TECHNISCHE UNIVERSITÄT DRESDEN
Fakultät Informatik, Institut für Technische Informatik, Professur Rechnerarchitektur

Einführung in die Technische Informatik

Programmierung von Parallelrechnern

Zellescher Weg 12
Willers-Bau A 205
Tel. +49 351 - 463 - 35450

Nöthnitzer Straße 46
Raum 1044
Tel. +49 351 - 463 - 38246

Wolfgang E. Nagel (wolfgang.nagel@tu-dresden.de)

ZIH
Center for Information Services & High Performance Computing

Gliederung

- Message Passing Programmierung
  - MPI (Message Passing Interface)
- Shared Memory Programmierung
  - OpenMP
- Partial Global Adress Space (PGAS) Programmierung
  - UPC

Message Passing Programmierung

- Message Passing Programmiermodell ist Abstraktion eines Parallelrechners mit verteiltem Speicher
- Message Passing Programm besteht aus einer Anzahl von Prozessen mit zugeordneten lokalen Daten
- Jeder Prozess kann auf seine lokalen Daten zugreifen und mit anderen Prozessen Informationen durch expliziten Nachrichtenaustausch austauschen
- Üblicherweise führt jeder Prozess das gleiche Programm aus (SPMD-Stil)
  - Abzweigung im Programm in Abhängigkeit von der Prozessnummer
- Kommunikations-Operationen
  - Werden dem Programmierer in Form einer Programmbibliothek zur Verfügung gestellt
  - Müssen im Message Passing Programm explizit angegeben werden
- Kommunikationsbibliotheken
  - Heute meist portabel und standardisiert
  - PVM (Parallel Virtual Machine) war Quasi-Standard
  - MPI (Message Passing Interface) ist heute weit verbreitet

Message Passing Programmierung

MPI-Forum

- 40 Organisationen
- Hauptsächlich Forschungseinrichtungen und Hersteller von Message Passing Programmpaketen

MPI als Standard

Liefert Programm-Schnittstellen für Kommunikationsoperationen in Verbindung mit Fortran und C, ab MPI-2 auch für C++

(in dieser Vorlesung Beschränken uns auf C-Schnittstellen)

- Mai 1994: MPI-Standard Version 1.0
- Juni 1995: MPI-Standard Version 1.1
- Juli 1997: MPI-Standard Version 2.0
- Juni 2008: MPI-Standard Version 2.1
- Sept. 2009: MPI-Standard Version 2.2
- Sept. 2012: MPI-Standard Version 3.0

MPI-1 Basisfunktionen

- MPI-1 enthält mehr als 120 Funktionen
- 6 Basisfunktionen reichen jedoch für ein ersten MPI-Programm

– **Initialisierung der MPI-Umgebung**

int MPI_Init(int *argc, char ***argv)

- Erste MPI-Funktion in einem MPI-Programm
- Darf nur einmal aufgerufen werden

– **Ermittlung der Prozessnummer in der mit einem Kommunikator assoziierten Gruppe**

int MPI_Comm_rank(MPI_Comm comm, int *rank)

MPI-1 Basisfunktionen

– **Ermittlung der Prozess-Anzahl in der mit einem Kommunikator assoziierten Gruppe**

int MPI_Comm_size(MPI_Comm comm, int *size)

– **Einzeltransferoperation zum Senden**

– **Einzeltransferoperation zum Empfangen**

– **Beenden der MPI-Umgebung**

int MPI_Finalize(void)

Lokale Sicht (Sicht des aufrufenden Prozesses)

Blockierende Kommunikationsanweisung

- Eine MPI-Kommunikationsoperation heißt blockierend, wenn die Rückkehr zum aufrufenden Prozess bedeutet:
  - Alle Ressourcen (z.B. Puffer), die für den Aufruf benötigt wurden, können erneut für andere Operationen genutzt werden.

Nichtblockierende Kommunikationsanweisung

- Eine MPI-Kommunikationsoperation heißt nichtblockierend
  - Wenn die aufgerufene Kommunikationsanweisung die Kontrolle zurückgibt
    - Bevor die durch sie ausgelöste Operation vollständig abgeschlossen ist und
    - Bevor eingesetzte Ressourcen (z.B. Puffer) wieder benutzt werden dürfen

- Die Operation ist erst dann vollständig abgeschlossen, wenn alle Ressourcen wieder verwendet werden können

Globale Sicht

Zusammenspiel der an einer Kommunikation beteiligten Prozesse wird durch die Eigenschaften synchroner oder asynchroner Kommunikation beschrieben

Synchrone Kommunikation

- Eigentliche Übertragung einer Nachricht findet nur statt, wenn Sender und Empfänger zur gleichen Zeit an der Kommunikation teilnehmen

Asynchrone Kommunikation

- Sender kann Daten versenden ohne sicher zu sein, dass der Empfänger bereit ist, die Daten zu empfangen

Einzeltransferoperationen (Punkt-zu-Punkt-Kommunikation)

- Einfachste Form des Datenaustausches zwischen Prozessen
- Genau zwei Prozesse beteiligt, die beide eine Kommunikationsanweisung ausführen müssen, z.B.:
  - Sendeprozess (Sender)

    int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - Empfangsprozess (Empfänger)

    int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

    buf Sendepuffer bzw. Empfangspuffer

    count Anzahl der zu sendenden bzw. Obergrenze für die Anzahl der zu empfangenden Elemente

