# Automatische Verteilung in Pangaea

André Spiegel

Freie Universität Berlin
Institut für Informatik, Takustraße 9, D-14195 Berlin
spiegel@inf.fu-berlin.de

**Zusammenfassung** Pangaea ist ein System, das zentralisierte Java-Programme automatisch verteilen kann, basierend auf statischer Quelltext-Analyse und unter Verwendung beliebiger Verteilungsplattformen wie RMI oder CORBA. Pangaea reduziert die Komplexität verteilter Programmierung, indem es die Idee der Verteilungstransparenz konsequent weiterführt: Sowohl die Entscheidung für eine bestimmte Verteilungsstrategie, als auch deren programmtechnische Umsetzung geschieht in Pangaea nicht nur transparent, sondern automatisch. Der Einsatz statischer Analyse zahlt sich insbesondere dadurch aus, daß er Optimierungen erlaubt, die für eine rein laufzeit-basierte Automatik unmöglich sind.

## 1 Einführung

*Pangaea*[1] ist ein System, das zentralisierte Java-Programme automatisch verteilen kann. Basierend auf statischer Quelltext-Analyse trifft Pangaea zunächst eine *abstrakte Entscheidung*, wie ein gegebenes Programm verteilt werden sollte, um bestimmten Rahmenbedingungen und Optimierungskriterien zu genügen. Die so gewonnene Verteilungsstrategie gibt an, welche Objekte auf welchem Rechner liegen sollen, wann und wie Objektmigration einzusetzen ist, etc. Pangaea *realisiert* diese Strategie dann, indem es den Programmcode für eine gegebene Verteilungsplattform transformiert, unter möglichst guter Ausnutzung der Fähigkeiten und Merkmale dieser Plattform. Pangaea kann demnach als ein *verteilender Übersetzer* betrachtet werden: die Quellsprache ist reines Java, ohne Einschränkungen oder zusätzliche Konstrukte, die Zielsprache ist die verteilte Java-Variante der zu verwendenden Verteilungsplattform.

Mögliche Anwendungsgebiete von Pangaea liegen dort, wo Programme verteilt werden sollen, die als zentralisierte Anwendungen geschrieben wurden. Ein Beispiel sind umfangreiche Web-Applets, die man in einen Client- und einen Server-Teil zerlegen kann, um sie auf kleinen Endgeräten wie PDAs auszuführen. Ein anderer Anwendungsbereich ist paralleles Rechnen: Pangaea erlaubt es dem Programmierer, einen nebenläufigen Algorithmus unter Verwendung von Threads als zentralisiertes Programm zu formulieren, ohne Rücksicht auf den Verteilungsaspekt, um den sich Pangaea nach Fertigstellung des Programms automatisch kümmert.

Pangaea befindet sich derzeit in der Implementierungsphase. In diesem Papier geben wir einen Überblick über das System und skizzieren einige der Ergebnisse, die wir bisher erzielt haben.

[1] *Pangaea* ist der Name des Urkontinents, in dem bis vor etwa 200 Millionen Jahren die gesamte Landmasse der Erde zentralisiert war [10]. Durch die Kontinentalverschiebung entstand aus Pangaea dann die verteilte Welt, wie wir sie heute kennen.

## 2 Verteilung und statische Analyse

Pangaea ist nicht an eine bestimmte Verteilungsplattform gebunden, sondern kann die Fähigkeiten und Merkmale beliebiger Plattformen ausnutzen. Um den Nutzen statischer Verteilungsanalyse einzuschätzen, ist es darum sinnvoll, ein ideales Verteilungsmodell zugrundezulegen, das die heute verfügbare Technik in die Zukunft extrapoliert, auch wenn es von gegenwärtigen Plattformen wie RMI oder CORBA erst in Teilen realisiert wird. Bei der Verteilung eines konkreten Programms auf einer konkreten Zielplattform entscheidet Pangaea von Fall zu Fall, welche Fähigkeiten die Plattform tatsächlich besitzt und wie sie sich einsetzen lassen.

### 2.1 Ein ideales Verteilungsmodell

Wir nennen ein Programm zentralisiert, wenn sich alle seine Laufzeitobjekte im selben Adreßraum befinden. Das Programm zu verteilen, bedeutet für uns, diese Objekte auf eine Menge von lose gekoppelten Rechnern zu plazieren. Die Verteilung hat dabei keinerlei Auswirkungen auf die Ausführungslogik des Programms: ein sequentiell formulierter Algorithmus läuft auch nach der Verteilung sequentiell, während ein nebenläufig, mithilfe von Threads programmierter Algorithmus, der im zentralisierten Fall in der Regel durch *time slicing* ausgeführt wird, nach der Verteilung jedoch echt parallel ablaufen kann. Interaktive, Client/Server-artige Anwendungen sind oft rein sequentielle Programme; man verteilt sie nicht, um parallele Ausführung zu erreichen, sondern um sie in inhärent verteilten Umgebungen, zum Beispiel dem Internet, einzusetzen.

Die Verteilungsplattform erlaubt es den Objekten, über Rechnergrenzen hinweg miteinander zu kommunizieren, und zwar durch entfernten Methodenaufruf oder auch entfernten Feldzugriff. Ohne Beschränkung der Allgemeinheit nehmen wir außerdem an, daß Objekte entfernt erzeugt werden können; auch auf Plattformen wie CORBA [4] oder RMI [9], wo es keine ausdrückliche Fernerzeugung gibt, läßt sie sich leicht simulieren.

