

Hochleistungsrechnen Hybride Parallele Programmierung

Dr. Panagiotis Adamidis
Deutsches Klimarechenzentrum

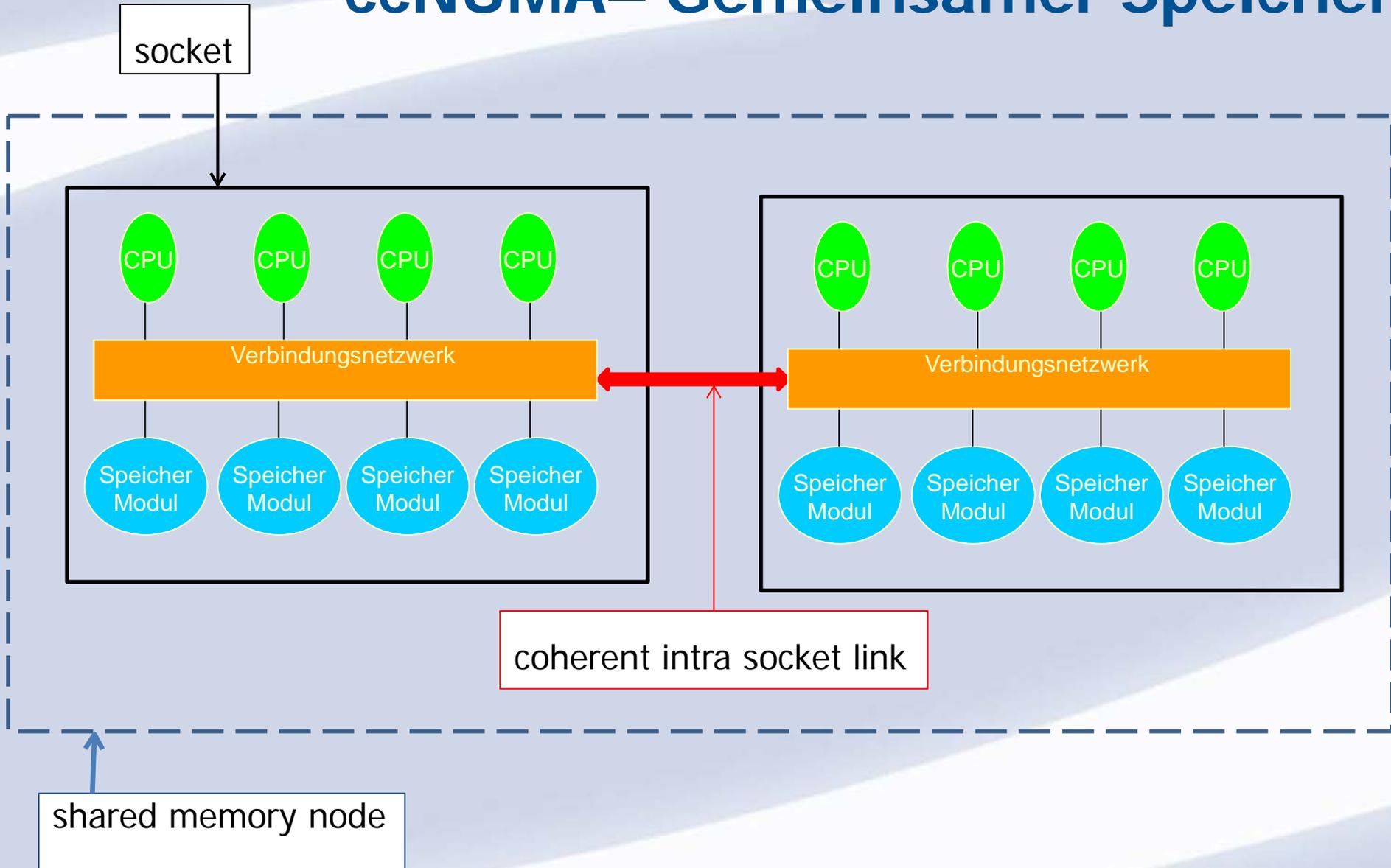
Inhaltsübersicht

- Einleitung und Motivation
- Programmiermodelle für hybride Architekturen
- Thread Safety
- Abbildungsprobleme
- Zusammenfassung

ccNUMA – Gemeinsamer Speicher

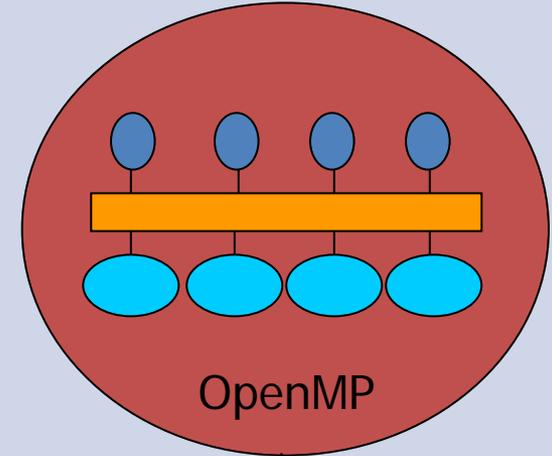
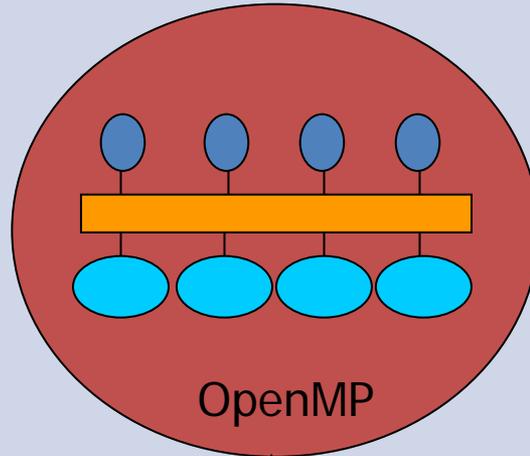
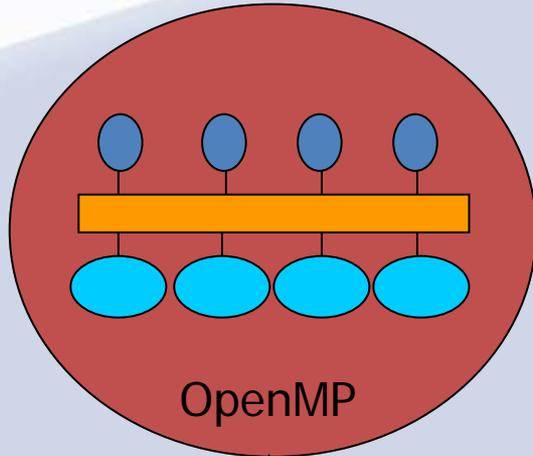
- Hardware mit verteiltem Speicher
- Logisch ist ein gemeinsamer Speicher sichtbar (ein Adressraum)
- Unterschiedliche Zugriffszeiten je nachdem ob Zugriffe im lokalen oder entfernten Speicher stattfinden
- cache-coherent Nonuniform Memory Access

