Parallelrechnerseminar IWR TU Dresden
Einführung und Message Passing Systeme

N. Schmeißer*
Forschungszentrum Rossendorf e.V.,
Abt. Kommunikation und Datenverarbeitung

28. Oktober 1996

*Telefon: (+49 351) 260/2207, E-Mail: nils@schmeisser.com,
URL: http://www.fz-rossendorf.de/FVTK/MITARB/schmei

Inhaltsverzeichnis

1 Parallelrechner

- heute zwei Kategorien von Parallelrechnern ("echte" und Cluster)
- Gegenüberstellung von Vor- und Nachteilen der beiden Kategorien
- FZR-Cluster als Beispiel
- **Was ist wichtig?**
  - hohe Knotenleistung
  - schneller Datenaustasuch
  - Systemsicherheit
  - Kosten
- Bemerkenswert: Transputer, insbesondere INMOS Chips ermöglichen sehr einfach Realisierung eines Parallelrechners mit message passing

2 Kommunikationsschemata

- zwei grundlegende Schemata: shared memory und message passing

  shared memory : Prozessoren nutzen einen gemeinsamen Speicher, d.h. sie können direkt auf ein und das selbe Datum zugreifen, shm erfordert oft explizite Synchronisation

  message passing : Prozesse müssen sich Nachrichten schicken, um auf ein Datum zugreifen zu können (impliziert Synchronisation, in dem Sinne, das ein Prozeß erst auf ein Ergebnis zugreifen kann, wenn es an ihn versendet wurde)
- Vergleich von Eigenschaften
- Geschwindigkeiten/Bandbreiten

Architektur	Bandbreite
SPP 1000 (shared memory)	250 MB/s
Cray T3D (shared memory)	300 MB/s
IBM SP/2	40 MB/s
Intel iPSC	2.8 MB/s
Ethernet	1.2 MB/s
FDDI	12 MB/s
ATM-100	12 MB/s

- distributed shared memory
  (Convex SPP, Hypernode-Modell, lokale Speicher verknüpft über CTI-Netzwerk)

3 Paralleles Programmieren

3.1 Programmier Paradigmen

- **master-slave** : ein Prozeß überwacht die Arbeit aller anderen, dieser master-Prozeß ist für die Verteilung von Aufgaben (workload allocation, load balancing) zuständig und sammelt Ergebnisse ein

  node-only : eine Anzahl gleichberechtigter Prozesse arbeitet an einer Aufgabe, es gibt keinen dedizierten Prozeß, i.A. werden gleicher Instanzen eines Programmes auf jedem Knoten ausgeführt (→ SIMD)

- dieses Schema läuft unter dem Namen "crowd computation" und wird normalerweise in drei Schritten ausgeführt:

  - Initialisierung (starten alle Prozesse und Aufbau von Topologien)
  - Berechnung
  - Einsammeln der Ergebnisse (im Prinzip kann jeder Prozeß seine Ergebnisse selbstständig in einer zentralen Ressource ablegen (→ Ausschlußsynchronisation))

  Beispiel:Berechnung eines Ausschnittes aus der Mandelbrotmenge

  - die Berechnung der Punkte kann unabhängig voneinander erfolgen
  - master startet n-1 slaves (bei n physische Prozessoren)
  - master teilt jedem slave eine Menge von zu berechnenden Punkten zu
  - ist ein slave fertig, erhält er eine neue Aufgabe (load balancing)
  - master sammelt alle Punkte und baut den gesamten Ausschnitt zusammen

- weitehin oft verwendetes Schema ist die "**tree-computation**", gelangt man im Berechnungsgraphen an eine Stelle, an der $n \geq 2$ Teilaufgaben parallel ausführbar sind, so werden n-1 neue Prozesse gestartet, der startende Prozess folgt i.A. der ersten Alternative

- **workload allocation**: kann auf zwei Arten erfolgen

  data decomposition : die zu bearbeitenden Daten werden verteilt (siehe Mandelbrot- Beispiel

  function decomposition : eine Anzahl voneinander verschiedener Unabhängiger Teilaufgaben wird auf die vorhandenen Prozessoren (Prozesse) verteilt

3.2 Kommunikation (message passing)

- allg.: message passing ist der Austausch von Nachrichten (die Daten enthalten)