Für eine effiziente Verteilung nicht-trivialer Programme sind außerdem Mobilitätsmechanismen unabdingbar, d.h. Mechanismen zur Migration, Replikation, oder zum Caching von Objekten. Zwei prinzipiell verschiedene Arten solcher Mechanismen lassen sich unterscheiden. *Synchrone Mechanismen* sind solche, die an den Kontrollfluß des Programms gebunden sind, d.h. immer dann, wenn die Ausführung eine bestimmte Stelle des Codes erreicht, wird eine entsprechende Veränderung der Objektplazierung vorgenommen. Beispiele dafür sind explizite Migrationsanweisungen im Code (wie in JavaParty [6]) oder auch strukturiertere Techniken, etwa Wertübergabe von Objekten (*objects-by-value* [7]) oder das aus Emerald bekannte *pass-by-move* und *pass-by-visit* [3]. Ein *asynchroner Mechanismus* besteht dagegen aus einer Instanz im Laufzeitsystem, die die Interaktionen zwischen den Objekten protokolliert und gegebenenfalls, asynchron, die Verteilung so anpaßt, daß z.B. die Zahl der entfernten Interaktionen minimiert wird. Wenige Java-basierte Plattformen enthalten bisher einen solchen Mechanismus, zum Beispiel aber das FarGo-System [2].

### 2.2 Die Bedeutung statischer Analyse

Verteilungen automatisch, durch statische Analyse zu finden, zahlt sich in mehrfacher Hinsicht aus. Zum einen entlastet es den Programmierer von einer Routinearbeit, die zunächst einfach erscheinen mag (und es darum verdient, automatisiert zu werden); bei näherem Hinsehen zeigt sich jedoch, daß es Optimierungen gibt, die der Programmierer durchaus übersehen könnte, und die im Detail oft mühsam durchzuführen sind. Die Informationen, die solche Optimierungen erlauben, sind andererseits auf keinem anderen Weg als durch statische Analyse zu gewinnen: Auch für eine ideale Verteilungsplattform, die Objekte zur Laufzeit automatisch und transparent plazieren kann, wäre statische Analyse daher sinnvoll, wenn nicht sogar unverzichtbar. Beispiele für solche nur statisch zu gewinnenden Informationen, und die zugehörigen Optimierungen, sind:

- die Erkennung *konstanter Objekte* (immutable objects); diese können frei repliziert werden, müssen nicht fernaufrufbar sein, und brauchen nicht vom Laufzeitsystem überwacht zu werden,
- die Bestimmung des *dynamischen Gültigkeitsbereichs* von Objektreferenzen; wobei sich zum Beispiel zeigen kann, daß bestimmte Objekte nur privat, innerhalb anderer Objekte oder Subsysteme benutzt werden, daher nicht fernaufrufbar sein müssen und ebenfalls für die Plazierungsentscheidungen des Laufzeitsystems irrelevant sind,
- die Erkennung von Möglichkeiten zur *synchronen Objektmigration*, was asynchroner Migration durch das Laufzeitsystem vorzuziehen ist, denn asynchrone Entscheidungen sind teuer und können erst getroffen werden, *nachdem* eine schlechte Objektverteilung sich bereits für einige Zeit manifestiert hat.

Auf der anderen Seite gilt natürlich, daß statische Analyse immer nur eine Annäherung des tatsächlichen Laufzeitverhaltens liefern kann. Das Ziel muß darum sein, die statische Analyse möglichst gut mit dem Laufzeitsystem zusammenarbeiten zu lassen – diejenigen Entscheidungen, die statisch getroffen werden *können*, soll der Algorithmus erkennen und dem Laufzeitsystem abnehmen; in anderen Fällen wird dynamisch entschieden werden müssen.

## 3 Verwandte Arbeiten

Unseres Wissens gibt es zwei andere Projekte, in denen statische Analyse mit dem Ziel automatischer Verteilung eingesetzt wurde oder wird: eines auf der Basis der Sprache *Orca* [1], das andere unter Verwendung der *JavaParty* Plattform [5]. Beide Projekte konzentrieren sich auf parallele Programmierung, also die automatische Verteilung nebenläufiger, meist numerischer Algorithmen, während das Pangaea-System auch auf interaktive, Client/Server-artige Anwendungen zielt.

Das *Orca*-Projekt konnte erfolgreich zeigen, daß statische Analyse dem Laufzeitsystem helfen kann, bessere Plazierungs- und Replikationsentscheidungen zu treffen; die Effizienz kommt sehr nahe an diejenige von manuell verteilten Programmen heran. Während Orca eine objekt-*basierte* Sprache ist, deren Definition gerade im Hinblick auf

mögliche statische Analyse einfach gehalten wurde, versuchen wir in Pangaea, ähnliche Ergebnisse in einer weit verbreiteten, objekt-*orientierten* Sprache wie Java zu erzielen.

