Testen paralleler Anwendungen im HPC-Bereich

Fokus: MPI+X am Beispiel des ESSEX-Projektes

DLR

Motivation

- Workflow: Performance-Optimierung
  → Korrektheit von Änderungen!

- Heterogene Hardware
  → Korrektheit auf verschiedenen Plattformen

- Komplikationen:
  - Parallelität auf mehreren Ebenen
  - Low-level Code
  - Hardware-spezifischer Code
  - Spezifischer Code je nach Dimension der Daten (Blockgröße)
  - Automatisch generierter Code

→ Effizientes Arbeiten benötigt automatische Tests

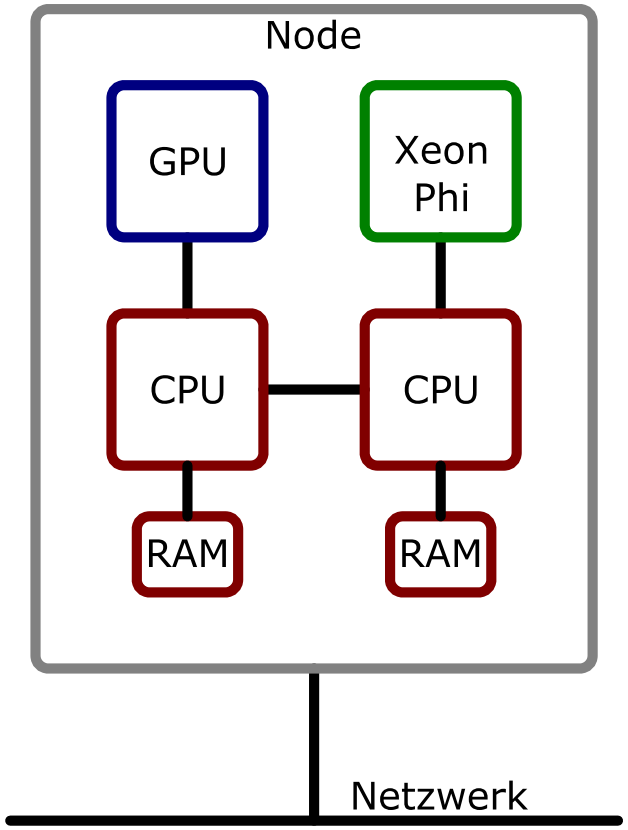
ESSEX-Projekt

Equipping **S**parse **S**olvers for **Ex**ascale:

- aus dem SPPEXA-Programm der DFG
- Projektpartner: Erlangen, Greifswald und Wuppertal
- **Thema: Berechnung von Eigenwerten großer, dünnbesetzter Matrizen**

Hardware-Nutzung im ESSEX-Projekt

- Ziel-Architektur: heterogenes Cluster
  - MPI + Tasks + OpenMP per CPU
    + CUDA per GPU
    + OpenMP per Xeon Phi
- Automatische Code-Generierung
  - CPU-abhängig (SSE, AVX)
  - SIMD / SIMT Code
- Rechenkernel-Auswahl zur Laufzeit
  - abhängig von Daten-Dimensionen



Test-Konzepte in ESSEX

- Testen von Schnittstellen
  - für Bausteine von Algorithmen
  - für Rechenkernel
    - Trilinos (stabil) – Prototypen – aus ESSEX (schnell)

- Möglichst „serielle Sicht“ (Parallelität im Hintergrund)
  - gezielte Suche nach bestimmten Fehlerklassen / Szenarien
  - Testfälle sukzessiv komplexerer Konfiguration

- „Templated“ Test-Code (durch C-Präprozessor-Macros)
  - single/double, reell/komplex
  - verschiedene Blockgrößen

