```

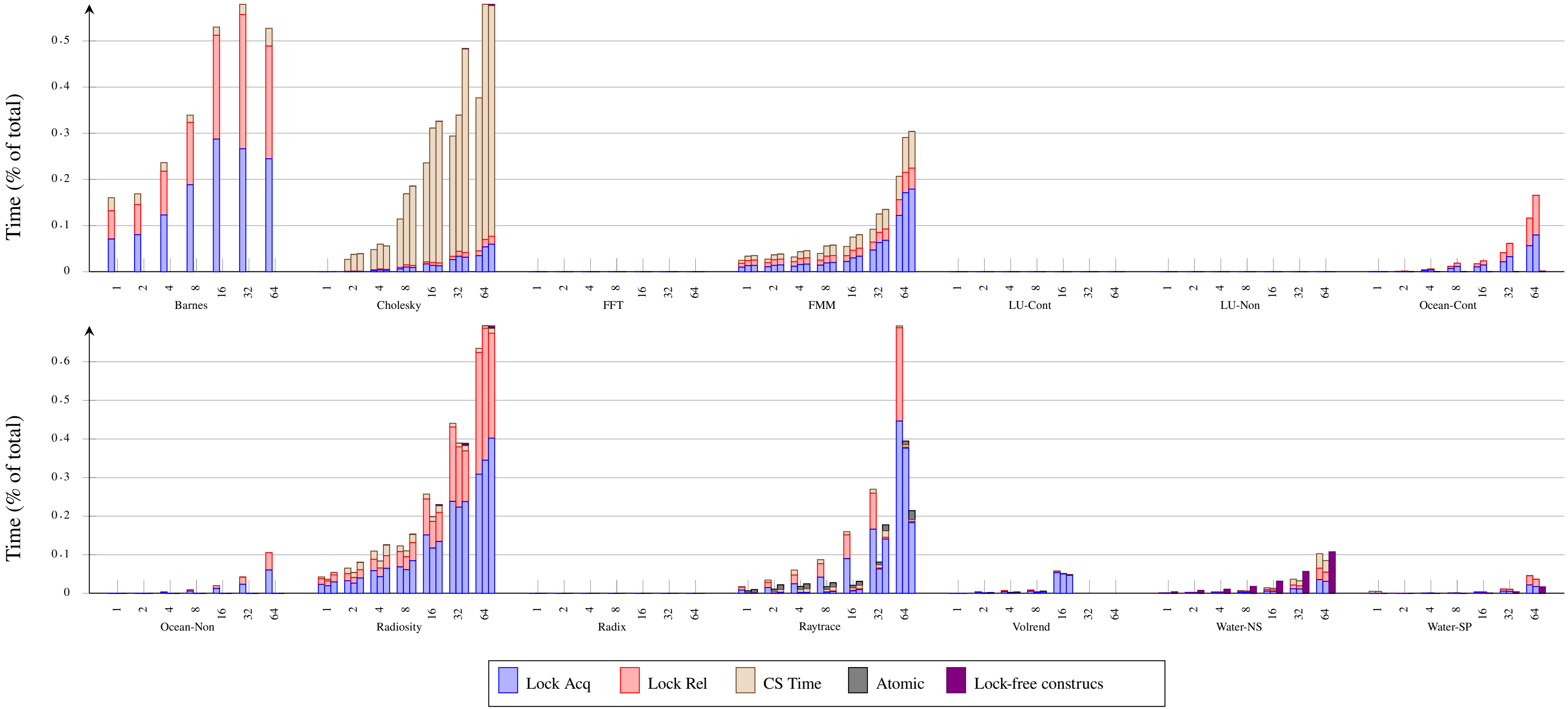
poteng.c.in 159 & poteng.c.in 253

```
/* Lock */
ALOCK(gl->MolLock, mol % MAXLCKS);
for ( dir = XDIR; dir <= ZDIR; dir++) {
    temp_p = VAR[mol].F[DEST][dir];
    temp_p[H1] += PFORCES[ProcID][mol][dir][
    H1];
    temp_p[O] += PFORCES[ProcID][mol][dir][O
    ];
    temp_p[H2] += PFORCES[ProcID][mol][dir][
    H2];
}
AULOCK(gl->MolLock, mol % MAXLCKS);
```

```
/* Lock-free */
for ( dir = XDIR; dir <= ZDIR; dir++) {
  FETCH_AND_ADD_DOUBLE(&(VAR[mol].F[DEST][
    dir][H1]), PFORCES[ProcID][mol][dir][H1
     ]);
  FETCH_AND_ADD_DOUBLE(&(VAR[mol].F[DEST][
    dir][O]), PFORCES[ProcID][mol][dir][O]);
  FETCH_AND_ADD_DOUBLE(&(VAR[mol].F[DEST][
    dir][H2]), PFORCES[ProcID][mol][dir][H2
     ]);
}
```

interf.c.in 156 & interf.c.in 167 & interf.c.in 179

Lock effects



Percent of time spent in critical sections out of total execution. The three bars per core count represent the original version, the straightforward C11 atomics, and the lock-free version respectively. The critical section time (CS) corresponds to the time spent in the critical section for the lock-unlock case, while for the C11 atomics and the lock-free constructs the original critical-section work is subsumed by the operations/constructs themselves.