MPI-1 Einzeltransferoperationen (Punkt-zu-Punkt-Kommunikation)

datatype Typ der zu sendenden Elemente (alle Elemente einer Nachricht müssen gleichen Typ haben) bzw. Typ der zu empfangenden Elemente

dest Nummer des Zielprozesses, der die Daten empfangen soll

source Nummer des Prozesses, von dem die Nachricht empfangen werden soll

tag Markierung der Nachricht (Unterscheidung verschiedenen Nachrichten desselben Senders)

comm Kommunikator

- Bezeichnet die Gruppe von Prozessen, die sich Nachrichten zusenden können
- Default-Kommunikator MPI_COMM_WORLD umfasst alle Prozesse eines parallelen Programms

status Datenstruktur, die Informationen über die empfangene Nachricht enthält

- status.MPI_SOURCE (Sender von dem Nachricht empfangen wurde)
- status.MPI_TAG (Markierung der empfangenen Nachricht)
- status.MPI_ERROR (Fehlercode)

Vordefinierte MPI Datentypen und korrespondierende Datentypen in C

MPI Datentyp	C-Datentyp
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_BYTE	

Beispielprogramm: Ping-Kommunikation Teil 1

```
#include <stdio.h>
#include <mpi.h>
int main(argc, argv)
int argc;
char *argv[];
{
  int size, ProcId, i, buf;
  MPI_Status status;
  i = MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcId);
```

Beispielprogramm: Ping-Kommunikation Teil 2

```
if (ProcId == 0)
    {
    buf=5;
    MPI_Send (&buf, 1, MPI_INT, 1, 100, MPI_COMM_WORLD);
    }
else
    {
    MPI_Recv(&buf, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
    printf ("Process 1 received %d from process 0\n", buf);
   }
 MPI_Finalize ();
return 0;
}
```


Interne Realisierung eines Nachrichtentransfers

- Daten werden aus dem Sendepuffer in den Systempuffer kopiert und Nachricht wird zusammengesetzt (z.B. Ergänzung eines Headers)
- Nachricht wird über das Netzwerk des Parallelrechners vom Sender zum Empfänger geschickt
- Daten werden vom Empfänger aus dem Systempuffer in den Empfangspuffer kopiert

MPI_Send( ) und MPI_Recv( ) sind blockierende, asynchrone Operationen

- MPI_Send( ) kann gestartet werden, wenn die zugehörige MPI_Recv( )-Operation noch nicht gestartet wurde
  - MPI_Send( ) blockiert jedoch so lange, bis der angegebene Sendepuffer wiederverwendet werden kann
- MPI_Recv( ) kann gestartet werden, wenn die zugehörige MPI_Send( )-Operation noch nicht gestartet wurde
  - MPI_Recv( ) blockiert jedoch so lange, bis der angegebene Empfangspuffer die erwartete Nachricht enthält

Nichtblockierende Kommunikation mit MPI_Isend( ) und MPI_Irecv( )

- Ziel ist die bessere Ausnutzung von Systemressourcen durch Vermeidung von blockierungsbedingten Wartezeiten
- Viele Parallelrechner haben für jeden Knoten eine separate Kommunikationshardware bzw. einen separaten Kommunikationsprozessor für den Kopier- und Übertragungsvorgang
  - Während dieser Zeit kann der Prozessor andere Berechnungen ausführen, die aber den Sendepuffer nicht verändern dürfen
- Prinzip
  - Phase 1: sendender oder empfangender Prozess initiiert die Kommunikation und fährt direkt mit seinen Aktionen fort, ohne den Erfolg des Datentransfers abzuwarten
  - Phase 2: zweite explizite Operation, die die Kommunikation abschließt bzw. überprüft, ob die Kommunikation vollständig abgeschlossen ist

MPI-1 Einzeltransferoperationen (Punkt-zu-Punkt-Kommunikation)

Nichtblockierende Sendeoperation (Phase 1)

- Startet Sendevorgang ohne sicherzustellen, dass nach Abschluss der Operation die Nachricht aus dem Sendepuffer kopiert wurde

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

  - Bedeutung der Parameter ist die gleiche wie bei MPI_Send( )
  - Zusätzlicher Parameter von Typ MPI_Request
    - Für den Programmierer nicht direkt zugreifbare Datenstruktur
    - Enthält Informationen über den Status der Ausführung der jeweiligen Operation

TECHNISCHE UNIVERSITÄT DRESDEN
Wolfgang E. Nagel
ZIH Center for Information Services & High Performance Computing

Nichtblockierende Empfangsoperation (Phase 1)

- Startet Empfangsoperation, bringt diese aber nicht zum Abschluss
- Information an das Laufzeitsystem, dass Daten im Empfangspuffer abgelegt werden können

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

  - Bedeutung der Parameter ist die gleiche wie bei MPI_Recv( )
  - Zusätzlicher Parameter von Typ MPI_Request

- Daten im Empfangspuffer erst benutzen, wenn Kommunikation abgeschlossen ist

MPI-1 Einzeltransferoperationen (Punkt-zu-Punkt-Kommunikation)

Abschluss der Kommunikation (Phase 2)

- Test, ob nichtblockierende Operation abgeschlossen ist

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

  - Wenn die durch request bezeichnete nichtblockierende Operation beendet ist → flag=1 ansonsten flag=0
  - Bei einer abgeschlossenen nichtblockierenden Empfangsoperation enthält die Datenstruktur status die Informationen, die bei MPI_Recv( ) beschrieben wurden
    - Bei einer nicht abgeschlossenen Empfangsoperation sind Einträge von status nicht definiert
  - Bei einer Sendeoperation sind die Einträge von status bis auf status.MPI_ERROR ebenfalls nicht definiert

- Warten auf den vollständigen Abschluss der nichtblockierenden Operation

  int MPI_Wait(MPI_Request *request, MPI_Status *status)

  - Blockiert den ausführenden Prozess, bis die von request bezeichnete Operation vollständig beendet ist
  - Bei einer Sendeoperation kann nach MPI_Wait( ) der Sendepuffer überschrieben werden
  - Bei einer Empfangsoperation können nach MPI_Wait( ) die Daten im Empfangspuffer benutzt werden

MPI-1 Einzeltransferoperationen (Punkt-zu-Punkt-Kommunikation)

