Praktikum für verteilte und parallele Systeme
WS 2008/09

MPI on Cell

Lukas Laschitz, Stefan Hofbauer und
Matthias Janetschek

MPI 1

- Message Passing Interface
- MPI-Programmiermodell: meist SPMD (Single Program, Multiple Data)
- Alle Variablen sind privat, gehören also ihrem jeweiligen Prozess
- Kommunikation erfolgt über Routinen einer speziellen Kommunikations-Bibliothek: MPI-Library

MPI 1

- MPI fuer C #<include mpi.h>
- Am Anfang des Programms die MPI_INIT(&argc, &argv); Anweisung
- Am Ende des Programms MPI_Finalize();
- Bestimmung der Anzahl von Prozessen MPI_Comm_size(MPI_COMM_WORLD,&np);
- Bestimmung des eigenen Rangs MPI_Comm_rank(MPI_COMM_WORLD,&myr)

Punkt zu Punkt Kommunikation

Zwei Modi : blockierend, nicht blockierend

vier verschiedene Arten:

- synchron
- buffered
- auto
- ready

Blockierend

Senden:

MPI_Bsend MPI(gepuffert)

MPI_Rsend (ready-Modus)

MPI_Send (Standard)

MPI_Ssend (synchron)

```
MPI_Comm_rank(MPI_COMM_WORLD, &r);

if(r==0)
{
   MPI_Send("Hello, World\n", 14, MPI_CHAR,1,0,MPI_COMM_WORLD);
}
else
{
   char buf[256];
   MPI_Status status;
   MPI_Recv(buf, sizeof(buf), MPI_CHAR,0,0,MPI_COMM_WORLD,&status);
   printf("Received: %s\n",buf);
}
```

Nicht Blockierend

Senden:

MPI_Ibsend (gepuffert)

MPI_Irsend (ready-Modus)

MPI_Isend (Standard)

MPI_Issend (synchron)

Zusatzabfragen:

MPI_Test

MPI_Wait

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

Zusammengesetzte Funktionen

MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,recvbuf, recvcount,recvtype, source, recvtag, comm, status)

MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

Kollektive Kommunikation

- Synchronisation aller Prozesse durch eine Barriere
- Broadcast einer Nachricht von einem Prozess zu allen anderen
- Scatter/Gather

