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
Case study: PaRSEC’s task scheduling algorithm

Core local queues

Shared Global queue (overflow)
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*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

![Graph showing execution time vs local queue length](image-url)
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

![Graph showing the relationship between local queue size and the number of failed task-stealing attempts. The median value is indicated by a straight line.](image-url)
L2 Cache Misses (L3 show same pattern)
Successful Close Stealing

[Graph showing the relationship between local queue size and tasks stolen from close local queues. The x-axis represents local queue size, ranging from 0 to 22, and the y-axis represents the number of tasks, ranging from 0 to 2x10^6. The graph peaks around a local queue size of 10, with median values indicated.]
Successful Close & Far Stealing

![Graph showing tasks stolen from close local queues vs. local queue size]

Key:
- Red line: Stolen from any Local Queue
- Blue line: Stolen from Other Socket

Y-axis: Tasks stolen from close local queues
X-axis: Local Queue Size
Successful Shared Queue Stealing

![Graph showing the relationship between tasks stolen from the global queue and local queue size.]
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?
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.

<table>
<thead>
<tr>
<th>HPC Application</th>
<th>Quantum Chemistry Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math library</td>
<td>Distributed Factorization</td>
</tr>
<tr>
<td>Task runtime</td>
<td>Data Dependency</td>
</tr>
<tr>
<td>MPI</td>
<td>One Sided Communication</td>
</tr>
<tr>
<td>Libibverbs</td>
<td>RDMA completion</td>
</tr>
</tbody>
</table>
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 tool infrastructure is already there
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

```c
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)

```c
void *counter_handle;

void small_test_init( void ){
    papi_handle_t *handle = papi_sde_init("TEST");
    papi_sde_create_counter(handle, "Evnt",
                              PAPI_SDE_long_long,
                              &counter_handle);

    ...
}
```
SDE code for creating a recorder (push mode)

```c
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)

```c
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)

```c
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);
...
}
```
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);
...
}
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);
}

SDE code for updating created counters/recorders
Performance overheads in simple benchmark

SDE overheads, Haswell E5-2650 v3

Execution time (ns)

Increment created counter

Record value
Performance overhead in PaRSEC

Scheduler: Global Dequeue

Scheduler: Local Flat Queues

Scheduler: Absolute Priority

Scheduler: Local Lists
Performance overhead in HPCG

![Graph showing performance overhead in HPCG with different versions and number of cores.](image)
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.