Whitepaper: Software-Defined Events (SDEs) in MAGMA-Sparse

ECP PEEKS: Production-ready, Exascale-Enabled, Krylov Solvers for Exascale Computing

Heike Jagode
Anthony Danalis
Hartwig Anzt
Ichitaro Yamazaki
Mark Hoemmen
Erik Boman
Stanimire Tomov
Jack Dongarra

Innovative Computing Laboratory, University of Tennessee
Sandia National Laboratory

December 20, 2018

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

Motivation

For domain scientists working with complex simulation codes, it is a burden to track down performance bottlenecks or numeric properties in the numeric software backends. In particular, if simulation codes utilize different numerical linear algebra (NLA) libraries in a recursive fashion, expert knowledge is necessary to extract algorithm-specific characteristics such as Sparse Matrix-Vector Product (SpMV) count, the residual or iteration count of a backend solver. At the same time, this information can be very useful to identify critical sections, optimize the code stack, or assess whether swapping a backend NLA library promises performance benefits. State-of-the-art is that collecting this information requires interacting with the low-level NLA libraries, potentially adding flags for verbose execution and/or recompilation. Even more daunting is the lack of a standard that specifies how this information can be accessed in iterative sparse linear algebra libraries, and the domain scientist has to browse the library-specific documentation and/or the code to identify the entry points.

This situation motivates to standardize the way how these algorithm-specific performance characteristics can be accessed as Software-Defined Events (SDEs). Precisely, we expose the SDEs through a standard API in the same way hardware counters are exposed. To that end we instrument the MAGMA-sparse library with counters from the PAPI performance library that is established as de-facto standard in the community for accessing hardware counters. This allows domain scientists to monitor the behavior of low-level linear algebra algorithms without the need of expert knowledge about the full software stack of the simulation code.

Design and Functionality

Using PAPI SDEs in a complex simulation code requires to initially include the PAPI header in the top-level driver, and define, how many SDEs will be exposed.

```
// include PAPI header
#include <papi.h>

// set the number of PAPI SDE events
#define NUM_PAPI_SDE_EVTS 13
```

Next, the PAPI SDEs are declared in the code. Here we declare a set of 13 SDEs for the MAGMA-sparse NLA backend. Acknowledging that “min”, “max”, “median”, and quantiles are not very meaningful for the residual of an iterative solver, we declare them to illustrate the powerfulness of PAPI SDEs. Also, we mention that “sde:::MAGMA::IterativeResidual_RCRD_D” is an SDE storing the complete residual history. This array is allocated by PAPI, and dynamically extended to meet the requirements. A performance analysis revealed the overhead of re-allocation being negligible.

```
/******************************************************************************
    Declaration of SDEs
******************************************************************************/

    const char *sde_event_name[] = {"sde:::MAGMA::numiter_I",
                                    "sde:::MAGMA::InitialResidual_D",
                                    "sde:::MAGMA::FinalResidual_D",
                                    "sde:::MAGMA::IterativeResidual_D",
                                    "sde:::MAGMA::SolverRuntime_D",
                                    "sde:::MAGMA::SpmvCount_I",
                                    "sde:::MAGMA::IterativeResidual_RCRD_D:CNT",
                                    "sde:::MAGMA::IterativeResidual_RCRD_D",
                                    "sde:::MAGMA::IterativeResidual_RCRD_D:MIN",
                                    "sde:::MAGMA::IterativeResidual_RCRD_D:Q1",
                                    "sde:::MAGMA::IterativeResidual_RCRD_D:MED",
                                    "sde:::MAGMA::IterativeResidual_RCRD_D:Q3",
                                    "sde:::MAGMA::IterativeResidual_RCRD_D:MAX"};
```

The initialization of the declared SDEs should only happen by default, but can easily be controlled via a environment variable. For the MAGMA-sparse library as backend, we use the environment variable PAPI_SDE_MAGMA.

```
/*****************************************************************************
    Initialization of SDEs
*****************************************************************************/

const char *is_papi_sde_on = getenv("PAPI_SDE_MAGMA");

    if ( is_papi_sde_on != NULL ) {
        retval = PAPI_library_init( PAPI_VER_CURRENT );
        if( retval != PAPI_VER_CURRENT ){
            fprintf( stderr, "PAPI_library_init failed: %d. Ensure that the
            %PAPI library used at run time is the same version used during
            %linking\n",PAPI_VER_CURRENT );
        }
        event_set = PAPI_NULL;
        retval = PAPI_create_eventset( &event_set );
        if( retval != PAPI_OK ){
            fprintf( stderr, "PAPI_create_eventset failed\n" );
        }

        for (j=0; j<NUM_PAPI_SDE_EVTS; j++) {
            retval = PAPI_event_name_to_code( (char *)sde_event_name[j],
            &event_codes[j] );
            if( retval != PAPI_OK ) {
                fprintf( stderr, "PAPI_event_name_to_code(%s)
                %failed.\n",sde_event_name[j]);
            }
        }

        retval = PAPI_add_events( event_set, event_codes, NUM_PAPI_SDE_EVTS );
        if( retval != PAPI_OK ){
            fprintf( stderr, "PAPI_add_events failed\n" );
        }
    }
```

Starting from this point, the SDE counters monitor the metrics in the algorithms invoked.

