Modelle der Parallelverarbeitung

12. Message Passing Interface

Thomas Worsch

Fakultät für Informatik
Karlsruher Institut für Technologie

Sommersemester 2017

Überblick

Überblick

BlueGene/P in Jülich

TOP 500 Liste November 2007

		Cores	TFlop/s peak	TFlop/s max
1	LLNL, USA			
	eServer Blue Gene, IBM	212992	596.4	478.2
2	FZ Jülich (FZJ)			
	Blue Gene/P Solution, IBM	65536	222.8	167.3
3	New Mexico Comp.App.Cent.			
	SGI Altix ICE 8200, SGI	14336	172.0	127.0
4	TATA SONS, India			
	Clust.Platf. 3000, Infini., HP	14240	170.9	117.9
5	Government Agency, Sweden			
	Clust.Platf. 3000, Infini., HP	13728	146.4	102.8

TOP 500 Liste Juni 2013

		Cores	TFlop/s peak	TFlop/s max
1	Nat. Univ. Defense, China			
	Tianhe-1	3120000	54902.4	33862.7
2	ORNL, USA			
	Titan	560640	27112.5	17590.0
3	LLNL, USA			
	Sequoia (BG/Q, IBM)	1572864	20173.2	17173.2
4	RIKEN, Japan			
	K computer, Fujitsu	705024	11280.38	10510.00
5	ANL, USA			
	Mira (BG/Q, IBM)	786432	10066.33	8162.38

Parallele Programmumgebungen: Ziele und Wege

Ziele:

- Effizienz
- Portabilität
- leichte Programmierbarkeit

Wege(?):

- direkter Hardwarezugriff
- parallele Programmiersprachen
  - evtl. pseudo via „Kommentare/Pragmas“ mit Bedeutung
- Unterprogrammbibliotheken

MPI

- *Message Passing Interface*
- der de facto Standard
- für Programmierung vom Standpunkt Nachrichtenaustausch
- `www.mpi-forum.org`
- spezifiziert
  - Konzepte, Datentypen, Schnittstellen von Funktionen
  - für Punkt-zu-Punkt-Kommunikation, kollektive Operationen, parallele E/A, …
- mehrere freie Implementierungen verfügbar (mpich, openmpi, …)

Überblick

Geschichte

- MPI 1
  - 1994: MPI 1.0
  - 1995: MPI 1.1
  - 1996: MPI 1.2 (?)
  - 2008: MPI 1.3
- MPI 2
  - 1996: MPI 2.0
  - 2008: MPI 2.1
  - 2009: MPI 2.2
- MPI 3
  - 2012: MPI 3.0
  - 2015: MPI 3.1

Begriffe

- Idee (vereinfacht): *SPMD*
  - *ein* Programm wird mehrfach gestartet
- es kommunizieren *Prozesse* (nicht Prozessoren)
- in *Kommunikatoren*
  - Gruppe, Kontext, virtuelle Topologie
- wichtige Funktionsgruppen von MPI2:
  - Punkt-zu-Punkt-Kommunikation (zweiseitig)
  - kollektive Kommunikation
  - einseitige Kommunikation (RMA)
  - parallele Ein-/Ausgabe

Hello World

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int myRank, numProcs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  printf("my rank: %d in comm of size %d.\n",
         myRank, numProcs);

  MPI_Finalize();
}
```

Hello World

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int myRank, numProcs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  printf("my rank: %d in comm of size %d.\n",
         myRank, numProcs);

  MPI_Finalize();
}
```

Hello World

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int myRank, numProcs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  printf("my rank: %d in comm of size %d.\n",
         myRank, numProcs);

  MPI_Finalize();
}
```

Hello World

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int myRank, numProcs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  printf("my rank: %d in comm of size %d.\n",
         myRank, numProcs);

  MPI_Finalize();
}
```

Hello World

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char* argv[]) {
  int myRank, numProcs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  printf("my rank: %d in comm of size %d.\n",
         myRank, numProcs);

  MPI_Finalize();
}
```

Running „Hello World“

- `mpicc hello-world.c -o ./hello`
- `mpiexec -n 4 ./hello`
- Ausgabe: (vielleicht)

```
my rank: 2 in comm of size 4.
my rank: 1 in comm of size 4.
my rank: 0 in comm of size 4.
my rank: 3 in comm of size 4.
```

Running „Hello World“

- `mpicc hello-world.c -o ./hello`
- `mpiexec -n 4 ./hello`
- Ausgabe: (vielleicht)

```
my rank: 2 in comm of size 4.
my rank: 1 in comm of size 4.
my rank: 0 in comm of size 4.
my rank: 3 in comm of size 4.
```

Running „Hello World“

- `mpicc hello-world.c -o ./hello`
- `mpiexec -n 4 ./hello`
- Ausgabe: (vielleicht)