Übertragungsmodi

Standardmodus

- Bisher vorgestellte Funktionen zur Punkt-zu-Punkt-Kommunikation
  - Blockierende Kommunikation mit MPI_Send( ) und MPI_Recv( )
  - Nichtblockierende Kommunikation mit MPI_Isend( ) und MPI_Irecv( )

Synchroner Modus

- Blockierendes Senden mit MPI_Ssend( )
  - MPI_Ssend( ) hat gleiche Parameter mit gleicher Bedeutung wie MPI_Send( )
- Nichtblockierendes Senden mit MPI_Issend( )
  - MPI_Issend( ) hat gleiche Parameter mit gleicher Bedeutung wie MPI_Isend( )

Puffermodus

- Blockierendes Senden mit MPI_Bsend( )
  - MPI_Bsend( ) hat gleiche Parameter mit gleicher Bedeutung wie MPI_Send( )
- Nichtblockierendes Senden mit MPI_Ibsend( )
  - MPI_Ibsend( ) hat gleiche Parameter mit gleicher Bedeutung wie MPI_Isend( )
- Puffer für das Zwischenspeichern von Nachrichten muss vom Programmierer zur Verfügung gestellt werden →Nutzung der Anweisung

  int MPI_Buffer_attach(void* buffer, int size)

  - size gibt Größe des Puffers in Byte an
- Freigabe des Puffers mit

  int MPI_Buffer_detach(void* buffer_addr, int* size)

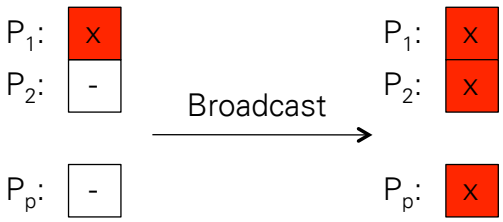
Kollektive Kommunikationsanweisungen

- An einer kollektiven Kommunikationsanweisung nehmen alle Prozesse eines Kommunikators teil
- Häufig verwendete Kollektive Kommunikationsanweisungen sind:

Komm.-Anweisung	MPI-Funktion
Broadcastoperation	MPI_Bcast( )
Akkumulationsoperation	MPI_Reduce( )
Gatheroperation	MPI_Gather( )
Scatteroperation	MPI_Scatter( )
Multi-Broadcastoperation	MPI_Allgather( )
Multi-Akkumulationsoperation	MPI_Allreduce( )
Totaler Austausch	MPI_Alltoall( )

Einzel-Broadcast

- Broadcast-Root schickt die gleiche Nachricht an alle anderen Prozesse
  - Broadcast-Root ist im Beispiel P_1



MPI-1 Kollektive Kommunikationsanweisungen

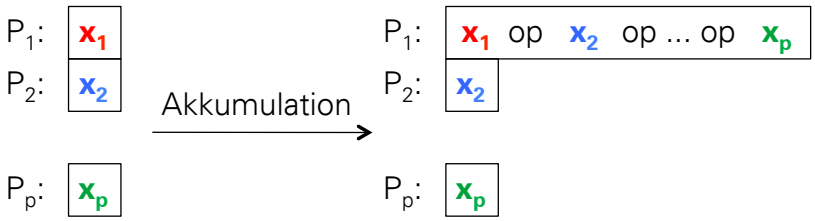
- MPI-Funktion

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

  - buffer Startadresse des Sendepuffers
  - count Anzahl der Elemente im Sendepuffer
  - datatype Datentyp der Elemente des Puffers
  - root rank des sendenden Prozesses (Broadcast-Root)
  - comm Kommunikator

Einzel-Akkumulation

- Jeder Prozess schickt eine Nachricht mit Daten gleichen Typs an einen Root-Prozess (im Beispiel P_1)
- Einzelne Nachrichten werden entsprechend der ausgewählten Reduktions-Operation (op) elementweise miteinander verknüpft



- MPI-Funktion

int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

- sendbuf — Startadresse des Sendepuffers, in dem jeder beteiligte Prozess seine lokalen Daten für die Akkumulationsoperation zur Verfügung stellt
- recvbuf — Startadresse des Empfangspuffers, wird vom Root-Prozess zur Verfügung gestellt (Ergebnis der Akkumulationsoperation)
- count — Anzahl der von jedem Prozess zur Verfügung gestellten Elemente
- datatype — Datentyp der Elemente
- op — Reduktions-Operation
- root — rank des Root-Prozesses
- comm — Kommunikator

MPI-1 Kollektive Kommunikationsanweisungen

- MPI stellt eine Reihe von vordefinierten Reduktions-Operationen zur Verfügung

Darstellung	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Summe
MPI_PROD	Produkt
MPI_LAND	logisches Und
MPI_BAND	bitweises Und
MPI_LOR	logisches Oder
MPI_BOR	bitweises Oder
MPI_LXOR	logisches exclusives Oder
MPI_BXOR	bitweises exclusives Oder
MPI_MAXLOC	maximaler Wert und dessen Index
MPI_MINLOC	minimaler Wert und dessen Index

MPI-1 Kollektive Kommunikationsanweisungen

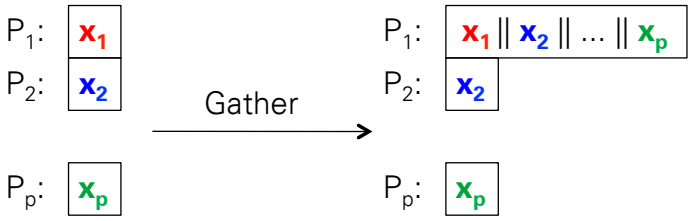
- Es können auch benutzerdefinierte Reduktionsoperationen erstellt werden

int MPI_Op_create(MPI_User_function *function, int commute,
MPI_Op *op)