ccNUMA – Gemeinsamer Speicher

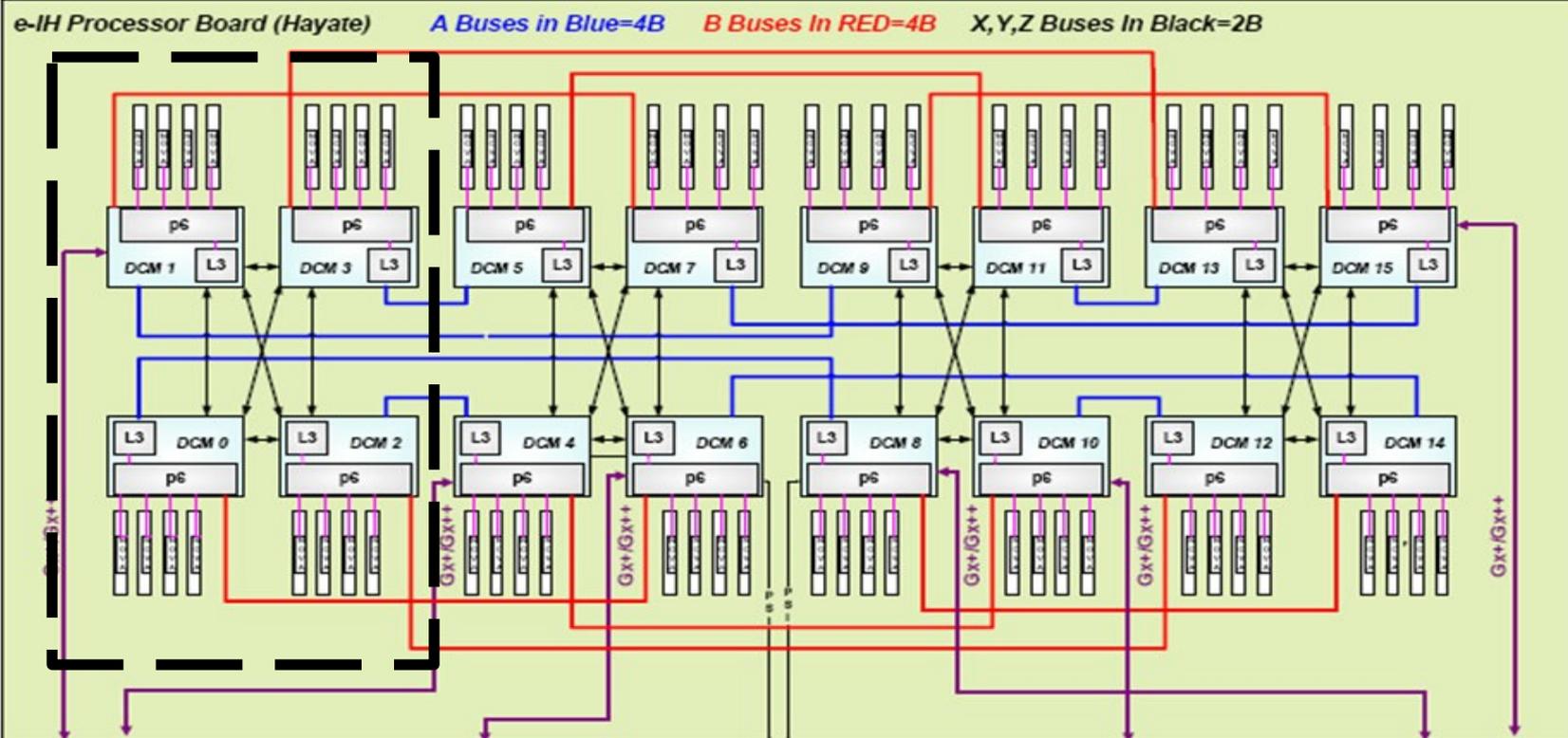


Die hybride Welt

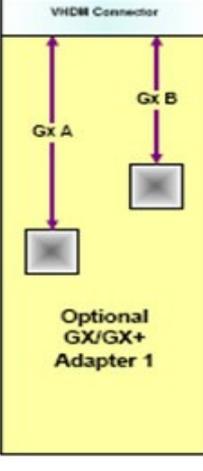
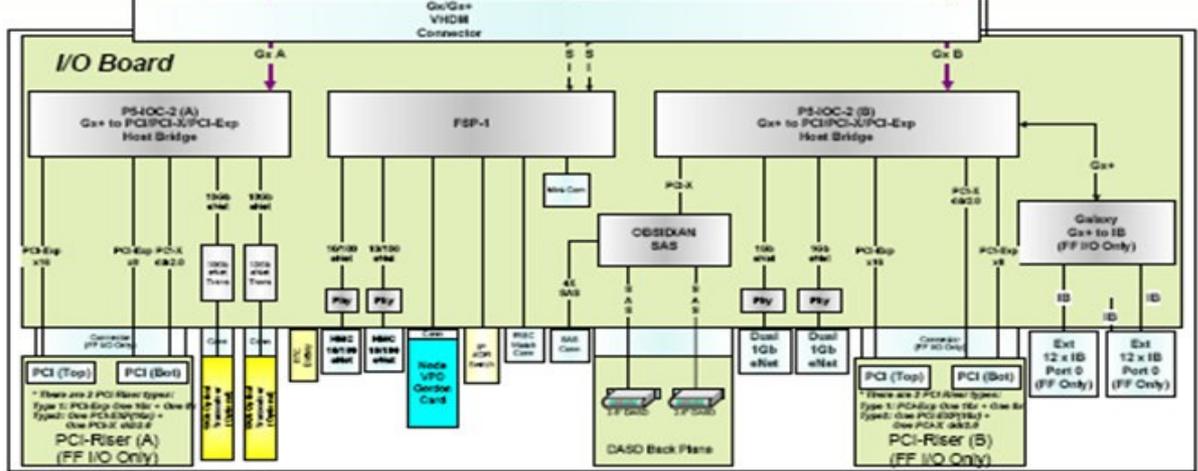
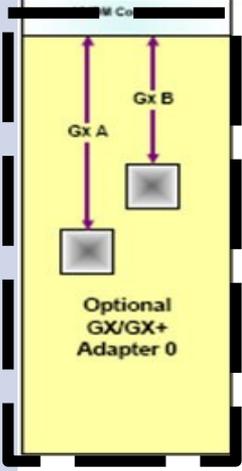
Knoten



Power6 p575 Node



Quad



Infiniband
 Galaxy-2
 Adapters

Engpässe von Hybriden Systemen

- Verbindungsnetzwerk
- Speicherbandbreite
- Ruhende Prozessoren/Kerne

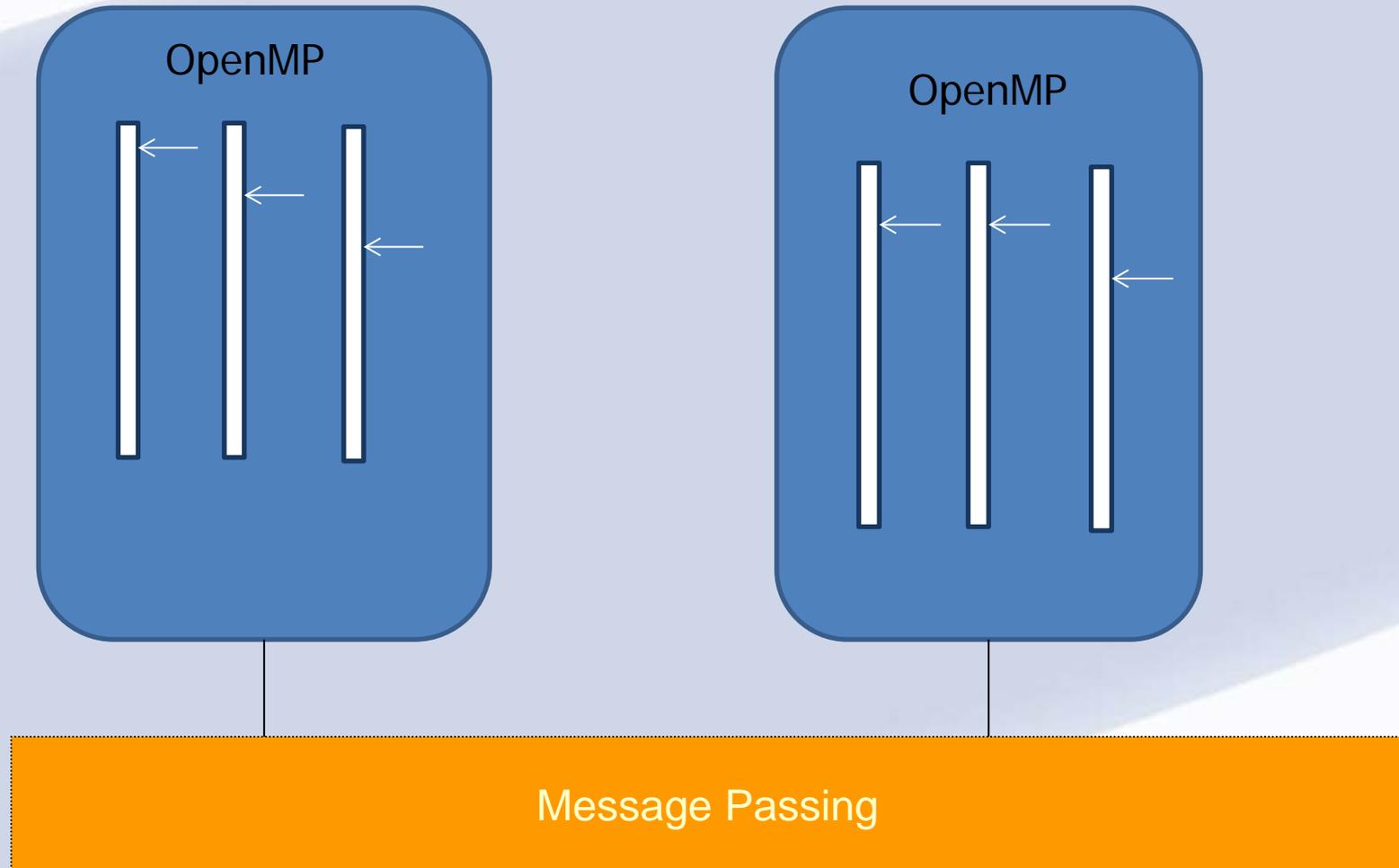
Warum hybride Parallelisierung

- Der logische Schritt um hybride Architekturen zu programmieren ist ein hybrides Programmiermodell
- Minimierung des MPI-Kommunikation Zusatzaufwands innerhalb von SMP-Knoten
- Hybride Programmiermodelle ermöglichen multilevel Parallelisierung von Anwendungen

Multilevel Parallelisierung

- Schritt 1
 - *coarse-grained parallelism*
 - Zerlegung des Problems in möglichst unabhängige Teilprobleme deren Berechnung gelegentlich Austausch von Informationen benötigt
 - Jedes Teilproblem wird auf ein MPI-Prozess abgebildet
- Schritt 2
 - *fine-grained parallelism*
 - Zusätzliche Parallelisierung mit OpenMP Direktiven z.B. auf loop-level

Prozesse und threads

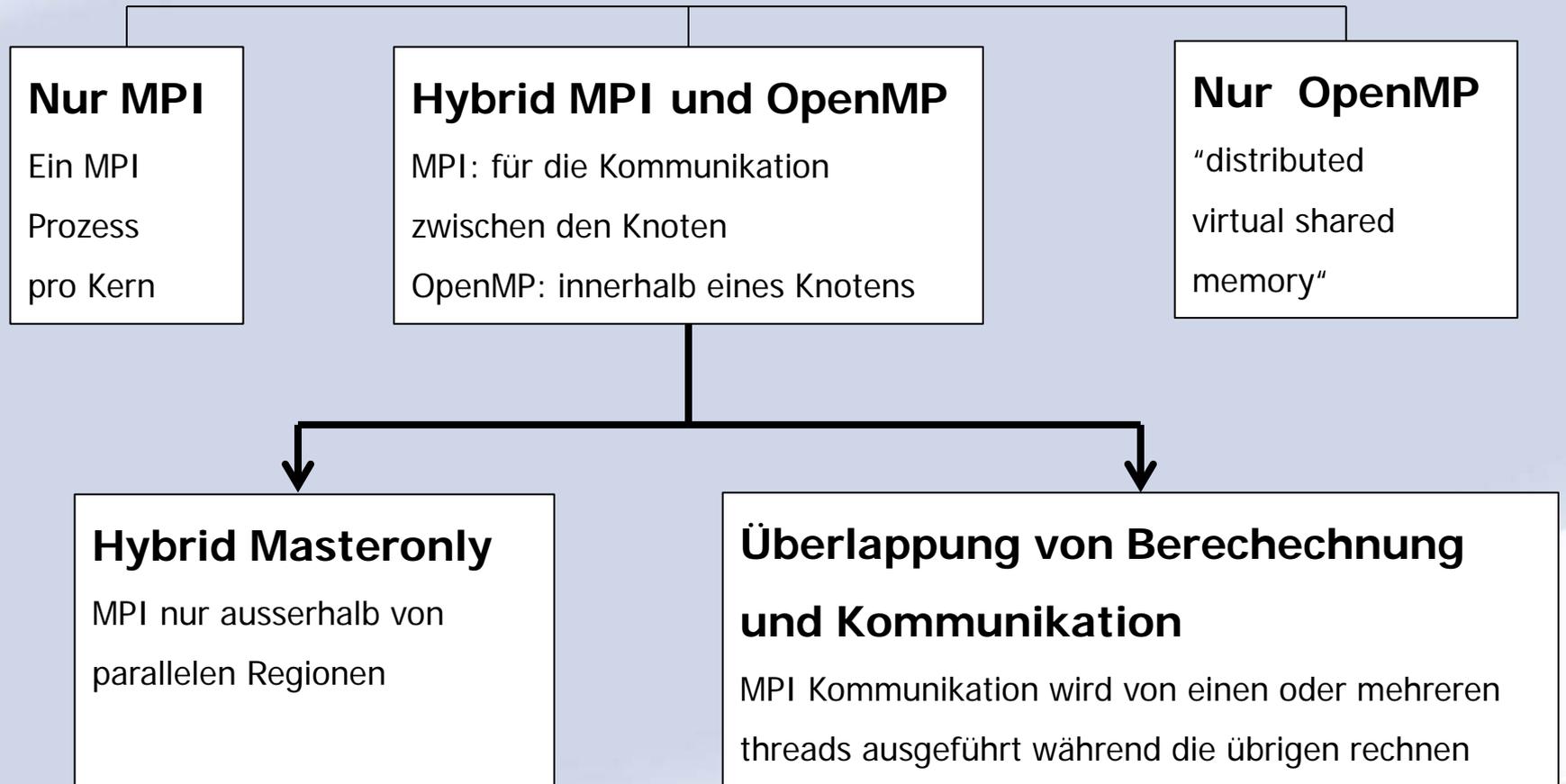


Prozesse und threads

- Einzelne threads sind nicht sichtbar ausserhalb ihres Prozesses
- Die threads führen MPI-Aufrufe im Auftrag ihres Prozesses aus
 - D.h. in den MPI-Routinen wird der **rank** des MPI-Prozesses benutzt und nicht die thread-id

