Eine Bibliothek mit parallelen Algorithmen und Datenstrukturen zur Lösung geometrischer Probleme

Holger Zickner* Holger Blaar† Matthias Bäsken

Fachbereich Mathematik und Informatik
Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg

Zusammenfassung

Es wird ein Vorhaben zur Erarbeitung einer Bibliothek mit parallelen geometrischen Algorithmen und Datenstrukturen zur Nutzung auf Parallelsystemen mit verteiltem Speicher vorgestellt. Spezielle verteilte Datenstrukturen sind hierbei erforderlich, um eine effiziente Kommunikation zu ermöglichen. Erste Ergebnisse mit Hilfe objektorientierter Verfahren wurden auf unterschiedlichen Hardwareplattformen für Algorithmen und verteilte Datenstrukturen an den Beispielen Sortieren und Berechnung konvexer Hüllen von Punktmengen erzielt.

1 Einführung

Die Entwicklung von Softwarekomponenten zur Lösung von Aufgaben bestimmter Problemklassen ist ein wichtiger Beitrag der Informatik zur kommerziellen und akademischen Softwareentwicklung. Ein wichtiges Ziel hierbei sollte es sein, neue Forschungsergebnisse zu effizienten Algorithmen und Datenstrukturen unmittelbar und schnell in Anwenderprogrammen nutzbar zu machen. Beispiele hierfür sind effiziente Sortier- und Suchalgorithmen, Datenstrukturen für Mengen und Folgen, Graph- und Netzwerkalgorithmen oder Algorithmen zur Lösung geometrischer Probleme. In der traditionellen sequentiellen Programmierung wurden in den letzten Jahren einige Softwarebibliotheken entwickelt, die mit Hilfe objektorientierter Verfahren dieses Ziel erreichen, z. B. die *Library of Efficient Data types and Algorithms* LEDA [MN95].
In diesem Beitrag soll ein Vorhaben zum Aufbau von Bibliotheken mit effizienten *parallelen* Algorithmen und Datenstrukturen für Parallelsysteme, insbesondere mit verteiltem Speicher, in seiner Anfangsphase vorgestellt werden. Eine besondere Problematik gegenüber sequentiellen Bibliotheken dieser Art stellt das Ziel dar, weitgehendst plattformunabhängige Software zu erarbeiten. In einem ersten Schritt werden möglichst einfache,

*e-mail: zickner@informatik.uni-halle.de
†e-mail: blaar@informatik.uni-halle.de

grobkörnige Parallelisierungen, z. B. *divide-and-conquer* Strategien, untersucht. Hierbei können effiziente Algorithmen der sequentiellen Bibliothek genutzt werden, wenn im praktisch relevanten Fall $p \ll n$ (p Prozessoren bzw. Rechnerknoten, n Eingaben) n/p Eingabewerte auf p Prozessoren verteilt werden. Interessant sind dann schnelle Mischverfahren, um die Gesamtlösung zu ermitteln. Als Programmierumgebungen werden PVM (*Parallel Virtual Machine*) bzw. MPI (*Message-Passing Interface*) sowie ähnliche proprietäre Entwicklungen (z. B. PARIX [Par93]) genutzt. Die Quellen werden objektorientiert in C++ erstellt.
Um die Nutzung paralleler Algorithmen aus Bibliotheken zu ermöglichen, müssen geeignete verteilte Datenstrukturen (z. B. verteilte Felder, verteilte Graphen) entwickelt werden. Bei deren Nutzung werden die erforderlichen Kommunikationsoperationen zwischen Prozessoren bzw. Rechnerknoten (Verteilen, Zusammenführen der Daten) ausgeführt, ohne daß der Nutzer der Bibliotheksfunktionen dies explizit veranlassen muß. Das verteilte Halten sowie das Bewegen zwischen Knoten muß effizient gestaltet werden, um einen möglichst geringen Kommunikationsoverhead zu erreichen.
Die für eine solche Bibliothek vorgesehenen parallelen Algorithmen sind u. a. interessant für Echtzeitanwendungen aus digitaler Bildverarbeitung, Mustererkennung und Computer Vision.
Als erste Beispiele werden eine Bibliothek mit parallelen Sortieralgorithmen, parallele Algorithmen zur Berechnung der konvexen Hülle einer Punktmenge und die Datenstruktur *Verteiltes Feld* vorgestellt.