```
my rank: 2 in comm of size 4.
my rank: 1 in comm of size 4.
my rank: 0 in comm of size 4.
my rank: 3 in comm of size 4.
```

Running „Hello World“

- `mpicc hello-world.c -o ./hello`
- `mpiexec -n 4 ./hello`
- Ausgabe: (vielleicht)

```
my rank: 2 in comm of size 4.
my rank: 1 in comm of size 4.
my rank: 0 in comm of size 4.
my rank: 3 in comm of size 4.
```

Running „Hello World“

- `mpicc hello-world.c -o ./hello`
- `mpiexec -n 4 ./hello`
- Ausgabe: (vielleicht)

```
my rank: 2 in comm of size 4.
my rank: 1 in comm of size 4.
my rank: 0 in comm of size 4.
my rank: 3 in comm of size 4.
```

- wir tun so, als dürften alle Ausgaben erzeugen …

Überblick

MPI_COMM_WORLD

- ein sogenannter Kommunikator
- umfasst *alle* Prozesse, die gestartet wurden

Kommunikator

- umfasst eine *Gruppe* von Prozessen
- legt einen *Kontext* fest, innerhalb dessen die Prozesse kommunizieren
- zwei Kommunikatoren
  - können die gleiche Gruppe umfassen,
  - definieren aber *immer* verschiedene Kontexte
- Kommunikationen in verschiedenen Kontexten haben nichts miteinander zu tun, stören sich nie, …
  - (die Wirklichkeit ist ein bisschen komplizierter)
- an Kommunikator kann *virtuelle Topologie* gebunden sein

Überblick

Pingpong (zwischen zwei Prozessen)

```
if (myRank == 0) {
  MPI_Send(buffer, msgLen, MPI_DOUBLE,
           1, TAG_PING, MPI_COMM_WORLD);
  MPI_Recv(buffer, BUFFERSIZE, MPI_DOUBLE,
           1, TAG_PONG, MPI_COMM_WORLD, &status);
}
if (myRank == 1) {
  MPI_Recv(buffer, BUFFERSIZE, MPI_DOUBLE,
           0, TAG_PING, MPI_COMM_WORLD, &status);
  MPI_Send(buffer, msgLen, MPI_DOUBLE,
           0, TAG_PONG, MPI_COMM_WORLD);
}
```

Pingpong (zwischen zwei Prozessen)

```
if (myRank == 0) {
  MPI_Send(buffer, msgLen, MPI_DOUBLE,
           1, TAG_PING, MPI_COMM_WORLD);
  MPI_Recv(buffer, BUFFERSIZE, MPI_DOUBLE,
           1, TAG_PONG, MPI_COMM_WORLD, &status);
}
if (myRank == 1) {
  MPI_Recv(buffer, BUFFERSIZE, MPI_DOUBLE,
           0, TAG_PING, MPI_COMM_WORLD, &status);
  MPI_Send(buffer, msgLen, MPI_DOUBLE,
           0, TAG_PONG, MPI_COMM_WORLD);
}
```

Pingpong (zwischen zwei Prozessen)

```
if (myRank == 0) {
  MPI_Send(buffer, msgLen, MPI_DOUBLE,
           1, TAG_PING, MPI_COMM_WORLD);
  MPI_Recv(buffer, BUFFERSIZE, MPI_DOUBLE,
           1, TAG_PONG, MPI_COMM_WORLD, &status);
}
if (myRank == 1) {
  MPI_Recv(buffer, BUFFERSIZE, MPI_DOUBLE,
           0, TAG_PING, MPI_COMM_WORLD, &status);
  MPI_Send(buffer, msgLen, MPI_DOUBLE,
           0, TAG_PONG, MPI_COMM_WORLD);
}
```

Pingpong (zwischen zwei Prozessen)

