ESSEX: Sparse iterative solvers, asynchronicity and fault tolerance

Faisal Shahzad, Moritz Kreutzer and Gerhard Wellein
FFMK workshop „Application Interfaces for an Exascale OS”
December 8, 2014
ESSEX - Equipping Sparse Solvers for Exascale
Equipping Sparse Scalable Solvers for Exascale (ESSEX)

Hardware
Fault tolerance
Energy efficiency
New levels of parallelism

Quantum Physics Applications
Extremely large sparse matrices:
eigenvalues, spectral properties,
time evolution

ESSEX

Exascale Sparse Solver Repository (ESSR)

FT concepts, programming for extreme parallelism
Sparse eigensolvers, preconditioners, spectral methods
Quantum physics / chemistry
ESSEX applications: Graphene, topological insulators, ...

Quantum Physics Applications

New levels of parallelism

Fault tolerance
Energy efficiency
ESSEX: “Co-Design” oriented project

Holistic Performance Engineering

- Applications
- Computational Algorithms
- Building Blocks

Fault Tolerance

Problem formulation, Quantum State Encoding (QSE)

Implementation, data structure, parallelization

Code/Energy Efficiency
Basic building blocks library: GHOST
General, Hybrid and Optimized Sparse Toolkit

- Basic tailored sparse matrix / vector operations
- CRS or SELL-C-σ* (unified format) storage schemes
- (Block-)SpMVM: SIMD intrinsic (AVX, SSE, MIC) & CUDA kernels
- Dense vector /matrices: row-/column-major storage

- **Supports** data & task parallelism (up to application level)
- MPI + OpenMP + tasks for concurrent execution
- Generic and hardware-aware task management

- Application layer triggered checkpoint / restart
- Asynchronous checkpointing via tasks
- Various checkpoint locations (node, filesystem)

Our (ESSEX) effort –
get a simple solution first –
application driven –
No silver bullet
1. **Iterative Solvers**
   - Kernel Polynomial Method (KPM)
   - Jacobi Davidson (JADA)

2. **Asynchronicity**
   - Handling node-level parallelism

3. **Fault Tolerance and Checkpointing/Restart**
ESSEX: Computational challenges / methods

Cover most aspects of large sparse eigenvalue problem

Compute approximation to complete eigenvalue spectrum of large sparse matrix $A$ (with $X = I$)


*The kernel polynomial method*
ESSEX: Start with simple but efficient iterative algorithms ("Kernel Polynomial Method")

```
for r = 0 to R-1 do
    |v⟩ = |rand()⟩;
    Initialization &
    computation of μ₀, μ₁
    for m = 1 to M/2 do
        swap(|w⟩, |v⟩);
        |u⟩ = H|v⟩;
        |u⟩ = |u⟩ - b|v⟩;
        |w⟩ = -|w⟩;
        |w⟩ = |w⟩ + 2a|u⟩;
        η₂m = ⟨v|v⟩;
        η₂m+1 = ⟨w|v⟩;
    end
end
```

Application: R random configurations (R=1,...,10²) or iterative loop

Algorithm: Compute Chebyshev moments

Basic building blocks: spMVM and sparseBLAS1

Checkpoint data: 2 vectors
Constant sparse matrix (H) – recompute

KPM approach can be implemented with only one global communication step
ESSEX: Iterative Eigensolvers – BJDQR

- Blocked Jacobi-Davidson QR Method

1: Setup initial subspace
2: while not converged do
3: Project the problem to a small subspace
4: Solve the small eigenvalue problem
5: Calculate an $n_b$ approximations and their residual
6: Lock converged eigenvalues
7: Shrink subspace if required (RESTART)
8: Approximately solve the $n_b$ correction equations
9: Block-Orthogonalize the new directions
10: Enlarge the subspace
11: end while
Converged eigenvectors (**LOCK**): incr. async. Backup in DP

Expanding subspace, full async. backup in SP after **RESTART**

**RESTART**: shrink subspace to \( j_{\text{min}} \) vectors
Basic parallelization approach: MPI + X

Motivation – Use X for many tasks at the same time
- Asynchronous Communication
- Asynchronous Checkpointing
- Concurrent worker teams (CPUs)
- Accelerators

Easy to use, low overhead, hardware-aware

*General, Hybrid and Optimized Sparse Toolkit
Developed by: Moritz Kreutzer, RRZE
Modern compute nodes

- Heterogeneous architectures on a single node

- Differ in programming paradigm
  - CPU: only native mode
  - GPU: only accelerator mode
  - Xeon Phi: accelerator or native mode

- Differ in performance

PU = processing unit
Work distribution

- Distinction between architectures via MPI processes:
  - exactly one process per GPU
  - at least one process per Xeon Phi
  - at least one process for all CPUs