Wir betrachten das *JavaParty*-Projekt als einen ersten Schritt in dieser Richtung. Der dort benutzte Analyse-Algorithmus konnte jedoch, wie die Autoren einräumen, für reale Programme noch keine überzeugenden Ergebnisse liefern. Wir glauben, einige der dafür verantwortlichen Probleme mit unserem Ansatz besser lösen zu können; eine ausführliche Diskussion findet sich in [8].

Was unsere Arbeit darüber hinaus von beiden genannten Projekten unterscheidet, ist, daß Pangaea noch andere Verteilungskonzepte behandelt außer reinen Plazierungs- und Replikationsentscheidungen. Pangaea ist außerdem nicht an eine bestimmte Verteilungsplattform gebunden, sondern bietet einen Abstraktionsmechanismus, der die Fähigkeiten verschiedenster existierender Plattformen ausnutzen kann und speziell so entworfen wurde, daß sich Pangaea auch an zukünftige Technik anpassen läßt.

## 4 Pangaea

Die Architektur von Pangaea ist in Abb. 1 dargestellt. Wir beschreiben das System zunächst im Überblick, um dann in den folgenden Abschnitten auf einzelne Bereiche genauer einzugehen.

Pangaeas Eingabe ist der Quelltext eines zentralisierten Java-Programms. Der *Analyzer* leitet daraus einen *Objektgraph* ab, der eine Annäherung der Laufzeitstruktur des Programms darstellt: Er beschreibt, welche Objekte es zur Laufzeit geben wird und wie sie miteinander kommunizieren (Einzelheiten dazu in Abschnitt 4.1). Der Analyzer entscheidet über die Verteilung des Programms durch Analyse dieses Objektgraphen. Vorgaben und Rahmenbedingungen dazu erhält er einerseits vom Programmierer, andererseits vom Plattform-Adapter für die zu verwendende Verteilungsplattform.

Der Programmierer legt die Rahmenbedingungen der gewünschten Verteilung fest, indem er in einer visuellen Darstellung des Objektgraphen einige wenige Objekte fest bestimmten Rechnern zuordnet[2]. In einer Client/Server-artigen Datenbank-Anwendung würde man zum Beispiel die Objekte der Benutzeroberfläche dem Client zuordnen; die Objekte, die Datenbankzugriffe durchführen, dem Server. Unter Maßgabe dieser Rahmenbedingungen vervollständigt der Analyzer die Verteilung, indem er etwa die übrigen Objekte so plaziert, daß sich möglichst wenig Kommunikation über die Verteilungsgrenze hinweg ergibt, d.h. er führt eine Graphpartitionierung durch. (Für nebenläufige Programme, bei denen es auf Lastverteilung ankommt, gelten etwas andere Kriterien, auf die wir hier aus Platzgründen nicht eingehen können.)

Der Analyzer berücksichtigt bei der Verteilung außerdem die Fähigkeiten der zu verwendenden Verteilungsplattform. Eine abstrakte Sicht dieser Fähigkeiten vermittelt der entsprechende *Plattform-Adapter*; er teilt dem Analyzer zum Beispiel mit, ob die Plattform über Objektmigration oder -replikation verfügt, oder ob sich eine bestimmte Java-Klasse mit dieser Plattform fernaufrufbar machen läßt oder nicht.

[2] Man könnte argumentieren, das Verfahren wäre wegen der Vorgaben durch den Programmierer bestenfalls semi-automatisch. Jede Automatik ist aber auf Eingaben angewiesen, und nichts anderes sind die anfänglichen Festlegungen des Programmierers in unserem System.

