

Does your tool support PAPI SDEs yet?

13th Scalable Tools Workshop

Anthony Danalis, Heike Jagode, Jack Dongarra

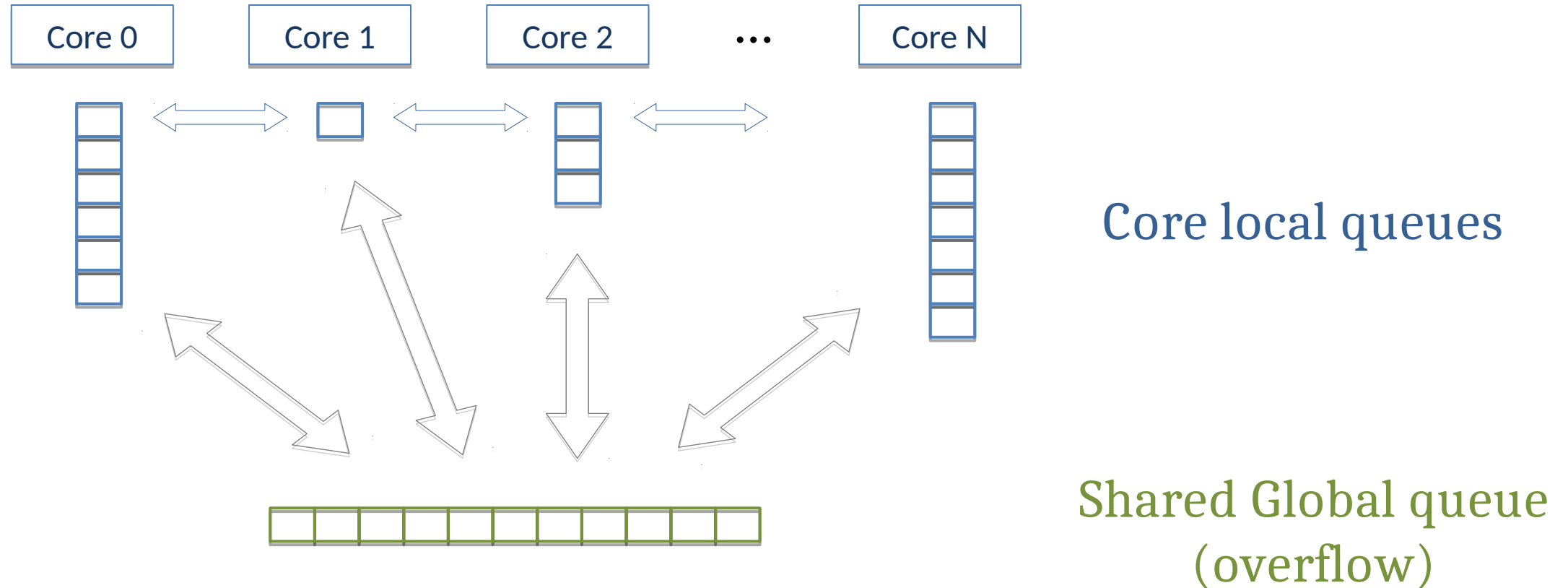
Tahoe City, CA

July 28-Aug 1, 2019

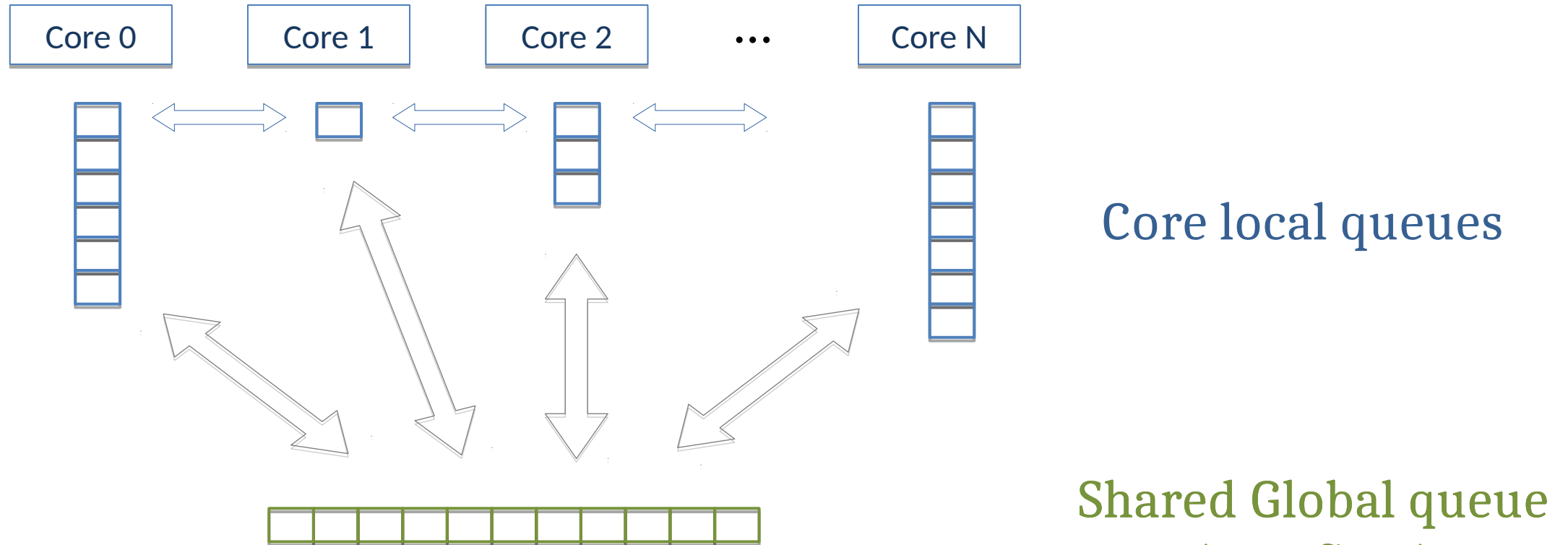


THE UNIVERSITY OF
TENNESSEE
KNOXVILLE

Case study: PaRSEC's task scheduling algorithm



Case study: PaRSEC's task scheduling algorithm



Thread Local Queues => High Locality
Overflow & Work Stealing => Load Balance

Parameter selection

Q1: How long should the local queues be?

Q2: Should a thread first steal from a close queue, any queue, or the shared queue?

Parameter selection

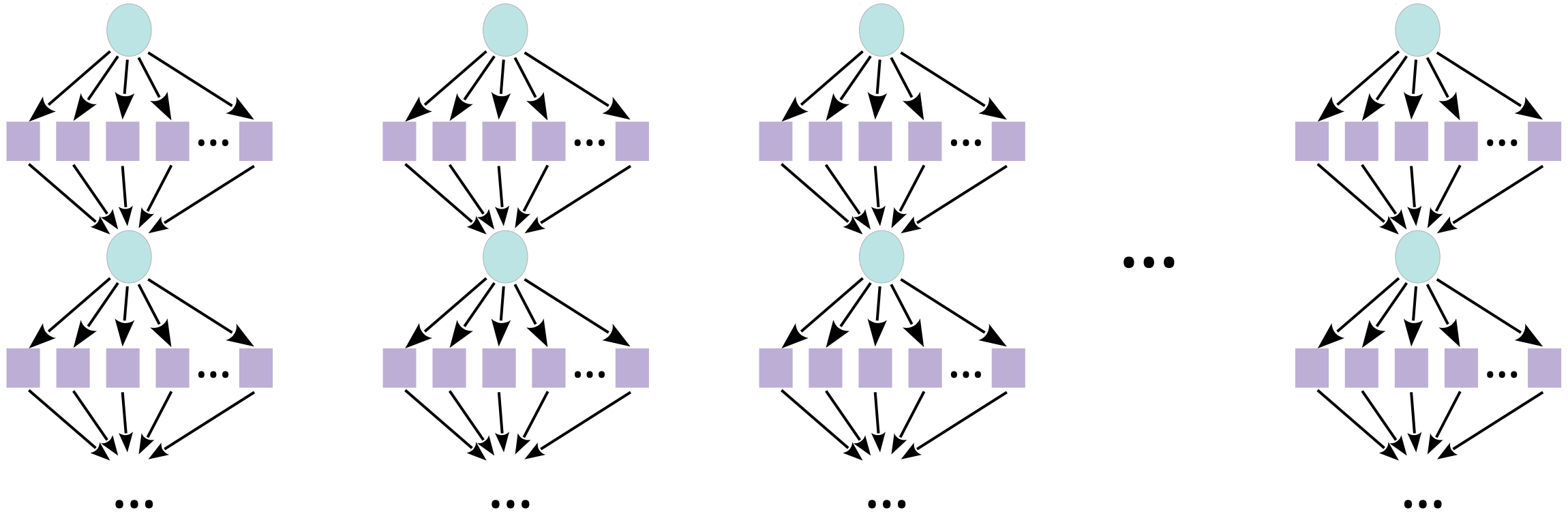
Q1: How long should the local queues be?

A: $4 * \text{Core_Count}$

Q2: Should a thread first steal from a close queue, any queue, or the shared queue?

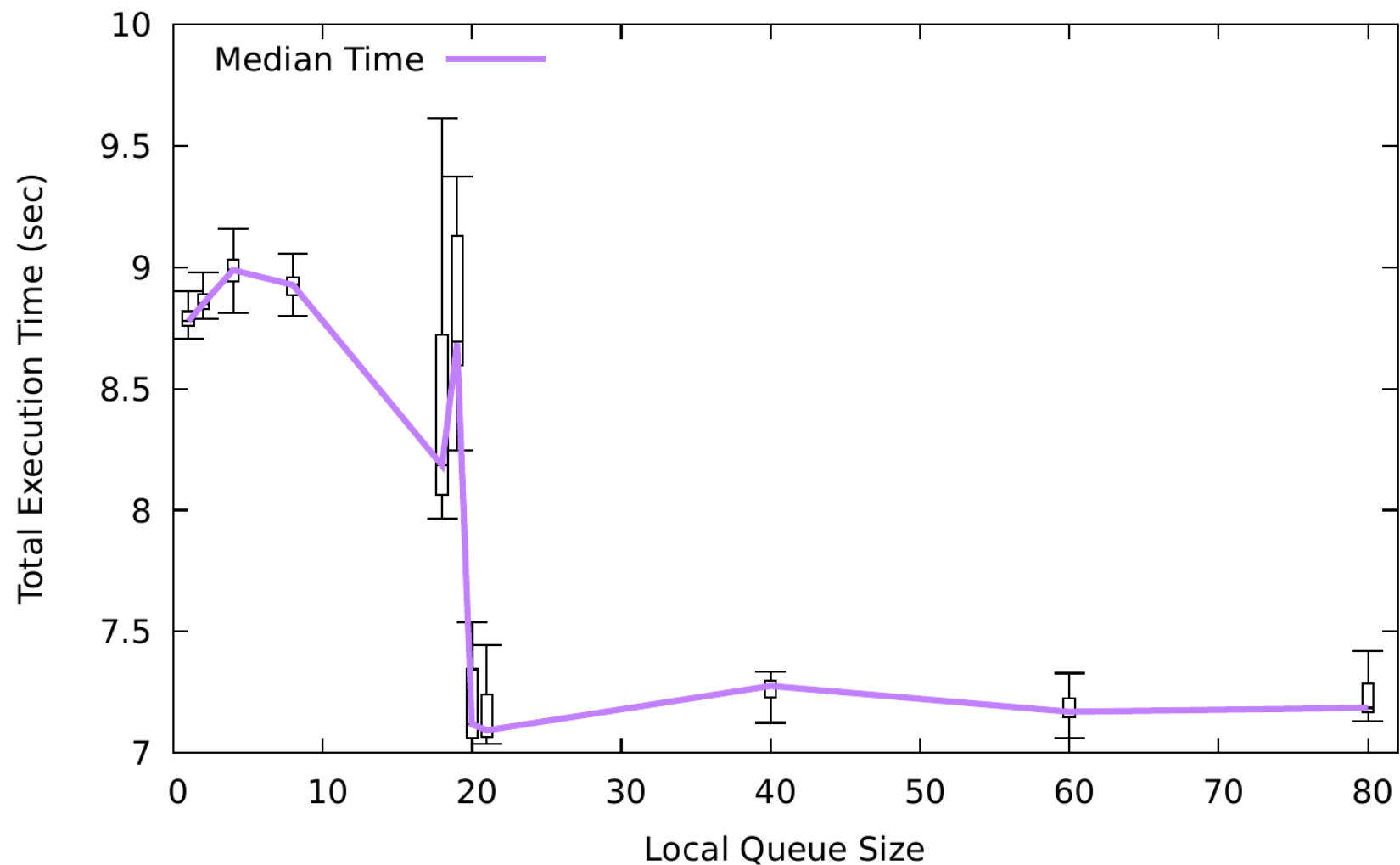
A: Any local queue (closest to farthest), then shared queue.