Barrier und Broadcast

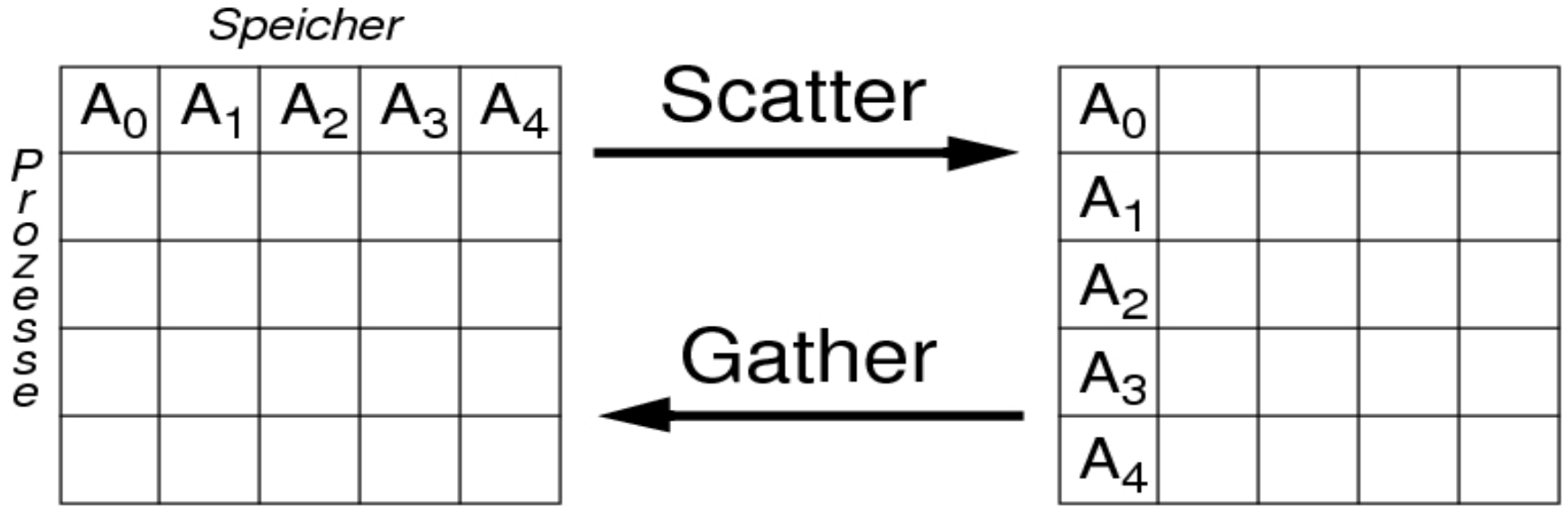
```
MPI_Barrier(MPI_COMM_WORLD);
```

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

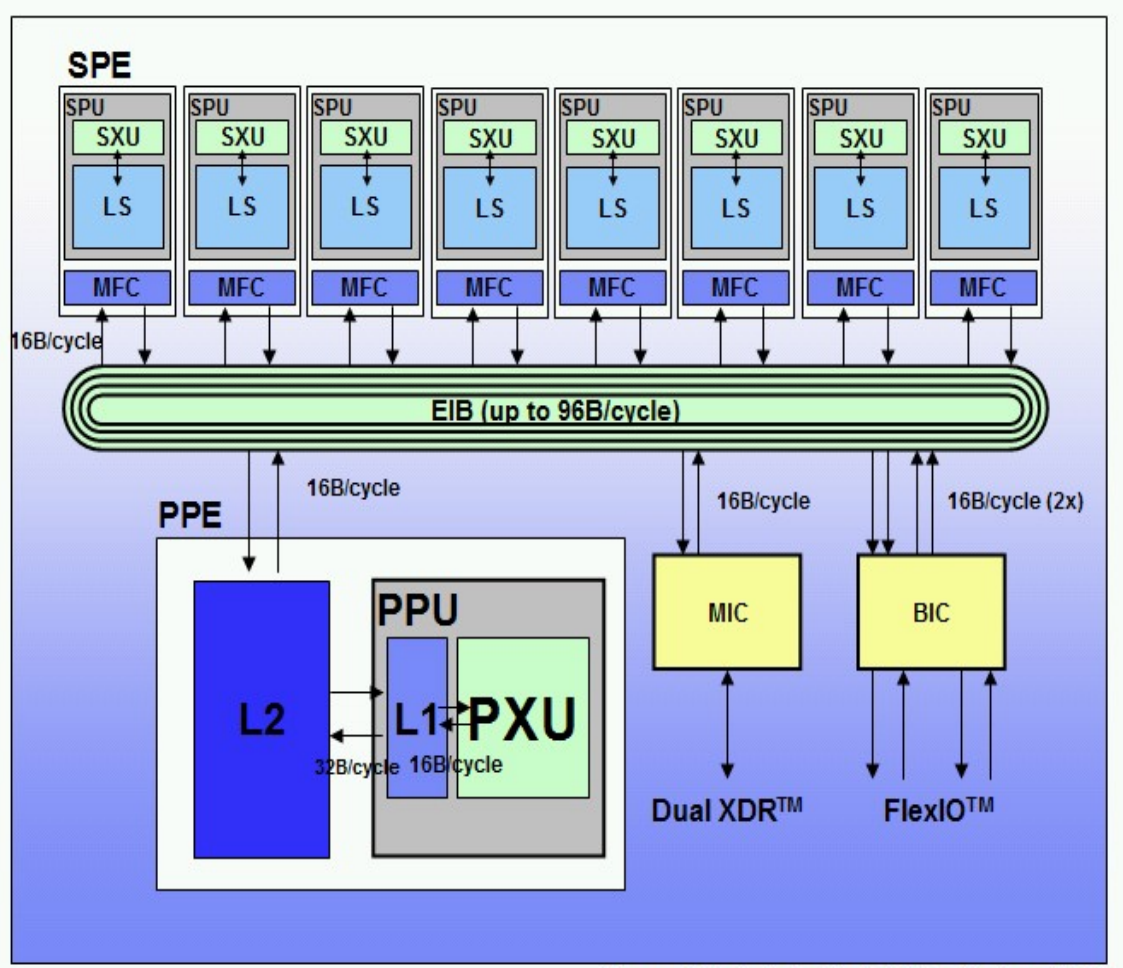
Gather/Scatter

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)