At the current stage, all Krylov subspace methods available in MAGMA-sparse are instrumented with PAPI SDEs. Also the LOBPCG eigensolver and the Iterative Refinement (IterRef) method are supporting PAPI SDEs. See Appendix for a complete list of the algorithms in MAGMA-sparse instrumented with PAPI SDEs.

From now on, the SDEs can be accessed as a standard PAPI counter, for example as environment variable or inside the top-level driver.

```
/*******************************************************************************
    Access SDE values
*******************************************************************************/

        if ( is_papi_sde_on != NULL ) {
            retval = PAPI_stop( event_set, values );
            if( retval != PAPI_OK ){
                fprintf( stderr, "PAPI_stop failed\n" );
            }
            printf(">>>> PAPI counter report:\n");
            printf("    %s: %lld\n",sde_event_name[0], values[0]);
            printf("    %s: %.4e\n",sde_event_name[1],
            %%GET_DOUBLE_SDE(values[1]));
            printf("    %s: %.4e\n",sde_event_name[2],
            %%GET_DOUBLE_SDE(values[2]));
            printf("    %s: %.4e\n",sde_event_name[3],
            %%GET_DOUBLE_SDE(values[3]));
            printf("    %s: %.4e\n",sde_event_name[4],
            %%GET_DOUBLE_SDE(values[4]));
            printf("    %s: %lld\n",sde_event_name[5], values[5]);
            printf("    %s: %lld\n",sde_event_name[6], values[6]);

            double *sde_ptr = GET_SDE_RECORDER_ADDRESS(values[7], double);
            for (j=0; j<values[6]; j++){
                printf("    %d: %.4e\n",j, sde_ptr[j]);
            }

            printf("\nPAPI SDE recorder statistics:\n");
            sde_ptr = GET_SDE_RECORDER_ADDRESS(values[8], double);
            printf("    %s: %.4e\n",sde_event_name[8], *sde_ptr);
            sde_ptr = GET_SDE_RECORDER_ADDRESS(values[9], double);
            printf("    %s: %.4e\n",sde_event_name[9], *sde_ptr);
            sde_ptr = GET_SDE_RECORDER_ADDRESS(values[10], double);
            printf("    %s: %.4e\n",sde_event_name[10], *sde_ptr);
            sde_ptr = GET_SDE_RECORDER_ADDRESS(values[11], double);
            printf("    %s: %.4e\n",sde_event_name[11], *sde_ptr);
            sde_ptr = GET_SDE_RECORDER_ADDRESS(values[12], double);
            printf("    %s: %.4e\n",sde_event_name[12], *sde_ptr);
            printf("<<<< PAPI counter report END\n");
        }
```

Usage Example

Using SDEs allows to easily monitor the characteristics and behavior of the NLA backend algorithms for a specific problem handled by the top-level application.

In Figure 1 we illustrate how the convergence of Krylov solvers can be visualized with the help of PAPI SDEs. This allows the domain scientist to quickly identify the method of choice for this specific problem.

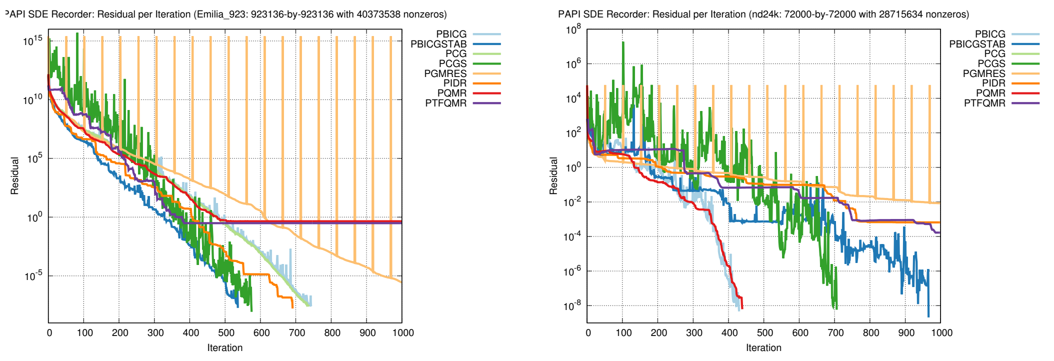

Figure 1: Convergence of different Jacobi-preconditioned MAGMA-sparse Krylov solvers. Test matrices are taken from the Suite Sparse Matrix Collection.

Appendix

MAGMA-sparse algorithms instrumented with PAPI SDEs

* BiCG	Biconjugate Gradient Method
* BiCGStab	Stabilized Biconjugate Gradient Method
* CG	Conjugate Gradient Method
* CGS	Conjugate Gradient Squares Method
* GMRES	Generalized Minimal Residuals
* IDR	Induced Dimension Reduction Solver
* IterRef	Iterative Refinement
* LOBPCG	LOcally Optimal Block Preconditioned CG (Eigensolver)
* QMR	Quasi-Minimal Residual
* TFQMR	Transpose-free Quasi-Minimal Residual

List of currently supported PAPI SDEs

```
* numiter = Number of iterations until convergence attained (I=integer)
* InitialResidual = Initial residual (D=double)
* FinalResidual = Final residual (D=double)
* IterativeResidual = Iterative residual (D=double)
* SolverRuntime = Total run-time of the solver (D=double)
* SpmvCount = Number of sparse matrix-vector multiplications (SpMV) (I=integer)
* IterativeResidual_RCRD = Array of all residuals until convergence
    (RCRD=recorder) (D=double)
```