
*Ein effizientes
Message-Passing-Interface
(MPI) für HiPPI*

cand. mach Thomas Beisel

Betreuer: Alfred Geiger

RUS #: IB 113

Datum: 21.2.1996

Rechenzentrum der Universität Stuttgart

Programmbeschreibung

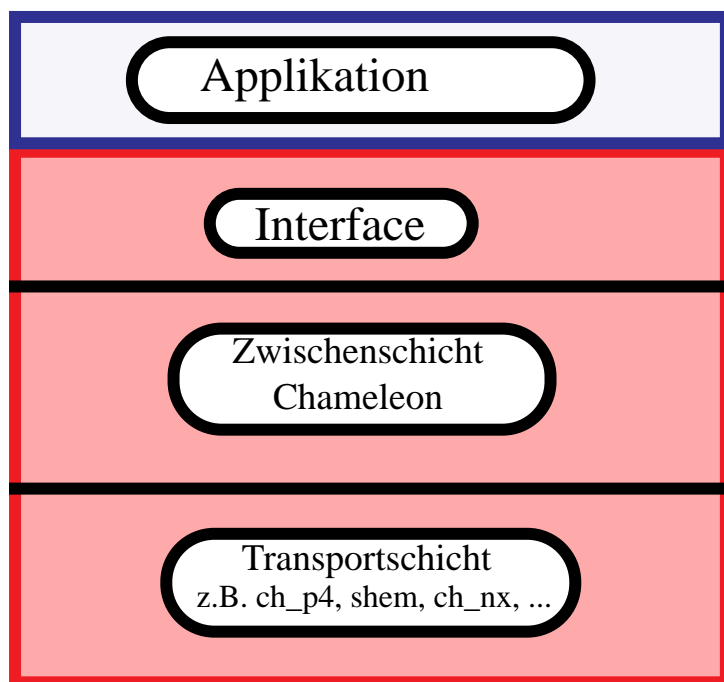
In dieser Programmbeschreibung wird auf den internen Aufbau von *MPI* eingegangen und begründet, warum eine Erweiterung der Funktionalitäten erforderlich ist. Der Lösungsansatz zur Erweiterung von *MPI* wird vorgestellt.

Der interne Aufbau von MPI

Die auf der Paragon installierte Version von *MPI* wurde vom *Argonne National Laboratory* entwickelt. Diese Version unterstützt sowohl Workstation-Cluster als auch verschiedene „echte“ Parallelrechner. Die Implementierung erfolgte nach einem Schichtenmodell, wodurch der Aufwand der Portierung auf weitere Hardwareplattformen minimiert wird. In Abbildung 35 ist der Aufbau von *MPI* und sein Zusammenwirken mit der Applikation des Benutzers schematisch dargestellt. Die Applikation ruft die *MPI*-Funktionen der Interface-Schicht zum Datenaustausch mit anderen Knoten auf. Diese *MPI*-Funktionen verwenden Routinen der Zwischenschicht (*Chameleon*), um die Anzahl der nötigen Funktionen der Transportschicht zu reduzieren. Dazu werden komplizierte Übertragungsfunktionen in

einfachere Basisfunktionen überführt, indem diese zum Teil mehrfach aufgerufen werden. Dadurch enthält die Transportschicht wenige einfache Übertragungsfunktionen und ist somit leicht auf eine neue Hardwareplattform zu portieren.

ABBILDUNG 35. Interner Aufbau von *MPI*



Die Transportschicht enthält die hardwareabhängigen Funktionen. Daraus folgt, daß MPI für jeden zu unterstützenden Rechner eine spezielle Transportschicht (*Device*) zur Verfügung stellen muß. Beispielsweise bildet das Device *ch_nx* die von der Zwischenschicht benutzten Transportfunktionen auf die Message-Passing Funktionen der Paragon ab. Ein anderes wichtiges Device ist *ch_p4*, das auf Workstationclustern eingesetzt wird. Zur Kommunikation werden bei dieser Version *Internet Sockets* eingesetzt. *Internet Sockets* sind

Kommunikationskanäle zwischen Rechnern, die durch ein Netzwerk miteinander verbunden sind.