Cell Broadband Engine



Source: M. Gschwind et al., Hot Chips-17, August 2005

Besonderheiten mit Relevanz für MPI

- Allgemeine Einschränkungen
- Speicher Alignments
- Kommunikationswege
  - DMA Calls
  - Mailboxes
  - Signals

Allgemeine Einschränkungen

- Sehr geringer Speicher
- Nur ein Thread (nonblocking schwierig)
- Nicht alle bekannten C-Funktionen nutzbar
- Executables nicht kompatibel (2 Libraries nötig)

Speicher Alignments

- Wichtig für DMA Transfers
- Nur aligned Daten können übertragen werden
- Kann dem Programmierer von unserer Implementierung nicht abgenommen werden

DMA Calls

- Größen von 1, 2, 4, 8 (mit Einschränkungen), 16 oder vielfachen von 16 Bytes möglich
- Maximal 16 Kbytes
- Ideal 128 Bytes (Cache line basiert)
- Sollten vom SPE initiiert werden (mehr SPEs, weniger Cycles, größere Queue)

Mailboxes

- 1 für max. 4 eingehende Nachrichten
- 1 für max. 1 ausgehende Nachrichten
- 1 für max. 1 ausgehende Nachrichten mit Interrupt
- 1 Nachricht hat 32bit, wird beim Lesen konsumiert
- Schreiben in vollen Channel blockiert

Signals

- 2 channels mit max. 1 Nachricht
- Überschreiben oder Verodern von Nachrichten
- 1 Nachricht hat 32bit, wird beim Lesen auf 0 gesetzt

Beispiel Threadcreation: mpi_init

```
void mpi_init(int no_spe) {
        cbu = (cb_union*)_malloc_align(no_spe * sizeof(cb_union),7);
        for (int i = 0; i < no_spe; i++) {
                datas[i].speid = spe_context_create (0, NULL);
                spe_program_load (datas[i].speid, &spe_handle);
                cbu[i].io.speid=i;
                cbu[i].io.no_spe=no_spe;
                ...
                datas[i].argp = (unsigned long long *) &cbu[i];
                pthread_create (&datas[i].pthread, NULL, &ppu_pthread_function, &datas[i])
        }
}
```

```
typedef struct ppu_pthread_data {
	spe_context_ptr_t speid;
	pthread_t pthread;
	void *argp;
} ppu_pthread_data_t;
ppu_pthread_data_t datas[8];

extern spe_program_handle_t spe_handle;

void *ppu_pthread_function(void *arg) {
	ppu_pthread_data_t *datap = (ppu_pthread_data_t *)arg;
	unsigned int entry = SPE_DEFAULT_ENTRY;
	spe_context_run(datap->speid, &entry, 0, datap->argp, NULL, NULL);
	pthread_exit(NULL);
}
```

SPE-SPE Mailboxing: PPE Code

```
if ((cb[i].control_area = spe_ps_area_get(datas[i].speid, SPE_CONTROL_AREA)) == NULL){
    printf("ERROR: spe_get_ps_area failed for control area(%d)\n", errno);
    return -1;

}
```