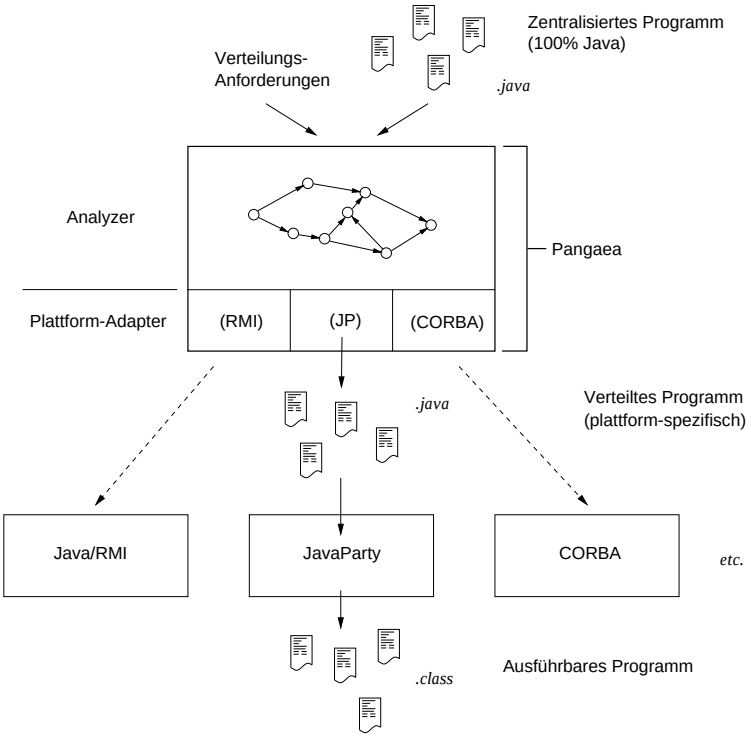


**Abb. 1.** Die Architektur von Pangaea

Nach Abschluß der Analyse bekommt der Plattform-Adapter vom Analyzer annotierte Versionen der Syntaxbäume des unverteilten Programms. Die Annotationen geben zum Beispiel an, welche Klassen fernaufrufbar sein müssen, welche serialisierbar, und welche `new`-Anweisungen zu Objekt-Fernerzeugungen werden sollen. Der Adapter generiert daraufhin den Quelltext des Programms neu, so daß ein verteiltes Programm für die gewählte Plattform entsteht; dazu können auch automatisch generierte Schnittstellenbeschreibungen oder Konfigurationsdateien gehören (für Einzelheiten siehe Abschnitt 4.2). Die Verteilungsplattform ist schließlich dafür verantwortlich, aus dem verteilten Code ein ausführbares Programm zu machen (was etwa Vertretergenerierung und Übersetzung einschließt), und es unter der Steuerung des Laufzeitsystems auszuführen.

### 4.1 Vom Quelltext zum Objektgraph

Der Algorithmus, der aus dem Quelltext einen Objektgraphen ableitet, ist derjenige Teil von Pangaea, von dem alle übrigen Analysen entscheidend abhängen. Unser Algorithmus unterscheidet sich von vielen anderen Ansätzen zur statischen Analyse dadurch, daß er auf der Ebene einzelner Objekte arbeitet, nicht nur der Typen dieser Objekte.

Letzteres ist zwar für gängige Optimierungen in Übersetzern, etwa die statische Bindung polymorpher Aufrufe, meist ausreichend, nicht jedoch für die Verteilung von Programmen. Wir haben den Algorithmus an anderer Stelle im Detail beschrieben [8] und müssen uns im folgenden auf einen knappen Überblick beschränken.

Das Ergebnis unseres Algorithmus' ist ein Graph, dessen Knoten die Laufzeitobjekte des Programms repräsentieren. Zwischen den Knoten gibt es drei Arten von Kanten, nämlich Erzeugungskanten, Referenzkanten und Benutzungskanten. (Wir sagen, daß ein Objekt $a$ ein Objekt $b$ *benutzt*, wenn $a$ Methoden von $b$ aufruft oder auf Felder von $b$ zugreift.) Der Graph *approximiert* die tatsächliche Laufzeitstruktur wie folgt:

- Manche der Knoten im Graphen stehen nicht für ein einzelnes, konkretes Laufzeitobjekt, sondern für eine unbestimmte Anzahl von Objekten eines bestimmten Typs. Wir nennen solche Knoten *indefinite Objekte*. Für jeden Typ des Programms kann es im Graphen mehrere konkrete oder indefinite Objekte geben; ein indefinites Objekt steht also nicht einfach für *alle* Instanzen eines bestimmten Typs (wodurch die Analyse zu einer typ-basierten Analyse degenerieren würde), sondern für eine bestimmte Teilmenge dieser Instanzen.
- Referenz- und Benutzungskanten sind *konservativ*, d.h. der Graph enthält eventuell mehr Kanten, als es der tatsächlichen Laufzeitstruktur entsprechen würde, aber niemals weniger. Die *Abwesenheit* einer Kante ist also eine sichere Information, nicht umgekehrt.
- Der Algorithmus behandelt Objekte – zumindest im endgültigen Graphen – als unstrukturierte Behälter von Referenzen, abstrahiert also von ihren internen Details. Wir sagen daß ein Objekt $a$ eine Referenz auf ein anderes Objekt $b$ besitzt, wenn diese Referenz irgendwann zur Laufzeit im Kontext von $a$ erscheinen kann, gleichgültig ob in einer Instanzvariable, als temporärer Wert eines Ausdrucks, etc.

Der Objektgraph wird in fünf Schritten aufgebaut:

**Schritt 1.** Bestimme die *Menge der Typen*, aus denen das Programm besteht. Sie ergibt sich aus der transitiven Abhängigkeitshülle der `main`-Klasse des Programms, d.h. sie enthält alle Typen, die in dem Programm syntaktisch referenziert werden.

**Schritt 2.** Erzeuge einen *Typgraph*, der Benutzungsbeziehungen und Datenflußbeziehungen auf der Typebene beschreibt. Eine Benutzungskante zwischen zwei Typen $A$ und $B$ bedeutet, daß Objekte des Typs $A$ Objekte des Typs $B$ zur Laufzeit benutzen können; eine Datenflußkante gibt an, daß Referenzen eines Typs $C$ von Objekten eines Typs $A$ zu Objekten eines Typs $B$ propagiert werden können, zum Beispiel als Parameter eines Methodenaufrufs. Der Typgraph wird durch einfache syntaktische Analyse gewonnen, wobei Beziehungen zwischen Typen grundsätzlich auch für Objekte beliebiger Subtypen der beteiligten Typen gelten.

**Schritt 3.** Bestimme die *Objektpopulation* des Programms, d.h. eine endliche Repräsentation der potentiell unbeschränkten Menge der zur Laufzeit existierenden Objekte. Der Kerngedanke ist hier, die `new`-Anweisungen des Programms in *initiale* und *nicht-initiale* Allokationen einzuteilen: Eine initiale Allokation ist eine `new`-Anweisung, die sicher genau einmal ausgeführt wird, wann immer der Typ, in dem sie erscheint, instantiiert wird (zum Beispiel weil die Anweisung in einem Konstruktor steht).