Neben dieser Implementierung von *MPI* gibt es noch weitere Versionen, meist von Rechnerherstellern selbst, bei denen auf die Zwischenschicht verzichtet wird. Diese sind dann nur auf einem Rechnertyp einsetzbar, aber erreichen die bestmögliche Leistung.

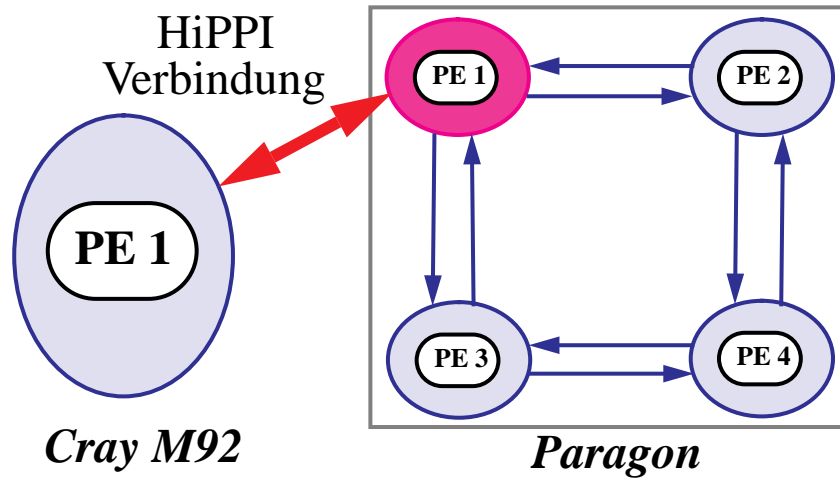
Allen *MPI*-Versionen ist gemeinsam, daß sie nicht mehrere „*Devices*“ gleichzeitig unterstützen. Das bedeutet, daß ein Datenaustausch über *MPI*-Funktionen z.B. auf der *Paragon* entweder nur innerhalb des Rechners erfolgen kann, oder das Programm muß mit einer *MPI*-Library gelinkt werden, die das „*ch_p4-Device*“ verwendet und damit laufen alle Kommunikationen über die langsameren Socketverbindungen. Es ist also *nicht* möglich, innerhalb des Parallelrechners *MPI*-Funktionen zu verwenden, die die hardwarenahen Message-Passing-Routinen ausnützen und für einen Datenaustausch mit einem anderen Rechner Socketverbindungen einzusetzen.

Zusammenwirken von *Paragon* und *Cray M92* im Meta-Computing-Szenario

Die Kommunikation zwischen den Knoten des Parallelrechners erfolgt über das *ch_nx-Device*, während für einen Datenaustausch mit der *Cray M92* eine schnelle HiPPI-Verbindung zum Einsatz kommt.

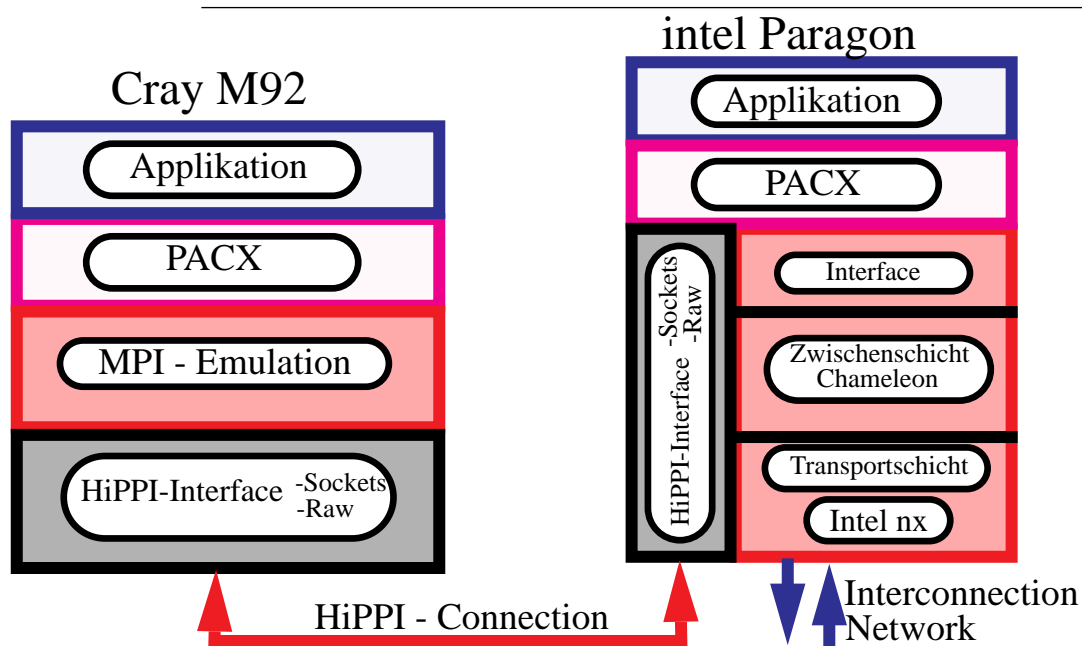
Bei der Kopplung der beiden Rechner spielt der Knoten 1 auf der *Paragon* eine Schlüsselrolle. Er hat ausschließlich die Aufgabe, die Verbindung zur *Cray* herzustellen. Aus Sicht des Programmierers repräsentiert dieser Knoten die *Cray M92*.