2 Der Datentyp dist_array

Mit dem abstrakten Datentyp `dist_array` (*distributed array*) wurde ein spezieller Arraytyp als Grundlage für verteilt arbeitende Algorithmen auf *message-passing* Systemen erarbeitet. Der Datentyp wurde als C++ Klasse speziell für PARIX implementiert, läßt sich aber ohne größeren Aufwand an andere *message-passing* Systeme anpassen. Er entspricht in seiner Funktion und Anwendung normalen Arrays. Der Unterschied zu einfachen C++ Arrays oder LEDA Arrays besteht darin, daß die Daten gleichmäßig über alle Prozessoren verteilt sind.
Parallele Algorithmen auf Rechnern mit verteiltem Speicher erfordern im allgemeinen eine Verteilung der Daten auf die Knoten durch explizite Kommunikation. Durch Verwendung des Datentyps `dist_array` erfolgt die Verteilung der Daten automatisch, die Kommunikation bleibt dem Nutzer verborgen.
Ein weiterer Grund für die Verteilung der Daten ist die Ausschöpfung der lokalen Speicherressourcen. Probleme mit größeren Anforderungen an den Speicher können überhaupt erst effizient behandelt werden, wenn die Daten während des gesamten Programmablaufs verteilt gehalten werden.
Der Datentyp `dist_array` wird durch eine C++ Klasse realisiert. Die Elemente eines `dist_array` sollen gleichmäßig auf alle Prozessoren verteilt werden. Bei einem verteilten Array der Größe n verwaltet jeder Prozessor ein Teilarray der Größe n/p. Dadurch, daß ein Array eine feste Größe besitzt, läßt sich die Position eines bestimmten Elements einfach berechnen. Das i-te Element des verteilten Arrays befindet sich auf dem Prozessor

i *div* $\frac{n}{p}$ an der Position i *mod* $\frac{n}{p}$ in dessen Teilarray (*div* steht für ganzzahlige Division, *mod* für den Divisionsrest).
Probleme treten beim Zugriff auf die Elemente auf. In der folgenden Codesequenz werden zwei Variablen deklariert:

```
dist_array<int> A(100);
int             x;
```

Auf jedem Prozessor existiert eine Instanz der Variablen `x`, die auch auf jedem Prozessor einen anderen Wert annehmen kann. `A` ist ein verteiltes Array. Das heißt jeder Knoten enthält $100/p$ Elemente des Arrays. Ein bestimmtes Element, A[10] beispielsweise, existiert nur genau einmal auf dem Parallelrechner. Eine Zuweisung wie `x=A[10]` ist noch relativ unproblematisch. Der Wert von `A[10]` wird an alle Prozessoren verteilt und `x` zugewiesen.
Bei `A[10]=x` tritt das Problem auf, daß p verschiedene Instanzen von `x` mit möglicherweise unterschiedlichen Werten existieren. Die Implementierung legt fest, daß immer der Wert von `x` auf Prozessor 0 zugewiesen wird. Das ist sinnvoll, da Prozessor 0 über eine direkte Verbindung zum Hostrechner verfügt. So ist das Einlesen der Werte eines verteilten Arrays aus einem File oder von Tastatur am effizientesten.
Das folgende Beispielprogramm demonstriert die einfache Anwendung des Datentyps `dist_array`.

```
#include<dist_array.h>
#include<iostream.h>
#include<stdlib.h>

#define N 10

main()
{
 dist_array<int> A(N);
 int i;
 for(i=0; i<N; i++) A[i]=rand();

 A.shellsort();

 for(i=0; i<N; i++) {
  int x=A[i];
  if (PC_MyProcID==0) cout << x << '\n';
 }
}
```