Testing Benchmark

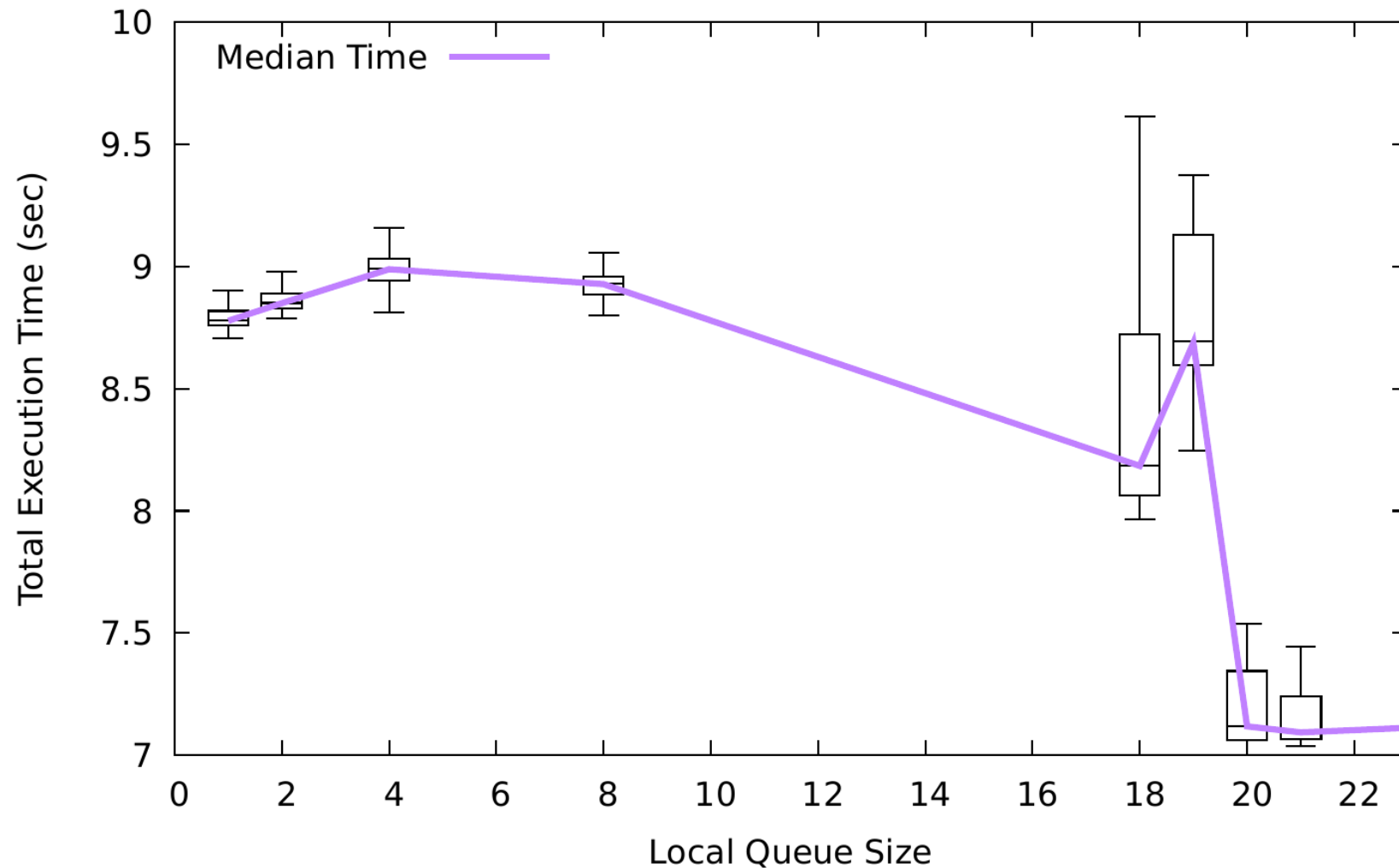


- 20 Independent Fork-Join chains x 20 (or 25) Tasks per fork.
- Memory bound kernel, with good cache locality.
- 20 Cores on testing node.

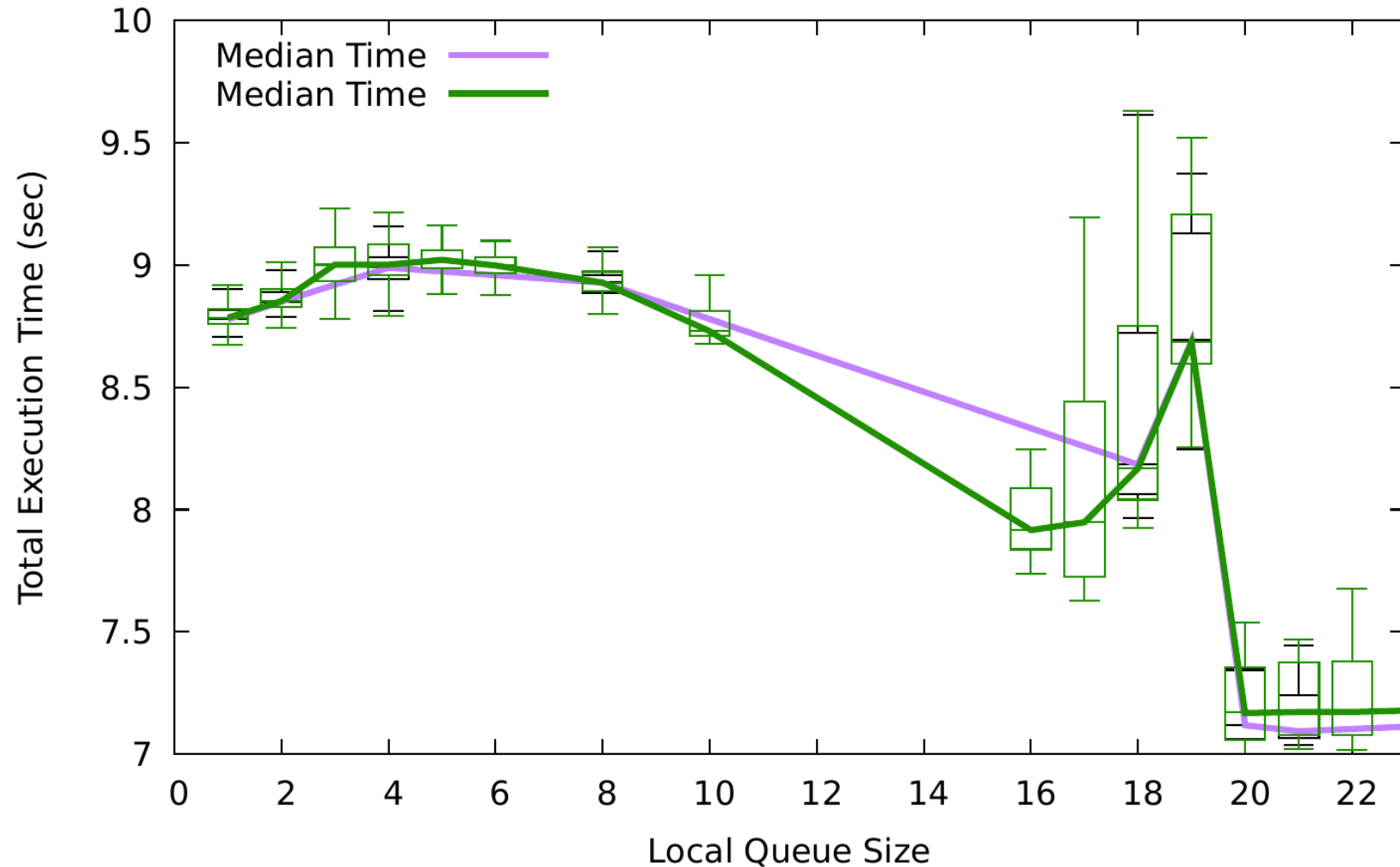
Execution time vs Local Queue Length



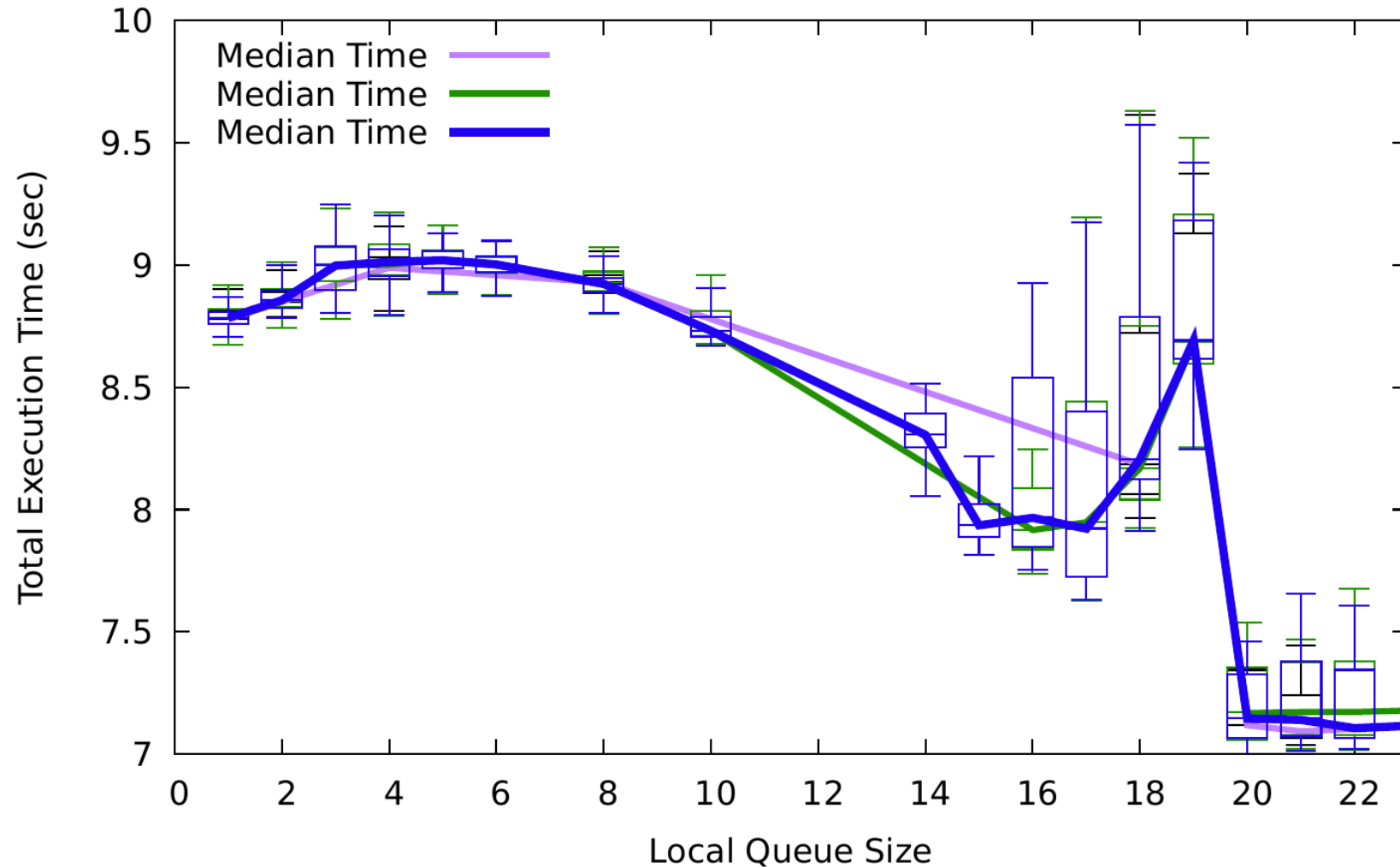
Execution time vs Local Queue Length (zoom)