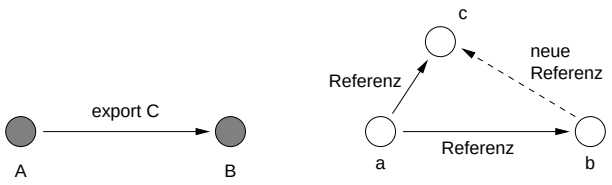


**Abb. 2.** Export einer Referenz im Typgraph (links) und im Objektgraph (rechts)

Wir betrachten die statischen Methoden und Felder der Klassen des Programms als *statische Objekte*, die automatisch erzeugt werden und in jedem Fall Teil der Objektpopulation sind. Die initialen Allokationen, die von diesen statischen Objekten ausgeführt werden, ergeben transitiv die Menge der *initial erzeugten Objekte* des Programms, deren Existenz der Algorithmus also sicher voraussagen kann. Bei den nicht-initialen Allokationen ist hingegen nicht sicher, wie oft, wenn überhaupt, sie zur Laufzeit ausgeführt werden. Für jede nicht-initiale Allokation, die in dem Typ eines Objekts auftaucht, werden daher – transitiv – indefinite Objekte zur Population hinzugefügt. Wir erhalten so die Knoten des Objektgraphen sowie die Erzeugungskanten und, da Erzeugung in der Regel Referenz impliziert, erste Referenzkanten.

**Schritt 4.** Propagiere die Referenzkanten im Objektgraphen, basierend auf der Datenflußinformation aus dem Typgraphen. Vergleiche hierzu Abb. 2: Enthält der Objektgraph beispielsweise ein Objekt $a$, das Referenzen auf zwei Objekte $b$ und $c$ besitzt, und gibt der Typgraph an, daß der entsprechende Typ $A$ Referenzen des Typs $C$ an den Typ $B$ exportiert, dann wird eine neue Referenzkante von $b$ zu $c$ in den Objektgraph eingefügt. Dieser Vorgang wird im gesamten Graphen solange wiederholt, bis ein Fixpunkt erreicht ist. So ergibt sich, welche Objekte welche anderen Objekte „kennen“ und also auch benutzen können.

**Schritt 5.** Füge Benutzungskanten in den Objektgraphen ein, und zwar überall dort, wo ein Objekt $a$ ein Objekt $b$ „kennt“ und laut Typgraph eine Benutzungsbeziehung zwischen den beiden zugehörigen Typen besteht.

Der Algorithmus ist insgesamt von polynomieller Komplexität und damit auch für größere Programme geeignet. Wir haben ihn vollständig implementiert und auf eine Reihe nicht-trivialer Beispiele mit bis zu 10.000 Zeilen Code angewendet. Die Ergebnisse sind vielversprechend: Nach Rechenzeiten, die höchstens im Minutenbereich liegen, läßt sich bei allen bisher betrachteten Programmen die Laufzeitstruktur gut anhand des Objektgraphen erkennen, und die Detailauflösung ist hinreichend zum Verteilen der Programme (siehe auch die Fallstudie in Abschnitt 5).

### 4.2 Die Plattform-Adapter

Pangaeas Plattform-Adapter befinden sich noch im Planungs- und Experimentierstadium. Ihre Aufgabe ist zum einen, dem Analyzer in abstrakter Weise mitzuteilen, welche Fähigkeiten die jeweilige Plattform besitzt; sie verbergen andererseits das Detailwissen, wie ein bestimmtes Verteilungskonzept programmtechnisch auf der Plattform realisiert

wird. Wir beschränken uns im folgenden auf den einfachsten Fall einer Plattform, die lediglich Fernaufrufe und Fernerzeugungen von Objekten erlaubt, lassen also etwaige Mobilitätsmechanismen außer acht.

Während der Analysephase hat der Adapter die Aufgabe, dem Analyzer mitzuteilen, ob bestimmte Klassen mit Hilfe dieser Plattform fernaufrufbar gemacht werden können oder nicht. Falls die Plattform bespielsweise keine entfernten Zugriffe auf Instanzvariablen erlaubt, muß der Adapter überprüfen, ob die gegebene Klasse öffentliche Instanzvariablen besitzt oder nicht. Der Analyzer berücksichtigt diese Information beim Festlegen der Verteilungsgrenze.

Nach abgeschlossener Analyse generiert der Adapter den Quelltext des Programms neu, wobei ihm der Analyzer durch Annotationen vorgibt, welche Klassen fernaufrufbar sein müssen und welche `new`-Anweisungen zu Fernerzeugungen werden sollen.

Eine *Fernerzeugung* ist, konzeptionell gesehen, eine `new`-Anweisung mit einem zusätzlichen Parameter, der angibt, auf welchem entfernten Rechner das Objekt alloziert werden soll. Programmtechnisch kann dies durch eine vorherige Anweisung an das Laufzeitsystem realisiert werden (so etwa in JavaParty), oder durch einen Aufruf eines Factory-Objekts auf dem gewünschten Rechner (wie in CORBA). Die entsprechende Code-Transformation ist in beiden Fällen trivial.

