VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Implementace distribuovaných datových struktur do knihovny OOSol

Implementation of distributed data structures in OOSol library

2010 Bc. Martin Stachoň

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava*.

V Ostravě 6. května 2010 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2010 .....

Děkuji rodičům za umožnění mého studia.

Dále bych poděkovat interní grantové agentuře FEI VŠB - TU Ostrava za podpoření projektu Vývoj paralelních algoritmů pro výpočetně náročné inženýrské úlohy, v rámci něhož tato diplomová práce vznikala. Také bych chtěl poděkovat všem mým kolegům z týmu Distribuované datové struktury a paralelní algoritmy za připomínky, rady, testovací příklady a diskuze o paralelních algoritmech.

Abstrakt

Tato práce se zabývá implementací distribuovaných datových struktur do knihovny OOSol s využitím existujících datových struktur a algoritmů v objektově orientovaném middleware CORBA. Popisuje základní aspekty technologie CORBA a její použití pro implementaci distribuovaného uložení blokových a blokově diagonálních matic na mnoho procesorů. Dále popisuje implementaci základních operací nad distribuovanými maticemi, které umožňují paralelní výpočty s existujícími algoritmy, například metodu sdružených gradientů. Práce obsahuje testování na modelových úlohách, které ukazuje rychlost a škálovatelnost této implementace. Uvedeno je také porovnání technologie CORBA s MPI.

Klíčová slova: paralelní a distribuované algoritmy, CORBA, MPI, middleware, systém uložení matic, paralelní škálovatelnost, OOSol

Abstract

This thesis deals with implementation of distributed data structures in OOSol library utilizing existing data structures and algorithms using object oriented middleware CORBA. Basic aspects of CORBA technology are described, as well as its use for implementation of distributed storage of block and diagonal block matrices on a large number of processors. Furthermore, implementation of basic operations on distributed matrices are described, which allow parallel computation using existing algorithms, e.g. conjugate gradient method. Testing on model problems is included, which shows the speed and scalability of this implementation. Also, a comparsion of CORBA with MPI is made.

Keywords: parallel and distributed algorithms, CORBA, MPI, middleware, matrix storage system, parallel scalability, OOSol

Seznam použitých zkratek a symbolů

CORBA	–	Common Object Request Broker Architecture
MPI	–	Message Passing Interface
POA	–	Portable Object Adapter
OOP	–	Object Oriented Programming, Objektově Orientované Programování
IIOP	–	Internet Inter-ORB Protocol
GIOP	–	General Inter-ORB Protocol
OMG	–	Object Management Group
TCP/IP	–	Transmission Control Protocol/Internet Protocol
ORB	–	Object Request Broker
SII	–	Static Invocation Interface
DII	–	Dynamic Invocation Interface
AMI	–	Asynchronous Messaging Interface
IOR	–	Interoperable Object Reference
CG	–	Conjugate gradient method

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Cíle a zadání

1.1 Zadání diplomové práce

Implementace distribuovaných datových struktur do knihovny OOSol

1. Návrh vhodné struktury distribuovaných matic.
2. Analýza vhodné technologie na bázi CORBA, Java.
3. Implementace distribuovaných plných i řídkých matic do knihovny OOSol.
4. Aplikace distribuovaných datových struktur na řešení rozsáhlých soustav lineárních rovnic.
5. Testování a srovnání zvolených technologií s jinými dostupnými technologiemi.

1.2 Knihovna OOSol

OOSol (Object Oriented Solvers) je objektově orientovaná knihovna, vyvíjená zaměstnanci a studenty na katedře aplikované matematiky VŠB-TU Ostrava. Knihovna má sloužit k objektově orientované implementaci numerických metod a algoritmů, se zaměřením na optimalizační úlohy. OOSol je napsán v jazyce C++.

V současnosti OOSol zahrnuje zejména třídy pro :

- Datové struktury vektor, plná matice, řídká matice ve formátu compressed row storage a ve formátu skyline (implementace skyline formátu matic do knihovny OOSol byla tématem mé bakalářské práce).
- Algoritmy CG, simplex, MPGRP, SMALBE.
- Faktorizace matic.