```
if (myRank == 0) {
  MPI_Send(buffer, msgLen, MPI_DOUBLE,
           1, TAG_PING, MPI_COMM_WORLD);
  MPI_Recv(buffer, BUFFERSIZE, MPI_DOUBLE,
           1, TAG_PONG, MPI_COMM_WORLD, &status);
}
if (myRank == 1) {
  MPI_Recv(buffer, BUFFERSIZE, MPI_DOUBLE,
           0, TAG_PING, MPI_COMM_WORLD, &status);
  MPI_Send(buffer, msgLen, MPI_DOUBLE,
           0, TAG_PONG, MPI_COMM_WORLD);
}
```

Senden und Empfangen

- `MPI_Send(void *b, int n, MPI_Datatype d, ....)`
  - Sendepuffer `b`
  - Anzahl `n` zu versendender Elemente vom
  - Datentyp `d`
- `MPI_Recv(void *b, int n, MPI_Datatype d, ....)`
  - Empfangspuffer `b`
  - für maxmimal `n` Elemente vom
  - Datentyp `d`
- beide Operationen *blockierend*:
  nach Beendigung der Aufrufe kann auf die Puffer „ohne Überraschungen“ zugegriffen werden

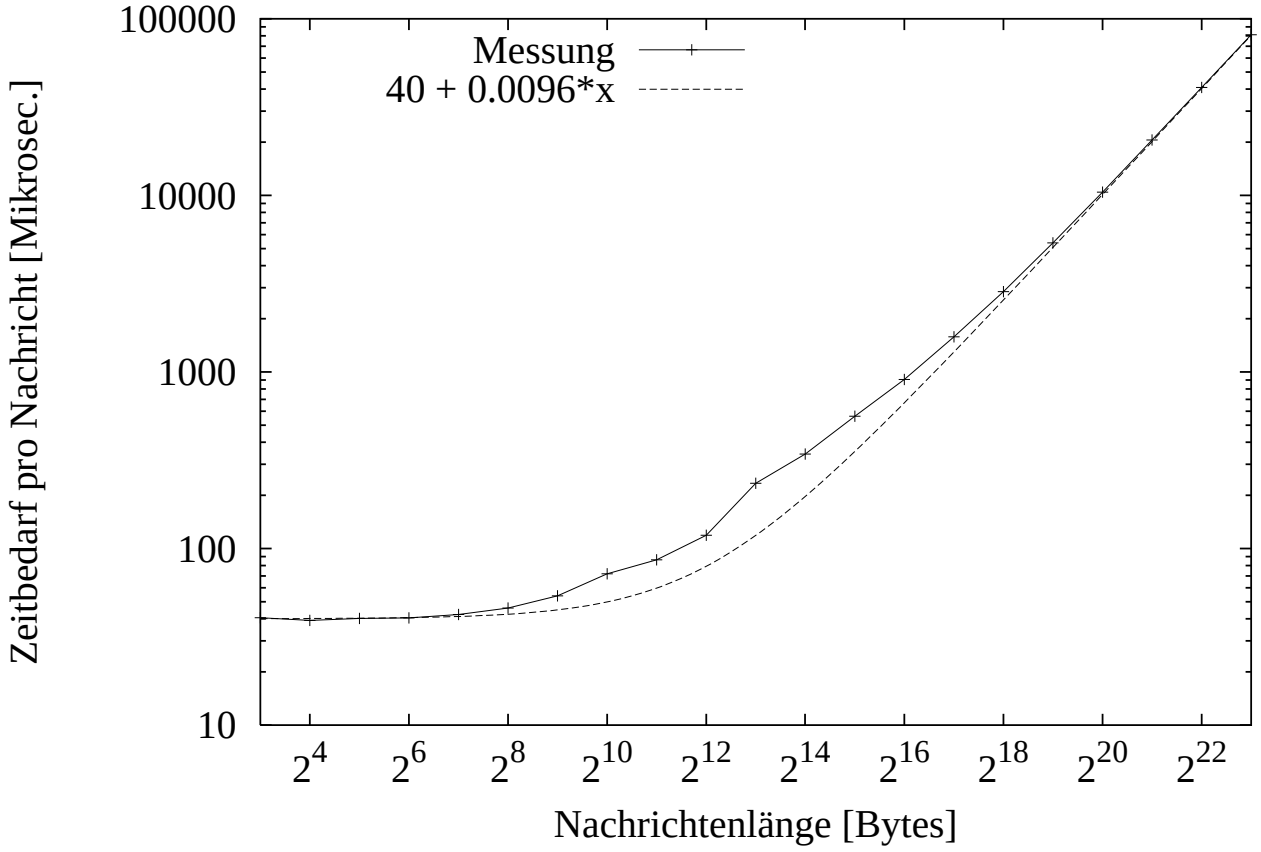
Senden und Empfangen