ABBILDUNG 36. Verteilung der Applikation auf die Knoten im Falle *Uranus*



Mit Hilfe dieser Art der Kopplung, der auf der *Cray M92* eingeführten MPI-Emulationsschicht sowie der auf beiden Rechnern eingeführten *PACX*-Zwischenschicht (*PARallel Computer eXtension*) kann die *Cray* indirekt MPI-Funktionen nutzen, um mit Knoten auf der *Paragon* zu kommunizieren. Bild 37 veranschaulicht den Aufbau.

ABBILDUNG 37. Funktionsschema der MPI - HiPPI-Extension



Die PACX-Schicht und die MPI-Emulationsschicht stehen dem Programmierer in Form einer Library zur Verfügung. Der PACX-Schicht kommen je nach Knoten verschiedene Aufgaben zu. Auf der Cray M92 werden hier Daten, vom Cray-Format in das standardisierte IEEE-Format¹ konvertiert. Anschließend ruft die PACX-Schicht, eine der in der MPI-Emulationsschicht enthaltenen Funktionen auf, die für die Kommunikation mit der Paragon direkt (z.B. MPI_Send()) oder indirekt (z.B. MPI_Pack()) notwendig sind. Diese erzeugen Befehle für den Interpreter der PACX-Schicht auf dem Paragon-Knoten Nr.1 (vgl. Abbildung 36) und schicken sie zusammen mit den zugehörigen Daten an diesen Knoten. Der Interpreter enthält die für ein Meta-Computing-Szenario notwendigen

1. Das Datenformat IEEE definiert, mit welcher Genauigkeit Zahlen dargestellt werden[18]. Dabei wird die Reihenfolge der Datenbytes innerhalb der Zahl nicht vorgeschrieben (Little/Big-Endian).

Funktionen. Sie rufen auf der Paragon die entsprechenden MPI-Funktionen unter Verwendung der *Cray*-Daten auf.

In umgekehrter Richtung, wenn z.B. Knoten 2 eine Nachricht an die *Cray M92* schicken will, so wird diese durch MPI-Routinen zuerst an den Knoten 1 der Paragon übertragen. Auf Knoten 1 durchläuft die Nachricht die MPI-Schicht und wird dann über die *PACX*-Schicht und die *HiPPI*-Interface-Schicht zur *Cray M92* übertragen. Auf diesem Knoten läuft der Interpreter, der auf Befehle von der *Cray M92* wartet und diese dann in MPI-Befehle (z.B. Datenaustausch mit anderen Knoten) umsetzt.

Für die Kommunikation innerhalb der Paragon ist die *PACX*-Schicht nicht notwendig, da hier die „original“ MPI-Funktionen zur Verfügung stehen. Diese Schicht ist dennoch auf allen Knoten vorhanden, damit auf allen beteiligten Knoten der Paragon ein einheitliches Programm verwendet werden kann.