- function ist die vom Programmierer zur Verfügung gestellte Operation
- commute gibt an, ob es sich um eine kommutative Operation (communte=1) handelt oder nicht (communte=0)
- op berechnet eine Operations-Datenstruktur, die später als Parameter an die Operation MPI_Reduce( ) übergeben werden kann

Gather

- Jeder Prozess schickt an den Root-Prozess (im Beispiel P_1) eine Nachricht
- Root-Prozess sammelt Nachrichten auf
  - Ohne Anwendung einer Reduktions-Operation



- MPI-Funktion MPI_Gather( )

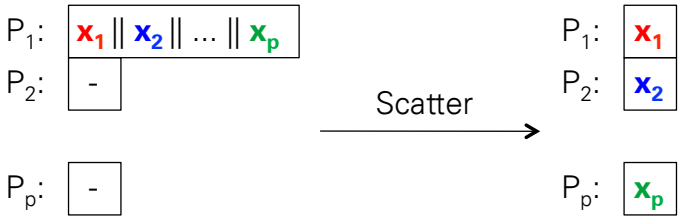
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)

- sendbuf — Sendepuffer in den jeder Prozess sendcount Elemente vom Typ sendtype zur Verfügung stellt
- recvbuf — vom Root-Prozess zur Verfügung gestellter Empfangspuffer, in dem von jedem Prozess recvcount Elemente vom Typ recvtype empfangen werden
- root — Root-Prozess der Gather-Operation

- MPI-Funktion MPI_Gatherv( )
  - Stellt Verallgemeinerung von MPI_Gather( ) dar, die es gestattet, dass jeder Prozess eine evtl. unterschiedliche Anzahl von Elementen zur Verfügung stellt
    - recvcout wird durch Integer-Feld recvcouts ersetzt
    - Zusätzliches Integer-Feld displs, kennzeichnet Einträge im Empfangspuffer

Scatter

- Ein Root-Prozess (im Beispiel P_1) schickt an jeden anderen Prozess eine eventuell unterschiedliche Nachricht



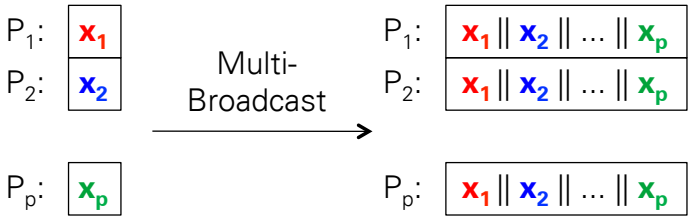
- MPI-Funktion MPI_Scatter( )

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm

  - Es gibt auch MPI_Scatterv( ) als Verallgemeinerung von MPI_Scatter( )

Multi-Broadcast

- Es gibt keine Broadcast-Root, stattdessen führt jeder Prozess eine Einzel-Broadcastoperation aus

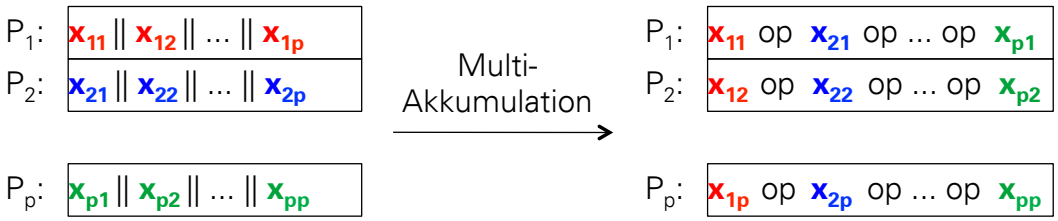


- MPI-Funktion MPI_Allgather( )

int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

Multi-Akkumulation

- Jeder Prozess führt eine Einzel-Akkumulation aus, d.h. jeder Prozess stellt für jeden anderen Prozess eine eventuell unterschiedliche Nachricht zur Verfügung
- Die Nachrichten für den gleichen Empfangsknoten werden mit einer vorgegebenen Reduktionsoperation kombiniert, so dass an jedem Empfangsknoten eine Nachricht eintrifft.

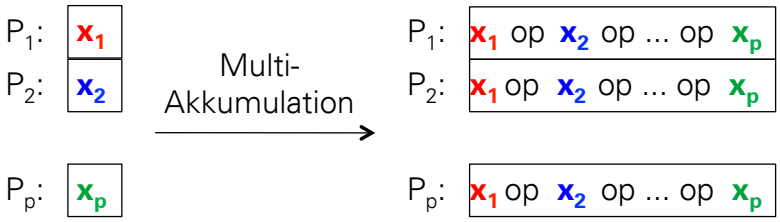


- MPI-Funktion MPI_Allreduce( )

int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

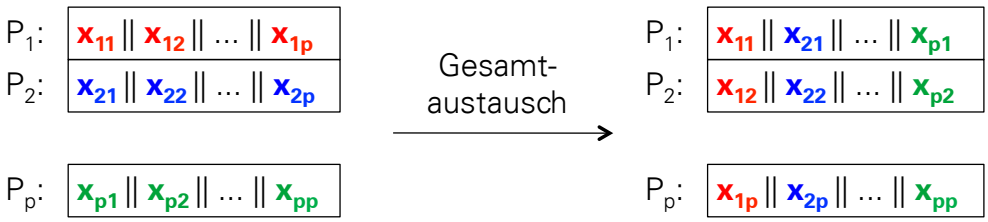
MPI-1 Kollektive Kommunikationsanweisungen

- Eingeschränkte Funktionalität in der MPI-Funktion MPI_Allreduce( )
  - Jeder Prozess stellt nur ein Datum zur Verfügung und besitzt nach Abschluss der Operation das gleiche Ergebnis