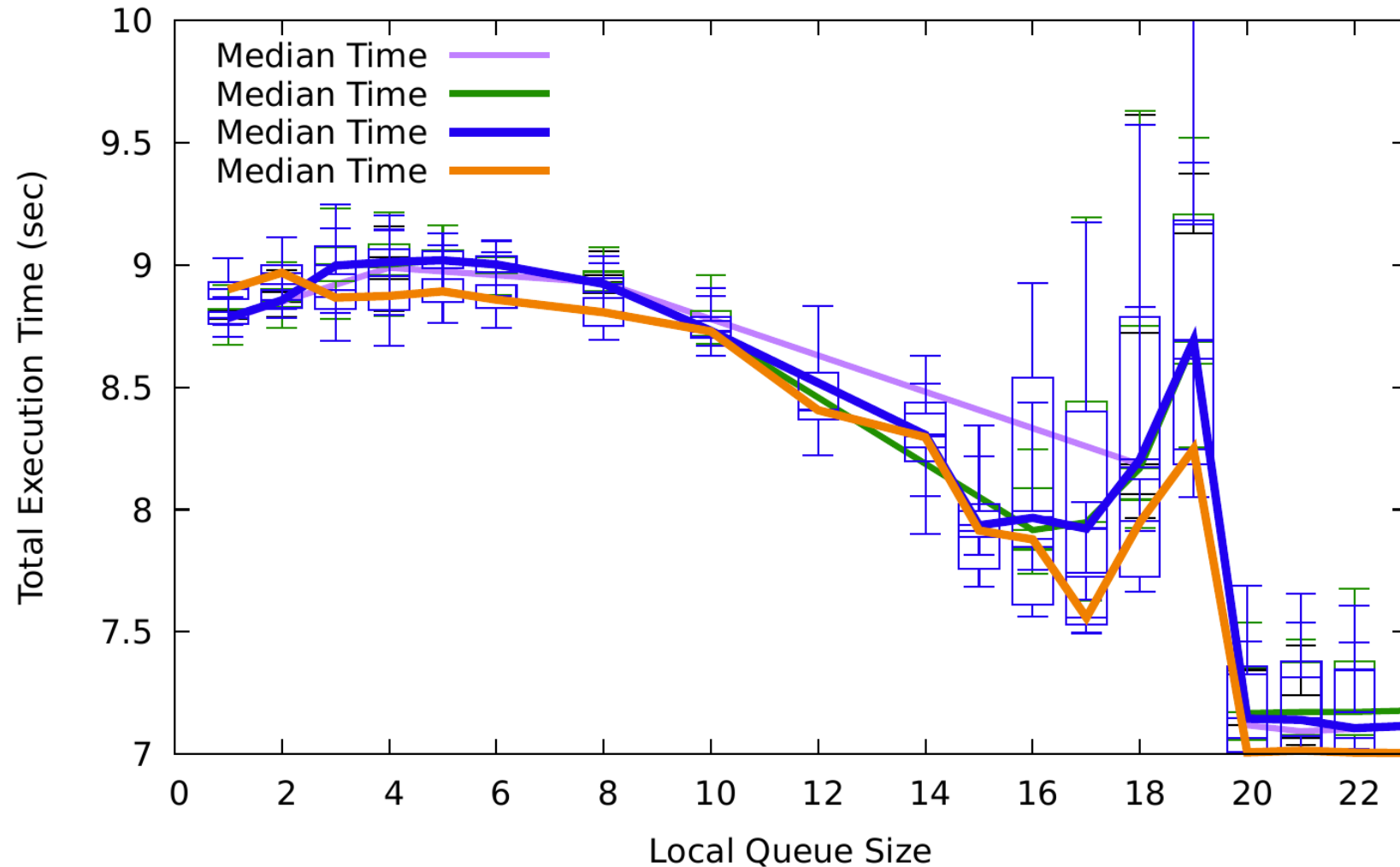
Execution time vs Local Queue Length (zoom 2)



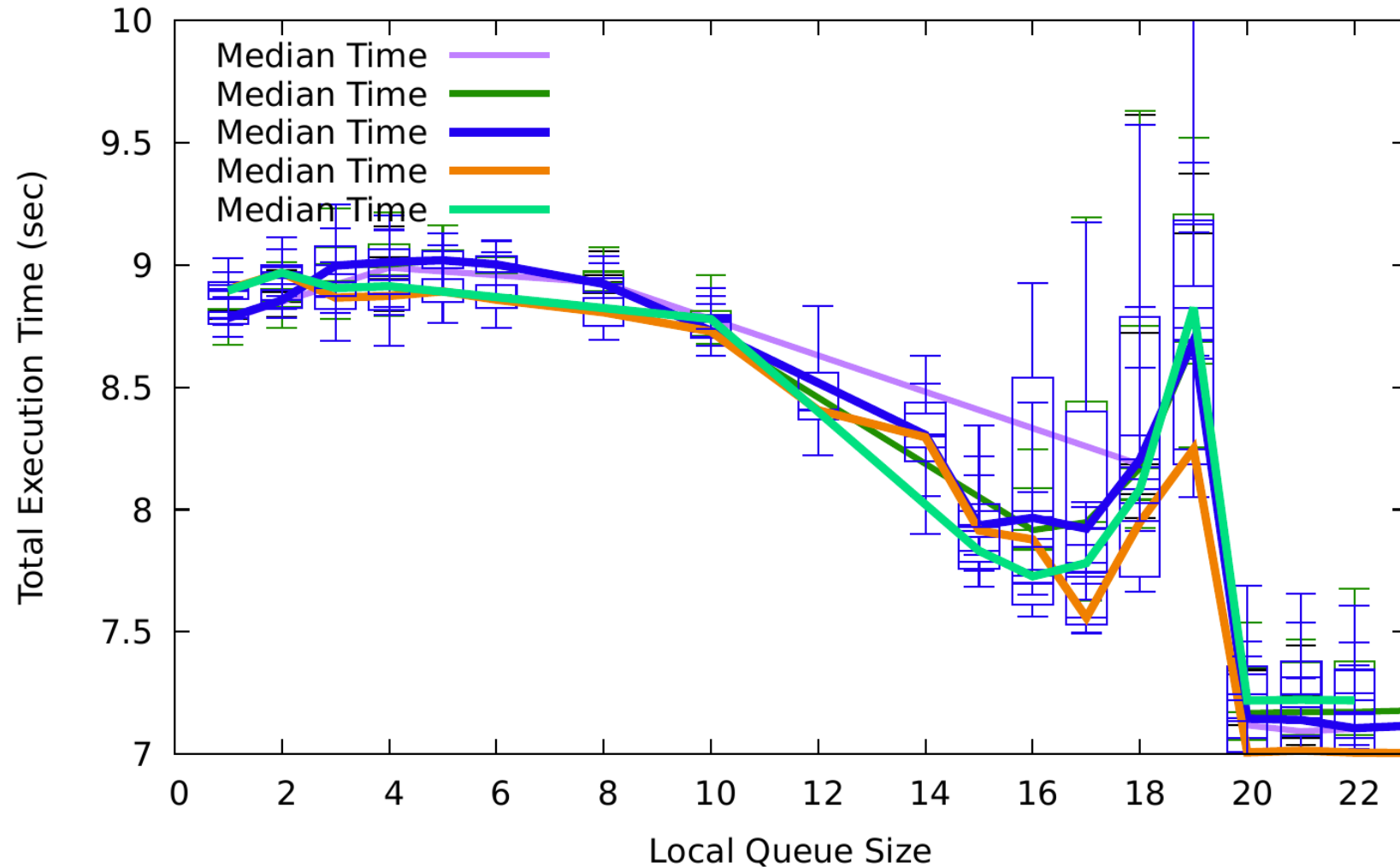
Execution time vs Local Queue Length (zoom 3)



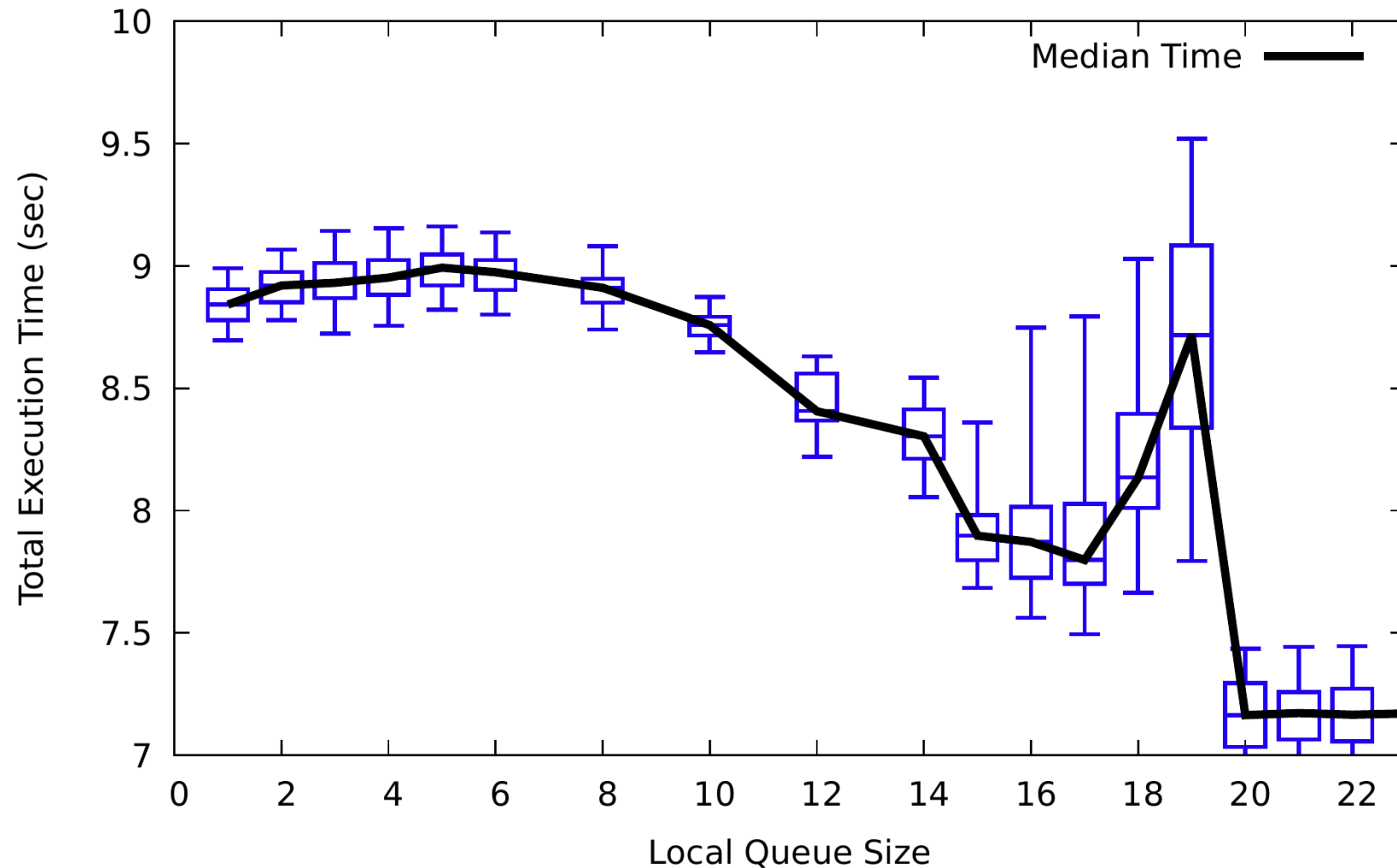
Execution time vs Local Queue Length (zoom 4)