Eine Objektklasse *fernaufrufbar* zu machen ist hingegen auf verschiedenen Plattformen mit sehr unterschiedlichem Aufwand verbunden. In JavaParty genügt es, ein einzelnes Schlüsselwort (`remote`) in die Klassendefinition einzufügen, während bei CORBA eine IDL-Beschreibung generiert werden muß und je nach Implementierung einige Eingriffe in die Vererbungshierarchie erforderlich sind. Der Plattform-Adapter hat darüber hinaus zwei wichtige Aufgaben:

1. Er kann durch Quelltext-Transformation Fähigkeiten simulieren, die die Plattform für sich nicht aufweist. Falls zum Beispiel entfernte Variablenzugriffe nicht möglich sind, kann der Adapter entsprechende Zugriffsmethoden generieren, die dann entfernt aufgerufen werden können.
2. Er muß sicherstellen, daß die fernaufrufbare Version der Klasse dieselbe Semantik besitzt wie die ursprüngliche Version. Dies ist nicht trivial, da alle uns bekannten Verteilungsplattformen geringfügige Änderungen der Semantik in Kauf nehmen. Zum Beispiel werden *Array*-Parameter bei Fernaufrufen normalerweise serialisiert, während sie im lokalen Fall per Referenz übergeben werden. Technisch ist die Erhaltung der Semantik jedoch durchaus möglich, sobald man, anders als es bei herkömmlicher Middleware geschieht, Quelltext-Transformationen zuläßt. Man sieht das leicht, wenn man sich klarmacht, daß alle erwähnten Plattformen sowohl einen entfernten Referenzmechanismus, als auch Mechanismen zur entfernten Wertübertragung besitzen. Die lokale Aufrufsemantik läßt sich damit in jedem Fall abbilden. Um zum Beispiel einen Array-Parameter semantikerhaltend, also per Referenz zu übergeben, muß das Array in ein fernaufrufbares Objekt eingekapselt werden, was per Transformation des Quelltextes leicht zu bewerkstelligen ist. (Um große Mengen teurer Fernzugriffe auf dieses Objekt zu vermeiden, kann es ggf. zum Empfänger migriert werden.) Nur falls die Analyse sicherstellen kann, daß der Empfänger nur *lesend* auf das Array zugreift, darf die übliche Wertübergabe per Serialisierung zugelassen werden.

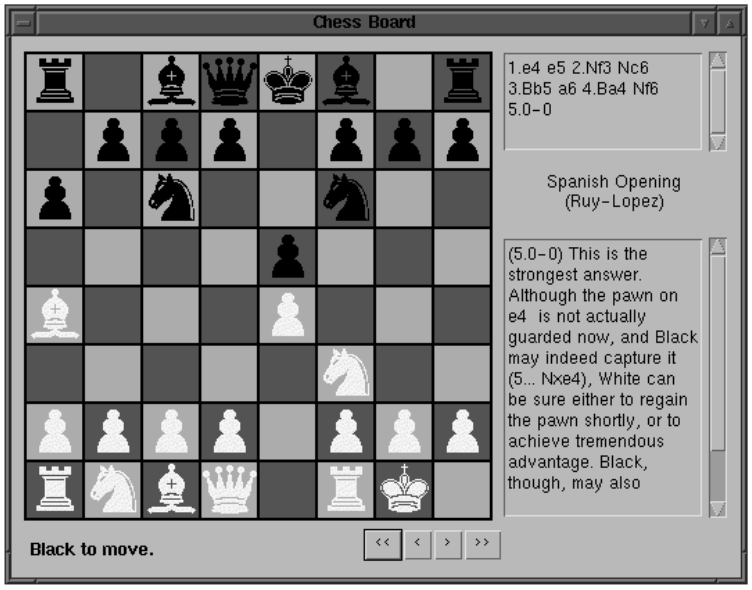


**Abb. 3.** Eine Datenbank für Schach-Eröffnungen

## 5 Eine Fallstudie

Wir betrachten als Beispiel eine in Java implementierte, graphische Datenbank für Schach-Eröffnungen (Abb. 3). Der Benutzer kann auf dem Schachbrett die Figuren bewegen, das Programm sucht in der Datenbank nach dem Namen der entsprechenden Eröffnung und zeigt diesen, zusammen mit einem eventuellen Kommentar zu dem Zug, auf dem Bildschirm an. Die Datenbank besteht aus einer einfachen Textdatei (derzeit, bei sehr rudimentärem „Eröffnungswissen“, etwa 25 kByte groß). Das Programm selbst enthält etwa 2.500 Zeilen Code in 40 Java-Klassen.

Um das Programm im Internet benutzen zu können, soll es so verteilt werden, daß die graphische Oberfläche (als Web-Applet) auf einem Client-Rechner läuft, während die Datenbank auf dem Server verbleibt (bei einer realistisch großen Datenbank wäre es nicht praktikabel, sie mit auf den Client zu laden). Das Optimierungskriterium für die Verteilung des Programms soll hier ein möglichst gutes interaktives Antwortzeitverhalten sein. In dieser rein sequentiellen Anwendung ist das gleichbedeutend damit, möglichst wenig Fernaufrufe durchzuführen.