1.3 Cíle diplomové práce

Cílem diplomové práce je rozšířit knihovnu OOSol, která dosud obsahovala pouze sekvenční (běžící pouze na jednom procesoru) datové struktury a algoritmy, o jejich distribuované verze, které umožní využít výpočetní prostředky složené z mnoha procesorů.

Nasazení distribuovaného OOSolu na výkonné výpočetní clustery, umožní řešit rozsáhlé praktické úlohy, které by na jednom počítači nebyly řešitelné vůbec, nebo ve velmi dlouhém čase.

Distribuovaný OOSol přinese urychlení i pro výpočty na běžných osobních počítačích, neboť i zde se dnes projevuje trend paralelizace a dokonce i přenosné počítače běžně obsahují několik procesorů.

Pro vývoj distribuovaných datových struktur v rámci této diplomové práce jsem si stanovil následující cíle :

- Zařadit systém do existující třídní hierarchie, použít pokud možno co nejvíce existujícího sekvenčního kódu. Pokusit se minimalizovat potřebu přepisovat existující sekvenční algoritmy znovu do distribuované verze.
- *Implementace v C++.* Jelikož je celá knihovna OOSol psaná v jazyce C++, i distribuované struktury jsem se rozhodl implementovat v C++ z důvodu výkonu a viz předchozí bod.
- *Přenositelnost.* Systém musí byt přenositelný mezi platformami, jak hardwarovými, tak i operačními systémy. Prakticky to znamená dodržení standartu ISO C++ a používání přenositelných externích knihoven.
- *Přehledný, udržovatelný kód.* Implementace by měla být jasná, přehledná a dobře čitelná. To ulehčí jak použití těchto struktur dalšími programátory, tak i vývoj.
- *Objektová orientace.* Používat co nejvíce objektově orientované programování.
- *Použití otevřených technologií.* Využívat technologie, jejichž specifikace jsou volně dostupné. Používat externí knihovny, které jsou k dispozici volně k používání (například pod licencí LGPL nebo BSD). Tím chci zabranit vazbě na proprietární technologie, které by omezily přenositelnost a přidaly by licenční, případně i finanční požadavky.
- *Škálovatelnost.* Implementace by měla být škálovatelná s počtem procesorů.

1.4 Obsah diplomové práce

V kapitole 2 uvádím přehled existujících komunikačních technologií a zavádím jejich rozdělení na datové, procedurální a objektové. V kapitole 3 popisuji princip distribuovaných objektů a jeho použití pro paralelní výpočty a propojení výpočetní infrastruktury. V kapitole 4 pak zdůvodňuji, proč jsem z dostupných technologií zvolil právě CORBA. Uvedeny jsou také její výhody a nevýhody.

Kapitola 5 je pak věnována podrobnějšímu popisu technologie CORBA. Uvedeny jsou hlavně prvky specifikace důležité pro implementaci v knihovně OOSol. V kapitole 6 je popsáno použití technologie CORBA pro implementaci distribuovaných datových struktur v knihovně OOSol. Popisuji, jak jsem zařadil svou implementaci do třídní hierarchie knihovny OOSol, jak jsem implementoval architekturu manager/worker a jak jsem řešil asynchronní operace. Nakonec popisuji implementaci distribuovaných struktur blokových matic a jejich použití v algoritmech.

V kapitole 7 je tato implementace testována na modelové úloze. Poslední kapitola 8 shrnuje dosažené výsledky této diplomové práce.

2 Přehled komunikačních technologií

V této části zmíním některé nejčastěji používané komunikační technologie (taktéž nazývané Middleware) pro vývoj paralelních a distribuovaných technologií a jejich výhody a nevýhody.

Komunikační technologie lze rozdělit na tři třídy - postupně od jednodušších k pokročilejším : remote messaging (datové), remote call (procedurální) a remote object (objektové). Toto rozdělení má paralelu ve vývoji programovacích jazyků, které můžeme podobně rozdělit na nestrukturované, procedurální a objektově orientované. Schematicky jsem to zobrazil na obrázku 1.