3 Parallele Sortieralgorithmen

Es wurden vier verschiedene parallele Sortieralgorithmen – *Odd-Even Transposition Sort, Shearsort, Shellsort* und *Bitonic Sort* – unter PARIX implementiert und in einer erweiter-

baren Bibliothek zusammengefaßt.
Die Algorithmen unterscheiden sich im wesentlichen durch die verwendete Topologie und ihre Laufzeit. *Odd-Even Transposition Sort* sortiert auf einem linearen Array n Werte in Laufzeit $O(n)$. *Shearsort* arbeitet mit einem zweidimensionalen Gitter. Bei einem quadratischen Gitter beträgt die Laufzeit $O(\sqrt{n}\log\sqrt{n})$. *Shellsort* und *Bitonic Sort* arbeiten mit einer Hypercube-Topologie. *Shellsort* hat eine Laufzeit von $O(\log n+k)$ mit $0 \leq k \leq n$). *Bitonic Sort* benötigt $O(\log^2 n)$ Schritte.
Alle vier Algorithmen setzen direkt auf der Implementierung des Datentyps `dist_array` auf. Es wird von gleichmäßig über alle Prozessoren verteilten Daten ausgegangen. Jeder Prozessor sortiert zunächst sein Teilarray mit einem sequentiellen Sortieralgorithmus. Grundlage der parallelen Algorithmen sind *merge-split* Operationen. Dabei kommunizieren jeweils zwei in der Topologie benachbarte Prozessoren miteinander. Die beiden Felder werden zu einem sortierten Feld gemischt und dann so auf die beiden Prozessoren verteilt, daß der Prozessor mit dem kleineren Index die untere Hälfte erhält.

3.1 Effizienz der parallelen Sortieralgorithmen

Tabelle 1 und Abbildung 1 zeigen die Testergebnisse für die bisher vier in der Bibliothek enthaltenen Sortieralgorithmen. Insbesondere der *Shellsort* Algorithmus erreicht bei allen getesteten Prozessorzahlen einen fast optimalen Speedup.

	Odd-Even		Shearsort		Shellsort		Bitonic	
p	$T_p[s]$	S_p	$T_p[s]$	S_p	$T_p[s]$	S_p	$T_p[s]$	S_p
1	43.3	1.00	43.3	1.00	43.3	1.00	43.3	1.00
2	21.9	1.98	21.9	1.98	21.8	1.98	21.8	1.98
4	11.2	3.88	11.2	3.87	11.1	3.90	11.2	3.86
8	5.9	7.31	5.9	7.34	5.5	7.88	5.9	7.31
16	3.5	12.49	3.4	12.55	2.8	15.40	3.2	13.46
32	2.4	18.13	2.0	21.56	1.5	29.88	1.9	22.33

Tabelle 1: Laufzeit und Speedup der Algorithmen auf dem Xplorer (n=100000)

Die beiden Algorithmen, die über eine Hypercube-Topologie kommunizieren, schneiden in der Praxis am besten ab. Sie benötigen zum Sortieren wesentlich weniger *merge-split* Operationen als die beiden anderen Algorithmen. Die Verwendung der für diesen Parallelrechner nicht optimalen Hypercube-Topologie fällt nicht so stark ins Gewicht. Die Prozessoren des Parsytec Xplorers sind zu einem zweidimensionalen Gitter verschaltet. PARIX ermöglicht jedoch auch die Kommunikation zwischen auf dem physischen Gitter nicht direkt benachbarten Knoten. Das Senden der Nachrichten über Zwischenknoten erfolgt automatisch.
Die Kosten für das Senden einer Nachricht setzen sich zusammen aus der Startup-Zeit t_S und der Zeit für die Übertragung der Nachricht. Wird die Kommunikation über mehrere Stationen explizit programmiert, fällt an jedem Knoten die Startup-Zeit an. Beim Routen der Nachrichten über Zwischenknoten fällt t_S nur einmal an. Bei diesem System ist t_S