- Each process gets assigned a weight deciding the share of work which depends on their relative performance

- Resource management:
  - Each process running inside an exclusive CPU set (no shared cores)
  - CPU sets may span several NUMA nodes
Example work distribution

- Amount of processes is the minimum: 3
- GPU is managed by a full core on the nearest socket
- CPU process spans two NUMA nodes
- Xeon Phi operated in native mode
  - one MPI process running on the coprocessor
Degrees of freedom in work distribution

- More than one CPU processes
  - maybe favorable in order to avoid NUMA problems: one per NUMA node
  - one process per core/PU also possible (in case OpenMP is not present/desired)

- Number of PUs for GPU management
  - at least one, but the other PU on the same core may have problems in this case

- More than one process on the Xeon Phi
Resource management

- Each process stores idle/busy states and locality information of each of its PUs (e.g. for initial state of CPU process)

<table>
<thead>
<tr>
<th>NUMA node</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- One shepherd thread will be created per PU:

- The shepherd threads wait for tasks to be put in the task queue
Task processing

- A task is defined by
  1. A function callback along with parameters
  2. The number of PUs to process the task
  3. The preferred NUMA node to process the task
  4. Additional flags

- Available flags (can be combined):
  - PRIO_HIGH: Put task to beginning of queue.
  - NODE STRICT: Execute task only on the given NUMA node.
  - NOT ALLOW CHILD: Do not allow a child task to use the task’s PUs.
  - NOT PIN: Do not register the task in the PU map.

- All tasks line up in a single queue
Simple tasking example
Asynchronous checkpoints via GHOST-task thread:

Parent task

Checkpoint task

ghost_task_create(ckpt_task_ptr, &CP_func, CP_obj,...)

ghost_task_enque(ckpt_task_ptr);

ghost_task_wait(ckpt_task_ptr);

update_CP(CP_obj);
//async. copy of CP is updated

CP_obj:
- object of ckpt_t type
- ckpt_t class is defined by programmer
- checkpoint object contains the asynchronous copy of the checkpoint

CP_func:
- This function takes an updated copy of CP_obj as argument and writes to PFS..
Pragmatic approach:

- Hide costs for checkpointing
  - Asynchronous
  - Remote Node checkpoints / hierarchical
  - Restart from node local data if possible

- Prototype experiences with GPI – Application based FT with CR
Checkpoint/Restart optimizations

1. Application level checkpointing
   - Minimal checkpoint data

2. Asynchronous checkpointing

3. Multi-level checkpointing (PFS/remote node/localFS)

4. Checkpoint compression

5. …

Hide / avoid costs of computational costs of checkpoints
Synchronous vs. asynchronous checkpointing

- Synchronous checkpointing:
  - Computation halts for I/O time
  - High execution time overhead

- Asynchronous checkpointing:
  - Dedicated threads for performing asynchronous I/O
  - Low execution time overhead
  - Checkpoint location: flexible (e.g. using SCR)
  - In-memory copy required.
Asynchronous Checkpointing

- Hybrid (MPI-OpenMP) configuration performance comparison

Cluster: LiMa, num. of nodes = 32, PFS = LXFS, Aggregated CP size = 200 GB/CP

- Total IO time: 436s
- Actual Overhead: 32s
Asynchronous checkpoints – Remote node

- Memory mapped – local filesystem

- Checkpoint data by
  - Exchanging MPI messages
    - Portable
    - May interfere with regular communication
      → Low priority checkpoint message scheduling?
  - rsync /remotenode/localFS
    - Generality? → efficient via IB?
    - May use (slow) alternative data path (e.g. GBit – if available)

- Failed node detection?!
- Restart MPI within same batch job and ignore FT in communication layer?!
Our (naïve) approach:

- Regular asynchronous Checkpoints (FS or remote node)
- Node failure detected by communication library (Communication library in valid state after node/process loss)
- Spare nodes are available – application replaces lost node
- Application driven restart from last checkpoint
AFT: GPI Introduction

  Developed by Fraunhofer IWTM
- Implements **GASPI** standard: [http://www.gaspi.de/software.html](http://www.gaspi.de/software.html) (Global Address Space Programming Interface)
- PGAS programming model

- Two memory parts
  - Local memory: local to each GPI process
  - Global memory: Accessible for other processes

- Enables fault tolerance
  - via providing **TIMEOUT** for every communication call.
AFT: GPI - Application requirements

- Algorithm based on PGAS model

- For effective fault tolerance
  - No global synchronization, barriers
  - Each GPI-process communicates with certain subset of GPI-processes (e.g. neighbors)
  - In case of failures, rest of the processes detect errors in results and correct them accordingly.