3.3 blockieren/nicht blockierend

- **non-blocking** : ein Sende-/Empfangsvorgang wird angestoßen, erst eine Komplettierungsanweisung (Synchronisation) sorgt für die Vervollständigung
  (Philisophie: der Vorgang enthält in vielen Fällen Zeitabschnitte, in denen sich der Prozeß im Leerlauf befindet, u.U. gbt es aber Teilaufgaben, die vom, zu empfangenden Datum unabhängig sind, diese könnten in dieser Zeit bearbeitet werden)

  blocking : ein Sende-/Empfangsprozeß gilt als beendet, sobald die Daten vollständig abgesetzt bzw. empfangen wurden ($\rightarrow$ idle times), impliziert Synchronisation
  unangenehmer Effekt: P_1 wartet auf Nachricht von P_2
  und umgekehrt $\rightarrow$ **dead-lock**

- diese Operationen werden auch als asynchroner Nachrichtenaustausch bezeichnet

- synchrone Operationen

  synchron : eine Sendung gilt erst als beendet, wenn der Empfang bestätigt wurde (rendevouz)

  ready : Sendung beginnt erst, wenn der Empfang gestartet wurde

3.4 andere Klassifikation

- **point-to-point** : ein Sender schickt Daten an genau einen Empfänger, abstrakter formuliert: immer genau zwei Prozesse haben einen gemeinsamen geöffneten Übertragungskanal

  collective comunications : hier "hängen" mehr als zwei Prozesse an einem Übertragungskanal, i.A. beziehen sich diese Operationen immer auf die Mitglieder einer (Prozeß-)Gruppe

  barrier synchronisation : alle Mitglieder einer Gruppe warten an einer bestimmten Stelle aufeinander, vielfach besteht auch die Möglichkeit nur eine bestimmte Anzahl von Mitglieder zu synchronisieren
  sollen bestimmte Prozesse synchornisiert werden, so verwendet man hierfür besser point-to-point Operationen

  broadcasting : ein Prozeß einer Gruppe sendet an alle anderen Mitglieder (u.U. auch an sich selbst) ein und die selbe Nachricht

gathering : bezeichnet den Vorgang des *Einsammelns* von (verteilten) Daten, es gibt zwei Interpretationen:

- nur ein Prozeß sammelt alle Daten
- alle Prozesse sammeln alle Daten (gather from all), z.B. eine Matrix, das ist die "message passing Variante des shared memory"

scatter : die Daten werden verteilt und nicht gesammelt, auch hier gibt es wieder die zwei Varianten (der Initiator sendet an nur einen oder an alle)

reduction : die Abfolge Einsammeln aller Daten, Anwenden einer Operation, Verteilen der Ergebnisse wird oft zu sogenannten Reduktionen zusammengefaßt

Beispiel:jeder Prozeß halte eine Menge von Teilchen (Position und Masse) es soll der Schwerpunkt des Systems bestimmt werden

$$\underline{x} = \frac{\sum_{j=1}^{k} \underline{x}_k \cdot m_k}{\sum_{i=1}^{k} m_k} \quad (1)$$

in den meisten Systemen sind Standardoperationen wie Summenbildung, Maximum- und Minimumbestimmung vorhanden

scan : mit scan bezeichnet man eine hierarchische Reduktionsoperation, d.h. die Operation arbeitet schrittweise über einer Folge von Daten D_i mit $D_0 \subseteq D_1 ... \subseteq D_n$

3.5 Synchronisation

- oft werden für die Berechnung der einzelnen Zweige eines Berechnungsgraphen unterschiedliche Zeiten benötigt, dies wird u.U. durch die Heterogenität und damit nicht notwendig gleiche Rechenleistung der Knoten eines Clusters verursacht, es ist deshalb notwendig, an bestimmten Stellen Prozesse zu synchronisieren, d.h. sicherzustellen, daß eine Berechnung erst fortgesetzt wird, wenn alle notwendigen Eingabedaten (die eventuell Ausgabedaten anderer Prozesse sind) zur Verfügung stehen ($\rightarrow$ Datenabhängigkeiten)

- beim Zugriff auf gemeinsame Ressourcen (Konsolen, Dateien o.a.) müssen Zugriffskonflikte verhindert werden (z.B. lesender Zugriff, während ein Schreibzugriff stattfindet)

  semaphor : gemeinsame Variable, die den Zustand einer Ressource kennzeichnet (besetzt oder frei), klassisch charakterisierbar als Signal/Ampel

  monitor : "zentralisierte Instanz zwischen Prozessen", kontrolliert Zugriffe auf eine Ressource, kann Warteschlangen bearbeiten

3.6 Prozessor Sets und Topologien

- Probleme lassen sich mitunter in unabhängige Teilaufgaben zerlegen, die ihrerseits wieder Parallelisierbar sind, man läßt ein solches Teilproblem dann auf einer Prozessor-Gruppe abarbeiten, innerhalb einer Gruppe gibt man Verbindungsstrukturen an und erhält eine $\rightarrow$ Topologie
- durch eine Topologie wird ein "Abstand zwischen Prozesseinheiten" festgelegt, d.h. das Maximum der Anzahl der Kanten auf dem kürzesten Weg von P_i nach P_j, dieses Maß wird auch "hops" genannt
- **Ring** : $h = \frac{n}{2}$