Stranou těchto imperativních přístupů je deklarativní programovaní (například funkcionální programování), jehož obdoby se začínají objevovat i v komunikačních technologiích.

2.1 Messaging (datové)

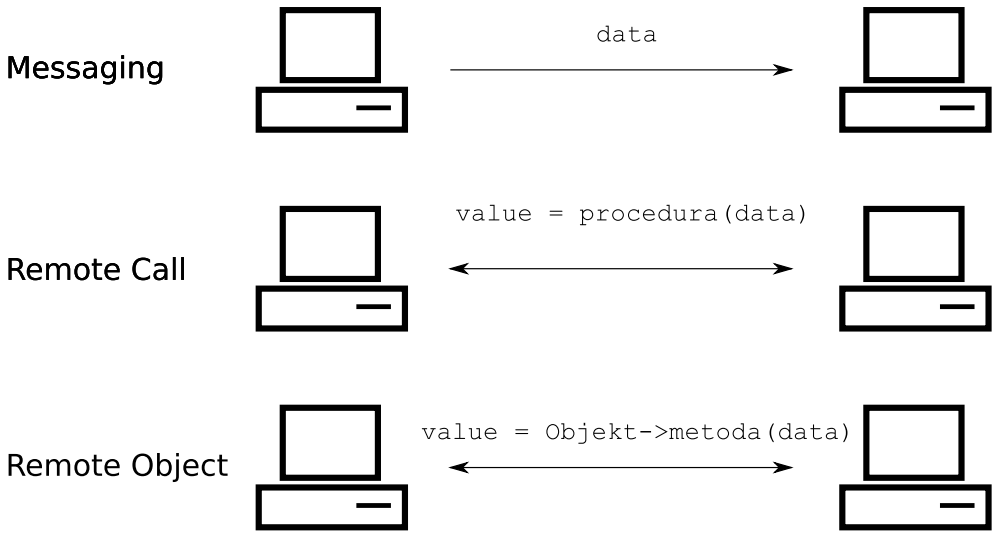
Jsou to technologie, které umožňují pouze posílat data mezi procesy. Tyto technologie lze samotné velmi snadno implementovat. Implementace programů, které je mají využívat, je ovšem náročnější. Programátor musí vytvořit formát, ve kterém data budou posílány, zajistit navázání komunikace mezi jednotlivými počítači a zajistit synchronizaci mezi přijímací a odesílací stranou.

- **Socket**. Socket API vyvinuté na univerzitě Berkeley se stalo de facto standardem pro socketové síťové rozhraní. Poskytuje základní funkce pro navázaní spojení, příjem a odesílání bajtů dat. Socket API lze najít téměř na všech operačních systémech.
- **MPI** je dnes nejčastěji používané rozhraní pro paralelní programování. Protokol nezávislý na platformě umožňuje cílenou i kolektivní komunikaci. MPI umožňuje efektivně využívat topologii sítě. Optimalizované implementace MPI jsou k dispozici pro mnoho programovacích jazyků i platforem.

2.2 Remote call (procedurální)

Tyto technologie umožňují volat procedury s daty jako parametry na vzdálených počítačích a získavat návratové hodnoty. Tento koncept se začal objevovat na konci 70. let minulého století.

- **DCE/RPC** je systém vzdáleného volání dnes již opuštěného softwarového balíku Distributed Computing Environment. DCE/RPC se později stalo základem technologií Microsoftu : DCOM, ODBC a Microsoft RPC.
- **RPC** Existuje v několika podobách. Sun RPC je k dispozici na většině Unixových platforem a používá ho dodnes používaný síťový souborový systém NFS. XML-RPC přenáší data ve formě XML po HTTP protokolu. Microsoft má vlastní verzi Microsoft RPC.



Obrázek 1: Rozdělení komunikačních technologií.