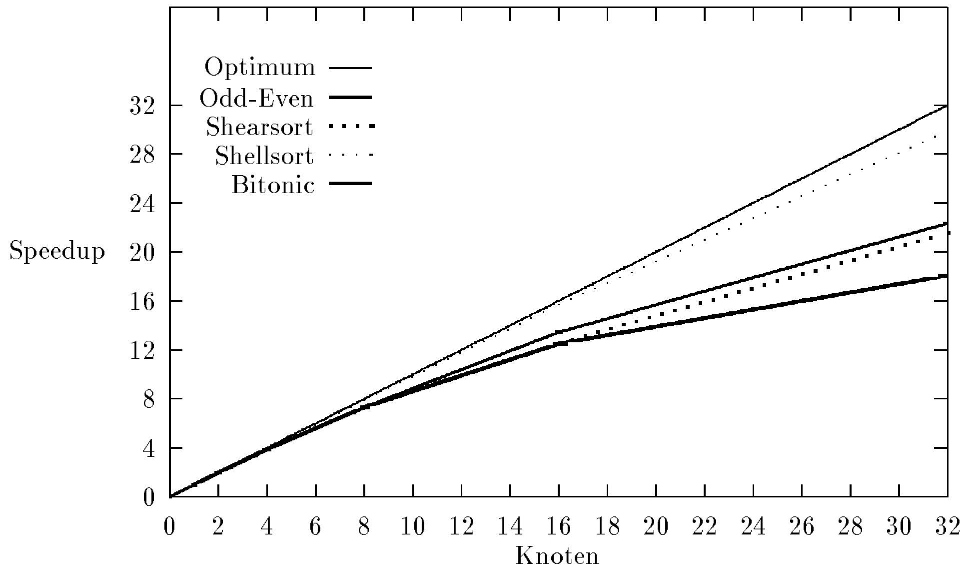

Abbildung 1: Speedupkurven der Algorithmen im Vergleich

relativ groß im Vergleich zur eigentlichen Transferzeit. Dadurch ist es oftmals günstiger, Topologien einzusetzen, die sich nicht optimal auf das Gitter des Xplorers abbilden lassen, wenn dafür der Algorithmus geringere Kommunikationskosten verursacht.
Tabelle 2 zeigt die parallele Laufzeit T_p (in s) und den Speedup S_p (= Verhältnis von sequentieller zu paralleler Laufzeit) für p Prozessoren von *Shellsort* auf dem Xplorer bei verschiedenen Prozessorzahlen und verschiedenen Problemgrößen.

	$n = 100$		$n = 1000$		$n = 10000$		$n = 100000$	
p	$T_p[s]$	S_p	$T_p[s]$	S_p	$T_p[s]$	S_p	$T_p[s]$	S_p
1	0.020	1.00	0.305	1.00	3.74	1.00	43.33	1.00
2	0.011	1.78	0.143	2.13	1.91	1.96	21.80	1.99
4	0.008	2.50	0.067	4.53	0.92	4.07	10.88	3.98
8	0.008	2.51	0.042	7.27	0.46	8.18	5.49	7.89
16	0.012	1.66	0.029	10.40	0.23	15.98	2.77	15.69
32	0.022	0.90	0.036	8.50	0.14	26.71	1.45	29.88

Tabelle 2: Laufzeit und Speedup von Shellsort auf dem Xplorer

Der Algorithmus zeigt bei allen Prozessorzahlen einen sehr guten Speedup, der fast linear mit der Knotenzahl wächst, falls n hinreichend groß ist. Wenn sich auf den einzelnen Knoten zu wenige Elemente befinden, verliert das lokale Sortieren, wo alle Prozessoren ohne Kommunikation parallel arbeiten können, an Gewicht, und der Anteil der Kommunikation an der Gesamtlaufzeit nimmt zu.