  Stern : alle PE's haben eine Kante mit einem zentralen Knoten, $h = 2$

  vollständiger Graph : jede PE ist mit jeder verbunden, $h = 1$

  Gitter/Torus : $h = n + m - 1 / h = c \cdot \sqrt{n}$

  shuffle exchange :

  Bäume : $h = 2 \cdot ld(n)$

  fat tree :

  Quadtree : $h = 2log_4(n)$

  Hypercube : $h = ld(n)$

4 Effizienz

- Speedup $s_n = \frac{T_1}{T_2}$; $s_n \leq n$
- Minsky'sche Vermutung: praktischer Speedup $O(logn)$
- Amdahl's Gesetz: sei f der normierte Anteil der sequentiellen Programmzeit, dann ist

$$t_n \geq f \cdot t_1 + (1 - f) \cdot \frac{t_1}{n} \quad (2)$$

$$s_n \leq \frac{n}{1 + f \cdot (n - 1)} \quad (3)$$

- **Grundregel: Kommunikationszeit $\ll$ Rechenzeit.**
- message passing time: $t_{n,h} = \alpha + n \cdot \beta + (h - 1) \cdot \gamma$
  α startup-time, β costs per byte, γ hop-delay, h number of hops
- **Grundregel: lange Nachrichten, kurze Wege**

5 Message Passing Systeme

- **UNIX V IPC**: semaphore, messages shared memory, sockets
- **PVM**: parallel virtual machine, Geist, Dongarra
  Einrichtung:
  1. Quellen holen und auf jeder Zielplattform übersetzten
  2. $(HOME)/pvm3/bin/$(PVM_ARCH) auf jeder Maschine anlegen (günstig ist hier ein NFS), $(HOME)/pvm3/lib anlegen und für jede Plattform eine Link auf die Plattformbibliothek setzen (z.B. $(HOME)/pvm3/lib/RS6K $\rightarrow$ /usr/local/pvm/lib/RS6K)
  3. executables auf jeder Zielplattform übersetzen und in die entsprechenden Verzeichnisse kopieren
  4. Überprüfen, ob $PVM_ROOT und $PVM_ARCH auf jeder Maschine gesetzt sind
  5. "pvm" starten (das ist die PVM-Konsole)
  6. parallele Maschine aufbauen ("add <hostname>"), man kann das auch automatisieren, indem man alle Knoten der Maschine in eine Datei schreibt (hostfile) und "pvm $< hostfile >$" aufruft
  7. Programm starten ("spawn <programmname>"), man kann die Konsole auch mit "quit" verlassen, und dann in der normalen Konsole arbeiten, ein Programm wird dann einfach durch Aufruf mit seinem Namen gestartet
  8. parallele Maschine herunterfahren ("halt")
- **MPI**: message passing interface, Gropp, Lusk
  MPI arbeitet meist (es gibt ein paar wenige Ausnahmen) nur auf einer koherenten Plattform, vorzugsweise nur auf Parallelrechnern und ist dort meist schon fertig installiert
  zum Start ruft man einfach "mpirun -np <knotenzahl> <programm>" auf
- **weitere**: p4, Linda, PARMACS
- **Chameleon**: Gropp, Smith
  Aufsatz auf Systeme, einheitliches Interface auf Kosten Funktionsumfang
  zunächst muß ein "Prozessverteiler" (z.B. PVM, MPI oder p4) installiert werden,
  die Erzeugung der executables richtet sich dann nach den Regeln der jeweiligen Schicht,
  man ruft dann nur noch "<programm> -np <knotenzahl>"