MPI_Xxxx(....., **rank**,

Programmiermodelle für hybride Systeme



Unterstützung von Multithreading

- Die MPI-Bibliothek muss den Zustand berücksichtigen, dass mehrere threads MPI-Routinen aufrufen, ohne sich dabei gegenseitig zu stören (thread safety)
- MPI-1 Standard unterstützt kein multithreading
- MPI-2 Standard hat multithreading Unterstützung

Unterstützung von Multithreading

- MPI_Init
 - Initialisierung ohne multithreading
- MPI_Init_thread
 - Initialisierung mit multithreading
- MPI_Init_thread (int*argc, char*(argv)[], int required, int*provided)

Unterstützung von Multithreading

- `MPI_Init_thread (int*argc, char*((*argv)[], int required, int* provided)`
 - `required`: der gewünschte multithreading Modus
 - `provided`: Rückgabewert, der den aktuellen Modus, der vom System unterstützt wird enthält

Unterstützung von Multithreading

- **MPI_THREAD_SINGLE**
 - Nur ein thread pro Prozess
 - Ähnlich wie MPI_Init
- **MPI_THREAD_FUNNELED**
 - Der Prozess ist multithreaded aber nur der master thread führt MPI-Aufrufe aus
- **MPI_THREAD_SERIALIZED**
 - Mehrere threads können MPI-Routinen aufrufen, aber nur einer zu einem gewissen Zeitpunkt

Unterstützung von Multithreading

- `MPI_THREAD_MULTIPLE`
 - Mehrere threads dürfen MPI-Routinen aufrufen ohne jede Restriktion
- Überlappung von Kommunikation und Berechnung erlauben
 - `MPI_THREAD_FUNNELED`
 - `MPI_THREAD_SERIALIZED` und
 - `MPI_THREAD_MULTIPLE`

Auswirkungen auf die Korrektheit

- Je höher der multithreading level desto höher die Komplexität der parallelen Algorithmen der Anwendungen
 - Einerseits kann man dadurch eine Effizienzsteigerung erreichen
 - Andererseits sind die Fehler die man dabei machen kann schwieriger zu finden
- Die Implementierung der MPI-Bibliothek ist auch komplexer

Unterstützung von Multithreading

- OpenMPI und mpich2 unterstützen alle multithreading level
- Debugging von hybrid MPI/OpenMP parallelen Programmen ist mit dem DDT debugger möglich
- Profiling und Trace Analysis ist mit dem VAMPIR tool möglich

Unterstützung von Multithreading

- `int MPI_Query_thread(int *provided)`
 - Ein thread kann den multithreading Modus abfragen, um sicher zu sein dass es erlaubt ist Aufrufe der MPI-Routinen zu machen
- `int MPI_Is_thread_main(int *flag)`
 - Ein thread kann herausfinden ob er der master thread ist
 - Wichtig bei `MPI_THREAD_FUNNELED`

Beispiel

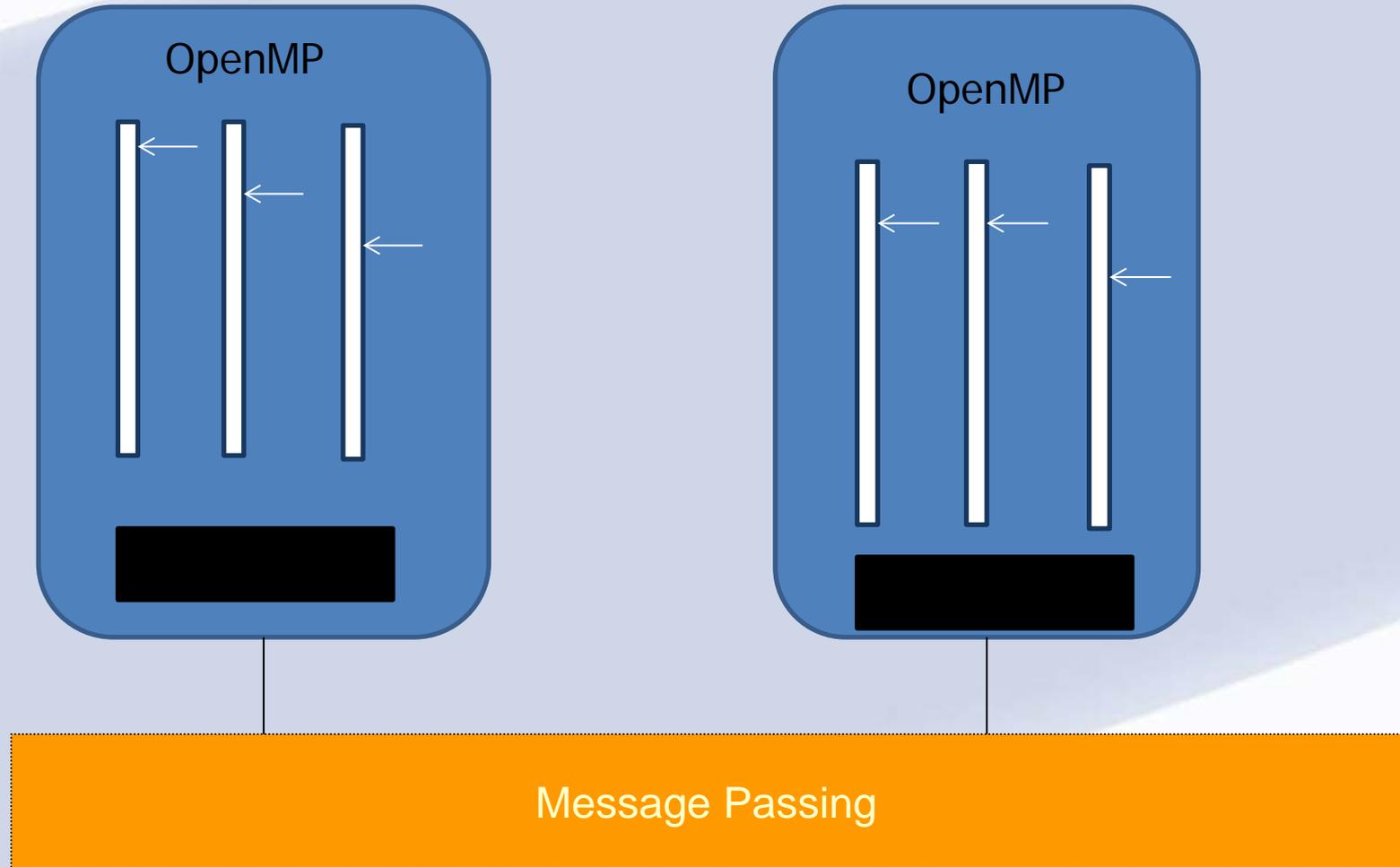
```
int thread_level, thread_is_main

MPI_Query_thread(&thread_level);
MPI_Is_thread_main(&thread_is_main);
If ((thread_level > MPI_THREAD_FUNNELED ) ||
    (thread_level == MPI_THREAD_FUNNELED && thread_is_main)) {
    .... /* diese threads duerfen MPI Routinen aufrufen */
else {
    printf("Fehler: Dieser thread darf keine MPI Kommunikation machen \n")
}
.....
```

Hybrid Master Only

- MPI-Aufrufe nur ausserhalb von parallelen Regionen
- Üblicherweise 1 MPI Prozess pro Knoten und 1 OpenMP thread pro Kern innerhalb des Knotens

Prozesse und threads



Hybrid Master Only

```
for (iteration = 1.....n)
{
  #pragma omp parallel
  {
    /* compute something */
  }
  /* ON MASTER THREAD ONLY */
  MPI_Send(sendbuffer, ..... )
  MPI_Recv(recvbuffer,..... )
}
```

Hybrid Master Only und OMP MASTER

-Innerhalb von OMP MASTER

```
#pragma omp barrier  
#pragma omp master  
    MPI_Xxx(.....)  
#pragma omp barrier
```

Hybrid Master Only und OMP MASTER

- `MPI_THREAD_FUNNELED` ist notwendig
- OMP MASTER garantiert keine Synchronisation
- OMP BARRIER ist notwendig um sicher zu gehen, dass der Kommunikationspuffer nicht von anderen threads benutzt wird

Beispiel mit MPI_Send

```
#pragma omp parallel
{
  #pragma omp for
    for(i=0;i<1000;i++)
      buf[i] = a[i];
  #pragma omp barrier
  #pragma omp master
    MPI_Send(buf,.....);
  #pragma omp barrier
  #pragma omp for
    for(i=0;i<1000;i++)
      buf[i] = c[i];
} /* omp end parallel */
```

- Vorteile
 - Keine MPI-Kommunikation innerhalb eines SMP Knotens
 - Kein Topologie Problem
- Nachteile
 - Während der Master kommuniziert schlafen alle anderen threads
 - Nur ein kommunizierender thread (master) kann in den meisten Fällen nicht die volle MPI-Bandbreite zwischen den Knoten ausnutzen