- **Simple Object Access Protocol** je protokol pro přenos strukturovaných dat v rámci Web Services. SOAP formátuje data pomocí XML a přenáší je protokolem HTTP, což má výhodu při použití přes veřejnou internetovou síť, jelikož firewally často jiné protokoly než HTTP nepovolují. Nevýhodou použití XML je, že se přenáší větší objem dat než u binárních formátů, navíc zpracování každé zprávy je výpočetně náročné. Proto je SOAP vhodný spíš pro webové služby, než pro high performance computing.

- **Apache Thrift**. RPC s kompatibilitou mezi mnoha programovacími jazyky.

2.3 Remote object (objektové)

Objektové technologie jsou logickým rozšířením procedurálních technologií. Začínají se objevovat na začátku 90. let minulého století. Volání procedur se stává voláním metod na objektech, které mohou být v síti dynamicky instancovány. Je použita dědičnost a další aspekty objektově orientovaného programovaní.

Příklady této třídy technologií jsou :

- **CORBA** Technologie CORBA je podrobněji popsána v kapitole 5

- **Java remote method invocation (RMI)** je vzdálené rozhraní pro jazyk Java. Díky tomu, že je vázáno na Javu, je RMI přehlednější a jednodušší než CORBA.

- **RMI-IIOP** Je implementace RMI nad protokolem CORBy. Umožňuje tak komunikaci mezi RMI a CORBA aplikacemi.

- **Enterprise JavaBeans**. EJB je komponentní architektura pro business logiku v Javě. Kromě rozsáhlých funkcí jako jsou persistence, transakce atd. umožňuje i použití distribuovaných objektů. K tomu je použit protokol RMI-IIOP, tedy CORBA.
- **Distributed Component Object Model (DCOM)** je technologie Microsoftu pro distribuované objekty, původně nazývaná Microsoft OLE. Je postavena na DCE/RPC. Později byla přidána možnost komunikovat pomocí HTTP. Ve své době byla tato technologie konkurentem CORBY, dnes je nahrazena technologií .NET Remoting.
- **.NET Remoting** je API uvedené v rámci .NET framework 1.0. V .NET 3.0 bylo nahrazeno technologií Windows Communication Foundation.
- **Windows Communication Foundation** je navržena jako architektura síťových služeb SOA (Service Oriented Architecture).
- **DDObjects** - již nevyvíjený middleware pro Borland Delphi, Pyro - Python Remote Objects, Distributed Ruby

2.4 Propojení technologií

Různé technologie je často možné propojovat a kombinovat. Například existují implementace MPI pomocí CORBy, nebo RMI pomocí CORBy. Také existují distribuované systémy, kde se uvnitř clusteru používá MPI, pro komunikaci s dalšími výpočetními prostředky pak CORBA. [12]

3 Distribuovaný objektový přístup

3.1 Distribuované a paralelní systémy

Víceprocesorové výpočetní systémy bývají často rozdělovány na *paralelní* a *distribuované*. Hranice mezi nimi není ostrá, označuji je touto přibližnou definicí :

- **Paralelní systém** je počítač se sdílenou pamětí, mnoha procesory (typicky identickými). Obvykle je na něm spuštěna jedna instance operačního systému. Architektura je předem známa.
- **Distribuovaný systém** tvoří mnoho počítačů s oddělenými paměťovými prostory, s vlastními instancemi operačního systému, propojené v počítačové síti. Jednotlivé počítače se mohou lišit v použitém softwaru i hardwaru. Počet procesorů a síťová topologie nemusí být předem známa.

Zatímco paralelní systémy se sdílenou pamětí jsou velmi drahé (kvůli nákladným paměťovým sběrnicím), distribuované systémy lze snadno stavět z běžných počítačů a operačních systémů. Rychlost takto vytvořeného systému, výpočetního cluster, se kromě výkonu jednotlivých počítačů (často nazývaných uzlů) také odvíjí od typu použité propojovací sítě.

Podobně můžeme rozlišit i distribuované programování pro distribuované systémy a paralelní programování pro paralelní systémy. Hranice mezi nimi je ještě menší. Dá se říct, že u paralelního programování jsou jednotlivé procesy silněji provázány, často bývají identické, kdežto u distribuovaného programování je třeba počítat s heterogenitou výpočetních prostředků, možnými výpadky atd.