Die Schicht „*HiPPI-Interface*“ ist für die Verbindung sowohl über *Internet Sockets* als auch im *HiPPI-Raw* Modus zuständig. Sie stellt Funktionen zum Aufbau der Verbindung und zur Datenübertragung bereit. Die Verbindung über *Internet Sockets* ist nicht an eine Netzwerktechnologie gebunden, sondern kann über jedes Übertragungsmedium erfolgen. Die Terminologie *HiPPI-Interface* wurde dennoch gewählt, da die Hauptanwendung im Raw-Modus stattfinden soll. Nur in diesem Modus ist eine befriedigende Übertragungsbandbreite zu erwarten.

Anwendung der *PACX*-Library

Dieser Abschnitt beschreibt, wie der Programmierer einer Applikation die *PACX*-Library einsetzt. Um die Unterschiede zu einem „normalen“ MPI-Programm zu verdeutlichen, wird zuvor gezeigt, wie ein solches entsteht.

Ein in der Programmiersprache C geschriebenes MPI-Programm sieht prinzipiell wie folgt aus:

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int main ( int argc, char *argv[] )
5 {
6     int mynode, myproc, numnode;
7
8     MPI_Init ( &argc, &argv );
9
10    :::::
11
12    MPI_Finalize ();
13
14    return ( 0 );
15 }
```

In den ersten beiden Zeilen liest der Preprozessor Funktionsdeklarationen¹ und vordefinierte Konstanten aus den Files *mpi.h* und *stdio.h*. In Zeile 4 beginnt das Hauptprogramm und direkt danach, in Zeile 6, erfolgt die Definition lokaler Variablen. In Zeile 8 erfolgt die Initialisierung der MPI-Umgebung. Im MPICH-Reference Manual [2] wird vorgeschrieben, daß dieser Funktionsaufruf so bald wie möglich nach dem Programmstart erfolgen soll. Zeile 10 symbolisiert die Berechnungen des Programmes. In Zeile 12 wird die MPI-Laufzeitumgebung und in Zeile 14 das Programm beendet, auch diese beiden Funktionen sollen möglichst unmittelbar aufeinander folgen.

Ist das Beispielprogramm im File *example.c* abgelegt und MPI in der Directory */usr/local/mpi* installiert, so kann das Programm mit den folgenden Kommandos übersetzt und anschließend gebunden werden.

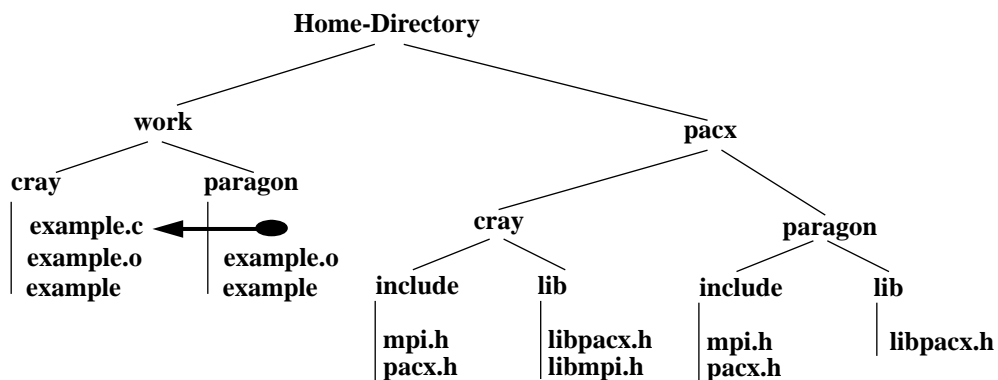
1. Eine Deklarationen macht dem Compiler den Datentype einer Variablen bzw. die Art einer Funktion (Datentyp des Funktionsergebnisses und der Argumente) bekannt.