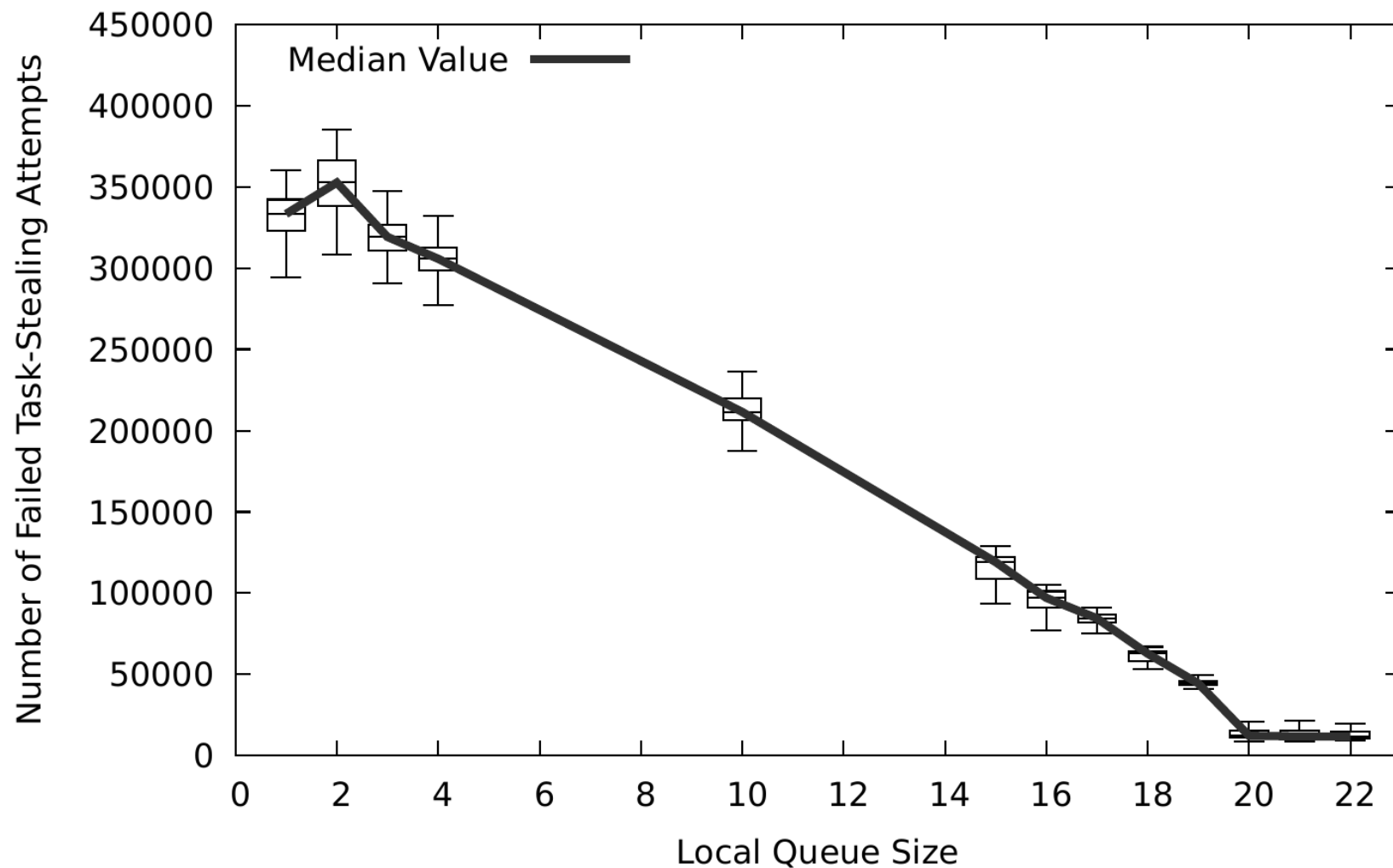
Execution time vs Local Queue Length (zoom 5)



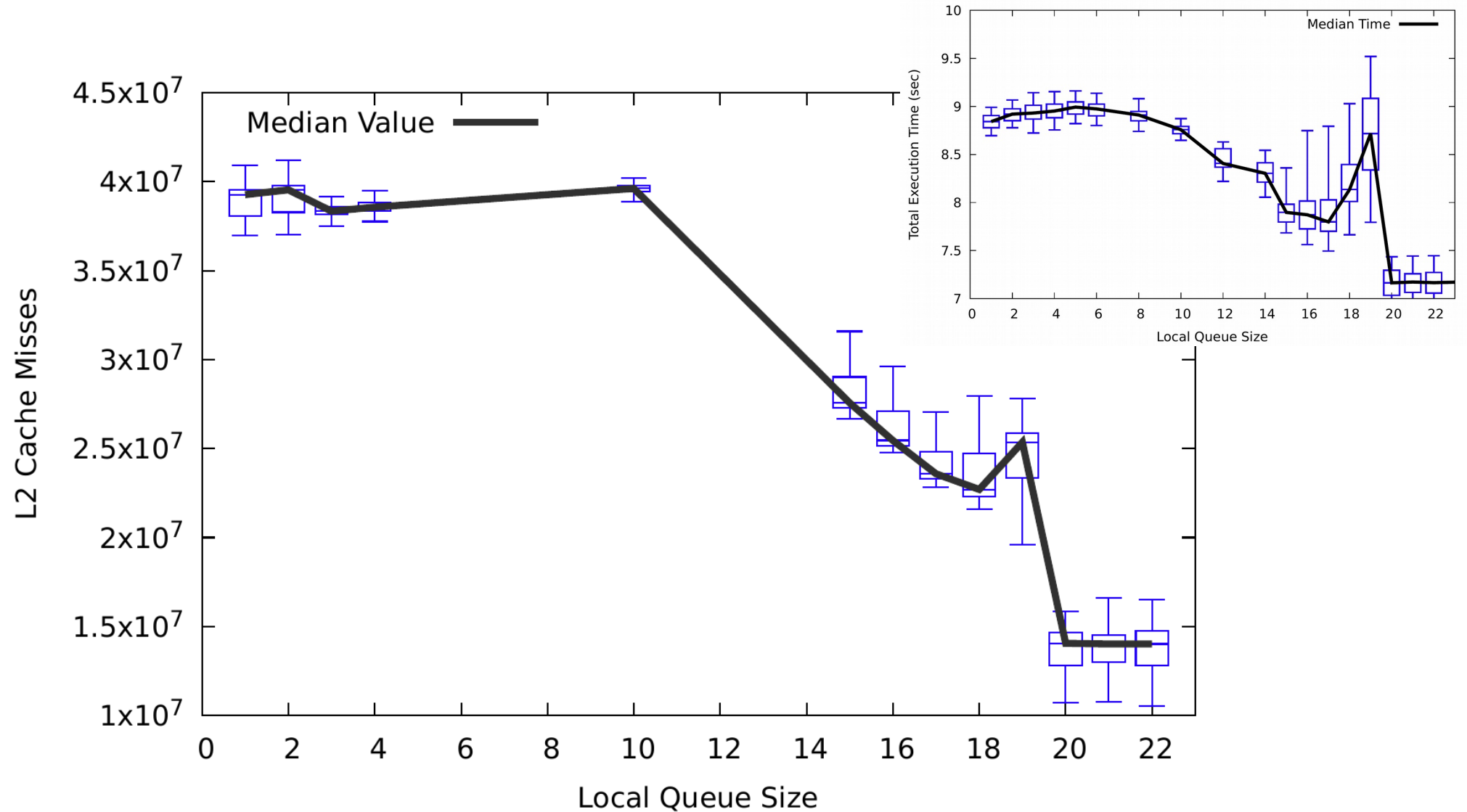
Execution time vs Local Queue Length (combined)