Tato terminologie však mnohdy nebývá dodržována a pojmy distribuovaný a paralelní bývají zaměňovány. Jejich cíl je stejný - umožnit řešení rozsáhlých úloh pomocí více procesorů.

3.2 Distribuované objekty

Zadání mé diplomové práce je implementace distribuovaných datových struktur. V současnosti se - jak pro paralelní, tak i pro distribuované systémy - používá, jak jsem zmínil v kapitole 2, většinou standart MPI (využívají ho například knihovny PETSc, MPQC, ScaLAPACK). Obvyklý způsob psaní a spouštění programů v MPI je bližší paralelním systémům - obvykle se na všech procesorech spouští stejný programový kód, který se pak na jednotlivých procesorech větví (SPMD - single program multiple data). [1]

Při použití obvyklého worker/manager modelu to znamená pro programátora napsat jeden kód, který se na různých místech musí větvit (if) a provádět různé činnosti podle toho, zda je daný uzel manager nebo worker. V jednom kódu se tak i uvnitř metod musí míchat kód pro managera a workera, což nevadí pro krátké programy. Pro složitější systémy se takto psaný kód stává velmi nepřehledným.

[1] MPI umožňuje spouštět na různých strojích různé programy, v praxi se tento přístup moc často nepoužívá

MPI navíc přenáší pouze data, takže i když je kód používající MPI objektový, musí programátor sestavit schéma, jak se data přenášená po MPI budou přiřazovat jednotlivým objektům a jejich metodám.

Z těchto důvodů je MPI hůře slučitelné s objektově orientovaným programováním, které se v dnešní době hojně používá ve všech softwarových odvětvích.

Rozhodl jsem se tedy svůj přístup k distribuci datových struktur postavit na principu *distribuovaných objektů*. Distribuovaný objekt je instancován na jednom uzlu distribuovaného systému, jeho data v paměti jsou tedy lokální pro jeden uzel. Distribuovaný objekt ovšem může také obsahovat odkazy na další distribuované objekty. Metody distribuovaného objektu lze volat jak lokálně, tak i vzdáleně, z jiného procesu v jiném paměťovém prostoru po síti.

Příkladem distribuovaného objektu je například objekt Matrix (reprezentující matici), který je instancován (a tedy uložen v paměti) na uzlu A. Proces, bežící na jiném uzlu B, může zavolat na vzdáleném objektu Matrix například metodu `norm_F()`, která vypočítá Frobeniovu normu matice. Uzel B odešle požadavek na volání uzlu A, který provede výpočet `norm_F()` a výsledek pošle zpět na uzel B. Pro proces na uzlu B není rozdíl mezi voláním metod lokálních a vzdálených objektů.