Gesamtaustausch

- Jeder Prozess sendet an jeden anderen beteiligten Prozess eine evtl. unterschiedliche Nachricht (Scatteroperation aus Sendersicht) und empfängt von jedem anderen beteiligten Prozess eine Nachricht (Gatheroperation aus Empfängersicht)
- Keine Reduktionsoperation



- MPI-Funktion MPI_Alltoall

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

MPI Ausblick

MPI-1

- Über 120 Funktionen, häufig angewandt werden auch
  - Funktionen zum Aufbau virtueller Topologien
  - Funktionen zur Bildung von Subarrays

MPI-2

- Enthält neben MPI-1 über 170 zusätzliche Funktionen
- Wesentliche Erweiterungen von MPI-2 sind:
  - Dynamische Prozessverwaltung
  - Einseitige Kommunikationsoperationen
  - Operationen zur parallelen Ein-/Ausgabe

MPI-3

- Wesentliche Erweiterungen von MPI-3 sind:
  - Nichtblockierende kollektive Kommunikationsanweisungen
  - Kollektive Kommunikationsanweisungen, die die Nachbarschaftsbeziehungen beachten
  - Dynamisches Kreieren von Fenstern für einseitige Kommunikationsoperationen
  - Nutzung von Shared Memory Fenstern
- Es sind noch keine Implementierungen von MPI-3 verfügbar

Gliederung

- Shared Memory Programmierung
  - OpenMP
- Partial Global Adress Space (PGAS) Programmierung
  - UPC

Shared Memory Programmierung

- Parallele Programmierung erfolgt auf der Grundlage von gemeinsamen Variablen
- Erfordert ein Speichermodell, in dem jede Ausführungseinheit (Thread) eines parallelen Programms während ihrer Abarbeitung auf einen gemeinsamen Adressraum zugreifen kann und die dort abgelegten Variablen lesen oder manipulieren kann
- Informationsaustausch zwischen den Threads erfordern keinen Nachrichten-austausch, stattdessen muss der Zugriff auf die gemeinsamen Variablen koordiniert werden

OpenMP

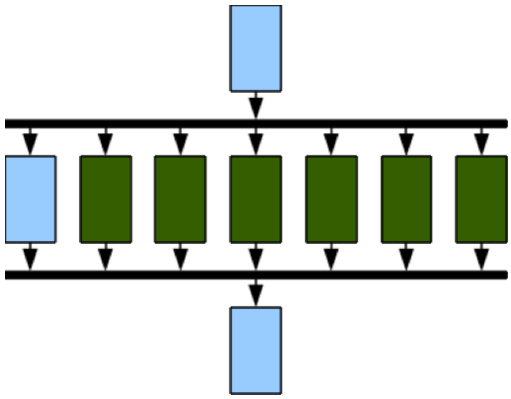
- OpenMP steht für Open-Multi-Processing
- Einheitlicher Standard für die Programmierung von Parallelrechnern mit gemeinsamen Adressraum
- Wurde insbesondere für den Bereich des wissenschaftlichen Rechnens spezifiziert
- Spezifikation von Übersetzerdirektiven, Bibliotheksfunktionen und Umgebungsvariablen
- Unterstützung der Schnittstellen für C, C++, FORTRAN
- SPMD-Stil

Entwicklung von OpenMP

- Oktober 1997 Version 1.0 für Fortran
- Oktober 1998 Version 1.0 für C/C++
- November 2000 Version 2.0 für Fortran
- März 2002 Version 2.0 für C/C++
- Mai 2005 Version 2.5 für C/C++ und Fortran
- Mai 2008 Version 3.0 für C/C++ und Fortran

Prinzipielle Arbeitsweise von OpenMP

- Programm startet mit Master Thread
- Master erzeugt bei Erreichen eines parallelen Abschnitts ein Team von Threads
- Die zu berechnende Aufgabe wird über das Team verteilt
- Nach Erreichen des Endes des parallelen Abschnitts arbeitet nur der Master Thread weiter



Wichtige Übersetzerdirektiven

- Wir beschränken uns auf die Schnittstellen für C
- Übersetzerdirektiven basieren auf den in C und C++ verwendeten #pragma-Direktiven
- Allgemeine Form einer Übersetzerdirektive

#pragma omp Anweisung [Parameter [Parameter]...]

  - Direktive muss in einer eigenen Programmzeile stehen
  - Direktiven werden oft in Verbindung mit Laufzeitfunktionen verwendet, die die Direktiven steuern
  - Jede Direktive wirkt nur auf die der Direktive direkt folgende Anweisung
    - Sollen mehrere Anweisungen durch die Direktive gesteuert werden, müssen diese zwischen { und } stehen und werden so zu einem Anweisungsblock zusammengefasst

Paralleler Bereich

- Wichtigste Direktive zur Steuerung der Parallelität

```
#pragma omp parallel
        {
        Anweisungsblock
        }
```

  - Zum Zeitpunkt “{“ erzeugt der Master-Thread ein Team von Threads
  - Der Zeitpunkt “}“ markiert das Ende des parallelen Abschnitts, danach arbeitet nur noch der Master-Thread
  - Natürliche Barriere an der Stelle, wo parallele Region in den Master-Thread übergeht

OpenMP – wichtige Übersetzerdirektiven

- Anzahl der Threads (N) für den parallelen Bereich wird üblicherweise mit der Laufzeitfunktion omp_set_num_threads(N) festgelegt
- Threadnummer kann mit omp_get_thread_num() abgefragt werden

OpenMP – wichtige Übersetzerdirektiven

Parameter der Direktive "Paralleler Bereich"

- Gemeinsame und private Variable der beteiligten Threads können definiert werden über die Parameter

  shared(list_of_variables) bzw.

  private(list_of_variables)

  - list_of_variables ist eine beliebige Liste von bereits deklarierten Programmvariablen
  - Damit kann die Direktive folgendermaßen erweitert werden

    #pragma omp parallel private (list_of_variables) shared(list_of_variables)