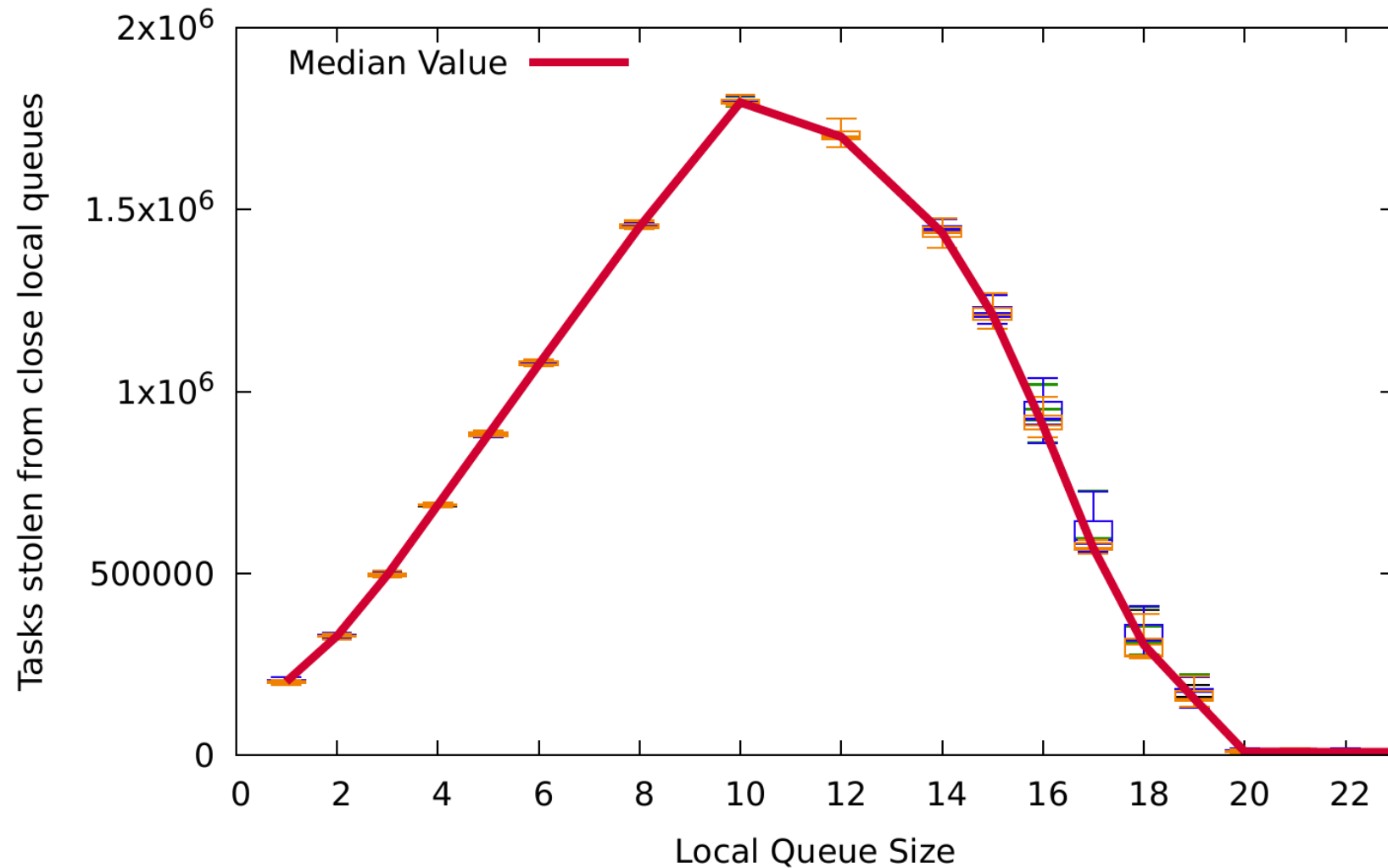
Failed Stealing Attempts



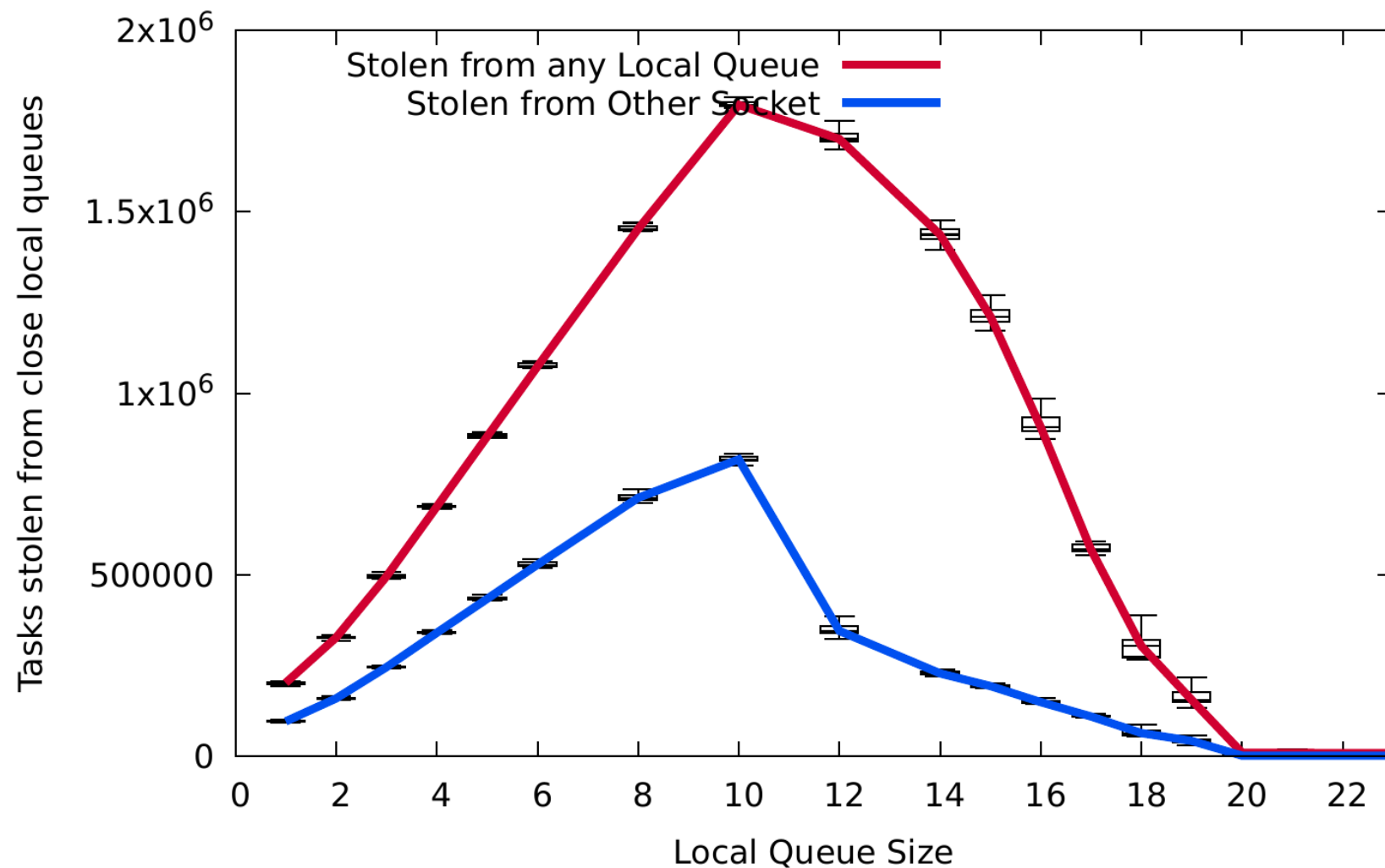
L2 Cache Misses (L3 show same pattern)