- Algorithm driven FT based applications
Prototype FT implementation

- **Idea:**
  - Running the program with \( n+m \) processes, where \( m \) is the number of idle processes.
  - Program initially utilizes \( n \) processes for work (work-group)
  - In case of a failed process in 'work-group', an idle process is added to the 'work-group'.
  - Processes in newly established 'work-group' restart the work from last checkpoint.
Prototype FT implementation: LBM

- Program flow:
Toy FT implementation: Health Check

- What GASPI provides:
  - A process local copy of 'health check vector'.
  - After each read/write from a process, the health check vector entry of that particular process is locally updated.
  - The entries of health vector are either 0 or 1.

- User side:
  - User can copy this 'health vector' via gaspi_state_vec_get() routine.
  - Deletion of old comm., creation of new comm., new communication structure, (checkpoint/restart) -> user responsibility
Prototype FT implementation: LBM

- Health check routine

```c
void send_msg_to_check_state(gaspi_state_vector_t health_vec, gaspi_rank_t *avoid_list){
    for(int i=0; i<numprocs; ++i){
        if(avoid_list[i]!=1)
        {
            ASSERT(gaspi_write(0, 0, i, 0, 0, sizeof(int), 0, GASPI_BLOCK));
        }
    }
    gaspi_return_t retval;
    retval = gaspi_wait(0, GASPI_BLOCK);
    ASSERT(gaspi_state_vec_get(health_vec));

    for(int i=0; i<numprocs; ++i){
        if(health_vec[i]==1){
            avoid_list[i]=1;
        }
    }
    if(myrank==0) print_health_vec(health_vec);
}
```
Toy FT implementation: local health check

P1 initiates a global health check. A consistent view of the worker communicator is formed.

P0 signals idle proc. to join worker comm.

Every process checks neighbours health

Create new worker comm.

Read last checkpoint (neighbour-level/PFS)
Toy FT implementation: LBM Benchmarks (I)

- Global health check, 16 nodes 1 failure
Toy FT implementation: LBM Benchmarks (II)

- Recovery time scaling, LOCAL Health check
Toy FT implementation: LBM Benchmarks (III)

- Recovery time scaling, GLOBAL Health check

![Graph showing recovery time scaling with number of nodes.](image-url)
GPI fault recovery overhead:

- Timeout returns for communication after failure
  - Only the communication to/from the failed process contributes to this overhead.

- Global health vector update to have consistent view of the health vector across all processes

- Rebuilding worker communicator
  - Process 0 signals the idle processes, which then joins the creation of new comm.

- Checkpoint fetching from neighbour (or PFS) and reinitializing
Prototype FT implementation: SPMVM

- Each process has point-to-point communication with many other processes.
- This communication pattern of a matrix is checkpointed at the start once.
- Restart:
  - 'Recovery-process' reads the matrix data & communication pattern of the dead-process.
  - Remaining processes redirect all communication from dead-process to recovery-process.
Neighbor level checkpointing for GPI (I)

- Development of Multi-level checkpointing infrastructure.
  - Based on library calls
  - Library thread responsible for transferring data in-between nodes and PFS.
  - Independent of communication library (MPI/GPI)

- Multi-level checkpointing with various layers of the application.
  - Different checkpoint frequency on various layers.
Neighbor level checkpointing for GPI (II)

1. Start
   - `cr_thread->cr_thread_init()`

2. Work
   - `pthread_create(..., &cp_monitoring_thread, ...)`

3. Write in-memory checkpoint

4. Signal library thread to transfer CP

5. Work finished?
   - No: Repeat work
   - Yes: Transfer CP to partner node or PFS

6. Check CP transfer
   - Yes: Skip to end
   - No: Sleep (idle)

7. End
Testing:

- Tested successfully up to 1000+ cores with 1-2 failures.

- Challenges using higher number of cores:
  - Seg. fault during deletion of old comm/ recreat new comm.
  - Issue using barrier for new comm.
  - Both issues are under investigation by Frauenhofer IWTM.

→ Bug in GPI library has been detected and will be fixed in next release.

- Future optimizations:
  - Reduction in health check time.
Concluding remarks:

Asynchronicity: User controlled node-level resource management still lacks simple and efficient support from OS.

FT: If you use checkpointing

- do it asynchronously (dedicated threads) – keep data in memory? – reduce interference with regular communication?
- use application specific knowledge
- restarting at runtime is a challenge with current communication libraries → You feel like a test pilot
- GPI is on a reasonable way – MPI-ULFM?

FT: Exascale “modus operandi” still unclear:

- Pool of spare nodes?
- Continue with remaining set
GHOST will become public available this year

If you are interested in testing, you are very welcome

ask us: https://blogs.fau.de/essex/

Thank you!

Questions?

Partially funded by DFG Priority Programme 1648

Partially funded by BMBF project FeTol