Application level asynchronous check-pointing / restart: first experiences with GPI

Faisal Shahzad and Gerhard Wellein
Dagstuhl Seminar „Resilience in Exascale Computing“
September 29, 2014
Erlangen Regional Computing Center:

- Tier-2 center in Germany
- Operates compute clusters (200,…,600 nodes)
- Scientist from University of Erlangen and northern Bavaria
- Strong application support group (collaboration with LRZ Munich)

HPC research focus:

- Node level performance engineering
- Hardware efficiency of sparse linear algebra, lattice Boltzmann solvers, stencil computations
- Hybrid/new programming parallel models

Leading PI of ESSEX project from SPPEXA
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, …
ESSEX: “Co-Design” oriented project

Holistic Performance Engineering

- Code/Energy Efficiency
- Computational Algorithms
- Building Blocks
- Applications

Implementation, data structure, parallelization

Fault Tolerance

Problem formulation, Quantum State Encoding (QSE)
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

$|v\rangle = |\text{rand}(\rangle$;
Initialization & computation of $\mu_0$, $\mu_1$

for $m = 1$ to $M/2$ do

swap($|w\rangle$, $|v\rangle$);

$|u\rangle = H|v\rangle$;

$|u\rangle = |u\rangle - b|v\rangle$;

$|w\rangle = -|w\rangle$;

$|w\rangle = |w\rangle + 2a|u\rangle$;

$\eta_{2m} = \langle v|v\rangle$;

$\eta_{2m+1} = \langle w|v\rangle$;

end

end

Application: $R$ random configurations ($R=1,\ldots,10^2$) 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
Our (ESSEX) effort –
get a simple prototype solution first –
application driven –
No silver bullet
Fault Tolerance Approaches

1. Algorithm Based Fault Tolerance (ABFT)

2. Message Logging

3. Redundancy

4. Fault Prediction (*proactive fault tolerance*)

5. Checkpoint/Restart (C/R)

Each of these fault tolerance approaches carries overhead in terms of time and/or resources

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.
Checkpoint overhead estimation model

**Synchronous Checkpointing**

- \( t_{O,s} = \) overhead for synchronous checkpoints
- \( t_{CP,s} = \) duration of a synchronous checkpoint
- \( S_{CP} = \) size of a single checkpoint in bytes
- \( B_{IO} = \) I/O bandwidth to the file system in bytes/s
- \( B_{M} = \) memory bandwidth of a node in bytes/s
- \( n = \) number of checkpoints

\[
\begin{align*}
    t_{O,s} &= n \cdot t_{CP,s} \\
    t_{O,s} &= n \cdot \frac{S_{CP}}{B_{IO}}
\end{align*}
\]

**Asynchronous Checkpointing**

- \( t_{O,a} = \) overhead for asynchronous checkpoints
- \( t_{CP,a} = \) duration of an asynchronous checkpoint
- \( S_{CP,node} = \) checkpoint size per node in bytes
- \( B_{M,CP} = \) memory bandwidth used for checkpoint-I/O in bytes/s

\[
B_{M} \cdot t_{O,a} = n \cdot B_{M,CP} \cdot t_{CP,a}
\]

\[
B_{M,CP} = \frac{m \cdot S_{CP,node}}{t_{CP,a}}
\]

\[
t_{O,a} = \frac{m \cdot S_{CP,node}}{B_{M}} \cdot n
\]

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
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 (w/ hwloc) task management

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

Asynchronous checkpoints via GHOST-task thread:

- 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.
APPLICATION DRIVEN AUTOMATIC FAULT TOLERANCE (AFT)

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
FT communication libraries

A long time ago it was no problem to

- tolerate the frequent loss of processes/nodes
- register new processes/nodes dynamically on demand

→ Parallel Virtual Machine (PVM)

Today:

- Several non-standard libraries, e.g. Charm++ or GPI

- Why is FT not part of MPI?
  - Complexity of MPI standard / MPI forum?
  - Restrict FT feature on small parts of MPI standard?
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.
GPI FT program flow:

- P0 signals idle proc. to join worker comm.
- P1 initiates a global health check. A consistent view of the worker communicator is formed.
- Every process checks neighbours health
- Create new worker comm.
- Read last checkpoint (neighbour-level/PFS)
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
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.
Concluding remarks:

If you use checkpointing
  - do it asynchronously
  - use dedicated threads
  - 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

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
The seminar topic

Resilience in Exascale Computing

How many users (scientific communities) need within a decade?

Which of them need?

Quality?
10,000 jobs @ 10 nodes

Capability!
1 job @ 100,000 nodes

Where is the sweet spot?
The seminar topic

Do we need resilience beyond 2025?

Who ensures/guarantees?

Hardware?

Resilience in Exascale Computing

FT algorithms?

Low level automatic SW solution – silver bullet?

Application with OS/HW support?

Conservation law of HPC
Flexibility * Performance = constant

Does overhead pay off?