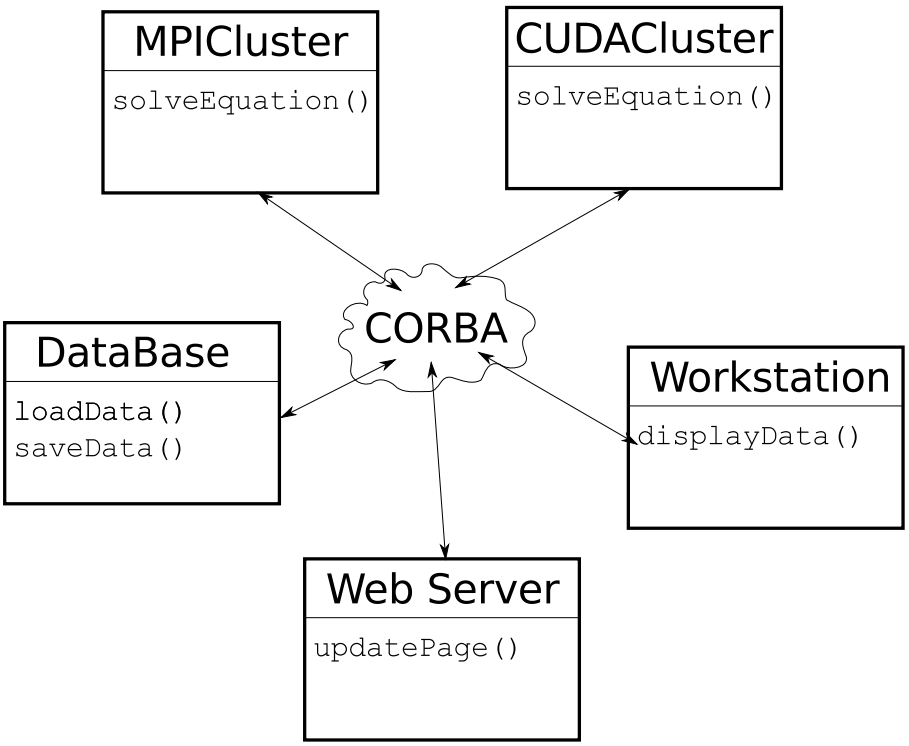
Jednotlivé distribuované objekty pak můžeme spojit do složitějších datových struktur. Příkladem může být objekt BlockMatrix, který reprezentuje blokovou matici a obsahuje odkazy na jednotlivé distribuované objekty Matrix. Pro výpočet `norm_F()` této BlockMatrix zavoláme `norm_F()` na každém objektu Matrix. Výsledná norma blokové matice je rovna odmocnině ze součtů čtverců norem jednotlivých bloků. Takto je proveden paralelní výpočet intuitivním způsobem - voláním metod jednotlivých objektů.

Distribuované objekty lze chápat jako rozšíření pojmu *enkapsulace*, známého z OOP. Při používání distribuovaných objektů nemusíme rozlišovat, zda je místní nebo vzdálený. Programátor nemusí ani vědět, že distribuovaný objekt je implementován pomocí tisíců dalších objektů distribuovaných v rozsáhlém výpočetním clusteru. Důležité je pouze rozhraní distribuovaného objektu.

Pro implementaci distribuovaných objektů jsem z dostupných možností zvolil technologii CORBA, která umožňuje distribuované objekty používat mezi různými platformami a programovacími jazyky. Tuto volbu zdůvodňuji v kapitole 4.

Distribuované objekty umožňují zavést abstrakce, které v datových technologiích, jako je MPI, nejsou možné. Jednotlivé distribuované objekty mohou reprezentovat datové struktury (jako je například zmíněná bloková matice). Mohou také reprezentovat různé výpočetní prostředky v síti. Na obrázku 2 je schématicky znázorněna sít s výpočetními zdroji přístupnými přes CORBA rozhraní. Distribuované objekty tak mohou být použity jako nástroj pro integraci rozmanitých výpočetních prostředků do jednotného, objektového rozhraní.

Nebo můžeme zavést abstrakci nad výpočetními prostředky a vytvořit rozhraní pro řešení úloh. Programátor může toto rozhraní použít například k řešení soustavy lineárních rovnic, a rozhraní se postará o rozdělení úlohy mezi dostupné výpočetní prostředky. Možnosti jsou opravdu široké.



Obrázek 2: Příklad využití distribuovaných objektů k integraci výpočetních prostředků.

4 Volba komunikační technologie

Při volbě vhodné komunikační technologie pro distribuované výpočty jsem zvažoval, zda mi umožní splnit cíle, které jsem si stanovil v kapitole 1. Vyloučil jsem všechny Messaging a Remote call technologie, neboť neumožňují přímo pracovat s objekty a nesplňují tedy požadavek na objektovou orientaci.

Ze zbývajících objektových technologií jsem se rozhodoval podle jejich přenositelnosti. Vyloučil jsem tak technologie DCOM, .NET Remote Objects a WCF, neboť jsou vázané na platformu Microsoft (částečné implementace pro Unix existují, nejsou však plně podporovány). Technologie jako RMI, DDObjects, Java Spaces, Pyro, Distributed Ruby jsem vyloučil z důvodu vazby na jazyky Delphi, Java, Python, respektive Ruby.