OpenMP – wichtige Übersetzerdirektiven

Beispielprogramm: Ausgabe der Threadnummer

```
#include <omp.h>
int main()
{
  int i;
  i = -1;
omp_set_num_threads(8);
#pragma omp parallel private(i)
 {
i = omp_get_thread_num();
printf( "my identity is: %d\no.k.?\n", i );
 }
}
```


OpenMP – wichtige Übersetzerdirektiven

Parallele Schleife

- Direktive zur Verteilung der Arbeit

```
#pragma  omp  for
for (i=lower_bound; i op upper_bound; incr_expr)
        {
        Schleifenrumpf
        }
```

  - op ∈ {<, <=, >, >=}

OpenMP – wichtige Übersetzerdirektiven

- Über den Parameter schedule können eine Reihe von Lastverteilungs-strategien ausgewählt werden
  - schedule(static, block_size)
  - schedule(dynamic, block_size)
  - schedule(guided, block_size)
  - schedule(runtime)

OpenMP – wichtige Übersetzerdirektiven

Nichtiterative parallele Bereiche

- Wenn eine nichtiterative Verteilung der durchzuführenden Berechnungen innerhalb eines parallelen Bereiches vorgenommen werden soll, wird die sections-Direktive verwendet
- Innerhalb einer sections-Direktive werden durch section-Direktiven Abschnitte gekennzeichnet, die unabhängig voneinander sind und damit parallel von verschiedenen Threads abgearbeitet werden können
- Jeder Abschnitt beginnt mit #pragma omp section und kann ein beliebiger Anweisungsblock sein
  - Für den ersten Anweisungsblock innerhalb der sections-Direktive kann #pragma omp section entfallen
- Implizite Synchronisation am Ende der sections-Direktive

OpenMP – wichtige Übersetzerdirektiven

- Syntax der sections-Direktive

```
#pragma  omp  sections
  {{ Anweisungsblock }
#pragma  omp  section
   { Anweisungsblock }
#pragma  omp  section
   { Anweisungsblock }
#pragma  omp  section
   { Anweisungsblock }
        ……..
  }/*omp end sections*/
```


OpenMP – wichtige Übersetzerdirektiven

Syntaktische Abkürzungen

- Wenn parallele Bereiche nur eine einzelne for- bzw. sections-Directive enthalten, kann man eine vereinfachte Schreibweise benutzen

  – #pragma omp parallel for
  for (i=lower_bound; i op upper_bound; incr_expr)
  {
  Schleifenrumpf
  }

  – #pragma omp parallel sections
  {{ Anweisungsblock }
  ……

Kritische Bereiche

- Zur Vermeidung von unkontrollierten parallelen Schreibzugriffen kann man mit der critical-Direktive Kritische Bereiche definieren
  - Kritische Bereiche dürfen zu jedem Zeitpunkt nur von jeweils einem Thread ausgeführt werden
- Syntax der critical-Direktive

```
#pragma omp critical
{
    Anweisungsblock
}
```


Gliederung

- Partial Global Adress Space (PGAS) Programmierung
  - UPC

Partial Global Adress Space (PGAS) Programmierung

Motivation

- Parallelisierung von HPC-Programmen erfolgt heute überwiegend mit MPI und OpenMP, die jedoch ihre Grenzen bezüglich der Prozessorzahl haben
  - OpenMP: Aufwand der Cache Kohärenzprotokolle als Basis für den globalen Adressraum ist für sehr hohe Prozessorzahlen nicht mehr vertretbar
  - MPI: Die für die Kommunikation bereitzustellenden Puffergrößen stellen für zukünftige Prozessorzahlen eine Begrenzung dar

  ➔ Es wird intensiv nach neuen Programmiermodellen gesucht, um zukünftige massiv parallele Hochleistungsrechner zu programmieren
    - Ein mittlerweile weit fortgeschrittener Ansatz ist das Programmiermodell Partial Global Adress Space (PGAS)
    - Die beiden bekanntesten Spracherweiterungen sind
      - Coarray-Fortran (mittlerweile Teil des FORTRAN 2008 Standards)
      - Unified Parallel C (UPC)

      →Wir konzentrieren uns auf UPC

Partial Global Adress Space (PGAS) Programmierung

Prinzip der PGAS-Programmierung

- Es werden nur bestimmte Teile des Adressraums als global definiert
- Innerhalb dieses partitionierten globalen Adressraums hat jeder Prozess direkten Zugriff auf geteilte Daten
- Das Programmiermodell ist unabhängig vom Speichermodell des Systems
  - Systeme mit gemeinsamen (globalen) Adressraum lösen Zugriffe auf die gemeinsamen Daten durch direkte Speicherzugriffe
  - Systeme mit verteilten (lokalen) Adressraum lösen Zugriffe durch Nachrichtenaustausch
- Jede Variable ist mit einem bestimmten Thread assoziiert, was bei einem Rechner mit verteiltem Speicher eine tatsächliche physikalische Verteiltheit bedeutet
  - Zugriff auf eine Variable benötigt dann unterschiedliche Zeiten im Sinne von NUMA
- SPMD-Stil

Prinzipielle Arbeitsweise von UPC

- Ist eine Erweiterung der Programmiersprache C zu einer parallelen Programmiersprache im SPMD-Stiel
  - UPC-Funktionalität wird im Allgemeinen durch Einbindung der UPC-Header <upc.h> in ein C-Programm realisiert
- UPC –Programm wird repliziert in mehreren Prozessen gestartet
  - Typisch ist eine Instanz (Thread) des Programms auf jedem Prozessor
- Einzelne Threads werden asynchron ausgeführt
- Programmfluss wird durch Synchronisationsvorgänge und Verzweigungen gesteuert
  - Basis ist eindeutiger Index für jeden Thread
- Datenaustausch erfolgt über global für alle Threads sichtbare „geteilte“ Variablen und Felder (Schlüsselwort `shared` bei der Deklaration
  - Z.B. `shared float a;` bzw. `shared float b[10];`