- `MPI_Send`
  `(...., int dest, int tag, MPI_Comm comm)`
  - Rang `dest` des Empfängers
  - im Kommunikator `comm`
  - „Aufkleber“ `tag`
- `MPI_Recv`
  `(...., int src, int tag, MPI_Comm comm, MPI_Status *status)`
  - Rang `src` des Senders (oder `MPI_ANY_SOURCE`)
  - im Kommunikator `comm`
  - „Aufkleber“ `tag` (oder `MPI_ANY_TAG`)
  - Infos in `status`

Regeln

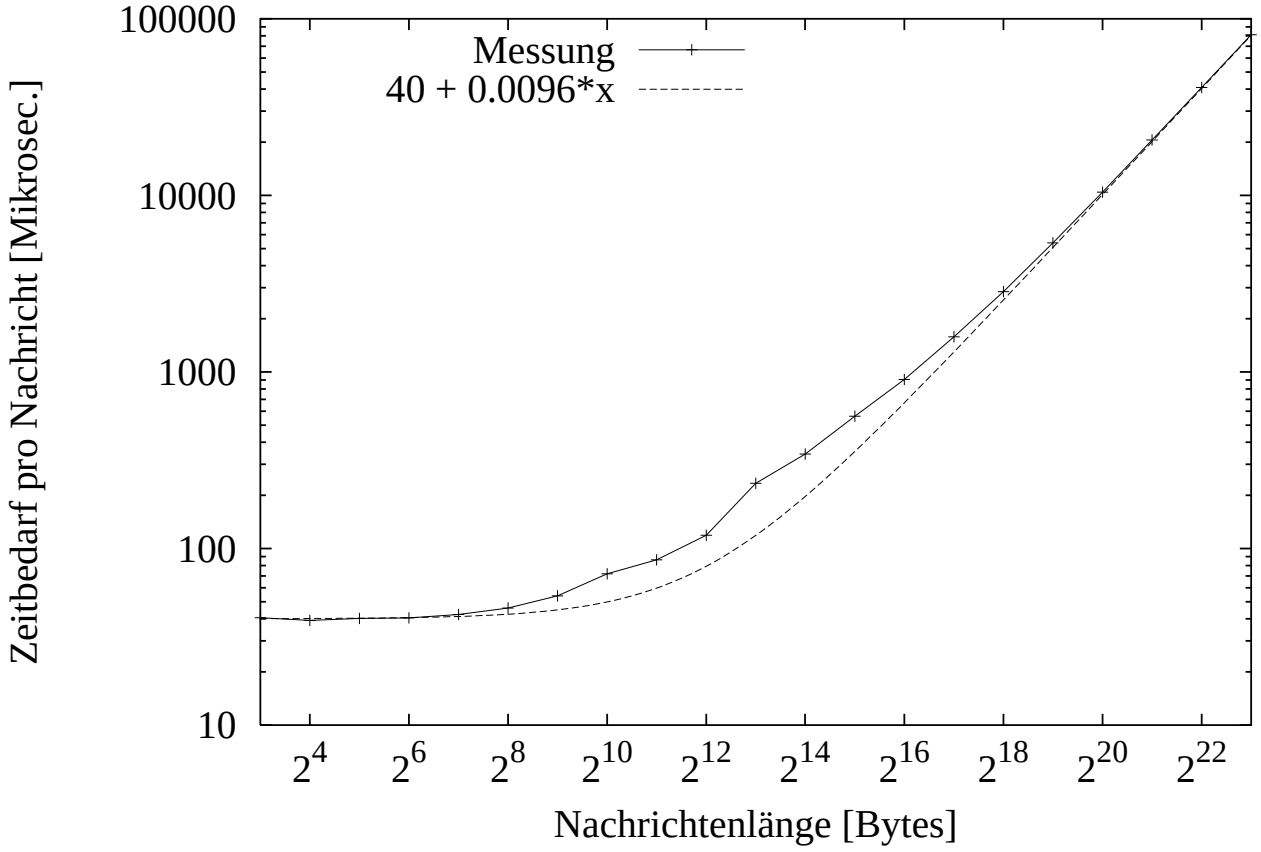
- Sende- und Empfangsoperation müssen zusammenpassen
  - gleicher Kommunikator
  - passende Sender-/Empfänger-Angaben
  - passende Tag-Angaben
  - („fast“) gleiche Daten-Typ/-Anzahl-Angaben
- fürs Senden gibt es weitere Varianten
- es gibt auch *nichtblockierende* Operationen
  - auf die Beendigung des „Datentransportes“ muss dann mittels anderer Funktionen gewartet werden

Ein typisches Messergebnis (alte IBM SP)



- aktuelle Maschinen sind deutlich schneller