4 Parallele Algorithmen zur Berechnung konvexer Hüllen

Ein wichtiges Problem der Algorithmischen Geometrie ist die Berechnung konvexer Hüllen. Die *konvexe Hülle* einer Punktmenge S von n Punkten in der Ebene ist die kleinste konvexe Menge, die S enthält. Schnelle Algorithmen zu deren Berechnung sind z. B. in der Bildverarbeitung und Mustererkennung, insbesondere für Echtzeitanwendungen, gefragt. Hier bieten sich parallele Algorithmen an, wobei sich einige sequentielle Algorithmen gut für eine Parallelisierung eignen. Für den im allgemeinen praktisch relevanten Fall $p \ll n$ sind gute Effizienzwerte erzielbar, da der Kommunikationsanteil dann relativ klein gegenüber dem Berechnungsanteil gehalten werden kann.
Sehr gut eignen sich *divide-and-conquer* Algorithmen zur Parallelisierung. Dabei wird zunächst für n/p Punkte je Prozessor (o. B. d. A. sei $n = kp$, $k \in N$) ein sequentieller Algorithmus eingesetzt. Mit Mischalgorithmen (z. B. [Sed93]) wird die gesuchte Gesamthülle in $O(\log p)$ Schritten erzielt. Die beste sequentielle Laufzeit für die Berechnung konvexer Hüllen ist (wie die für Sortieren) $O(n \log n)$. Das Mischen erfordert im *worst case* $O(n)$ Berechnungen. Parallel ist damit eine Laufzeit von $T_p = O(\frac{n}{p} \log \frac{n}{p}) + O(n \log p)$ für den Berechnungsaufwand erreichbar.
Nicht vernachlässigt werden können die Kommunikationskosten, die z. B. nach [KGGK94] für ein einfaches Modell zum Nachrichtenaustausch auf *message-passing* Systemen mit $t_c = t_S + mt_W$ angesetzt werden können. Die Startup-Zeit t_S und die Zeit t_W für den Transport eines Worts sind Systemkonstanten, m ist die Nachrichtenlänge in Worten. Die Gesamtkommunikationskosten sind in der Regel von der Netzwerktopologie abhängig. Für einen Hypercube z. B. erhält man hier $t_c = O(n)$ wegen des letzten (sequentiellen) Schrittes beim Mischen, also eine Gesamtlaufzeit von $O(\frac{n}{p} \log \frac{n}{p} + n \log p)$. Ein solcher Algorithmus ist kostenoptimal, denn es gilt $pT_p = O(n \log n) = T_1$ bei $p \ll n$.
In [AL93] wird ein *multiway divide-and-conquer* Algorithmus für eine PRAM mit $p = O(n)$ vorgestellt, der ein verbessertes paralleles Mischen der Teilhüllen ermöglicht. Alle Prozessoren sind in einem parallelen Schritt mit Laufzeit $O(n)$ an der Berechnung der Gesamthülle beteiligt. Diese Idee läßt sich auf den Fall $n = kp$ übertragen, wodurch die Zeit für das Mischen bei idealer Prozessorenauslastung reduziert werden kann.
Mehrere Algorithmen wurden auf verschiedenen Plattformen unter PVM in C++ implementiert. Die Tests zeigen für hinreichend großes n gute Effizienz.

4.1 Effizienz

Für erste Untersuchungen zu parallelen Verfahren zur Ermittlung konvexer Hüllen wurden folgende Algorithmen mit PVM [Oak94] in C++ implementiert und getestet. In den Laufzeiten ist der Kommunikationsaufwand t_c, wie oben angegeben, enthalten:

- eine Anpassung des Algorithmus von Leighton [Lei92] für $p \ll n$ (Leightons Verfahren arbeitet mit $p = n$ auf einer Pipeline-Topologie); Laufzeit $O(\frac{n^2}{p})$

- eine Parallelisierung des Einwickelverfahrens nach Jarvis (in [PS88]); Laufzeit $O(m\frac{n}{p})$, wobei m die Anzahl der Hüllecken ist

- ein einfaches paralleles *divide-and-conquer* Verfahren, in dem die sequentiell ermittelten Teilhüllen durch Mischen [Sed93] zur Gesamthülle kombiniert werden; Laufzeit $O(\frac{n}{p} log \frac{n}{p}) + logp\ O(n)$
- eine Parallelisierung des Algorithmus von Narayanaswami [Nar91], in dem in einer Vorstufe innere Punkte eliminiert werden; Laufzeit abhängig von Vorstufenparameter und Anzahl erforderlicher Iterationen

Als Eingabemengen wurden zufällig Punkte verteilt, z. B. in Quadrat, Kreis, Kreisring und Dreieck. Die Effizienzuntersuchungen wurden zunächst unter dem Aspekt ausgeführt, die prinzipielle Eignung einzelner Algorithmen für eine parallele Abarbeitung auf einem *message-passing* System, hier mit PVM, festzustellen. Einige typische Testergebnisse für verschiedene Eingabemengen, erzielt auf einem Netz von Sparc-Workstations, zeigen die Tabellen 3 bis 6.

	p = 1	p = 2		p = 4		p = 6	
	Zeit (s)	S_p	E_p	S_p	E_p	S_p	E_p
Quadrat	74.6	1.4	0.7	2.3	0.6	3.1	0.5
Kreis	70.3	1.4	0.7	2.4	0.6	3.2	0.5
Kreisring	69.7	1.4	0.7	2.3	0.6	3.1	0.5
Dreieck	107.6	1.4	0.7	2.4	0.6	3.3	0.5