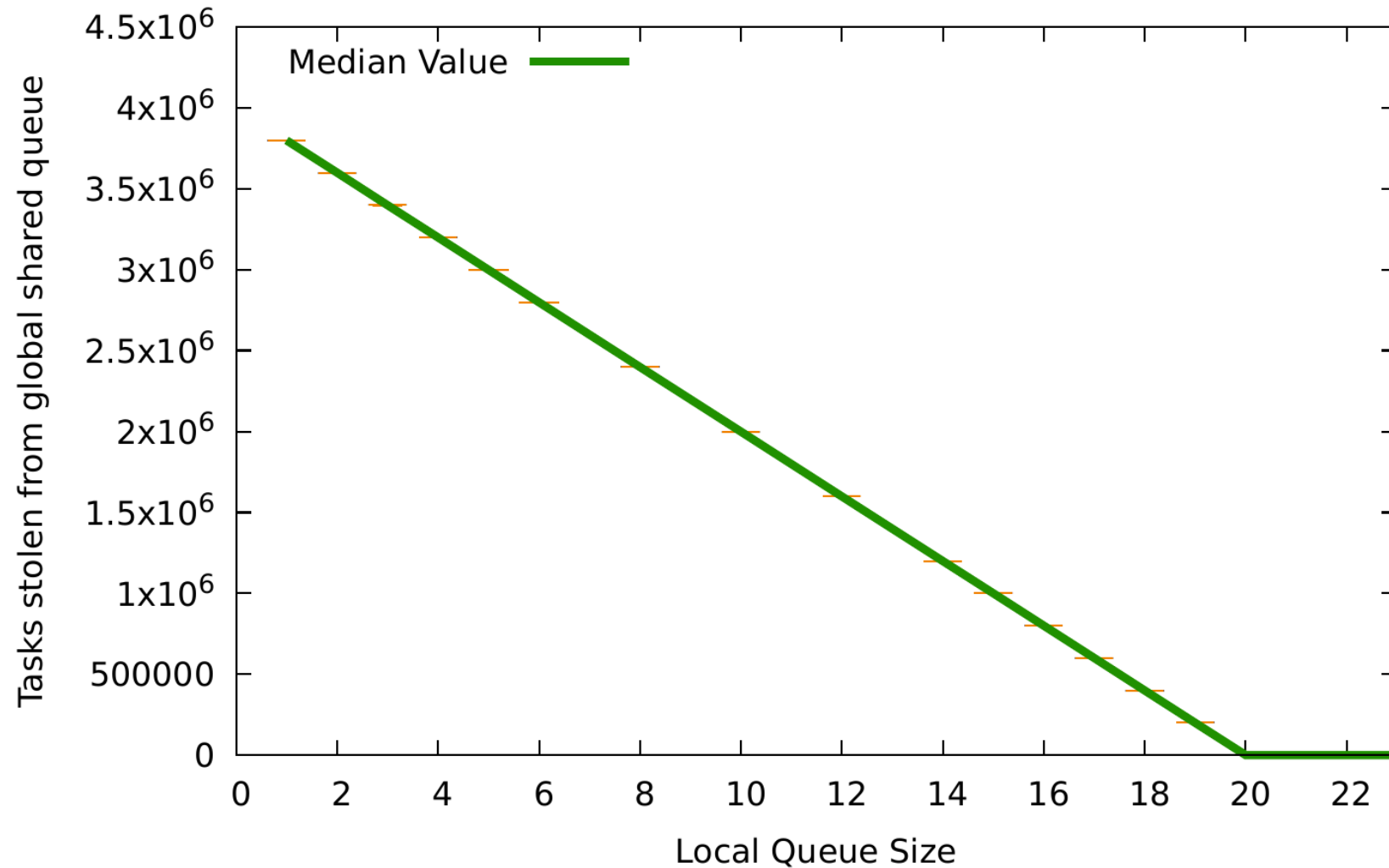
Successful Close Stealing



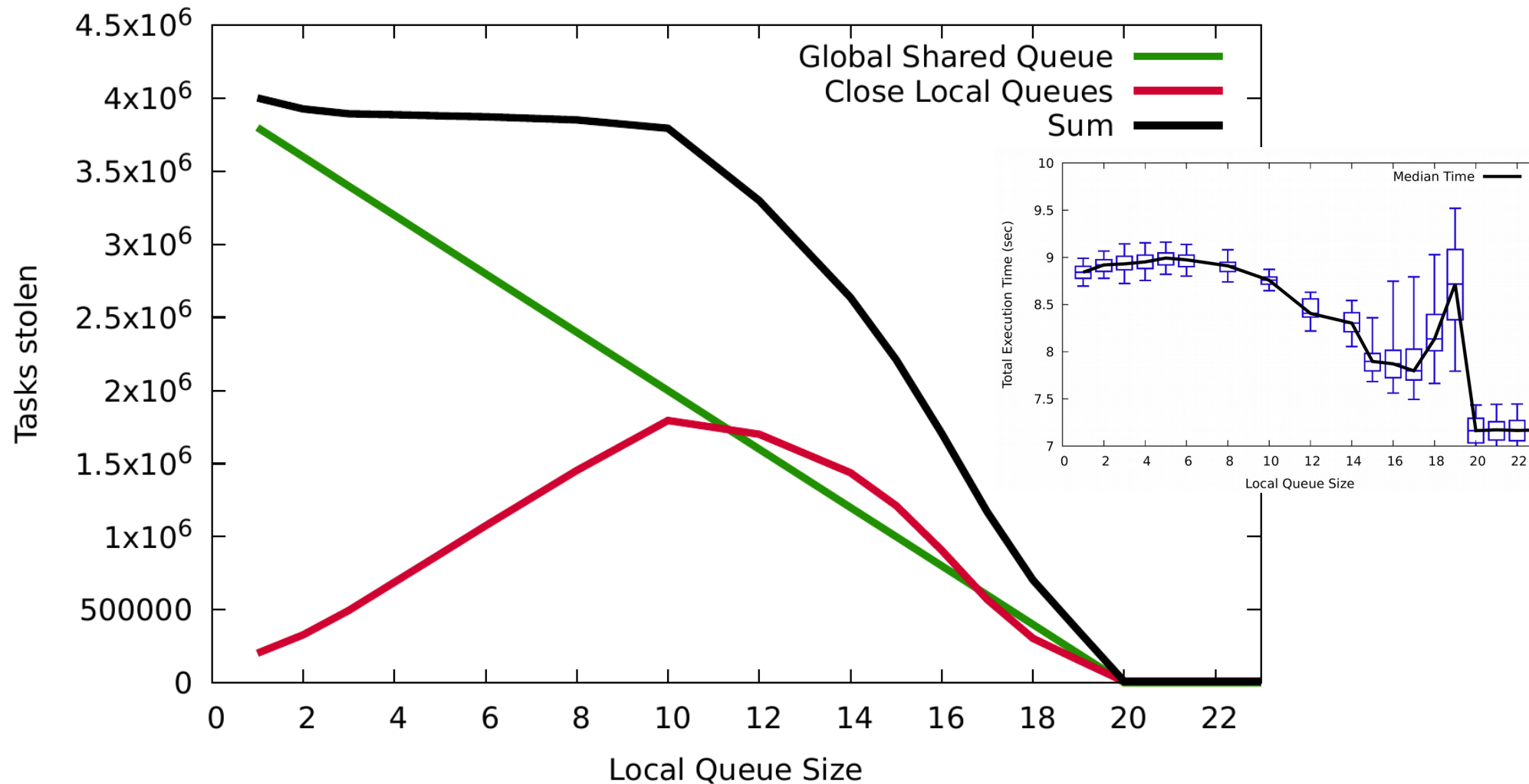
Successful Close & Far Stealing



Successful Shared Queue Stealing



Successful Local + Shared Queue Stealing



Unanswered questions

Q: So, what causes the bump?

Q: How did you measure all these things?

Unanswered questions

Q: So, what causes the bump?

A: I don't know!

Q: How did you measure all these things?

Unanswered questions

Q: So, what causes the bump?

A: I don't know!

Q: How did you measure all these things?

A: I am glad you asked.

What is missing from current infrastructure?

Events that occurred inside the software stack

There is no standardized way for a software layer to export information about its behavior such that other, independently developed, software layers can read it.

HPC Application	Quantum Chemistry Method
Math library	Distributed Factorization
Task runtime	Data Dependency
MPI	One Sided Communication
Libibverbs	RDMA completion

PAPI Software Defined Events

- De facto standard:

SDEs from your library can be read using the standard `PAPI_start()/PAPI_stop()/PAPI_read()`.

- Low overhead:

Performance critical codes can implement SDEs with zero overhead by exporting existing code variables without adding any new instructions in the fast path.

- Rich feature set:

PAPI SDE supports counters, groups, recordings, simple statistics, thread safety, custom callbacks.

The tool infrastructure is already there

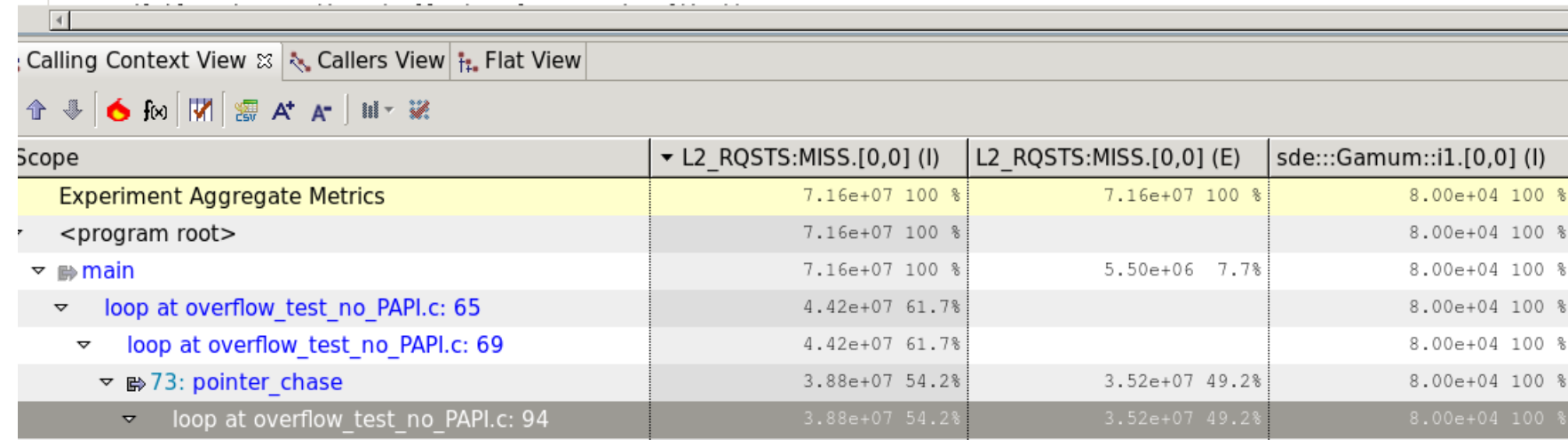
The screenshot displays a debugger window with a C source file named `overflow_test_no_PAPI.c`. The code is a test program designed to trigger a buffer overflow. It includes a `main` function that sets up a buffer, calculates a pointer offset, and enters a loop that writes data to the buffer. A `pointer_chase` function is also present, which updates a pointer and a sum. The code is annotated with comments explaining its purpose and the compiler's behavior. The debugger's `Calling Context View` is open, showing a table of performance metrics for various scopes.

```
1: int main(int argc, char **argv){
2:   int i, ret, len, pbb, code;
3:   uintptr_t *buff;
4:
5:   gamum_init();
6:
7:   /* Compute a weird value to mess up any algebraic simplification that the
8:    * compiler might do if we let everything to be provably zero. */
9:   junk = ((double)argc)/0.2254;
10:   for(i=0; i<DATA_SIZE; i++){
11:     junk_data[i] = 0.2468*junk*(double)+1.3774/((double)+1.2183);
12:   }
13:   junk2 = junk*1.234;
14:
15:   event cnt = 1;
16:   pbb = 64;
17:
18:   int buff_size = 3824*1024*1024/sizeof(uintptr_t);
19:   int stride = 218/sizeof(uintptr_t);
20:   int segment_size = pbb*4096/sizeof(uintptr_t); // default: 64 pages
21:   int LB = 163824;
22:   int UB = buff_size/stride;
23:   long long counter, values[3];
24:   buff = (uintptr_t *)malloc(buff_size*sizeof(uintptr_t));
25:   prepareArray_sections_random(buff, buff_size, stride, segment_size);
26:
27:   // Start work
28:   printf("##### main loop start\n");
29:   fflush(stdout);
30:
31:   for(len = LB; len < UB; len+=2){
32:     int j, rep;
33:
34:     printf("##### Starting len=%d\n", len);
35:     fflush(stdout);
36:     for(rep=0; rep < 4; rep++){
37:       for(j=0; j<DATA_SIZE; j++){
38:         junk = junk_data[j];
39:         pointer_chase(buff, len);
40:       }
41:     }
42:
43:     printf("##### main loop end\n");
44:     fflush(stdout);
45:     free(buff);
46:     printf("rst: %d\n", junk);
47:     return 0;
48:   }
49:
50:   void pointer_chase(uintptr_t *buff, int elem_count){
51:     int i, ret;
52:     uintptr_t *ptr;
53:     uintptr_t sum = 0;
54:
55:     ptr = (uintptr_t *)buff[0];
56:     if(i == 14094)
57:       gamum_work();
58:     ptr = (uintptr_t *) ptr;
59:     sum += (*ptr)*425361;
60:     junk += junk2*(double)sum/180125.7;
61:
62:     return;
63:   }
64:
65:   /* All sizes are in "uintptr_t" elements, NOT in bytes
66:    * Note: It is wise to provide an "array" that is aligned to the cache line size.
67:    */
68:   static void prepareArray_sections_random(uintptr_t *array, int len, int stride, int secSize){
69:     int elemCnt, maxElemCnt, sec;
70:     int curElemCnt, minIndex, takes;
71:     uintptr_t *p, *next;
72:     int curSecSize = secSize;
73:     int secCnt = 14*len/secSize;
74:     int *availableElements, remainingElemCnt;
75:
76:     p = (uintptr_t *)array[0];
77:     maxElemCnt = curSecSize/stride;
78:   }
79: }
```

Scope	L2_ROSTS-MISS [0.0] (I)	L2_ROSTS-MISS [0.0] (E)	sde::Gamum:~1 [0.0] (I)	sde::Gamum:~1 [0.0] (E)
Experiment Aggregate Metrics	7.16e+07 100 %	7.16e+07 100 %	8.00e+04 100 %	8.00e+04 100 %
<program root>	7.16e+07 100 %	7.16e+07 100 %	8.00e+04 100 %	8.00e+04 100 %
main	7.16e+07 100 %	5.50e+06 7.7%	8.00e+04 100 %	8.00e+04 100 %
loop at overflow_test_no_PAPI.c: 65	4.42e+07 61.7%		8.00e+04 100 %	8.00e+04 100 %
loop at overflow_test_no_PAPI.c: 69	4.42e+07 61.7%		8.00e+04 100 %	8.00e+04 100 %
73: pointer_chase	3.28e+07 45.8%	3.52e+07 49.2%	8.00e+04 100 %	8.00e+04 100 %
loop at overflow_test_no_PAPI.c: 94	3.52e+07 49.2%	3.52e+07 49.2%	8.00e+04 100 %	8.00e+04 100 %
overflow_test_no_PAPI.c: 97	3.52e+07 49.2%	3.52e+07 49.2%	8.00e+04 100 %	8.00e+04 100 %
96: gamum_work_	1.50e+06 2.1%	1.00e+05 0.1%		
97: killpg	1.00e+06 1.4%			
97: apic_timer_interrupt	7.00e+05 1.0%			
97: retint_signal	2.00e+05 0.3%			
96: papi_sde_inc_counter	1.00e+05 0.1%			
97: stub_0_sigreturn	1.00e+05 0.1%			
loop at overflow_test_no_PAPI.c: 70	5.40e+06 7.5%	5.40e+06 7.5%		
59: prepareArray_sections_random	2.68e+07 37.4%	8.00e+06 11.2%		
81: munmap	4.00e+05 0.6%			
loop at overflow_test_no_PAPI.c: 43	2.00e+05 0.3%	1.00e+05 0.1%		

The tool infrastructure is already there

```
94 for(i=0; i<elem count; i++){
95     if(0 == i%4096)
96         gamum_do_work ();
97     ptr = (uintptr_t *) *ptr;
98 }
99 sum += (*ptr)%425361;
100 junk += junk2+(double)sum/100125.7;
101
102 return;
103 }
104
105 /*
106 * All sizes are in "uintptr_t" elements, NOT in bytes
107 * Note: It is wise to provide an "array" that is aligned to the cache line size.
108 */
109 static void prepareArray_sections_random(uintptr_t *array, int len, int stride, int secSize){
110     int elemCnt, maxElemCnt, sec, i;
111     int currElemCnt, uniqIndex, taken;
112     uintptr_t **p, *next;
113     int currSecSize = secSize;
114     int secCnt = 1+len/secSize;
115     int *availableNumbers, remainingElemCnt;
116
117     p = (uintptr_t **)&array[0];
118
119     maxElemCnt = currSecSize/stride;
```