Ein typisches Messergebnis (alte IBM SP)



- aktuelle Maschinen sind deutlich schneller
- $T(m) = \alpha + \beta m$ typischerweise angenommene Näherung

Zeitmessung

- `double MPI_Wtime( void )`
- Aufruf `MPI_Wtime()`
  - liefert die Zeit in Sekunden
  - in dem aufrufenden Prozess
  - seit irgendeinem Zeitpunkt
- Auflösung der Uhr: `MPI_WTICK`

- im allgemeinen *keine Vergleichbarkeit* der Uhrzeiten verschiedener Prozesse
  - (man kann `MPI_WTIME_IS_GLOBAL` inspizieren)
- daher Pingpong-Messungen
  - Annahme: Kommunikation hin und her gleich schnell

Überblick

Kollektive Operationen

- umfassen *alle* an einem Kommunikator beteiligten Prozesse
- Annahme im Folgenden
  - es geht um sogenannte Intrakommunikatoren
  - Interkommunikatoren wurden gar nicht erwähnt

- manchmal alle Prozesse gleichberechtigt

  (`MPI_Barrier`, `MPI_Alltoall`, ...)
- manchmal einer ausgezeichnet („root“)

  (`MPI_Bcast`, `MPI_Reduce`, ...)

Beispiele

- `MPI_Barrier`: Synchronisation
- `MPI_Bcast`: etwas von einem an alle
- `MPI_Reduce`: von allen zu einem
  - `MPI_Allreduce`: Ergebnis an alle
- `MPI_Scan`
- `MPI_Scatter`: von einem je ein Teil an jeden
- `MPI_Gather`: je ein Teil von jedem zu einem
  - `MPI_Allgather`: Ergebnis an alle
- `MPI_Alltoall`: von jedem je ein Teil an je einen
- ...
- MPI-3: *nichtblockierende* kollektive Operationen

MPI_Barrier

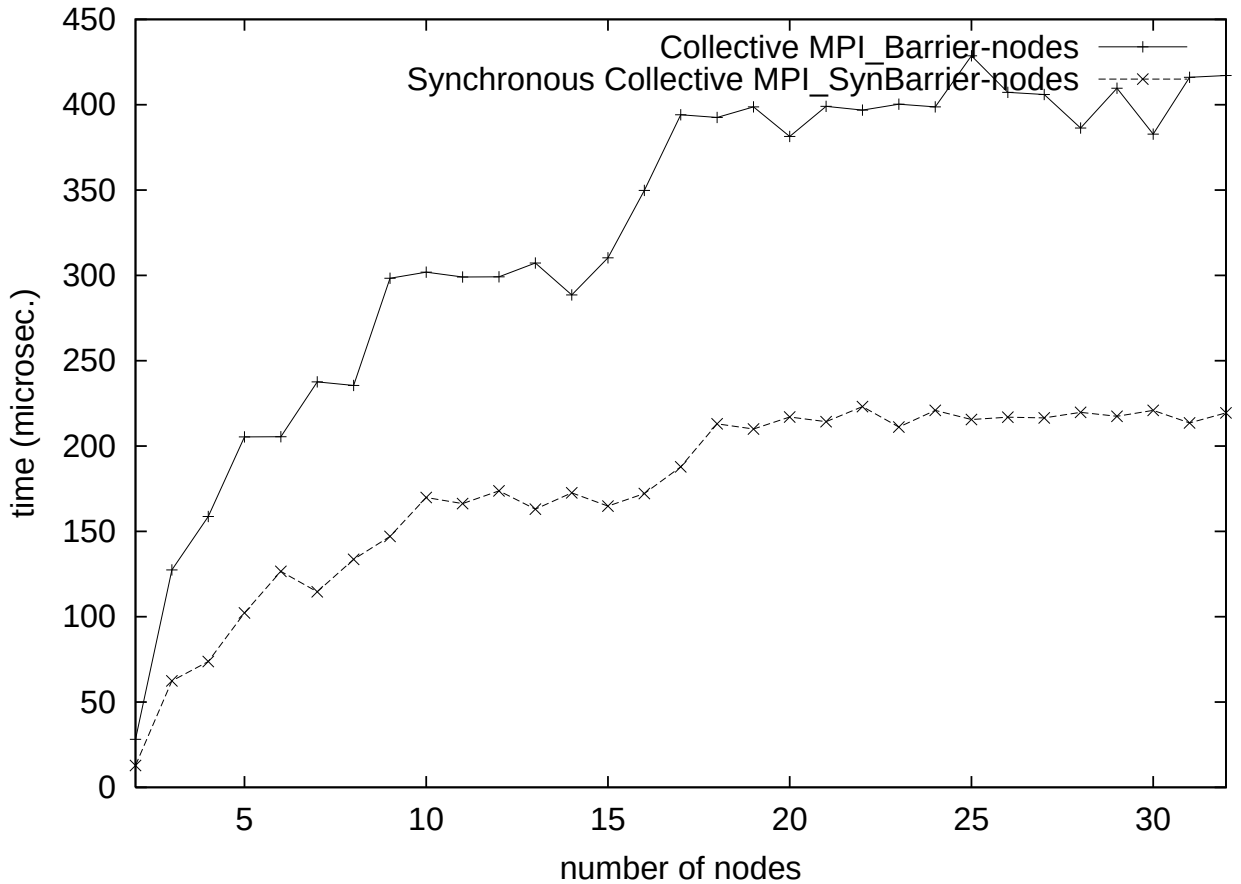
- *Synchronisation*
- Der Aufruf endet in jedem Prozess erst, wenn alle beteiligten Prozesse es aufgerufen haben.

- sparsam verwenden

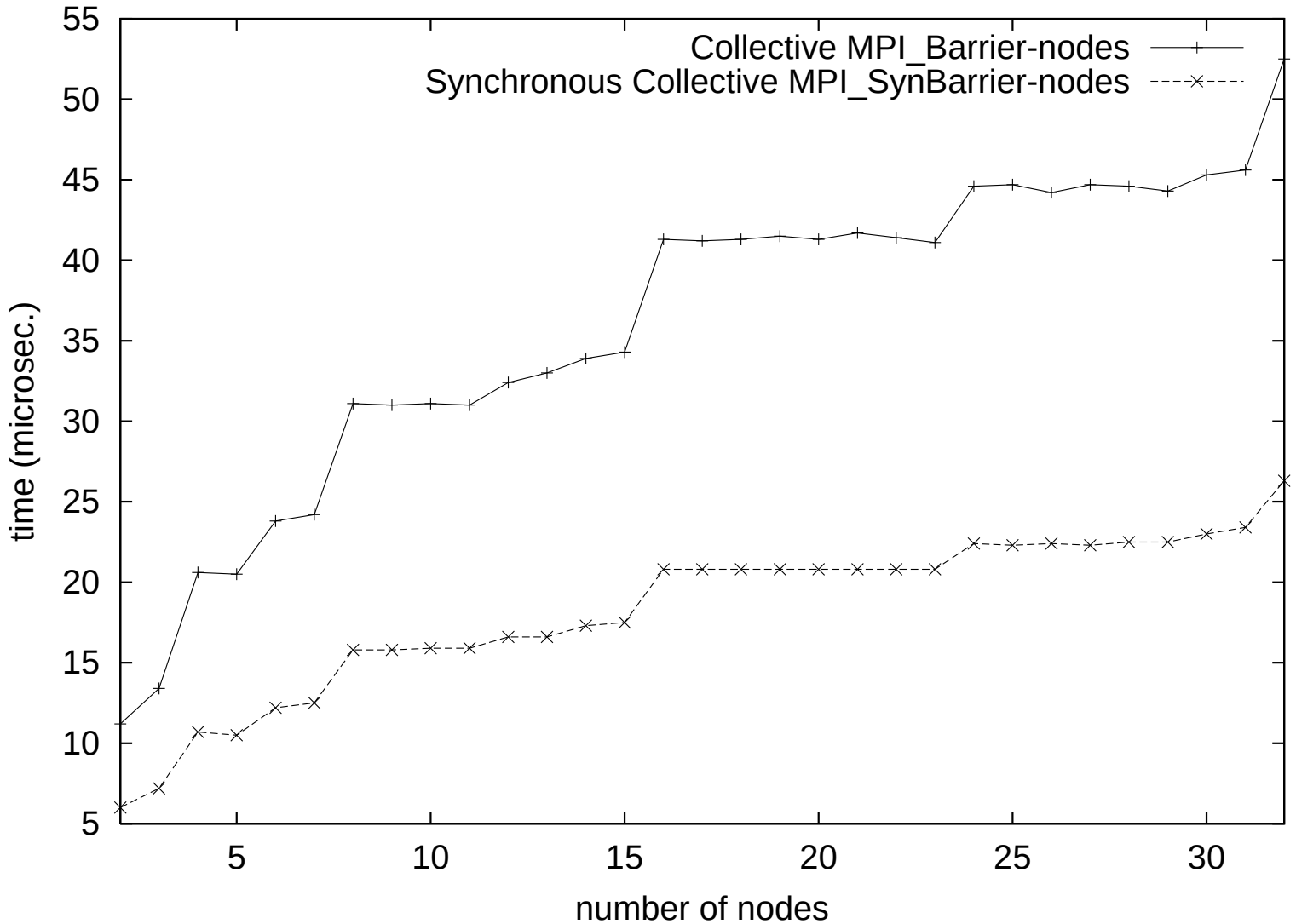
- `MPI_Barrier(` ⟨*comm*⟩ `)`

Barrier auf der ehemaligen IBM SP

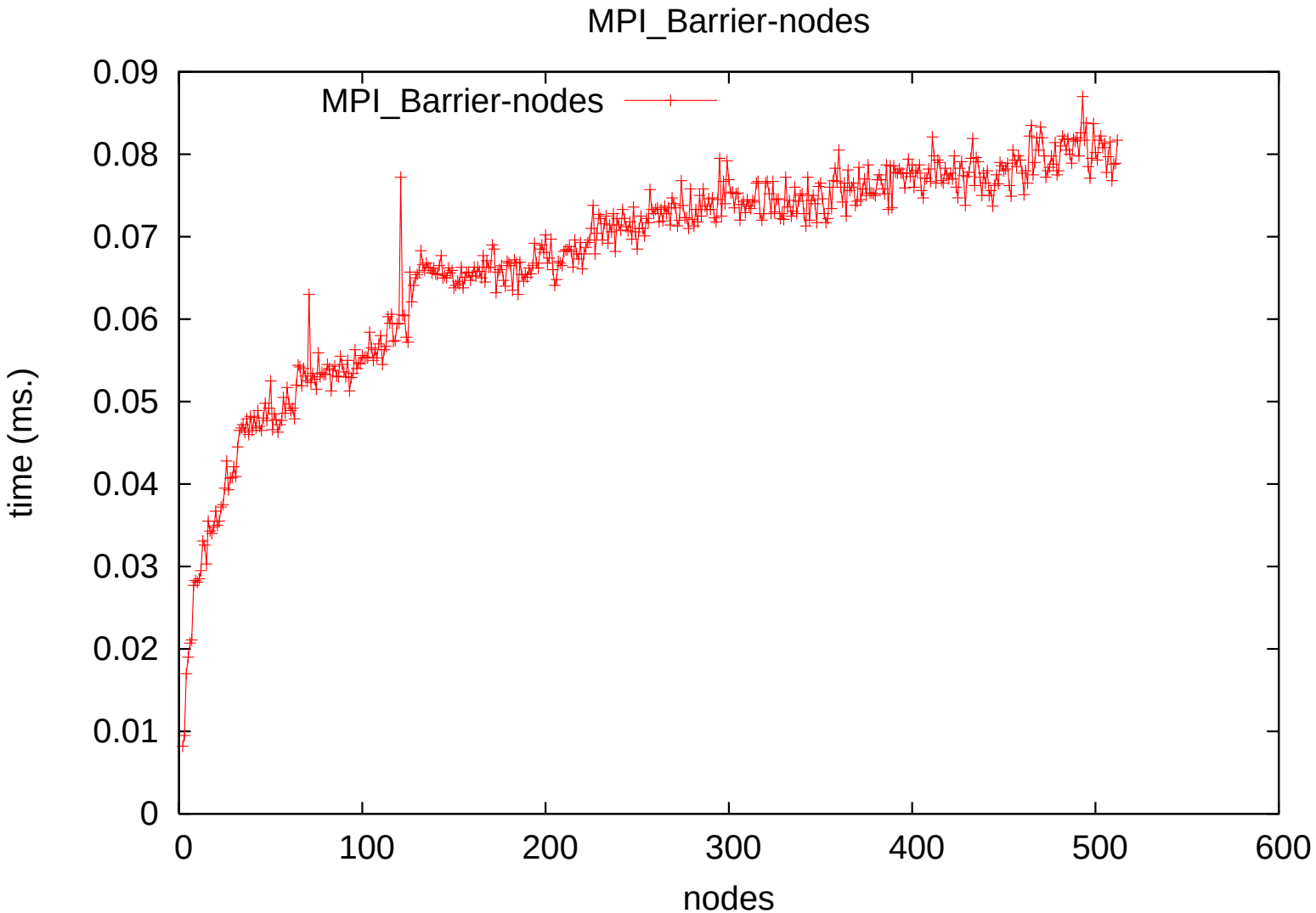
vermessen mit SKaMPI mit alter und neuer Messmethode