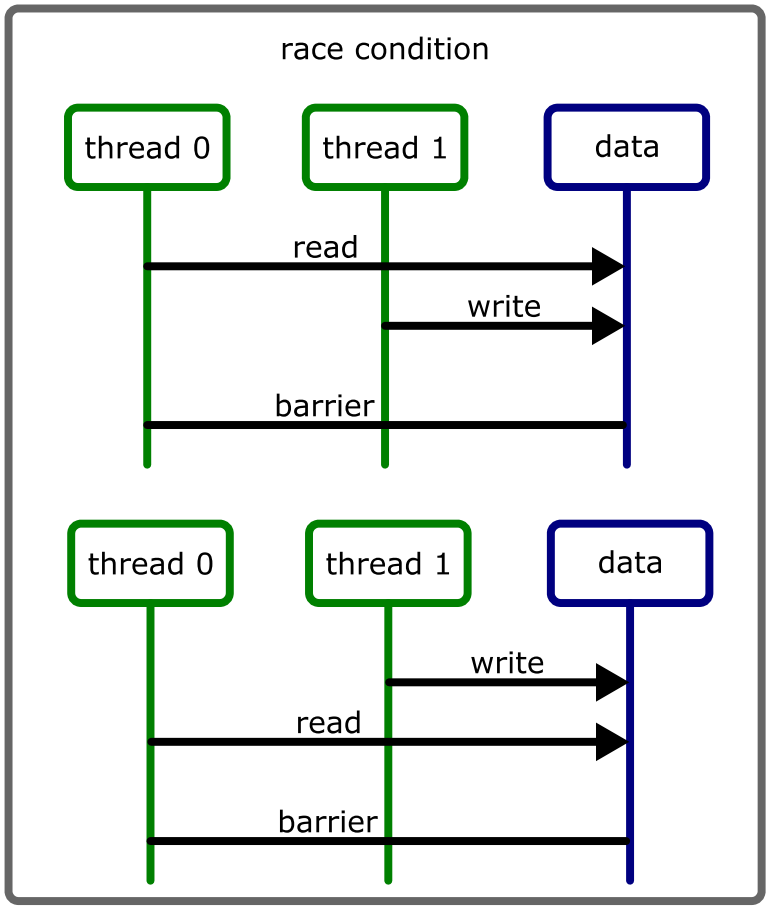
  → **~25% der Code-Zeilen nur für Tests**

  → ~450 Tests, jeder in 20 Konfigurationen (~9000 Testfälle)

Compiler-Instrumentalisierung für Multithreading

- Race Conditions, Deadlocks:
  - `-fsanitize=thread`
    (GCC / Clang)
- Zeiger-Probleme (Array-Indizes)
  - `-fsanitize=address`
    (GCC / Clang)
  - `-fcheck=bounds`
    (gfortran)

→ mehrere Test-Modi:
  - Debug + fsanitize=…
  - Release



Parallele Testframeworks

- Für Fortran: **pFUnit**
  - entwickelt von der NASA
  - unterstützt OpenMP und MPI
  - benötigt neueste Compiler
  - Open-Source: http://pfunit.sourceforge.net/

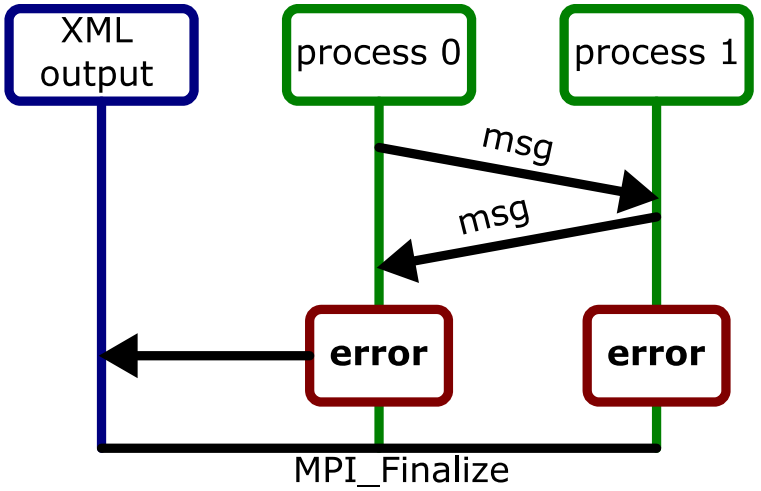
- Für C / C++: **???**
  - In Trilinos (http://trilinos.org/)

→ eigene MPI-Erweiterung für googletest

Test-Szenarien mit MPI (1)

Gleicher Fehler auf allen Prozessen:

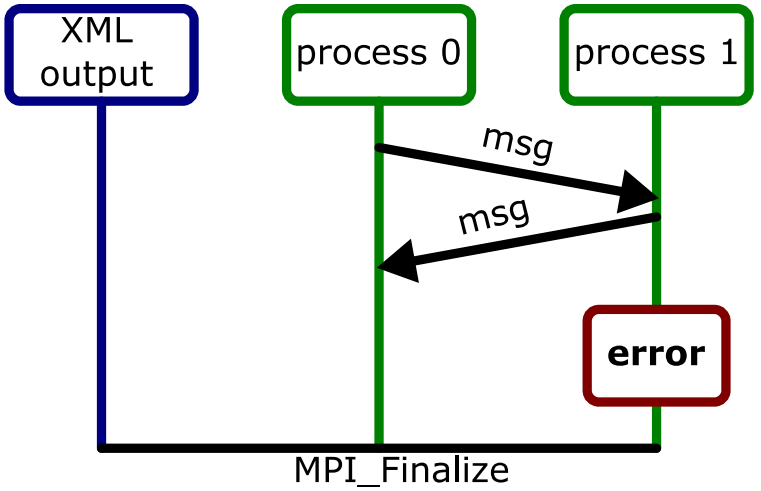
→ **Alles OK!**



Test-Szenarien mit MPI (2)

Nur Fehler in Prozess 1:

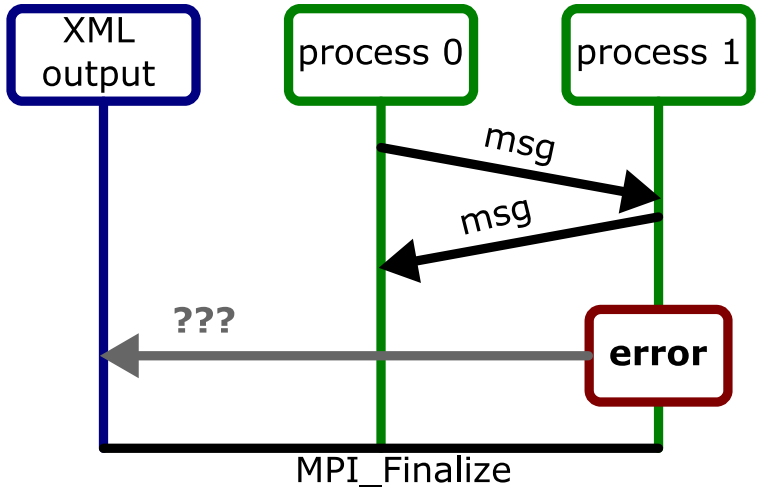
→ **Scheint alles OK!**



Test-Szenarien mit MPI (2)

Nur Fehler in Prozess 1,
alle Prozesse geben Fehler aus:

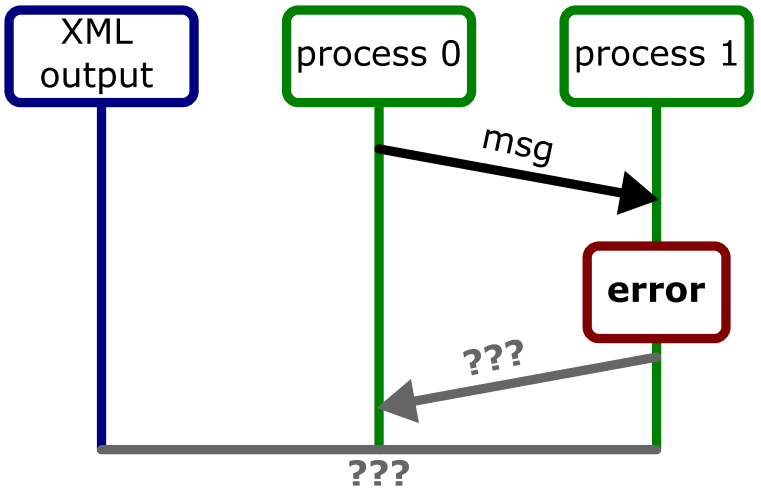
→ **Mehrere Prozesse schreiben in gleiche Datei!**



Test-Szenarien mit MPI (3)

Nur Fehler in Prozess 1,
Prozess 0 wartet auf Nachricht:

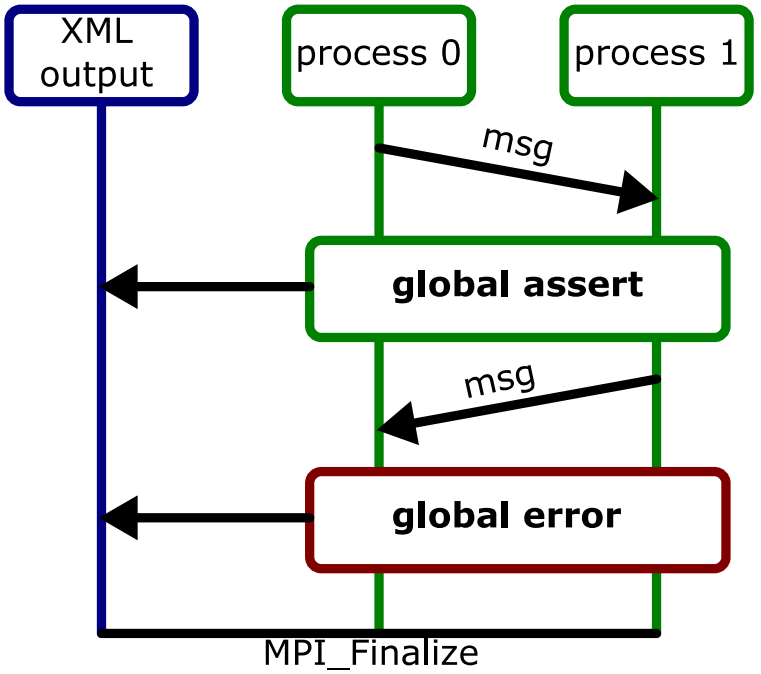
→ **Keine Ausgabe, terminiert nicht!**



Test-Szenarien mit MPI (3)

Globale Assertions,
Fehler werden an alle kommuniziert:

→ **Alles OK!**



MPI-Erweiterung für googletest

- CMake-Option zum Aktivieren von MPI
  - Ausgabe nur auf Prozess 0
  - eigener MPI-Communicator

- Globale Testoperationen (`ASSERT_*`, `EXPECT_*`)
  → Müssen auf allen Prozessen identisch sein!
  - Problem:

```
for(int i = 0; i < n_local; i++)
    ASSERT_NEAR(0., x[i]);
```

  → Terminiert nicht!

MPI-Erweiterung für googletest

- CMake-Option zum Aktivieren von MPI
  - Ausgabe nur auf Prozess 0
  - eigener MPI-Communicator

- Globale Testoperationen (`ASSERT_*`, `EXPECT_*`)
  → Müssen auf allen Prozessen identisch sein!
  - Lösung:

```
double tmp = 0;
for(int i = 0; i < n_local; i++)
    tmp = max(tmp,abs(x[i]));
ASSERT_NEAR(0.,tmp);
```

- Einschränkungen:
  - Nicht geeignet für „asynchrone“ Testfälle
  - Keine Erkennung von Kommunikationsfehlern
  - Keine „Death-Tests“ (MPI)

Tools für MPI

- MUST
  - erkennt Kommunikationsfehler
  - MPI-Wrapper: „mustrun“ statt „mpirun“
  - https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST

- valgrind mit MPI-Support:
  - erkennt Speicher/Buffer-Fehler
  - MPI-Wrapper: „LD_PRELOAD=… mpirun“
  - alternativ GCC/Clang mit `-fsanitize=address`

Testumgebung in ESSEX

- Jenkins (Continuous Integration system)
  - → **schnelle Übersicht**
  - verschiedene Konfigurationen
  - verschiedene Compiler / MPI-Bibliotheken
- Echte Hardware (keine virtuellen Buildknoten)
  - Aufwendige Tests
  - Nutzung der GPU
  - Erkennung der CPU-Topologie
- Emulation verschiedener CPU-Befehlssätze (AVX/SSE)
  - Intel SDE

Ausblick: Performance-Modellierung

Vorgehen:

1. Laufzeit modellieren (roofline)
2. Relevante HW-Bottlenecks messen (likwid)
3. Laufzeit mit Modell vergleichen
4. HW-Counter messen (likwid, ScoreP+PAPI)

→ Automatische Performance-Tests