Abbildung 4 zeigt, stark vereinfacht, den Objektgraph des Programms, wie ihn der bereits implementierte Teil von Pangaea auch tatsächlich ermittelt. Jeder Knoten steht für ein einzelnes Laufzeitobjekt, die Kanten geben Benutzungsbeziehungen, d.h. Methodenaufrufe an. Ausgeblendet aus dem Graph sind unter anderem bereits die Objekte, die Schachzüge oder Brettkoordinaten repräsentieren: Pangaea erkennt, daß es sich dabei um konstante Objekte handelt, die wie Werte behandelt werden können. Man erkennt außerdem, daß es zur Laufzeit zwei Instanzen der Klasse `Board` gibt: eine davon

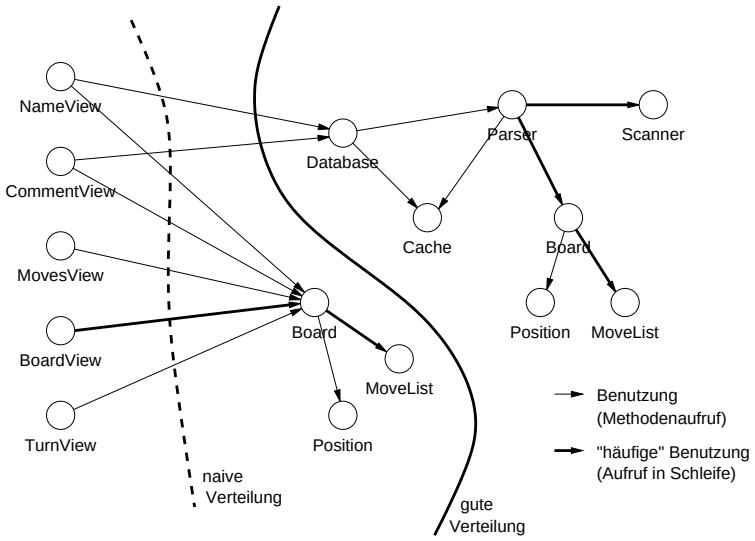


**Abb. 4.** Objektgraph (vereinfacht) mit Verteilungsgrenzen

bildet das Modell des Schachbretts, auf dem der Benutzer seine Züge macht, das andere wird vom `Parser` intern zum Interpretieren der Textdatei verwendet. Eine rein klassenbasierte Analyse oder Verteilung griffe hier also bereits zu kurz.

Eine naive Verteilung des Programms bestünde darin, die Objekte der Oberfläche (die `View`-Objekte am linken Rand) auf den Client zu legen, und die Anwendungslogik auf den Server. Bei diesem Programm führt das allerdings zu sehr schlechtem Antwortzeitverhalten – und zwar deshalb, weil die Oberflächenobjekte intensiv mit dem linken `Board`-Objekt kommunizieren. Insbesondere das `BoardView`-Objekt führt bei jedem Zug des Benutzers 64 Aufrufe durch, um den Zustand der einzelnen Schachfelder abzufragen.

Um das Kommunikationsvolumen entlang der Kanten im Graph abzuschätzen, ist bereits eine einfache Heuristik ausreichend. Wenn wir alle die Kanten, die Methodenaufrufe innerhalb einer Schleife repräsentieren, als „potentiell teuer“ markieren, und an diesen Stellen möglichst keine Verteilungsgrenze einziehen, ergibt sich in diesem Programm bereits die richtige Verteilung: Sie besteht darin, das linke `Board`-Objekt, und alle Objekte, die zu seinem „Innenleben“ gehören, mit auf den Client zu legen. Wie der Graph zeigt, ergibt sich dadurch außerdem, daß nur das `Database`-Objekt je über die Verteilungsgrenze hinweg aufgerufen wird und also fernaufrufbar sein muß, bei allen anderen Objekten kann dieser Aufwand entfallen.

Die Auswirkung auf das Antwortzeitverhalten ist drastisch. In Abbildung 5 und 6 ist die Reaktionszeit des Programms auf verschiedene Eingaben (Schachzüge) dargestellt, und zwar für eine unverteilte Version, die naiv verteilte und die gut verteilte Fassung (wir haben diese Verteilungen manuell auf der JavaParty Plattform realisiert; es mußten jeweils nur wenige Programmzeilen geändert werden). Der helle Teil der Balken bedeutet die Zeit bis zur ersten sichtbaren Reaktion des Programms, der dunklere Teil die Zeit