5.1 Programmrahmen

```
/*---------------------------------------*/
/*--- file: frame.c ----------------------*/
/*---------------------------------------*/

#ifdef PVM
  #include <pvm3.h>
#endif
#ifdef MPI
  #include <mpi.h>
#endif
#ifdef CHAMELEON
  #include <tools.h>
  #include <comm/comm.h>
#endif

static int myid;
static int nproc;

int worker(int argc,char **argv) {

  return 0;
}

int main(int argc,char **argv) {
  unsigned int i;
  int result;

  #ifdef PVM
    /* anzahl prozesse aus kommandozeilenargument -np <anzahl> bestimmen */
    int parent;
    int *tids;

    nproc=1;
    for (i=1;i<argc;i++) {
      if (!(strcmp(argv[i],"-np")))
        sscanf(argv[i+1],"%d",&nproc);
    }
    /* ermitteln der prozess id, der eltern und der prozessnummer */
    myid  =pvm_joingoup("frame");
    parent=pvm_parent();
    /* synchronisierter start von insgesamt ntask prozessen */
    if (parent==0) {
```

```
      tids=(int*)malloc(sizeof(int)*nproc);
      pvm_spawn("frame",argv,PvmTaskDefault,"",nproc-1,tids);
      free(tids);
    }
    pvm_barrier("frame",nproc);
    result=worker(argc,argv);
    pvm_exit();
  #endif

  #ifdef MPI
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    result=worker(arc,argv);
    MPI_Finalize();
  #endif

  #ifdef CHAMELEON
    myid =PImytid;
    nproc=PInumtids;
    result=PICall(worker,argc,argv);
  #endif

  return result;
}
```

6 Probleme

6.1 Vielteilchensystem

Ein klassisches Problem der Physik ist das sogenannte Vielkörper(teilchen) Problem. Das heißt, eine Menge von Teilchen mit bestimmten Eigenschaften (Position, Geschwindigkeit, Masse, Ladung) befindet sich in einem feldfreien Raum und man will das zeitliche Verhalten dieses Systems studieren.
Im nicht-relativistische Geschwindigkeitsbereich beschreiben die Gesetze der klassischen Newton'schen Physik das Verhalten der Teilchen:

$$\ddot{\vec{x}}_i \cdot m_i = \vec{F}_i^m + \vec{F}_i^q \tag{4}$$

mit

$$\vec{F}_i^m = \gamma \cdot \sum_{j=1}^{n} m_i \cdot m_j \frac{\underline{x}_i - \underline{x}_j}{||\underline{x}_i - \underline{x}_j||^3}, i \neq j \tag{5}$$

$$\vec{F}_i^q = -\frac{1}{4\pi\epsilon} \cdot \sum_{j=1}^{n} q_i \cdot q_j \frac{\underline{x}_i - \underline{x}_j}{||\underline{x}_i - \underline{x}_j||^3}, i \neq j \tag{6}$$

Im Fall der Kollision zweier Teilchen soll ein vollkommen elastischer Stoß angenommen werden, also

$$Impulserhaltungssatz : m_i \cdot \underline{\dot{x}}_i + m_j \cdot \underline{\dot{x}}_j = m_i \cdot \underline{\dot{x}}'_i + m_j \cdot \underline{\dot{x}}'_j, i \neq j \tag{7}$$

und

$$Energieerhaltungssatz : \frac{m_i}{2}\underline{\dot{x}}_i^2 + \frac{m_j}{2}\underline{\dot{x}}_j^2 = \frac{m_i}{2}\underline{\dot{x}}_i'^2 + \frac{m_j}{2}\underline{\dot{x}}_j'^2, i \neq j \tag{8}$$

Alle Teilchen sollen den gleichen Radius r haben. Ein Stoß findet dann statt, wenn $||\underline{x}_i - \underline{x}_j|| \leq 2r$.

- Schreiben Sie ein Programm, daß mit festen Zeitschritten dt die neuen Eigenschaften der Mitglieder eines n-Teilchen-Raumes (dim=3) berechnet. Hinweis: Um einen Urzustand des Systems zu generieren, verwenden Sie eine gleichmäßige Verteilung über dem Einheitswürfel. Um einen Stoßvorgang zu berechnen gehen Sie so vor:
  Gilt die Relation $< 2r$ so berechnen Sie den Zeitpunkt dt_t innerhalb des Zeitschrittes, in dem der Stoß stattfindet und zu dem sich die kollidierenden Teilchen berühren $(= 2r)$, führen Sie den Stoß aus und berechnen Sie die Eigenschaften der beteidigten Teilchen zur Zeit $dt - dt_t$ nach dem Stoß.

- Die Lösung dieses Problems läßt sich parallelisieren. Begründen Sie dies und führen Sie die Parallelisierung an Ihrem Programm aus. Achten Sie dabei auf Skalierbarkeit.

- Welches Parallelisierungsschema läßt sich sinnvoll anwenden (master-slave oder node-only, data oder function decomposition)?
- Testen Sie verschiedene Topologien (z.B. vollständiger Graph und Ring). Benutzen Sie broadcasting und point-to-point Kommunikation.
- Sinnvoll kann auch die Anwendung von Reduktionen sein.
- Die Lösung ist ein iterativer Vorgang. Welches Problem kann dabei auftreten (Stichwort: Genauigkeit)? Machen Sie einen Vorschlag, wie sie dieses Problem beseitigen können (Stichwort: Energie des Systems, Energieerhaltungssatz).
- Bestimmen Sie den speed-up für verschiedene Anzahl von physischen Prozessoren. Verwenden Sie auch verschiedene Teilchenzahlen $5 \leq n \leq 2000$.