```
% cc -c example.c -I/usr/local/mpi/include
% cc -o example example.o -L/usr/local/mpi/lib -lmpi
```

Die Option **-I** gibt an, in welcher Directory die Headerdatei *mpi.h* zu finden ist. Beim Binden des Programmes wird die Directory mit **-L** angegeben und **-lmpi** veranlaßt den Linker Funktionen aus der MPI-Library *libmpi.a* zu verwenden.

Damit dieses Beispielprogramm in einer Meta-Computing-Umgebung läuft, sind am Programm selbst keine Änderungen vorzunehmen. Es ergibt sich aber dennoch eine komplexere Situation, da für verschiedene Rechnertypen meist auch spezielle ausführbare Dateien erzeugt werden müssen. Steht die Home-Directory auf allen eingesetzten Rechnern zur Verfügung, so ist es zweckmäßig, für jeden Rechner ein Unterverzeichnis sowohl für *PACX* als auch für das Programm anzulegen. Die Verzeichnisstruktur sieht dann wie in der folgenden Abbildung aus:

ABBILDUNG 38. Verzeichnisstruktur zur Programmverwaltung in Meta-Computing-Szenarien



Das File *example.c* steht über einen link sowohl in der Directory *cray* als auch in *paragon* zur Verfügung. Bei einem link wird ein Verweis auf eine Datei oder ein Verzeichnis erzeugt. So ist die Quelldatei nur einmal vorhanden, wodurch Inkonsistenzen vermieden werden, und

es kann von beiden Verzeichnissen aus auf die Datei zugegriffen werden. Auf der *Cray M92* ist die MPI-Emulation in der Library *libmpi.a* enthalten und im gleichen Verzeichnis abgelegt wie *libpacx.h*. Auf der *Paragon* wird die vom System bereitgestellte MPI-Library verwendet.

Auf dem Parallelrechner muß zum Einfügen der *PACX*-Schicht in der Befehlszeile die Directory der von *PACX* zur Verfügung gestellten Header-Datei vor der von MPI stehen! Dabei wird ausgenutzt, daß die Directories in der angegebenen Reihenfolge durchsucht werden. Dadurch wird die von *PACX* bereitgestellte Header-Datei *mpi.h* verwendet. Diese wiederum bindet die „original“ MPI-Header-Datei und die für *PACX* notwendige Datei *pacx.h* ein. Es sind also zwei Header-Dateien mit dem Namen *mpi.h* vorhanden. Das begründet auch, daß *PACX* in einer eigenen Directory installiert sein muß.

Ist *PACX*, wie in Abb. 38, in der Home-Directory unter *pacx* installiert, so sind folgende Kommandos nötig, um das obige Programm für die *Paragon* und *Cray M92* zu erhalten:

1. **Paragon:**

```
% cc -c example.c -I${HOME}/pacx/paragon/include \
-I/usr/local/mpi/include
% cc -o example example.o -L${HOME}/pacx/paragon/lib \
-L/usr/local/mpi/lib -lpacx -lmpi
```

2. **Cray M92:**

```
% cc -c example.c -I${HOME}/pacx/cray/include
% cc -o example example.o -L${HOME}/pacx/cray/lib -lpacx -lmpi
```

Programmstart in der Meta-Computing-Umgebung

Vor dem Programmstart muß die Meta-Computing-Umgebung in dem Konfigurationsfile *.hostfile* beschrieben werden. Diese Datei wird in der Initialisierungsroutine von *PACX* gelesen und legt fest, welcher Knoten des Parallelrechners eine Verbindung zu einem Host aufbaut und welches Protokoll verwendet wird. Sie muß im aktuellen

Verzeichnis auf dem Parallelrechner liegen. Der Aufbau dieser Datei wird in der folgenden Abbildung am Beispiel der für *Uranus* verwendeten Konfiguration beschrieben.

ABBILDUNG 39. PACX - Konfigurationsdatei zur Beschreibung der Meta-Computing-Umgebung