Threadindexierung

- THREADS
  - Ist eine vordefinierte Programmvariable, in die die Anzahl der Threads abgespeichert wird, mit dem das Programm gestartet wurde
- MYTHREAD
  - Ist eine vordefinierte Programmvariable, in die der Index des eigenen Threads abgespeichert wird
  - Es werden die Werte 0,.....,THREADS-1 vergeben

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

- Globaler Adressraum von UPC hat folgendes Aussehen

Thread 0	Thread 1		Thread THREADS-1
gemeinsame Daten	gemeinsame Daten	. . .	gemeinsame Daten
private Daten	private Daten		private Daten

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

Deklaration und Zugriff

- Deklaration global sichtbarer und geteilter Objekte erfolgt, in dem das Schlüsselwort `shared` der Deklaration vorangestellt wird
  - Skalare Variablen werden nur auf dem ersten Thread abgelegt(Threadindex MYTHREAD==0),

    jedoch kann jeder Thread beliebig auf diese Variable lesend oder schreibend zugreifen

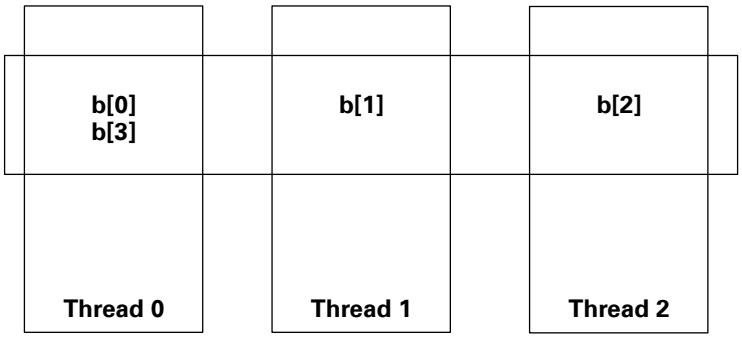
    z.B. `shared float a;`

- Deklaration geteilter Objekte innerhalb von Funktionen ist nicht möglich
  - `shared`-Variablen müssen entweder globale Variablen sei (Deklaration außerhalb der main-Funktion) oder man muss die dynamische Allokation geteilter Speicherbereiche verwenden

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

- Felder als geteilte Objekte
  - Wird ein Feld als geteiltes Objekt deklariert, werden die Werte einzeln nach dem Round-Robin-Verfahren auf den Speicher aller Threads verteilt

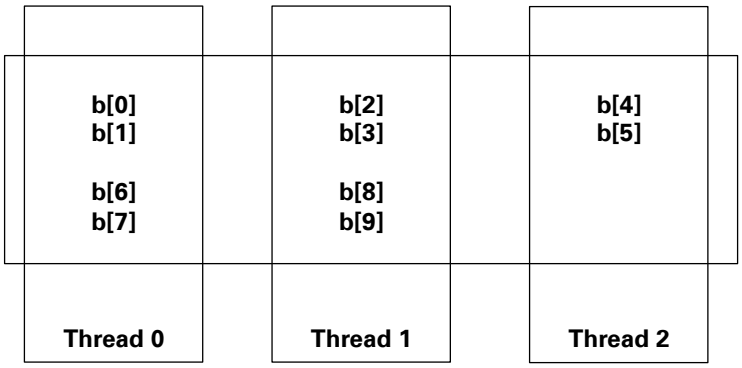
    z.B.: `shared float b[4];`



Unified Parallel C (UPC) - wichtige Sprachkonstrukte

  - Mit dem Präfix `shared [i]` wird festgelegt, dass immer i aufeinander folgende Werte des Feldes dem gleichen Thread zugeordnet werden

    z.B.: `shared [2] float b[10];`



- i=0 ist als Blockgröße „unendlich“ zu deuten, damit wird das gesamte Feld im Speicher des ersten Feldes abgelegt

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

Zeiger

- Herkömmliche Zeiger in C wie `int *ptr1;` können nur auf lokale Speicheradressen zeigen
- Soll ein Zeiger auf geteilte Speicheradressen zeigen, muss bei der Deklaration der jeweilige `shared` -Datentyp angegeben werden

  z.B.: `shared int *ptr2;`

  – Zeiger ptr2 ist jedoch nur innerhalb des Threads selbst sichtbar
- Ein Zeiger auf eine geteilte Speicheradresse, der über alle Threads hinweg sichtbar und modifizierbar ist, wird deklariert mit

```
shared int *shared ptr3;
```


Unified Parallel C (UPC) - wichtige Sprachkonstrukte

Dynamische Speicherallokation

- UPC stellt drei Funktionen für die dynamische Allokation von geteilten Speicher zur Verfügung
  - **upc_alloc**

    Nichtkollektive Funktion, die dazu dient, auf einem einzelnen Thread Speicher zu allokieren, der nachher für alle Threads sichtbar ist

    `shared void *upc_alloc(size_t nbytes);`

    Funktion liefert einen Zeiger auf einen Speicherblock der Länge `nbytes` zurück
  - **upc_global_alloc**

    Nichtkollektive Funktion, die dazu dient, über alle Threads verteilt Speicher zu allokieren, welcher nachher global für alle Threads sichtbar

    `shared void *upc_global_alloc(size_t nblocks, size_t nbytes);`

    Die Funktion liefert dem aufrufenden Thread einen Zeiger auf `nblocks * nbytes` zurück

- **upc_all_alloc**

  Kollektive Funktion durch die alle Threads gemeinsam `nblocks * nbytes` Speicher über die Threads verteilt allokieren

  ```
  shared void *upc_all_alloc(size_t nblocks, size_t nbytes);
  ```