Calling Context View | Callers View | Flat View

↑ ↓ 🔥 f(x) 📄 CSV A+ A- 📊 🗑️

Scope	L2_RQSTS:MISS.[0,0] (I)	L2_RQSTS:MISS.[0,0] (E)	sde:::Gamum::i1.[0,0] (I)
Experiment Aggregate Metrics	7.16e+07 100 %	7.16e+07 100 %	8.00e+04 100 %
<program root>	7.16e+07 100 %		8.00e+04 100 %
main	7.16e+07 100 %	5.50e+06 7.7%	8.00e+04 100 %
loop at overflow_test_no_PAPI.c: 65	4.42e+07 61.7%		8.00e+04 100 %
loop at overflow_test_no_PAPI.c: 69	4.42e+07 61.7%		8.00e+04 100 %
73: pointer_chase	3.88e+07 54.2%	3.52e+07 49.2%	8.00e+04 100 %
loop at overflow_test_no_PAPI.c: 94	3.88e+07 54.2%	3.52e+07 49.2%	8.00e+04 100 %

Simplest SDE code (library side)

```
static long long local_var;  
  
void small_test_init( void ) {  
    local_var = 0;  
    papi_handle_t *handle = papi_sde_init ("TEST");  
    papi_sde_register_counter( handle, "Evnt",  
                               PAPI_SDE_RO|PAPI_SDE_DELTA,  
                               PAPI_SDE_long_long,  
                               &local_var );  
  
    ...  
}
```

SDE code for registering a callback function

```
sometype_t *data;  
  
void small_test_init( void ){  
    data = ...  
    papi_handle_t *handle = papi_sde_init ("TEST");  
    papi_sde_register_fp_counter(handle, "Evnt",  
                                PAPI_SDE_RO|PAPI_SDE_DELTA,  
                                PAPI_SDE_long_long,  
                                accessor, data);  
  
    ...  
}
```

SDE code for creating a counter (push mode)

```
void *counter_handle;
```

```
void small_test_init( void ) {
```

```
    papi_handle_t *handle = papi_sde_init("TEST");
```

```
    papi_sde_create_counter(handle, "Evt",  
                           PAPI_SDE_long_long,  
                           &counter_handle);
```

```
    ...
```

```
}
```

SDE code for creating a recorder (push mode)

```
void *recorder_handle;  
  
void small_test_init( void ) {  
    papi_handle_t *handle = papi_sde_init("TEST");  
    papi_sde_create_recorder(handle, "RCRDR",  
                             sizeof(double),  
                             cmpr_func_ptr,  
                             &recorder_handle);  
  
    ...  
}
```

SDE code for creating a recorder (push mode)

```
void *recorder_handle;  
sde::TEST::RCRDR  
void small_test_init( void ) {  
    papi_handle_t *handle = papi_sde_init("TEST");  
    papi_sde_create_recorder(handle, "RCRDR",  
                             sizeof(double),  
                             cmpr_func_ptr,  
                             &recorder_handle);  
  
    ...  
}
```

SDE code for creating a recorder (push mode)

```
void *recorder_handle;  
    sde::TEST::RCRDR  
void small_test_init::TEST::RCRDR::CNT  
    papi_handle_t *handle = papi_sde_init("TEST");  
    papi_sde_create_recorder(handle, "RCRDR",  
                             sizeof(double),  
                             cmpr_func_ptr,  
                             &recorder_handle);  
  
    ...  
}
```

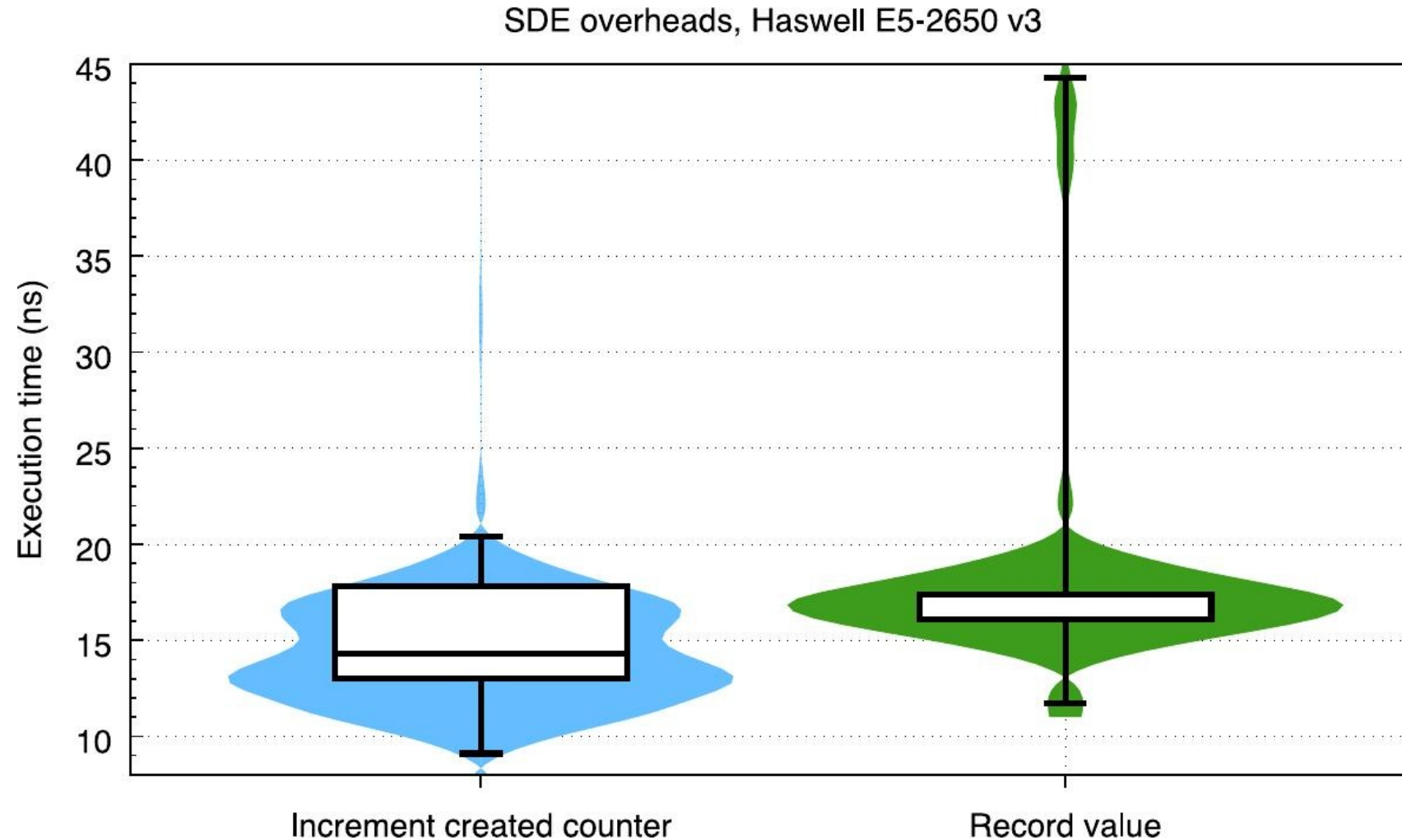

SDE code for creating a recorder (push mode)

```
void *recorder_handle;  
sde :: TEST :: RCRDR  
void small_test_init(void)  
papi_handle_t *handle = papi_sde_init("TEST");  
papi_sde_create_recorder(handle, "RCRDR",  
sde :: TEST :: RCRDR : Q1  
sde :: TEST :: RCRDR : MED  
sde :: TEST :: RCRDR : Q3  
...  
sde :: TEST :: RCRDR : MAX  
}
```

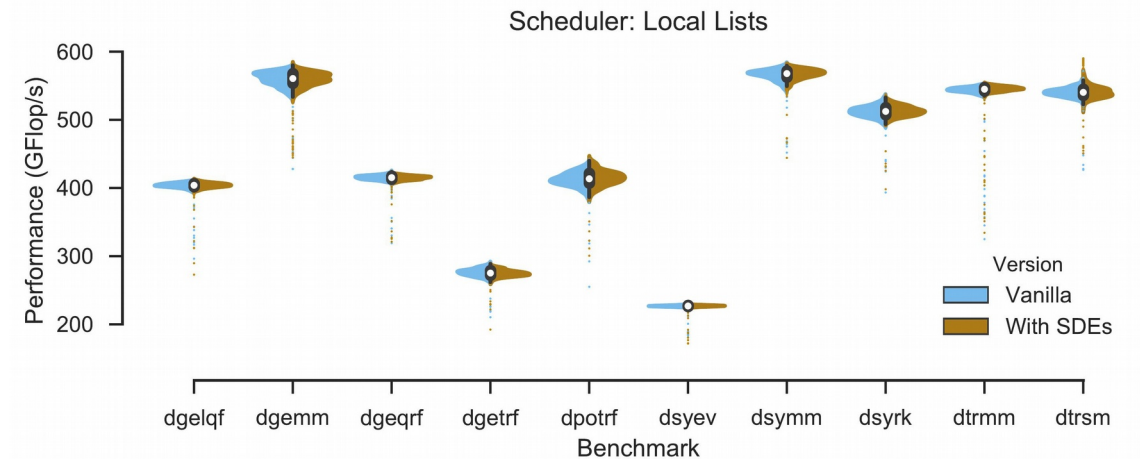
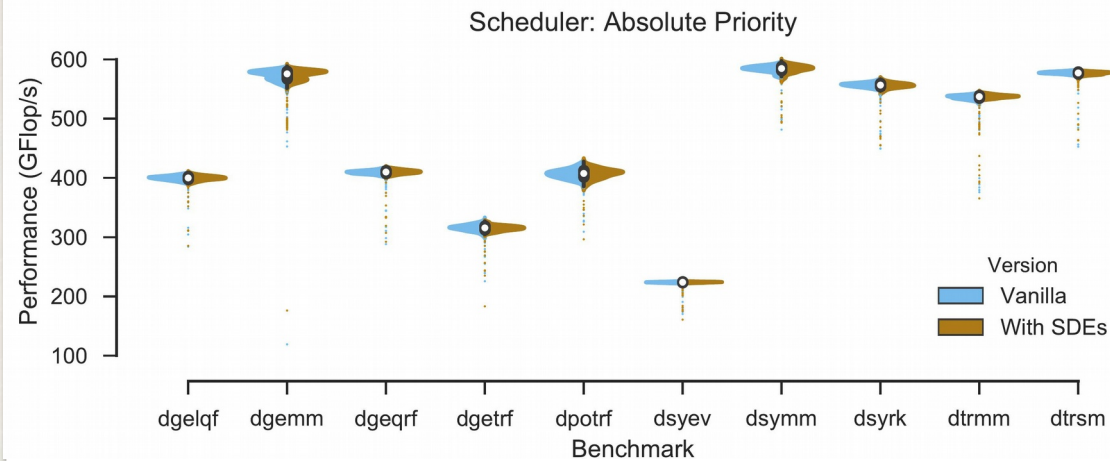
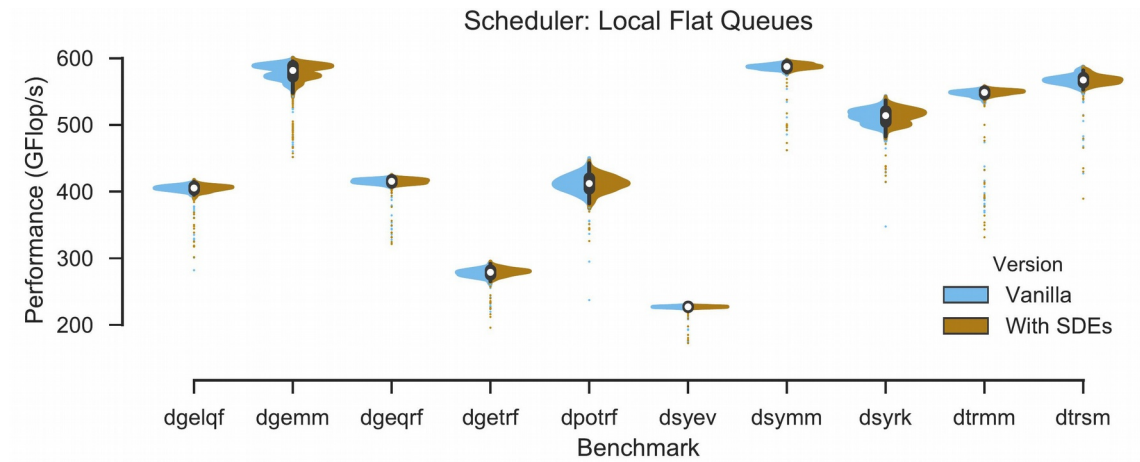
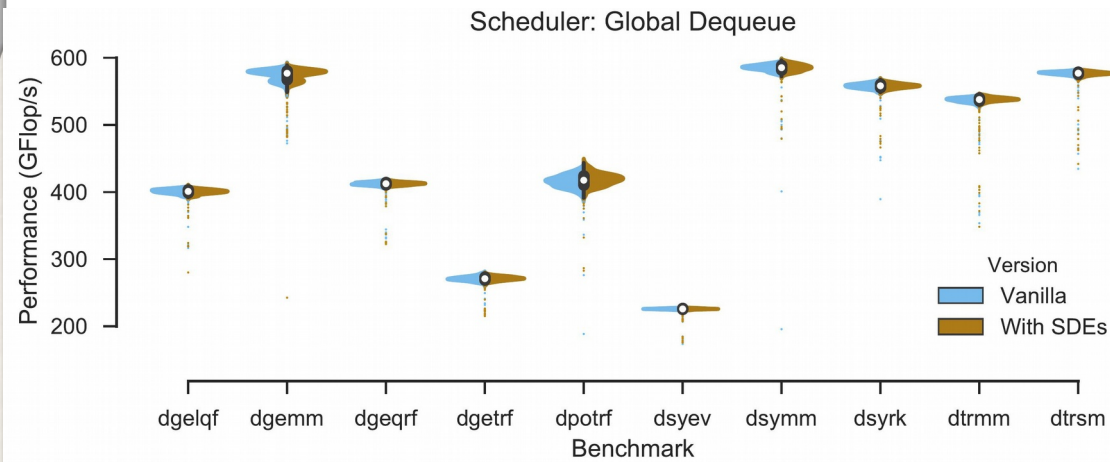
SDE code for updating created counters/recorders

```
void *counter_handle;  
void *recorder_handle;  
  
void push_test_dowork(void) {  
    double val;  
    long long increment = 3;  
  
    val = perform_useful_work();  
    papi_sde_inc_counter(counter_handle, increment);  
    papi_sde_record(recorder_handle, sizeof(val), &val);  
}
```