Tabelle 3: Laufzeit, Speedup und Effizienz für Leighton-Algorithmus, 10 000 Punkte

Die mit Abstand schlechtesten Laufzeiten des Leighton-Algorithmus sind dem Umstand geschuldet, daß er für eine Pipeline mit $p = n$ entwickelt und hier für $p \ll n$ mit *message-passing* implementiert wurde.

	p = 1	p = 2		p = 4		p = 7	
	Zeit (s)	S_p	E_p	S_p	E_p	S_p	E_p
Quadrat	5.94	1.9	0.95	3.1	0.8	4.2	0.6
Kreis	39.25	1.7	0.85	3.0	0.75	4.3	0.6
Kreisring	56.95	1.7	0.85	2.9	0.7	4.5	0.6
Dreieck	5.13	1.9	0.95	2.9	0.7	3.9	0.6

Tabelle 4: Laufzeit, Speedup und Effizienz für Einwickelverfahren, 100 000 Punkte

Entsprechende Tests auf einer Convex SPP1600 mit PVM zeigten prinzipiell ähnliche Ergebnisse. Einerseits ist ersichtlich, daß alle bisher untersuchten Algorithmen im Vergleich zu ihren sequentiellen Ausgangsverfahren ein gutes paralleles Laufzeitverhalten zeigen. Andererseits fallen aber sofort die sehr unterschiedlichen sequentiellen Zeiten auf. Ein Vergleich der parallelen Verfahren mit dem hier besten sequentiellen zeigt die Überlegenheit des *divide-and-conquer* Algorithmus. Dieser Umstand erweist sich mit Blick auf

	p = 1	p = 2		p = 4		p = 7	
	Zeit (s)	S_p	E_p	S_p	E_p	S_p	E_p
Quadrat	0.42	1.75	0.9	3.2	0.8	5.25	0.75
Kreis	1.05	1.75	0.9	2.8	0.7	5.0	0.7
Kreisring	1.79	1.9	0.95	3.3	0.8	5.95	0.85
Dreieck	0.57	1.95	0.95	3.8	0.95	6.3	0.9

Tabelle 5: Laufzeit, Speedup und Effizienz für *divide-and-conquer*, 100 000 Punkte

	p = 1	p = 2		p = 4		p = 7	
	Zeit (s)	S_p	E_p	S_p	E_p	S_p	E_p
Quadrat	1.32	1.9	0.95	3.1	0.8	4.4	0.6
Kreis	1.57	1.95	0.98	3.6	0.9	5.1	0.7
Kreisring	2.88	1.9	0.95	3.8	0.95	6.0	0.85
Dreieck	1.33	1.95	0.98	3.6	0.9	5.8	0.8

Tabelle 6: Laufzeit, Speedup und Effizienz für Narayanaswami-Algorithmus, $(30,30)$-Gitter in Vorstufe, 100 000 Punkte

das Gesamtziel als sehr günstig, da die schnellsten sequentiellen Algorithmen aus einer Bibliothek (z. B. LEDA) für die erste Phase sofort genutzt werden können. Für die zweite Phase sind dann lediglich Kombinations- bzw. Misch-Algorithmen erforderlich, die gegebenenfalls sequentiell schon vorliegen.
Das Laufzeitverhalten der parallelen Verfahren wurde bei Arbeit mit verteilten Daten ermittelt (s. 4.2). Das ist für viele praktische Fälle relevant, bei denen die Ermittlung konvexer Hüllen einen Zwischenschritt bildet. Wird jedoch das Ein- und Auslesen der Punktmenge bzw. der Hüllpunkte in die Laufzeit eingeschlossen, sinkt die Effizienz gerade auf *message-passing* Systemen ab.