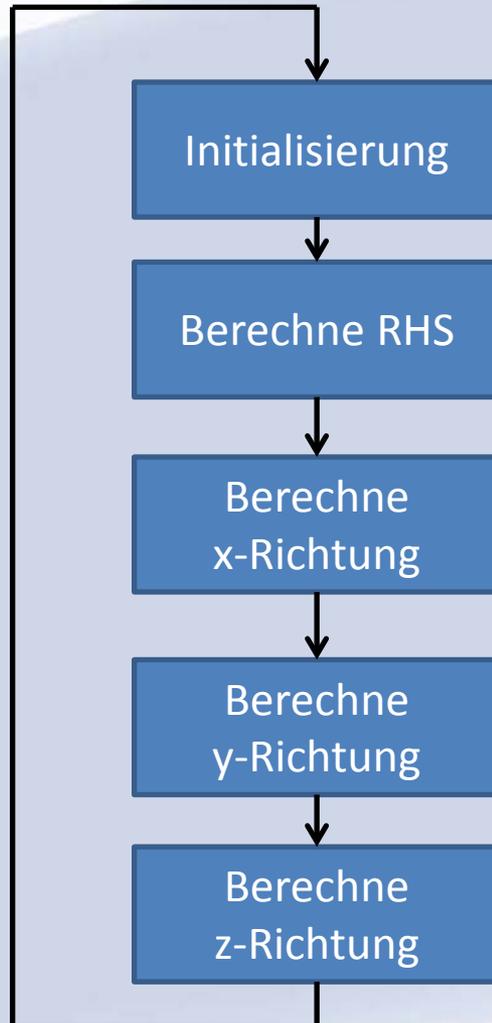
Hybrid mit Überlappung (Mixed Modell)

- Überlappung von Kommunikation und Berechnung
- Mehr als ein MPI-Prozess pro Knoten
- MPI-Aufrufe können von mehreren threads ausgeführt werden
- MPI-Bibliothek muss multithreading unterstützen

Hybrid Mixed

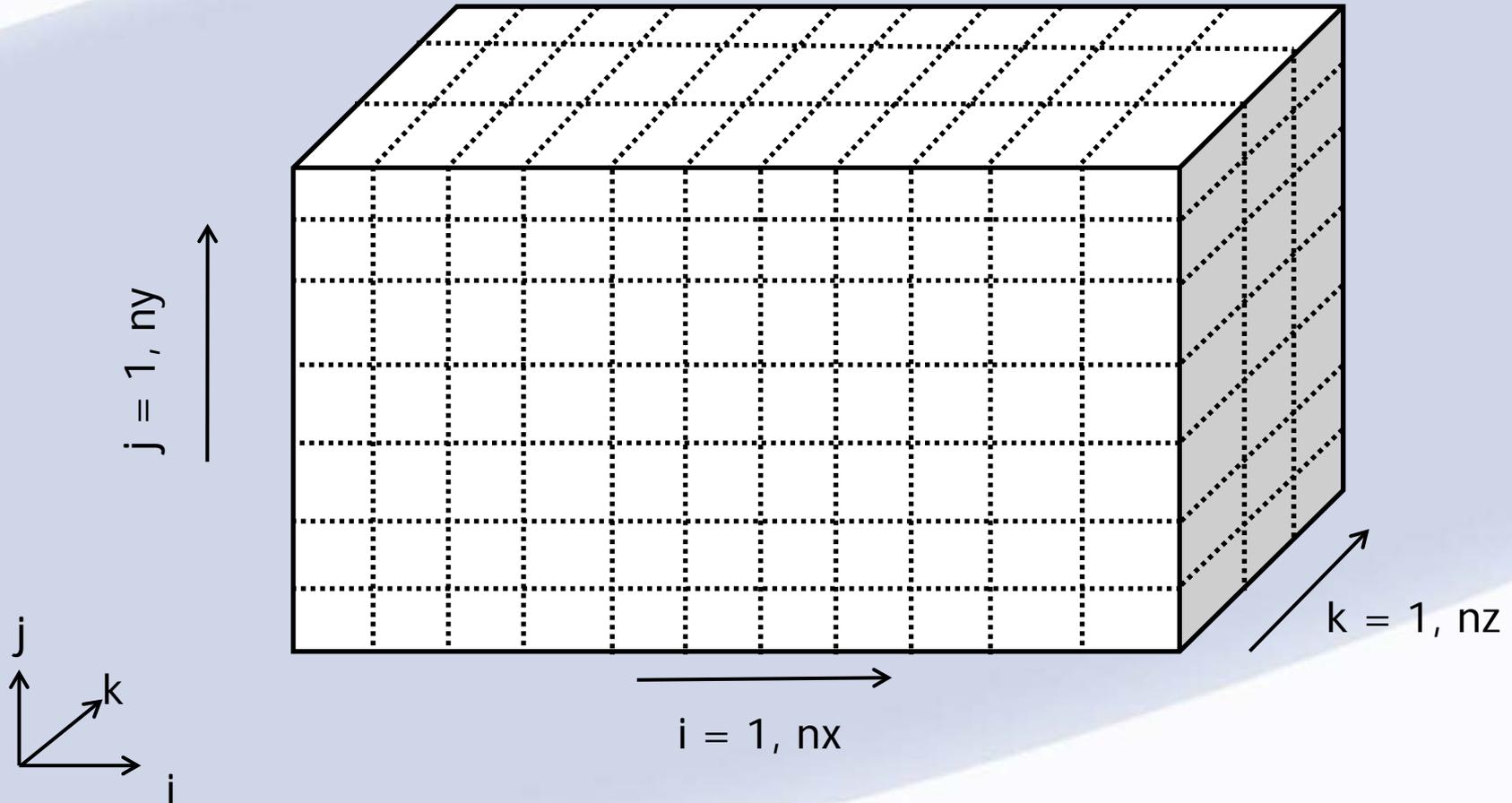
```
#pragma omp parallel
{
  if ( my_thread_id == id1) {
    MPI_Send(sendbuffer, .....)
  }
  else if ( my_thread_id == id2) {
    MPI_Recv(recvbuffer, .....)
  }
  else
  {
    /* compute something */
  }
}
```