6.2 Mandelbrot-Menge

Aus dem Iterationsprozeß $\bar{x}_{n+1} = \bar{x}_n^2 + \bar{c}$ wird die Mandelbrotmenge bestimmt. Es ist $\bar{x}_0 = 0 + 0i$, $\bar{c}$ variiert über einem vorgegebenen Intervall der komplexen Ebene. Der Prozeß fragt, ob $(\bar{x})$ konvergiert oder divergiert. Wenn die Folge divergiert, so fragt man, in wieviel Schritte $(\bar{x})$ braucht, um eine obere Schranke zu überschreiten. Diese Anzahl bestimmt den Wert (Farbe) eines Punktes der Mandelbrotmenge.
Algorithmus:

1. n_{max} = maximale Anzahl Iterationen
   d = Divergenzgrenze
2. $x_0 = 0 + 0i$, $x = x_0$, $k = 0$, setze c
3. $x = x^2 + c$
4. falls $|x| > d$, bestimme Farbwert aus k und Ende
5. $k = k + 1$
6. falls $k < n_max$ gehe zu 3.
7. Folge konvergiert für diesen Startwert, setze einen festen Farbwert

- Dieses Problem ist zutiefst parallel in sich selbst. Schreiben Sie ein Programm, das eine diskrete Menge von Punktwerten (Farben) von Punkten der Mandelbrot-Menge über einem komplexen Rechteck parallel berechnet. Achten Sie dabei auf eine effiziente Auslastung der Knoten Ihres "Parallelrechners" (load balancing).
- Messen Sie die time-to-solution, d.h. die Zeit, die vom Start des Programmes bis zu seiner Beendigung verflossen ist. Verwenden Sie zum Beispiel auf einem PVM-Cluster der Aufruf "time <programm>" und lesen Sie die sogenannte "wall clock" aus. Variieren Sie die Anzahl der Prozessoren und die Problemgröße. Bestimmen Sie den Speed-up für konstante Problemgröße erzeugen Sie ein Speed-up Diagramm. Was stellen Sie fest?
- Sie haben sich für ein master-slave Schema entschieden. Bauen Sie in genau einen der slave-Prozesse eine Blockierung ein, die im Ergebnis eines zufälligen Versuches aktiviert wird (Simulation des Ausfalls dieses Knotens). Wie können Sie einen solchen Fehler erkennen?

6.3 5-Philosophen Problem

An einen großen Saal, in welchem sich ein runder Tisch mit n_t Tellern, zwischen denen jeweils genau ein Gabel liegt, grenzen n_p Kammern, in denen sich jeweils genau ein Philosoph aufhält, der mit Denken beschäftigt ist. Zu einem nicht vorherstimmbaren Zeitpunkt wird ein Philosoph, unabhängig von den anderen, hungrig und will essen (es gibt Spaghetti). Damit er das kann, muß ein Teller mit zwei freien Gabeln verfügbar sein. Ist kein Teller verfügbar, muß der Philosoph warten und kann in der Zwischenzeit nicht weiterdenken. Zum Essen benötigt jeder Philosoph wiederum eine nicht vorherstimmbare Zeitdauer.

- Schreiben Sie ein paralleles Programm mit n_p unabhängigen Teilprozessen (Philosophen) und einem Master (Speisesaal), welches das Philosophen Problem realisiert. Protokollieren Sie das Verhalten der Philosophen und die Belegung der Plätze im Saal für eine bestimmte Zeitdauer.
- Testen Sie das/die von Ihnen gewählten Verfahren zur Ausschlußsynchronisation für $n_p < n_t$, für $n_p = n_t$ und insbesondere für $n_p > n_t$ und $n_p \gg n_t$.

Hinweis: Versuchen Sie sich an der Implemenation von Semaphoren und Monitoren. Verwenden Sie Standardalgorithmen, wie den von Dijkstra.

6.4 Intervallschachtelung

Aufspannen neuer Tasks (tree computation), Prozessgruppen

6.5 globale Optimierung

Aufspannen neuer Tasks (tree computation), Prozessgruppen

6.6 Funktionsdekomposition

Berechnungsgraph, Topologie, dynamische Konfiguration, non-blocking Operationen

6.7 Matrixoperationen

Multiplikation -¿ master-slave, point-to-point, broadcasting, Effizienz (1..n Prozesse)
Tansponierte