```
1 # PACX - Konfigurationsfile fuer Uranus
2 #
3 # Host          protokol    nodenumber    [command]
4 paragon        nx           0..n
5 ruscy-hi.rus   tcp           0
```

Beginnt eine Zeile mit dem Zeichen #, so handelt es sich um einen Kommentar, der nicht weiter beachtet wird. Jede Konfigurationszeile besteht aus mindestens drei Einträgen. Der erste Eintrag ist der Hostname, zu dem eine Verbindung aufgebaut werden soll. Der zweite Eintrag bestimmt das Protokoll, das zur Kommunikation verwendet wird. Als Protokoll sind derzeit folgende Einträge zulässig:

- **nx**: verwendet die Funktionen aus der MPI-Library.
- **tcp**: benutzt *Internet Sockets* zur Datenübertragung. Dieses Standard-Übertragungsprotokoll ist langsam, hat aber den Vorteil, daß es auf allen Unix-Systemen mit allen verfügbaren Netzwerktechnologien eingesetzt werden kann.
- **hippi_raw**: kommuniziert über eine HiPPI-Verbindung im Raw Modus.

Der dritte Eintrag bestimmt, welcher Knoten des Parallelrechners als Interpreter betrieben wird, dabei können Bereiche durch zwei Punkte oder mehrere Knoten durch Kommata getrennt werden (ohne Leerzeichen!). Der Rest der Zeile kann ein Kommando enthalten, das vom entsprechenden Knoten in einer Subshell ausgeführt wird. Dadurch ist es möglich, das Programm auf dem „remote“ Host durch das Kommando rsh automatisch zu starten¹.

1. Dieses Feature ist vorgesehen, aber noch nicht vollständig implementiert.

Für den Programmlauf wird nun für jeden Host eine Shell benötigt. Stehen diese bereit, so wird das Programm auf dem Parallelrechner gestartet. Sobald ein Knoten als Interpreter betrieben wird, stellt er die Verbindung zum „remote“ Host her und gibt eine Zeile mit dem Hostnamen und den verwendeten Port-Nummern aus. Stehen diese Informationen bereit, so wird das Programm auf dem „remote“ Host gestartet und bei der Initialisierung werden die Informationen abgefragt. Danach läuft das Programm wie gewohnt. Durch den Einsatz einer Shell je Knoten erhält man die Programmausgaben nach Knoten getrennt. Das hat sich als sehr übersichtlich erwiesen, so daß die Implementierung des automatischen Programmstarts vorerst zurückgestellt wurde.

Implementation der Funktionen

Im Rahmen dieser Diplomarbeit müssen nur diejenigen Routinen für ein Meta-Computing-Szenario am Beispiel von *Uranus* bereitgestellt werden, die in dieser Beispielanwendung zum Einsatz kommen. In diesem Kapitel werden diese in tabellarischer Form aufgelistet, die Arbeitsweise jeder Routine beschrieben und die Implementierung genauer erläutert.

Beschreibung der von Meta-Computing-Szenarien verwendeten MPI-Funktionen.

Um einen Überblick zu erhalten, welche Funktionen die MPI-Emulationsschicht zur Verfügung stellt, werden diese hier in einer Tabelle angegeben. Nur ein Teil der Funktionen wird vom Interpreter auf der Paragon benötigt. Diese sind in der Tabelle mit * gekennzeichnet. Der Tabelle ist weiterhin zu entnehmen, ob eine Funktion mit einem anderen Knoten kommuniziert (*remote*), oder ob sie *local*, d.h. nur auf einem Knoten bzw. in diesem Fall auf der *Cray M92* arbeitet.

Ein Eintrag gibt an, ob bei der Ausführung der Routine eine zeitaufwendige Datenkonvertierung durchgeführt wird, oder nicht.

TABELLE 3. Liste der verfügbaren Funktionen

Name der Funktion	Interaction <i>local/remote</i>	Datenkon vertierung
MPI_Init ()	remote	nein
MPI_Finalize () *	remote	nein
MPI_Abort () *	remote	nein
MPI_Comm_Rank ()	local	nein
MPI_Comm_Size ()	local	nein
MPI_Send () *	remote	ja
MPI_Receive () *	remote	ja
MPI_Bcast () *	remote	ja
MPI_Pack ()	local	ja
MPI_Unpack ()	local	ja

Die ersten drei Tabelleneinträge sind Funktionen zur Kontrolle der Programmumgebung. Die folgenden beiden Routinen sind zur Prozeßkontrolle nötig und die restlichen werden für den Datenaustausch mit anderen Knoten bzw. zum Bearbeiten der Daten vor oder nach einem Datenaustausch benötigt.

Neben den in der Tabelle aufgeführten Funktionen können alle weiteren aus der MPI-Library verwendet werden, solange sie *nur* auf dem Parallelrechner verwendet werden und nicht den „*remote*“-Knoten ansprechen. Ein Beispiel dafür ist die Funktion `MPI_IRecv ()`. In Abbildung 36 auf Seite 70 können die Knoten 2,3 und 4

MPI_Isend () ohne weiteres zum Datenaustausch verwenden, da hier der Interpreter auf Knoten 1 nicht angesprochen wird.

Programmbeschreibung