Beispiel aus Strömungsmechanik

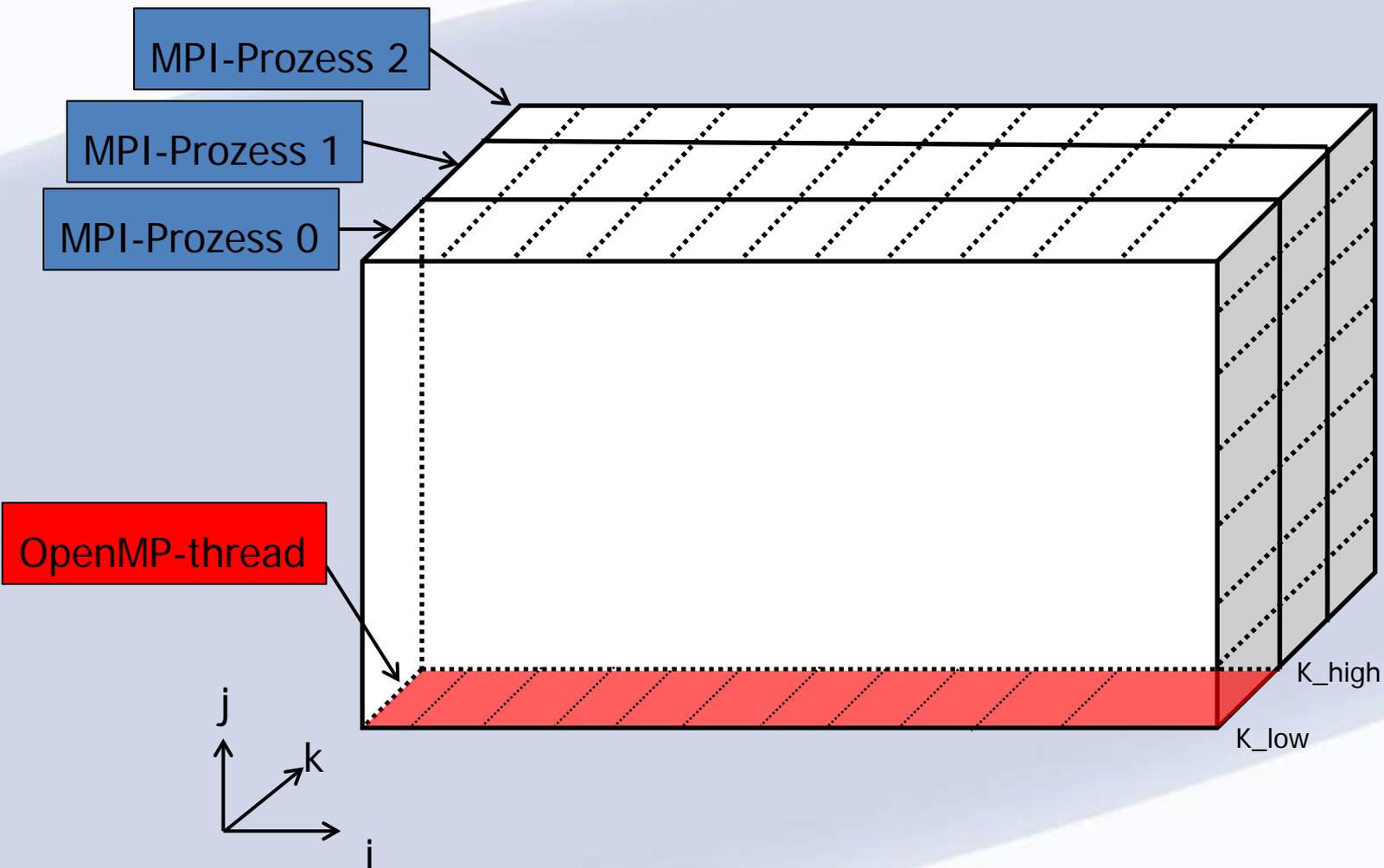


3D Navier Stokes Gleichungen

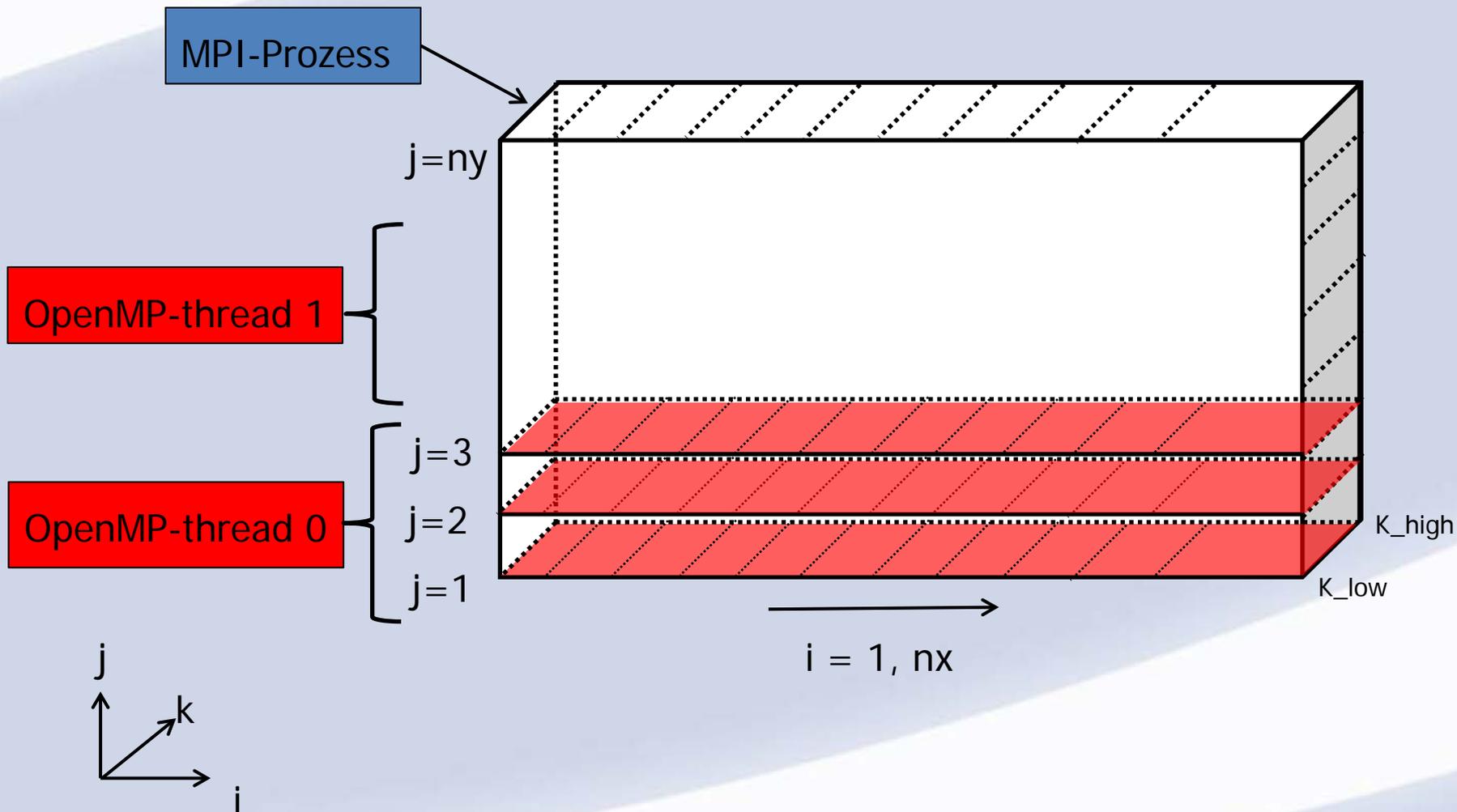
3 Dimensionales Rechengebiet



1D Gebietszerlegung in z-Richtung



1D Gebietszerlegung in z-Richtung



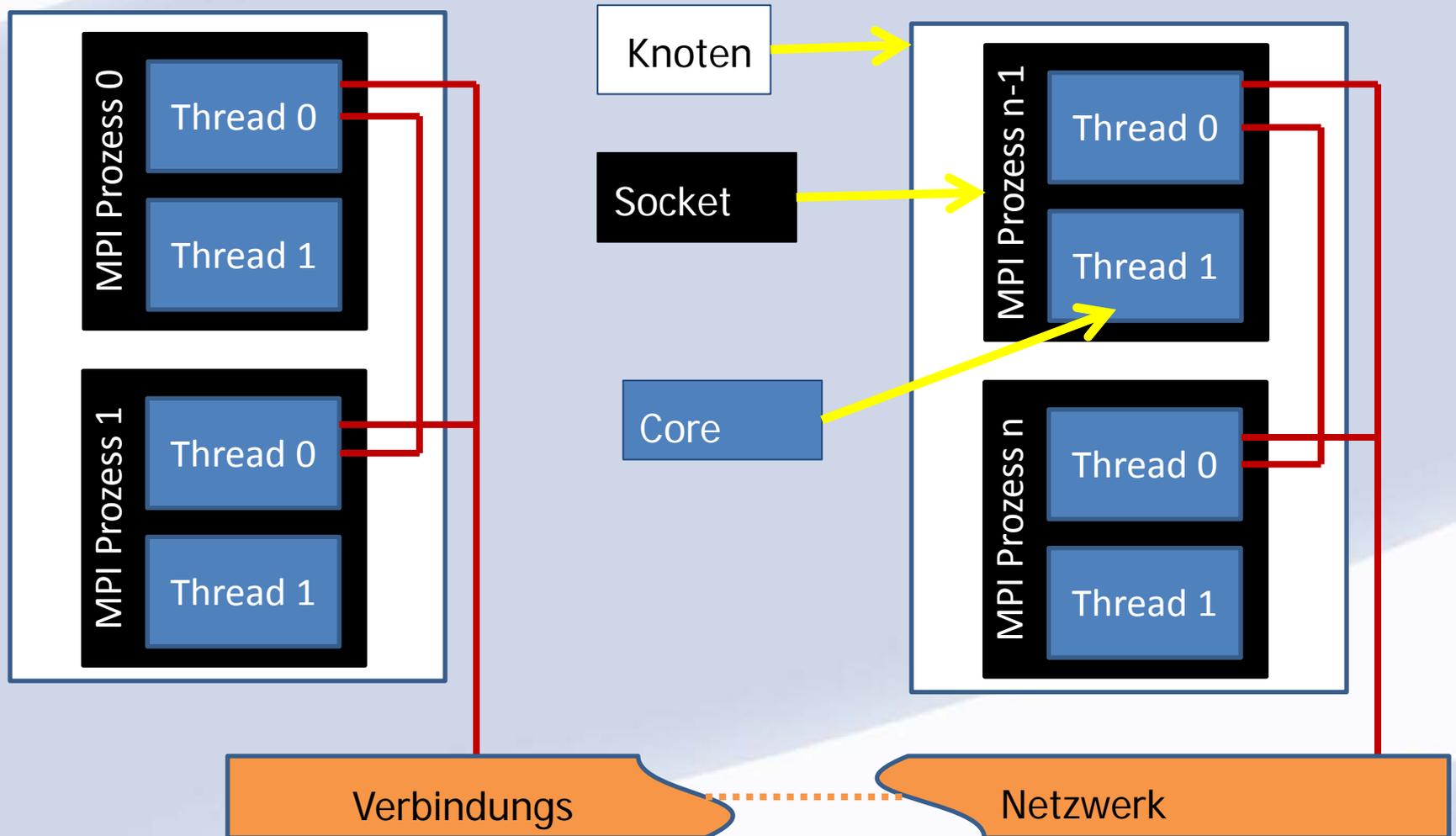
Hybrid Mixed

```
MPI_Init_thread(MPI_THREAD_MULTIPLE,.....)
#pragma omp for
for (j=1; j<ny; j++) {
    MPI_Receive(pid1, ..... )
    for (k=k_low; k<k_high; k++)
        for (i=1; i<nx; i++)
        {
            z[i,j,k] = z[i,j,k-1] + .....
        }
    MPI_Send(pid2,.....)
}
```

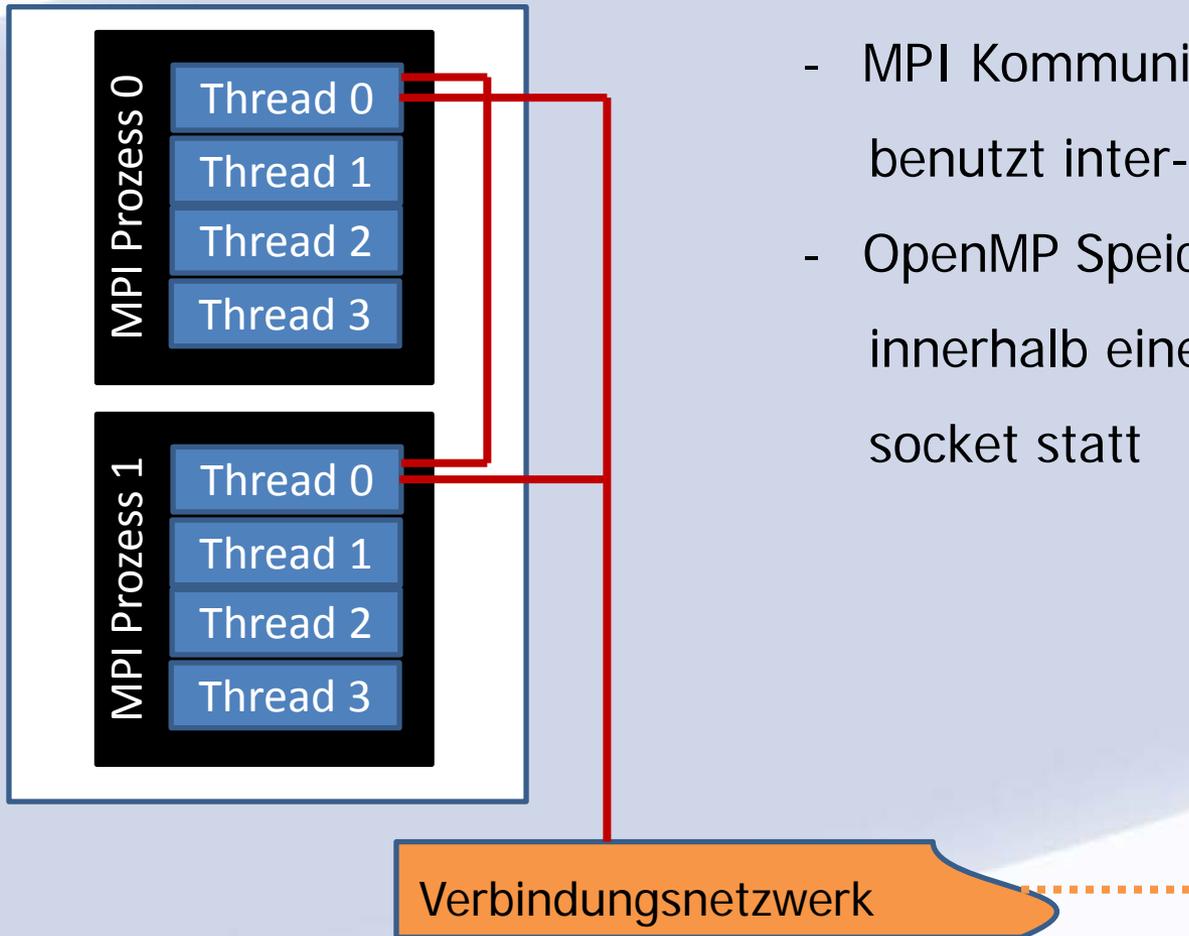
OpenMP und MPI Non Blocking Kommunikation

- Es ist nicht erlaubt
 - dass mehrere threads ein MPI_Wait oder MPI_Test auf eine und dieselbe MPI Non Blocking Operation (request Objekt) durchführen
- Es ist erlaubt
 - dass ein thread eine Non Blocking Operation startet und ein anderer diese abschliesst. Es dürfen aber nicht zwei threads versuchen diese zu beenden