Barrier auf einer T3E



Barrier auf der XC2



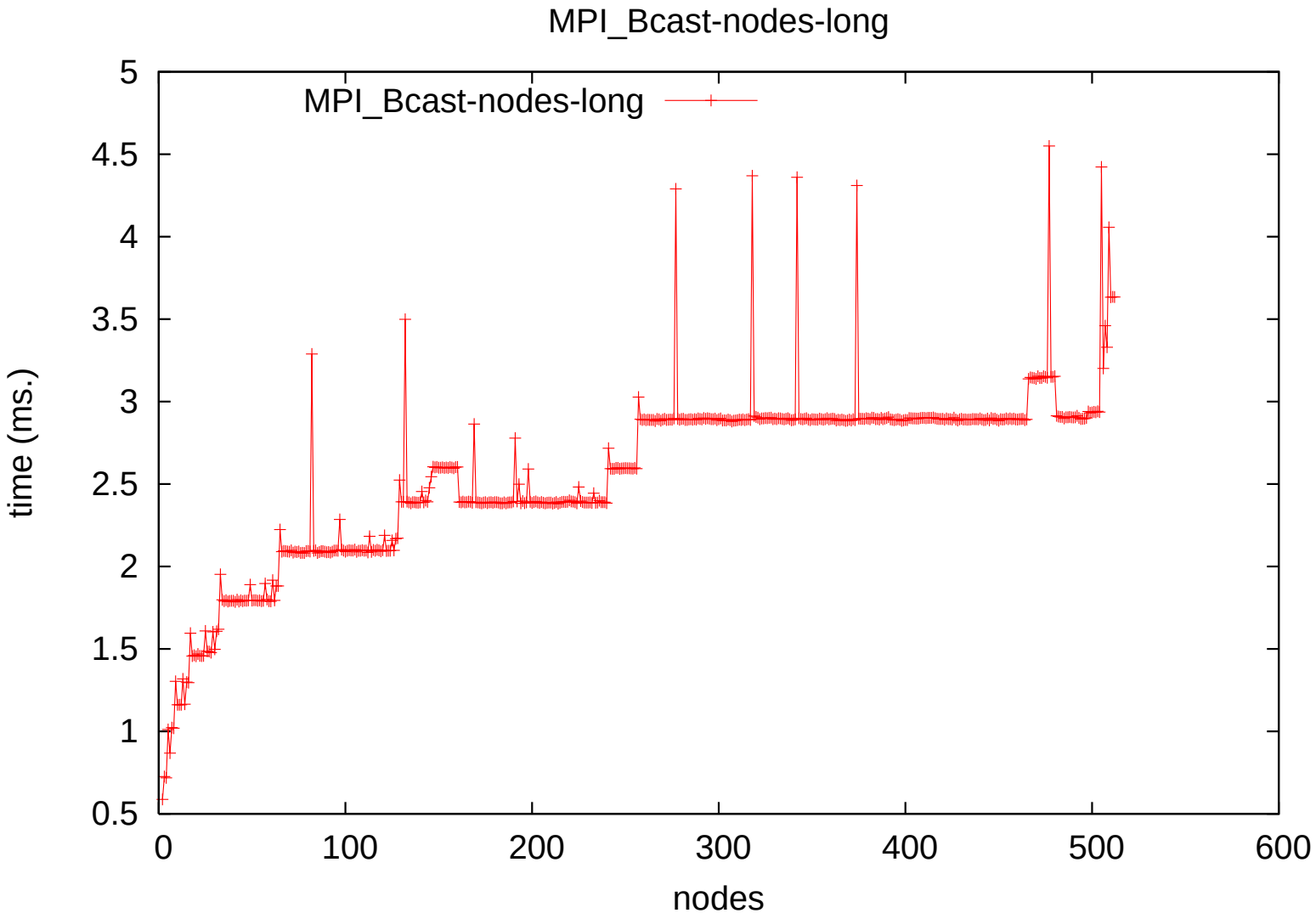
MPI_Bcast

- Verteilung *von einem alles an alle*
- `MPI_Bcast`(⟨*buffer*⟩,⟨*count*⟩,⟨*datatype*⟩,⟨*root*⟩,⟨*comm*⟩)
- Wie implementiert man das effizient?

MPI_Bcast

- Verteilung *von einem alles an alle*
- `MPI_Bcast`(⟨*buffer*⟩,⟨*count*⟩,⟨*datatype*⟩,⟨*root*⟩,⟨*comm*⟩)
- Wie implementiert man das effizient?
  - Sanders/Speck/Träff: Two-tree algorithms for full bandwidth broadcast, reduction and scan. Parallel Computing 35 (2009).

Broadcast auf der XC2 (256 KiB)



MPI_Scatter und MPI_Gather

- MPI_Scatter: ⟨*root*⟩ sendet i-ten Teil des Puffers an Prozess i
- MPI_Gather: „invers dazu“
- MPI_Allgather: wie MPI_Gather, aber Ergebnis bei allen

- MPI_Scatter(⟨*sendbuf*⟩, ⟨*sendcount*⟩, ⟨*sendtype*⟩,
  ⟨*recvbuf*⟩, ⟨*recvcount*⟩, ⟨*recvtype*⟩,
  ⟨*root*⟩, ⟨*comm*⟩)