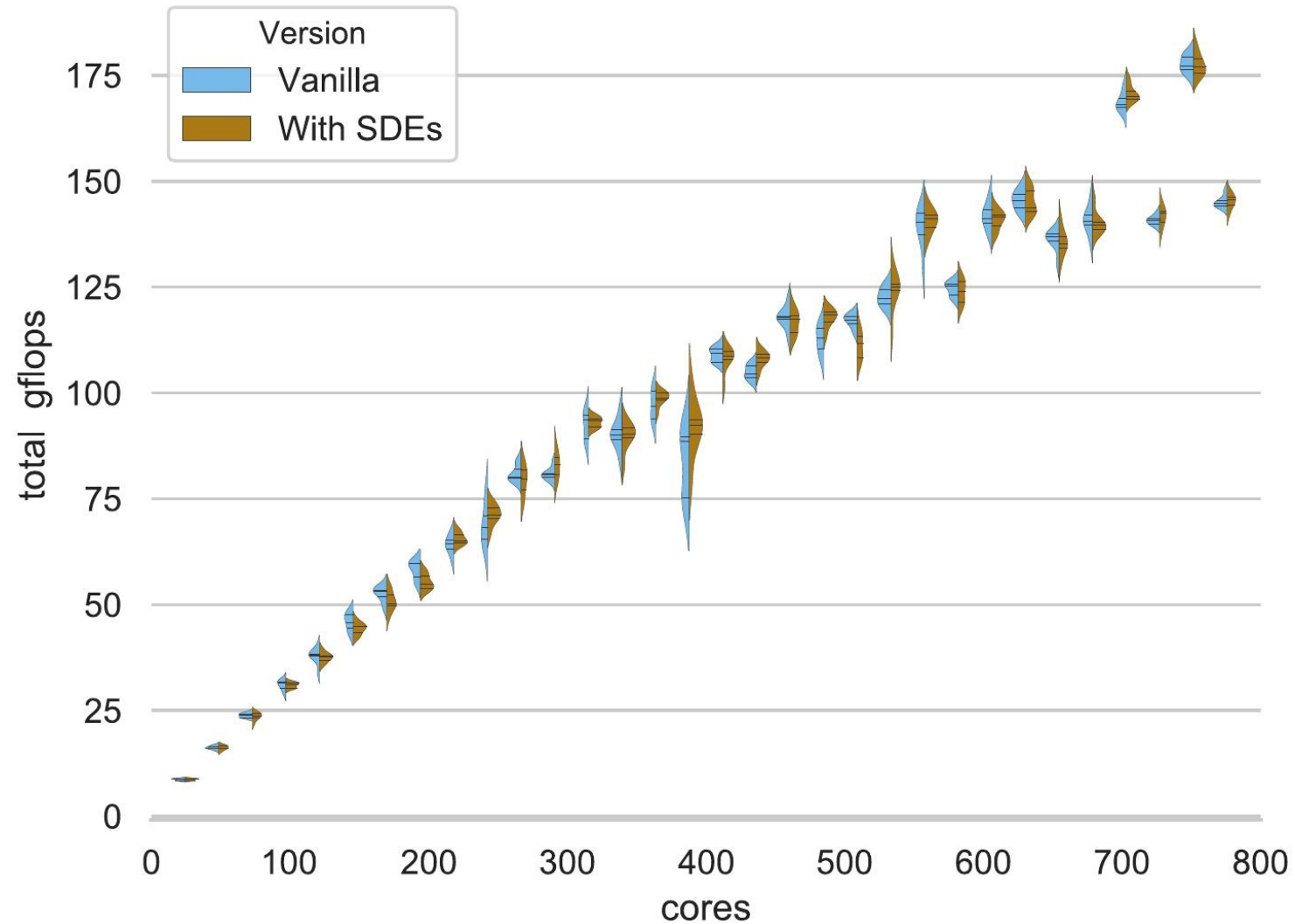
Performance overheads in simple benchmark



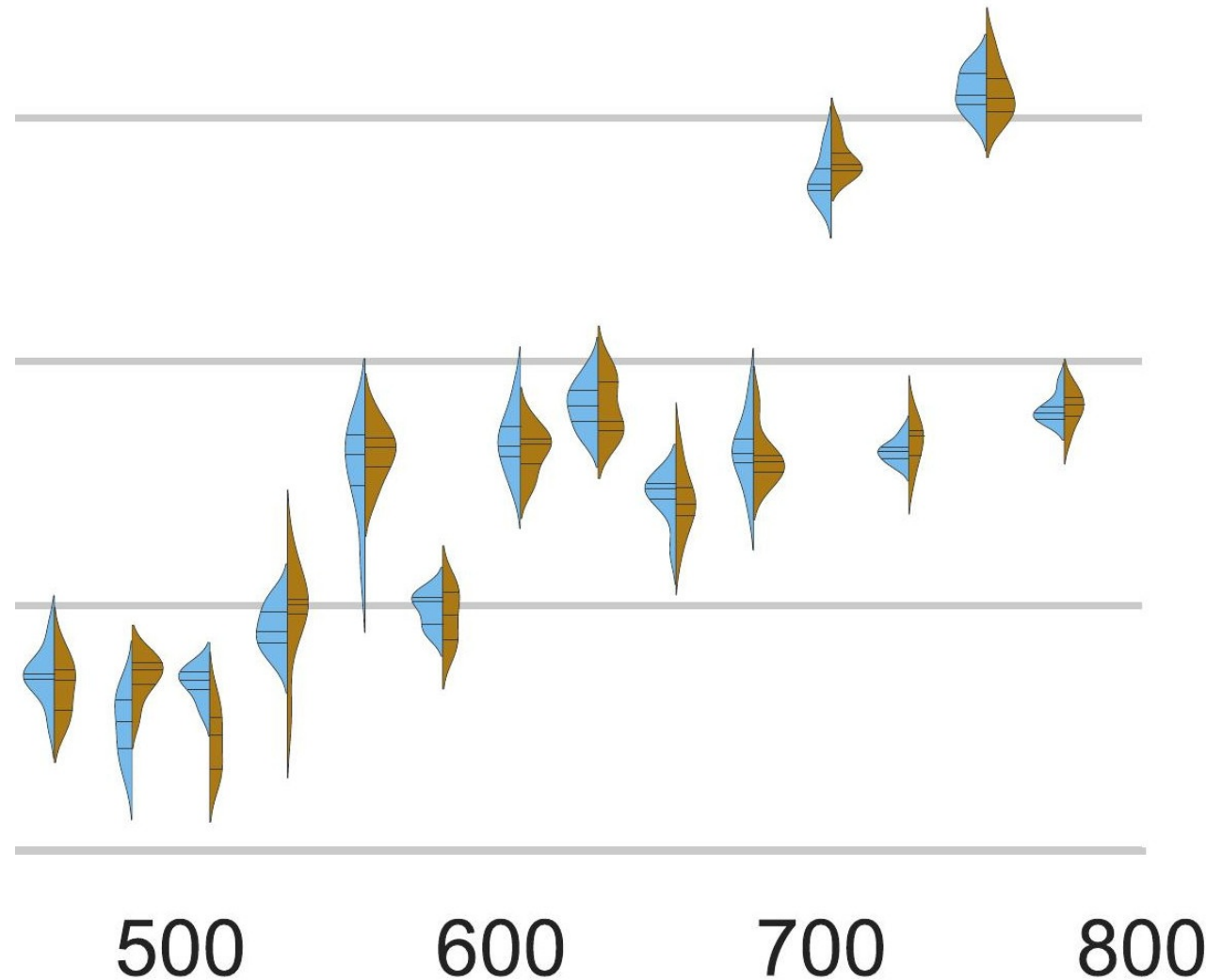
Performance overhead in PaRSEC



Performance overhead in HPCG



Performance overhead in HPCG (zoom)



Open Problem for our Community:

How do we associate useful context information with SDEs?

What meaningful information to associate with “TASKS_STOLEN”?

- ~~Code location~~
- Hardware events (e.g. cache misses)
- Patterns in history (e.g. last task before stealing event)
- Patterns in call-path/stack/originating thread

Conclusions

- Libraries/runtimes generate multiple useful software “events”.
- PAPI SDE allows any software layer to export events.
- SDEs can be read using the standard PAPI functionality.
- SDEs have minimal to **zero** performance overhead.
- SDEs might require different types of analysis by tools.