4.2 Verteilter Datentyp

Für effiziente parallele Berechnungen mit großen Datenmengen auf *message-passing* Systemen bzw. Systemen mit verteiltem Speicher sollten die Daten nicht nur verteilt bearbeitet sondern auch möglichst verteilt gehalten werden, um den Transfer zu minimieren. Um die Implementation zu erleichtern und sicherer zu ermöglichen, empfiehlt sich die Arbeit mit verteilten Feldern, wie in 2 dargestellt. Für die Nutzung der Algorithmen als Bibliotheksfunktionen war die Implementierung eines Datentyps *Verteiltes Feld* mit den entsprechenden PVM-Funktionen erforderlich. Dazu wurde eine einfache Klasse `ms_point_list` implementiert, die das verteilte Halten der Daten in Listen auf Basis der

LEDA-Liste ermöglicht:

```
class ms_point_list{

void absende(int tid, list<point>& pts, int zl, int dat){
// an Task tid zl Punkte aus pst mit Kodierung dat versenden
 int a;
 int info;
 double xw,yw;
 point t;

 pvm_initsend(dat);
 for (a=0;a<zl;a++){
  t=pts.pop();  // Punkt aus der Eingabeliste holen
  xw=t.xcoord();yw=t.ycoord();
  pvm_pkdouble(&xw,1,1); pvm_pkdouble(&yw,1,1);
 }
 info=pvm_send(tid,1800);
}
```

Die Funktion `absende` dient zum Verschicken von `zl` Punkten aus einer Liste `pts` an die Task mit der Nummer `tid`; sie nutzt die entsprechenden PVM-Funktionen und die LEDA-Liste.

5 Ausblick

Die bisherigen Untersuchungen waren ein erster erfolgreicher Test, um eine Softwarebibliothek mit effizienten *parallelen* geometrischen Algorithmen aufzubauen und durch eine spezielle *verteilte* Datenstruktur eine einfache Anwendung zu ermöglichen. Die explizite Kommunikation, die auf Parallelsystemen mit verteiltem Speicher erforderlich ist, wird mit dieser Datenstruktur vor dem Anwender verborgen.
Weitere parallele Algorithmen und Datenstrukturen zur Lösung kombinatorischer und geometrischer Probleme (Schnitte, Sichtbarkeit, Delauny-Triangulierung, Voronoi-Diagramme, Distanzprobleme) sollen entwickelt bzw. für diese Bibliothek(en) implementiert werden. Entsprechende Entwicklungen für MPI sollen sich anschließen.
Die Bibliotheken sind mit entsprechenden Anpassungen auf andere Parallelsysteme mit verteiltem Speicher übertragbar. Um die Portierung auf andere Plattformen zu erleichtern, also den Austausch der jeweiligen Kommunikations- und Synchronisationsfunktionen maschinell zu unterstützen, ist die Entwicklung entsprechender Werkzeuge vorgesehen. Zunächst werden zwei prinzipielle Wege untersucht. Eine Anwendung wird hinsichtlich der Kommunikations- und Synchronisationsfunktionen mit einem Pseudocode entwickelt, der in einer Vorstufe in die entsprechenden Funktionen der gewählten *message-passing* Umgebung übersetzt wird. Oder es werden die Quellen von einer parallelen Entwicklungsumgebung in eine andere übersetzt. Die erste Vorgehensweise bietet

sich für Neuentwicklungen an, beide sind jedoch problematisch wegen des unterschiedlichen Funktionsumfangs paralleler Entwicklungsumgebungen.

Literatur

[AL93] Selim G. Akl and Kelly A. Lyons. *Parallel Computational Geometry.* Prentice Hall, Englewood Cliffs, 1993.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of parallel algorithms.* Benjamin/Cummings, Redwood City, 1994.

[Lei92] Frank T. Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes.* Morgan Kaufmann, San Mateo, 1992.

[MN95] Kurt Mehlhorn and Stefan Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96 – 102, 1995.

[Nar91] C. Narayanaswami. A practical parallel convex hull algorithm. In H. P. Santo, editor, *Proc. First Intl. Conf. on Computational Graphics and Visualization Techniques (Compugraphics '91)*, volume II, pages 444–453, Sesimbra, Portugal, 1991.

[Oak94] Oak Ridge National Laboratory. *PVM 3 Users's Guide and Reference Manual*, 1994.

[Par93] Parsytec GmbH, Aachen. *PARIX Release 1.2 Software Documentation*, 1993.

[PS88] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction.* Springer, New York, 1988.

[Sed93] Robert Sedgewick. *Algorithmen in C.* Addison-Wesley, Bonn, 1993.