- Was macht MPI_Scatter gefolgt von MPI_Allgather ?!

MPI_Alltoall

jeder Prozess

- sendet i-ten Teil des Puffers an Prozess i
- legt die empfangenen Teile
  in der Reihenfolge der Senderänge in Empfangspuffer

- `MPI_Alltoall`(⟨*sendbuf*⟩, ⟨*sendcount*⟩, ⟨*sendtype*⟩,
  ⟨*recvbuf*⟩, ⟨*recvcount*⟩, ⟨*recvtype*⟩,
  ⟨*comm*⟩)

- weitere Varianten, je nachdem,
  was man über die Länge der Einzelteile etc. „weiß“

MPI_Reduce

- Datenelemente aller beteiligten Prozesse werden gemäß einer binären Operation verknüpft
- `MPI_Reduce`(⟨*sendbuf*⟩, ⟨*recvbuf*⟩, ⟨*count*⟩, ⟨*datatype*⟩, ⟨*op*⟩, ⟨*root*⟩, ⟨*comm*⟩)

- Reduktions-Operationen
  - diverse vordefinierte stehen zur Verfügung
  - assoziativ, evtl. kommutativ

  - man kann selbst weitere (assoziative) definieren

SKaMPI

- Mikro-Benchmark für MPI-Implementierungen
- kann mal wieder Betreuung brauchen !!!!!