Abbildung von Prozessen/Threads auf die Hardware (mixed Modell)



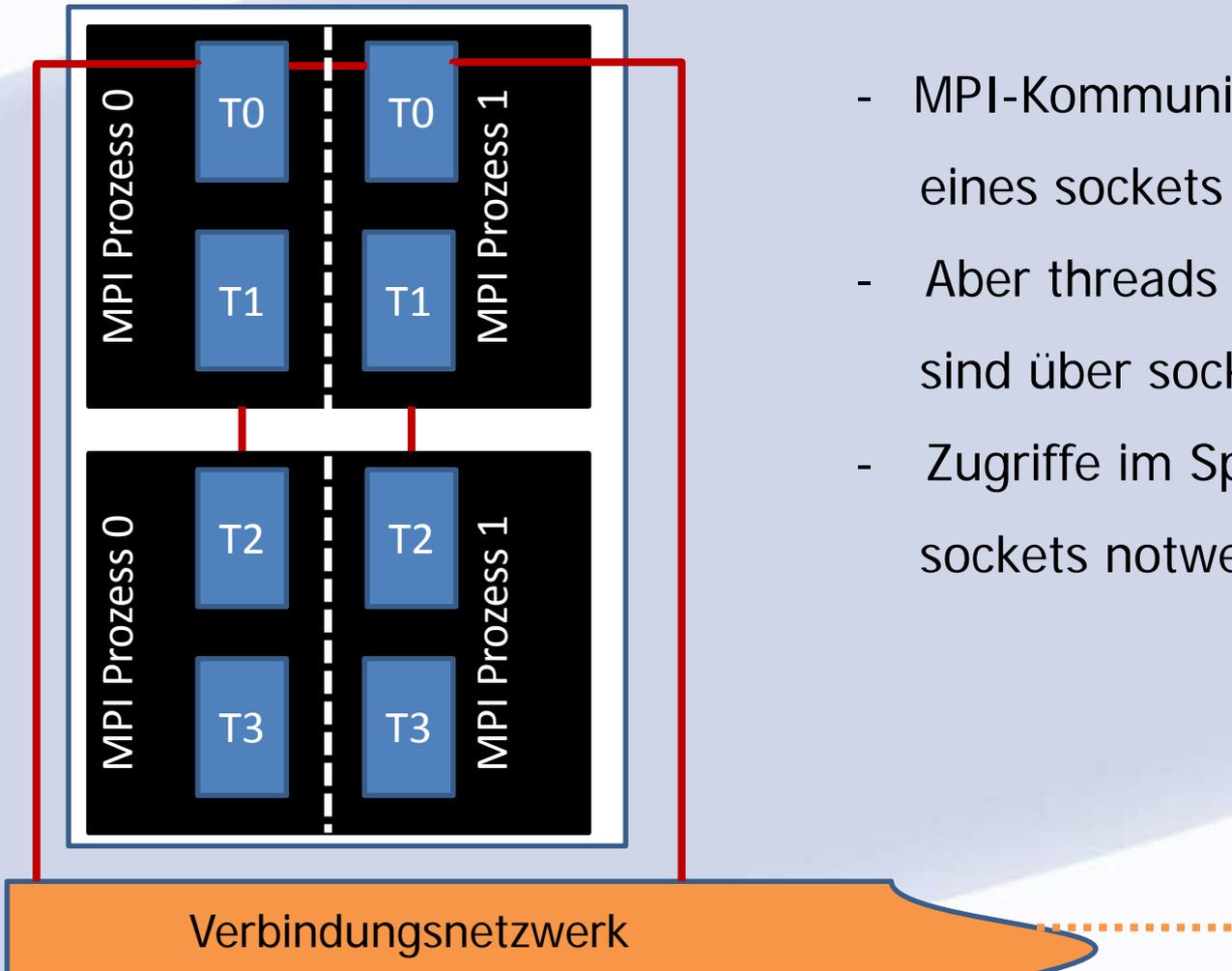
Abbildungsbeispiel 8 core 2 socket Knoten



- MPI Kommunikation im Knoten benutzt inter-socket Verbindungen
- OpenMP Speicherzugriffe finden innerhalb eines multicore socket statt

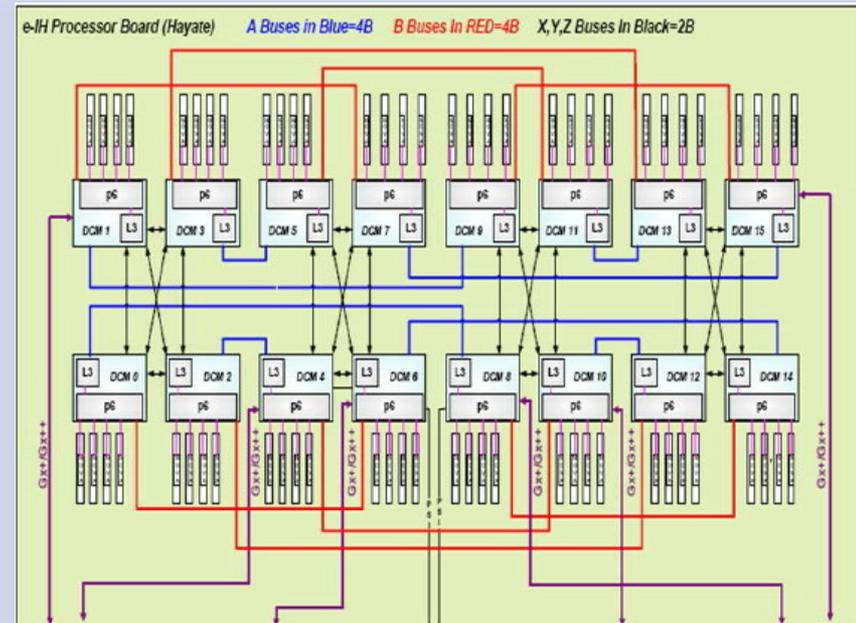
Abbildungsbeispiel 2 socket Knoten mit 4 cores/socket

- MPI-Kommunikation wird innerhalb eines sockets gehalten (master only)
- Aber threads des gleichen Prozesses sind über socket Grenzen verteilt
- Zugriffe im Speicher des anderen sockets notwendig



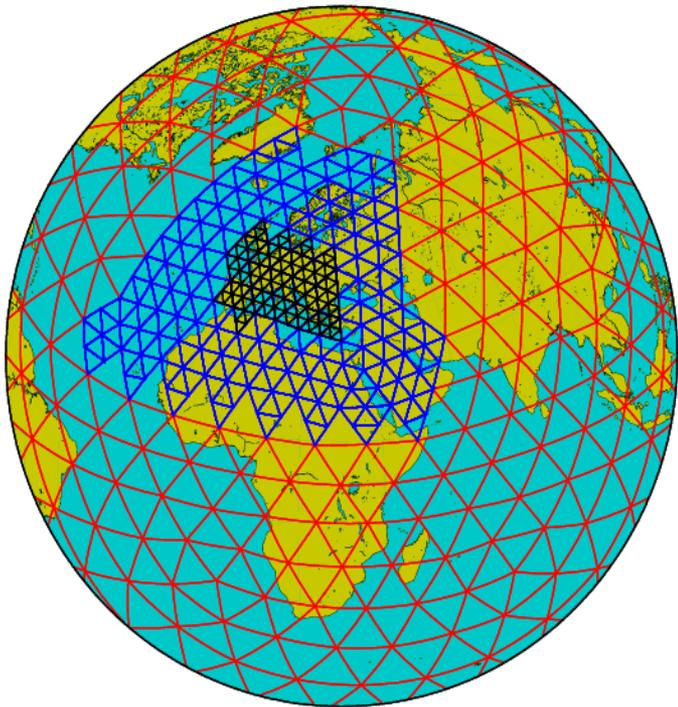
Power6 p575 Task Affinity

- ST mode 32 MPI Tasks/Knoten :
tasks_per_node = 32
task_affinity = core(1)
- SMT mode 64 MPI Tasks/Knoten :
tasks_per_node = 64
task_affinity = cpu(1)
- Hybrid ST mode
8 MPI Tasks x 4 OpenMP threads :
tasks_per_node = 8
task_affinity = core(4)
parallel_threads = 4
- Hybrid SMT mode
32 MPI Tasks x 2 OpenMP threads :
tasks_per_node = 32
task_affinity = cpu(2)
parallel_threads = 2



- Task Geometry Power6 p575
 - **task_geometry**={(*task id,task id,...*)(*task id,task id, ...*) ... }
 - **Beispiel:** Ein job mit 6 MPI Tasks auf 4 Knoten: **task_geometry**={(0,1) (3) (5,4) (2)}
- Finde die Optimale Abbildung der Prozess Topologie der Anwendung, auf die Topologie der Hardware
 - Finde optimalen Lastausgleich aller Recheneinheiten
 - Minimiere den Kommunikationsaufwand
 - Dieses Problem ist NP-complete

ICON auf IBM Power6 "blizzard"



- MPI (ST) 4 Knoten
32 MPI-Prozesse/Knoten
Wallclock : 779 sec
- MPI (SMT) 4 Knoten
64 MPI-Prozesse/Knoten
Wallclock : 549 sec
Gewinn : 29,5% zu MPI (ST)
- MPI/OpenMP (SMT) 4 Knoten
Pro Knoten:
32 MPI-Prozesse x 2 OpenMP Threads
Wallclock : 499 sec
Gewinn : 9% zu MPI (SMT)
Gewinn : 35,9% zu MPI (ST)

ICON auf IBM Power6 "blizzard"