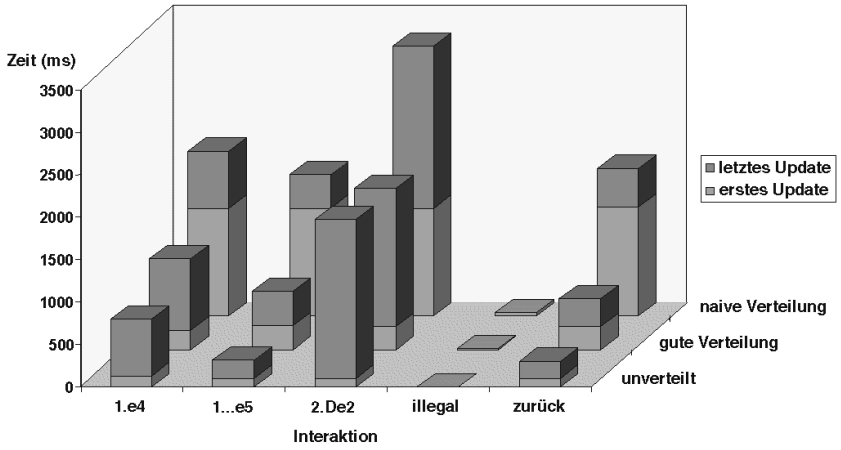


**Abb. 5.** Verteilung via Ethernet (10 Mbps)

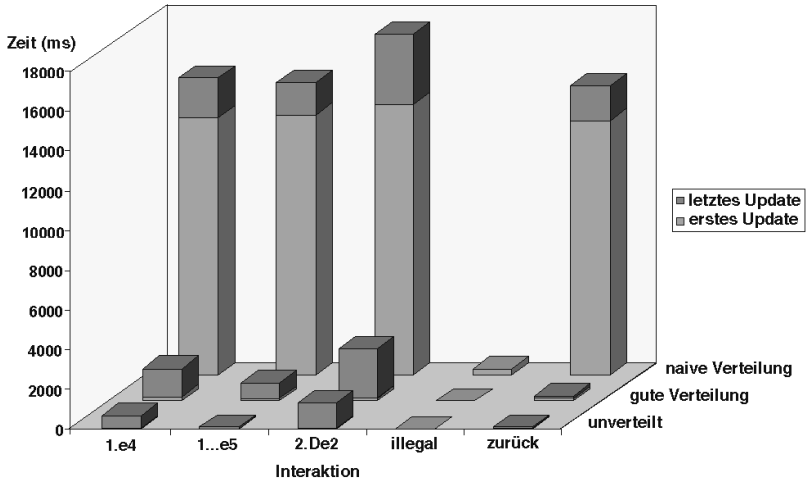


**Abb. 6.** Verteilung via Modem (28800 Bps)

bis zur vollständigen Abarbeitung der Eingabe. Wie sich zeigt, ist die naive Verteilung auch in einem schnellen Netzwerk mindestens unangenehm, und bei einer langsamen Verbindung schlicht inakzeptabel. Gut verteilt, kommt das Programm hingegen in allen Fällen dicht an die Leistung der unverteilten Version heran.

## 6 Zusammenfassung und Ausblick

Wir zeigen mit dem Pangaea-System, daß es möglich ist, zentralisierte Java-Programme automatisch zu verteilen. Das bedeutet einerseits, aufgrund von statischer Analyse eine *abstrakte Entscheidung* zu treffen, wie ein gegebenes Programm verteilt werden sollte, um bestimmten Rahmenbedingungen und Optimierungskriterien zu entsprechen. Wir zeigen andererseits, daß die *Umsetzung* dieser Entscheidung auf einer gegebenen Verteilungsplattform ein rein mechanischer Vorgang ist, der ebenfalls automatisch erfolgen kann. Pangaea kann damit als ein *verteilender Übersetzer* betrachtet werden: Genau wie ein traditioneller Übersetzer die Konstrukte einer Hochsprache möglichst effizient auf eine unterliegende Maschinenarchitektur abbildet, so übersetzt Pangaea Java-Programme für eine vorgegebene Middleware-Schicht.

Implementiert ist bisher der Algorithmus zur Gewinnung von Objektgraphen aus dem Quelltext, sowie die Benutzeroberfläche zur Manipulation dieser Graphen. Konstante Objekte werden bereits automatisch erkannt und können aus dem Graph ausgeblendet werden. Wir planen, den Programmierer die Verteilungsentscheidung zunächst noch manuell treffen zu lassen, durch Markieren *aller* Objekte im Graphen; später sollen Graphpartitierungsalgorithmen und weitergehende Analysen einen Großteil dieser Arbeit übernehmen. Einen Plattform-Adapter werden wir zunächst für die JavaParty-Plattform entwickeln, da hier bereits ein sehr hoher Grad von Verteilungstransparenz vorliegt. Später ist in einer eigenständigen Arbeit ein CORBA-Adapter geplant, sowie die Unterstützung einer experimentellen Verteilungsplattform, die derzeit an der Freien Universität Berlin entwickelt wird und insbesondere strukturierte Mobilitätstechniken untersuchen soll, gekoppelt an Mechanismen der Parameterübergabe.

## Literatur


1. Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and M. Frans Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, February 1998.
2. Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. Dynamic layout of distributed applications in FarGo. In *Proc. ICSE '99*, Los Angeles, May 1999.
3. Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
4. OMG. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, July 1995.
5. Michael Philippsen and Bernhard Haumacher. Locality optimization in JavaParty by means of static type analysis. In *Proc. Workshop on Java for High Performance Network Computing at EuroPar '98*, Southhampton, September 1998.
6. Michael Philippsen and Matthias Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.
7. André Spiegel. Objects by value: Evaluating the trade-off. In *Proc. PDCN '98*, pages 542–548, Brisbane, Australia, December 1998. IASTED, ACTA Press.
8. André Spiegel. Object graph analysis. Technical Report B-99-11, Freie Universität Berlin, July 1999.
9. Sun Microsystems. *Java Remote Method Invocation Specification*, February 1997.
10. Alfred Wegener. *Die Entstehung der Kontinente und Ozeane*. Vieweg, Braunschweig, 1915. 6. Auflage 1962.