  Funktion liefert jedem Thread einen Zeiger auf den Beginn des allokierten geteilten Speichers zurück

- **upc_free**

  Nichtkollektive Funktion mit der dynamisch allokierte Speicherbereiche wieder freigegeben werden

  ```
  upc_free(shared void *ptr);
  ```

  - Muss nur von einem Thread freigegeben werden

Synchronisation

- **upc_notify / upc_wait**
  - Kollektive Synchronisationsanweisungen, die immer paarweise auftreten
  - Aufruf `upc_notify` ist nicht-blockierend
  - Beim Erreichen der Anweisung `upc_wait` wird der Programmfluss des Threads solange unterbrochen, bis alle anderen Threads ebenfalls die `upc_notify` – Anweisung erreicht haben
  - Anweisungen können mit ganzzahligen Bezeichnern versehen werden z.B.:

```
...
upc_notify 3;
...
upc_wait 3;
...
upc_notify 4;
...
upc_wait 4;
```

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

- **upc_barrier**
  - An einer kollektiven Barriere `upc_barrier` pausiert jeder Thread bis alle Threads die Barriere erreicht haben
  - `upc_barrier`-Anweisung verhält sich äquivalent zu einem unmittelbar aufeinanderfolgenden Aufruf von `upc_notify; upc_wait`
- **upc_fence**
  - `upc_fence`-Anweisung kann von einem einzelnen Thread benutzt werden, um zu pausieren bis alle ausstehenden Zugriffe auf geteilte Daten durchgeführt sind
- **Locks**
  - Über Locks kann der Zugriff zu geteilten Ressourcen abgesichert werden
  - Locks werden in UPC durch den Datentyp `upc_lock_t` in Verbindung mit Funktionen zum Allokieren, Belegen und Freigeben realisiert

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

- `upc_all_lock_alloc(void)`
  - Kollektive Funktion, die ein neues Lock erzeugt
  - Liefert Zeiger auf das Lock zurück
  - Im Ausgangszustand ist das Lock unbelegt
- `upc_lock_free(upc_lock_t *ptr)`
  - Funktion löst das Lock auf und gibt den von ihm belegten Speicherplatz wieder frei
  - Aufruf muss nur von einem Thread aus erfolgen
- `upc_lock(upc_lock_t *ptr)`
  - Funktion versucht das Lock ptr zu belegen
  - Befindet sich das Lock vorher im unbelegten Zustand, wird das Lock belegt
  - Ist das Lock belegt, wartet die Funktion so lange bis das Lock wieder frei ist
  - Im UPC-Standard ist nicht festgelegt, in welcher Reihenfolge mehrere um ein Lock konkurrierende Threads den Zugriff erhalten

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

- `upc_lock_attempt(upc_lock_t *ptr)`
  - Funktion versucht ebenfalls das Lock ptr zu belegen, jedoch ohne zu blockieren, falls das Lock schon belegt ist
  - Rückgabewert 0: falls das Lock schon belegt ist
  - Rückgabewert 1: wenn das Lock durch den Funktionsaufruf belegt werden konnte
- `upc_unlock(upc_lock_t *ptr)`
  - Durch diese Funktion wird das Lock freigegeben

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

Hilfsfunktionen und kollektive Operationen

- **Alternative sizeof-Operationen**
  - `upc_localsizeof`
    - Für ein verteiltes Objekt liefert der Operator `upc_localsizeof()` die Größe des Speicherplatzes in Byte, die auf einem einzelnen Thread belegt wird
    - Rückgabewert entspricht der oberen Grenze des belegten Speicherplatzes über alle Threads
  - `upc_blocksizeof`
    - Operator `upc_blocksizeof()` gibt für ein geteiltes Objekt als Operand die Blockgröße bzw. den Blocking-Faktor zurück
  - `upc_elemsizeof`
    - Operator `upc_elemsizeof()` liefert für ein verteiltes Feld die Größe des Speichers, die von einem seiner Elemente belegt wird

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

- **Terminierung**
  - `void upc_global_exit(int status)`
    - Mit dieser Funktion werden alle Threads des UPC-Programms terminiert

- **upc_forall-Schleife**
  - `upc_forall`
    - Dient als Alternative zur manuellen Parallelisierung von Zählschleifen

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

Beispiel: Manuelle Aufteilung der Iterationen einer `for`-Schleife

```
#include <upc.h>
for (int i=0; i<100; i++) {
  if ( i % THREADS == MYTHREAD ) {
     //..
 }
}
```

- Durch die if-Anweisung führt jeder Thread nur die Iterationen der for-Schleife aus, für die der Threadindex gleich dem Wert der Zählvariablen modulo der Anzahl der Threads ist
- Iterationen werden im Round-Robin-Verfahren auf die Threads verteilt

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

Beispiel: Aufteilung der Iterationen mit `upc_forall`

```
#include <upc.h>
upc_forall (int i=0; i<100; i++; i) {
    //..
}
```

- `upc_forall`-Schleife hat einen vierten Ausdruck in der runden Klammer, der die Affinität der Iterationen zu den Threads angibt
  - Vierter Ausdruck wird im folgenden als *Affinitätsausdruck* bezeichnet
- Ein Thread führt eine Iteration genau dann aus, falls sein Threadindex gleich dem *Affinitätsausdruck* modulo der Anzahl aller Threads ist
- Im obigen Beispiel ist der *Affinitätsausdruck* i, damit werden die Iterationen nach dem Round-Robin-Verfahren auf die Threads verteilt

Unified Parallel C (UPC) - wichtige Sprachkonstrukte

Beispiel: Aufteilung der Iterationen mit `upc_forall`

```
#include <upc.h>
upc_forall (int i=0; i<100; i++; i/(THREADS-1)) {
    //..
}
```

- Die Auswahl von `i/(THREADS-1)` als *Affinitätsausdruck* führt zu einer blockweisen Aufteilung der Iterationen