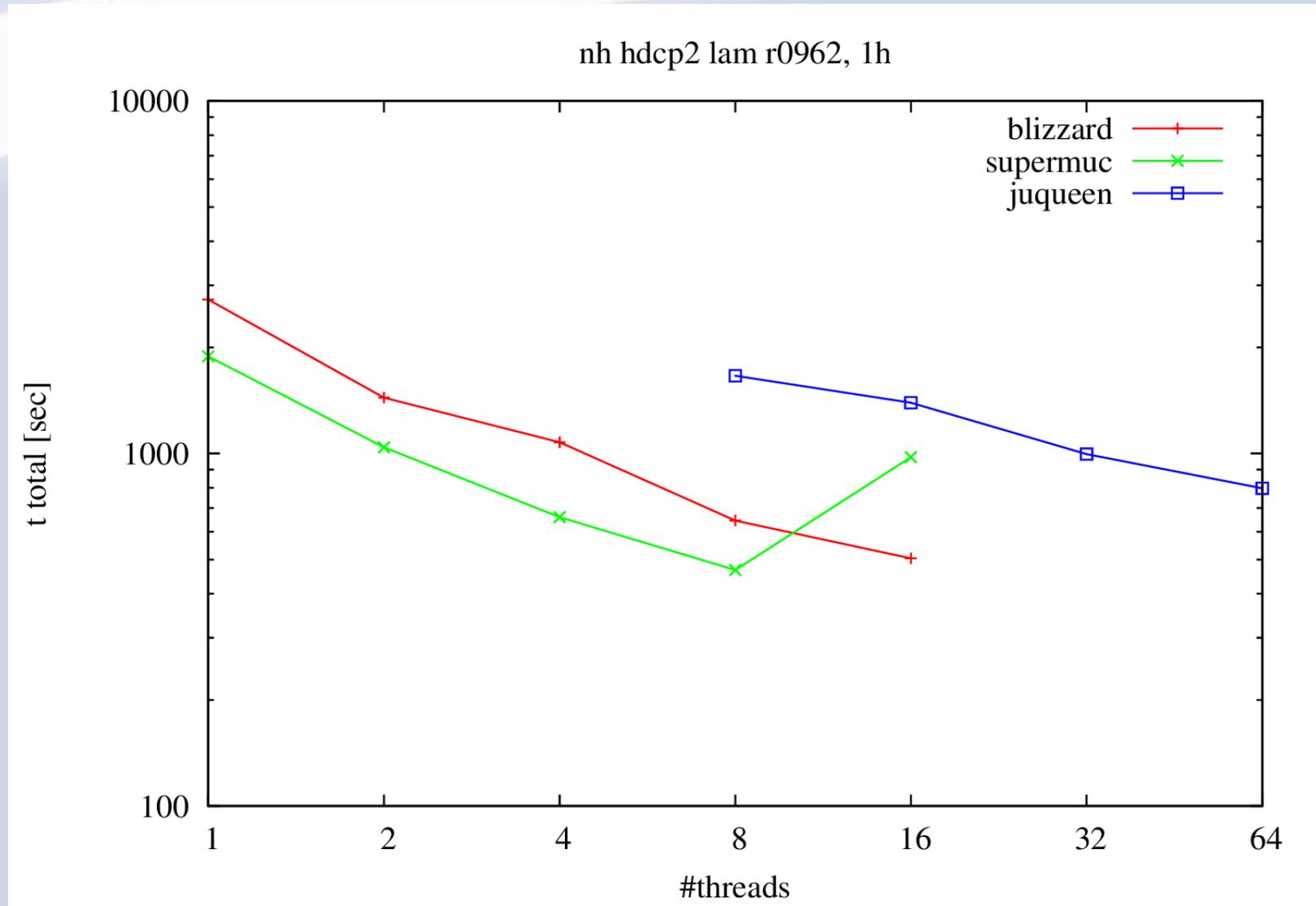
ST Mode

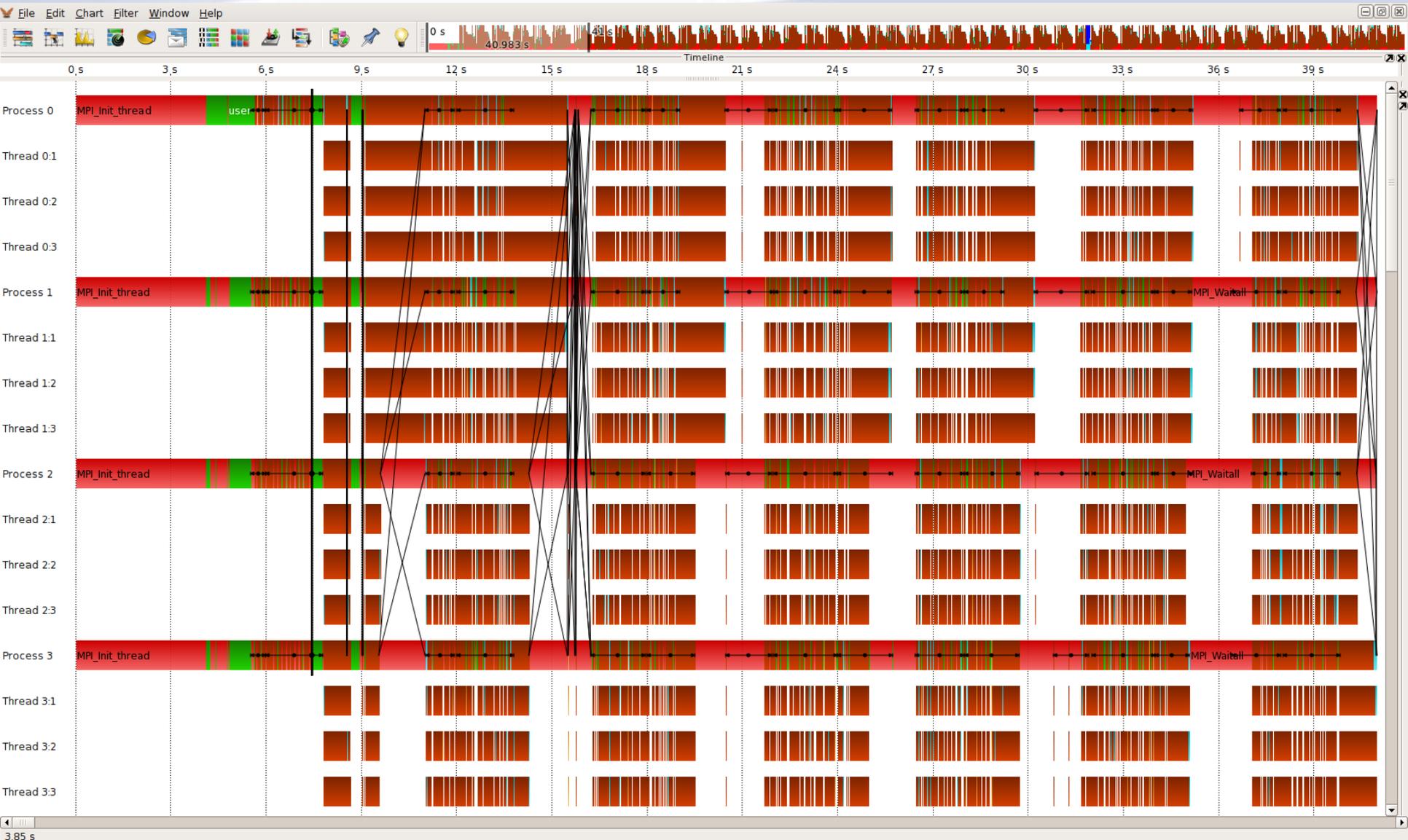
- MPI (ST) 1 Knoten
Wallclock : 2981 sec
- MPI (ST) 2 Knoten
Wallclock : 1516 sec
Strong Scaling=**1,96**
- MPI (ST) 4 Knoten
Wallclock : 779 sec
Strong Scaling=**3,82**
- MPI (ST) 8 Knoten
Wallclock : 410 sec
Strong Scaling=**7,27**

SMT Mode

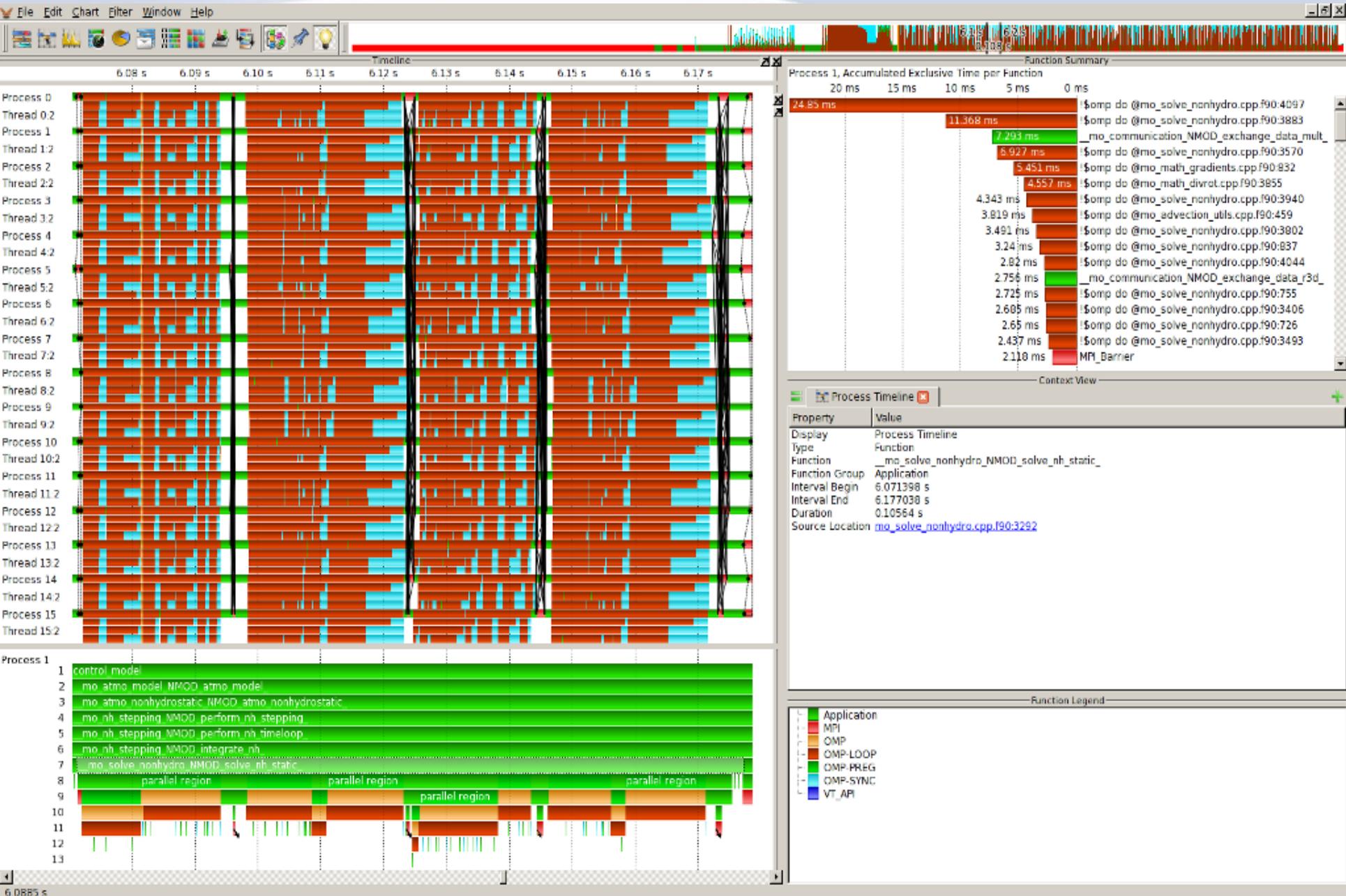
- MPI (SMT) 1 Knoten
Wallclock : 2021 sec
Strong Scaling=1,47 -> **75% vom Speedup auf 2 Knoten ST**
- MPI/OpenMP (SMT) 4 Knoten
32 MPI-Prozesse x 2 OpenMP Threads
Wallclock : 499 sec
Strong Scaling = 5,97 -> **82% vom Speedup auf 8 Knoten ST**