SPE-SPE Mailboxing: SPE Code

```
int send_msg_to_mailbox(int rank, int value, int tag_id) {
        // array of 4 is equvalent to quadword
        volatile int msg[4] __attribute__ ((aligned (128)));
        unsigned int ea;

        msg[3] = value;
        ea =  (unsigned int)cb[rank].control_area+12;

        mfc_put(&msg[3], ea, 4, tag_id, 0, 0);
        mfc_write_tag_mask(1<<tag_id);
        mfc_read_tag_status_all();
        //(void)spu_mfcstat(2);

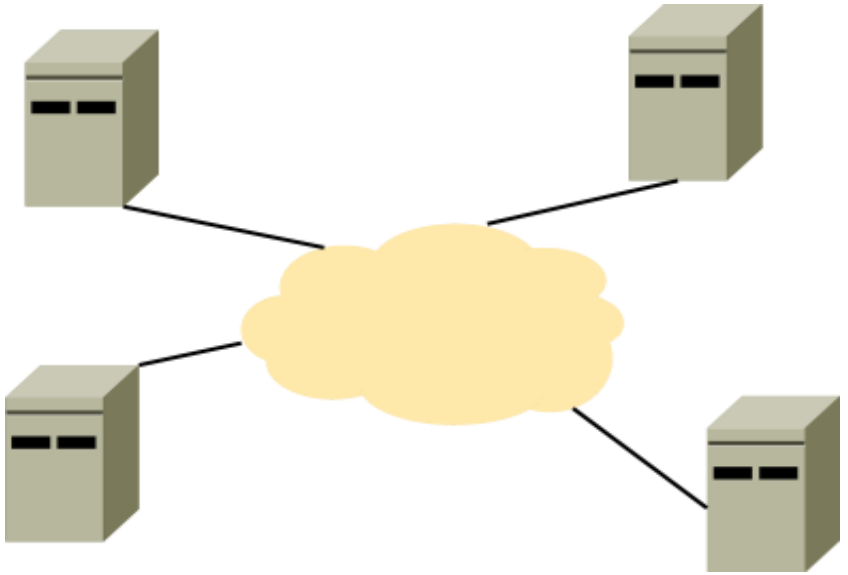
        return tag_id;
}
```

Probleme mit MPI

- Ideale Transfermethode (insbesondere für kleine Nachrichten) herausfinden
- Nachrichten falscher Größe an Voraussetzungen anpassen
- Verwendung von Mailboxes oder Signals führt zu Unverwendbarkeit für den User (kann aber schwer erzwungen werden)
- Speichernutzung gering halten

Was ist unser Ziel

- Kapselung der DMA-Befehle
- Bekannte API
- SPEs und PPU sind Knoten eines Distributed-Memory-Computers



Probleme

- MPI-Standard nimmt „vollwärtige“ Knoten an
- Normale Implementation hat keine Beschränkung hinsichtlich Verwaltungsoverhead
- Cell-Architektur „erzwingt“ viele DMA-Transfers

Vorgehensweise

- LS-Speicher sparsam verwenden (Daten-Strukturen und Programmcode)
- Wenn möglich in den Hauptspeicher auslagern
- DMA-Transfers nur wenn notwendig
- Möglichst keine Synchronization
- MPI-Standard-Treue vs. Performance/wenig Overhead

Vorgehensweise

2 Möglichkeiten:

1. Komplexen Ansatz optimieren und vereinfachen

2. Einfachen Ansatz bestmöglichst ausreizen und Standard-Verletzungen in Kauf nehmen

Was wollen wir implementieren (Kern-Implementation)

- Point to Point – Kommunikation
  - Blocking Buffered und Synchronous Kommunikation (Send, Bsend, Ssend, Recv, Probe, Cancel)
  - Non-Blocking Kommunikation (ISend, IBsend, ISsend, IRecv, Test, Wait)
- Collective Communication
  - Barrier
  - Broadcast, Scatter & Gather, All*

Was wollen wir implementieren (Erweiterungen)

- Persistent Communication
- Global-Reduction-Functions
- Scan
- Groups & Communicators

Unsere Implementierung

- (n-1) Incoming Message-Queues
- Status-Array für Outgoing-Messages (auch für Buffermanagement)
- Sender schreibt per DMA Message-Envelope in die Message-Queue des Empfängers
- Empfänger holt sich Daten per DMA und schreibt Rückmeldung in das Status-Array des Senders

Nachteile

- Nicht viele ausstehende Nachrichten möglich
- Globale Reihenfolge der Nachrichten wird nicht gewährleistet
- Non-Blocking Kommunikation nicht 100% non-blocking

Projekt-Plan

- Design-Phase & Einarbeiten in Cell-Programmierung
- Point to Point – Communikation
- Performance Tests & Optimierung der Point to Point – Communikation
- Collective Communication
- Performance Tests & Optimierung der Collective Communikation
- ...

Danke für die Aufmerksamkeit