Thread Scaling @ 128 MPI Tasks



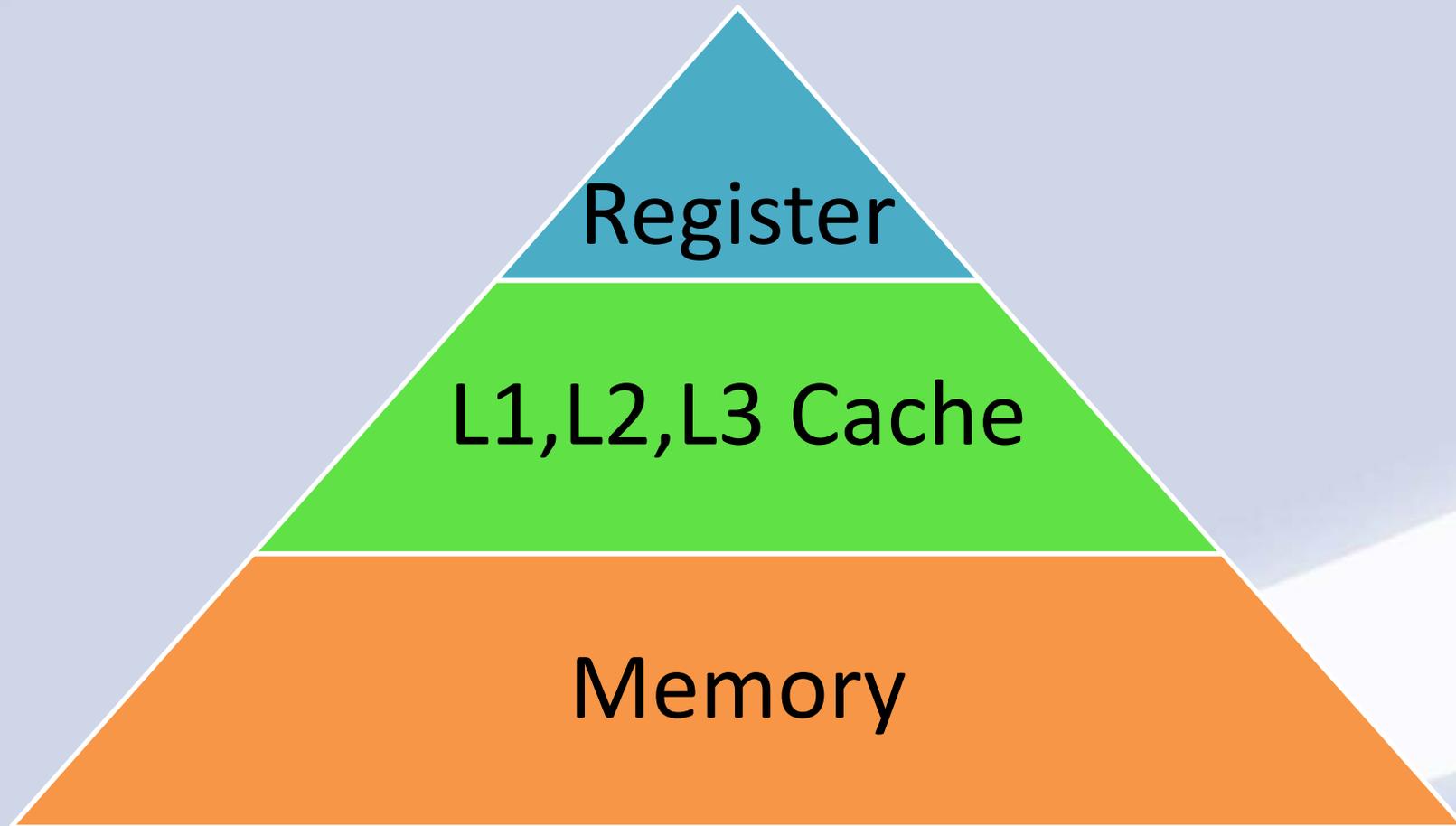


Static Schedule

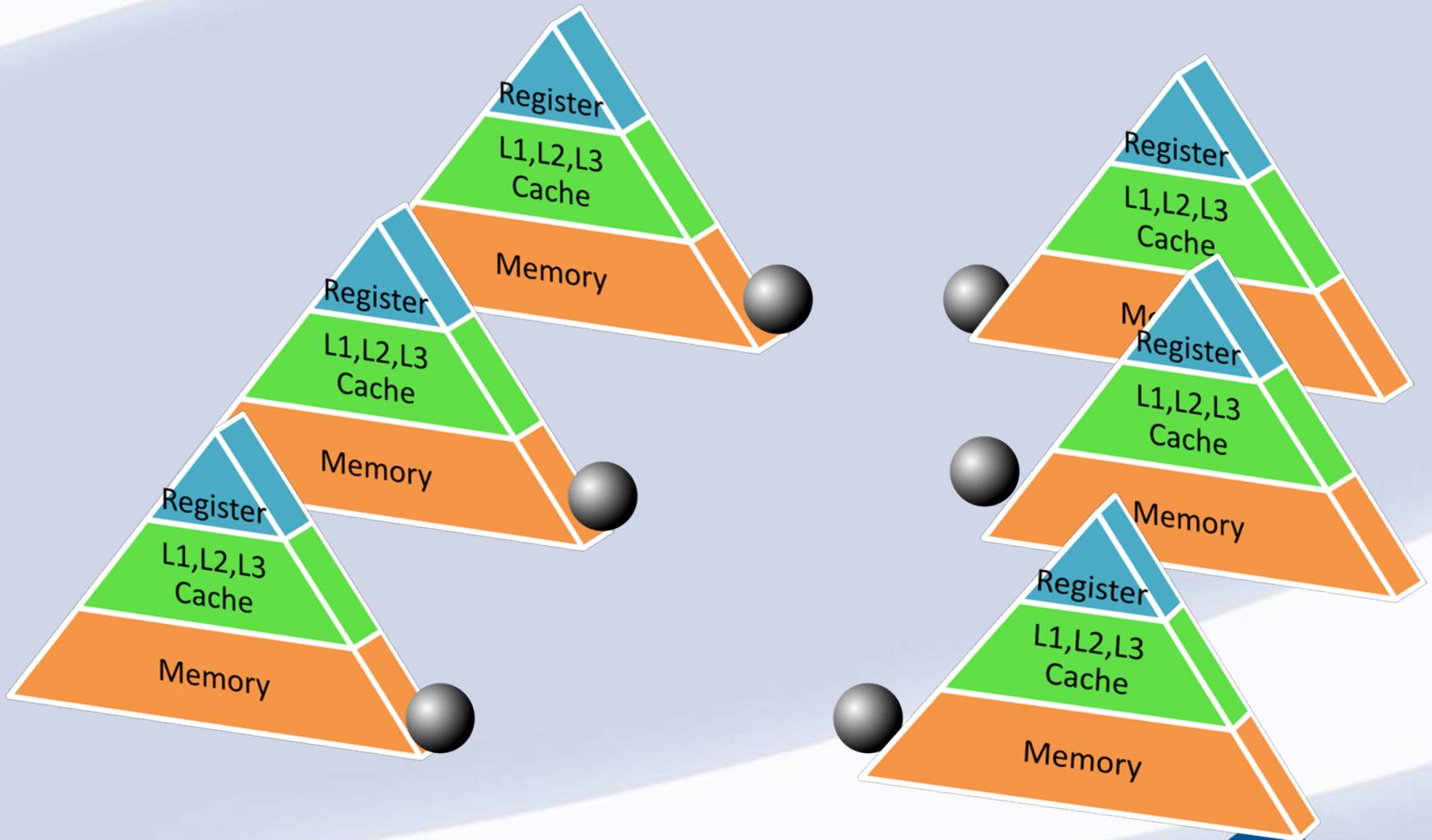


- Lösungsansätze
 - Die Prozess Topologie der Anwendung wird auf Graphen abgebildet
 - Das Lastausgleichproblem wird als Graphenpartitionierungs Problem formuliert
 - Die Partitionierungsalgorithmen basieren auf Heuristiken
 - Bekannte Bibliotheken: METIS, Jostle

Memory Hierarchy



Data Movement in Parallel Systems



Paralleler I/O in OpenMP

- OpenMP spezifiziert nichts über parallele I/O
- Programmierer ist verantwortlich das Parallele I/O im multi-threaded Programm sicherzustellen
- In Fortran, führt nicht synchronisierter I/O auf der gleichen “unit” zu einem nicht spezifiziertem Verhalten
- In hybriden Programmen kann man weiterhin parallele I/O in der MPI Welt machen

- Hybride MPI/OpenMP Parallelisierung hat das Potential die Skalierung von Anwendungen auf modernen hybriden Architekturen zu steigern
- MPI-Kommunikation innerhalb von SMP-Knoten kann dabei vermieden werden
- Multilevel Parallelisierung der Anwendungen wird dadurch ermöglicht

- Die MPI-Bibliothek muss thread safe sein
- Die Komplexität der hybrid parallelen Programmen ist grösser im Vergleich zu den nur MPI oder nur OpenMP parallelen Programmen
- Auswirkungen auf Korrektheit
 - Die Fehler sind schwieriger zu finden

- Auswirkungen auf Effizienz
 - Zusatzaufwand der Generierung und Synchronisation der threads
- Es ist schwierig die optimale Anzahl der OpenMP threads pro MPI-Prozess zu bestimmen
- Ein guter Lastausgleich zwischen allen Recheneinheiten des Systems hat sehr grosse Auswirkung auf die Effizienz des parallelen Programms

- “Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on large-scale Multicore Clusters”, Xingfu Wu and Valerie Taylor
- “Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming in Clusters of Multi-core SMP Nodes”, Georg Hager, Gabriele Jost, Rolf Rabenseifner
- <http://glaros.dtc.umn.edu/gkhome>
- <http://staffweb.cms.gre.ac.uk/~c.walshaw/partition>
- Bei Fragen: Panos Adamidis
DKRZ-Raum 213 (adamidis@dkrz.de)