Tyto jazyky nejsou vhodné pro implementaci numerických výpočtů z důvodu horšího výkonu ve srovnání s C++. Existují sice implementace i pro jiné jazyky, například implementace RMI pro C++, bohužel nejsou v produkčním stavu. [2]

Dále jsem vyloučil technologie, které jsou uzavřené a mají malou uživatelskou základnu, jako je například ICE. Zůstala mi tak jediná technologie, která umožňuje splnit všechny cíle mé diplomové práce - CORBA.

4.1 Výhody technologie CORBA

Výhody použití technologie CORBA jsem krátce shrnul v těchto bodech :

1. Umožňuje volání objektů jak v rámci jednoho paměťového prostoru, tak i vzdáleného po síti.
2. CORBA je mnoho let existující „průmyslový standard“, specifikace jsou veřejně dostupné. Existuje mnoho jak komerčních, tak volně dostupných implementací.
3. Standard CORBA je nezávislý na platformě a použitém programovacím jazyku.
4. Programy mohou zároveň komunikovat i počítat (pokud konkrétní implementace používá threading).
5. Měření provedená v publikacích [12, 11, 13, 14] dávají předpoklad, že rychlost a škálovatelnost je řádově srovnatelná s MPI.
6. CORBA umožňuje vývoj *fault tolerant* výpočetních systémů.

4.2 Nevýhody technologie CORBA

1. Není zcela snadné přecházet mezi jednotlivými implementacemi. Některé konkrétní prvky implementace nejsou specifikovány ve standardu CORBA a mezi jednotlivými implementaci se odlišují. Jedná se například o práci se skeleton třídami.

[2] Existují možnosti, jak technologie pro jiné jazyky použít v C++ pomocí wrapperu, například Java Native Interface mezi Javou a C++. To ovšem znamená přidání další mezivrstvy, která sníží přehlednost a rychlost programu.

2. Ne všechny implementace plně splňují všechny prvky specifikace, může se tedy projevit vzájemná nekompatibilita.
3. CORBA nemá metody pro kolektivní komunikaci.
4. CORBA přidává další stupeň abstrakce, proto může být komunikace méně efektivní než u čistě datových technologií.
5. Specifikace a API jsou místy zbytečně složité, mapování pro jazyk C++ není zcela snadno použitelné, programátor se musí starat o některé detaily, což může vést k chybám.

Více kritiky lze najít v [3].

5 Technologie CORBA

Tato kapitola je věnována úvodu do technologie CORBA. Nejsou zde popsány zdaleka všechny funkce, které jsou popsány ve specifikaci. Věnuji se hlavně konceptům a funkcionalitě, kterou jsou využil v rámci implementace struktur v knihovně OOSol. Případného zájemce odkáži na oficiální specifikace [15].

5.1 Historie

Standard CORBA (Common Object Request Broker Architecture) byl vytvořen v devadesátých letech konsorciem OMG (Object Management Group). OMG bylo založeno v roce 1989 několika společnostmi (mimo jiné HP, Apple Computer, IBM, Sun) za účelem vytvoření přenositelného standardu pro distribuované objekty. Později přibyly další projekty, mimo jiné standart UML (Unified Modeling Language), používaný v softwarovém inženýrství. Nyní má OMG přes 800 členských společností.

První specifikace CORBA 1.0 byla vydána v roce 1991. Ta ale obsahovala mnoho nejasností, nebyla přenositelná, neboť nebyl specifikován komunikační protokol, a obsahovala mapování pouze pro jazyk C. Další hlavní verzí byla specifikace 2.0, jejímž hlavním přínosem je definování abstraktního komunikačního protokolu GIOP a jeho konkrétní implementace pro TCP/IP - IIOP. Díky tomu je možná vzájemná komunikace mezi různými implementacemi ORB.

Následovaly další verze specifikace, které přidaly funkce jakými jsou šifrovaná komunikace, podpora jazyka Java, real-time, Naming service, Notification service a další. Aktuální verze specifikace CORBA je 3.1, vydaná v roce 2008 [15].

5.2 Filozofie

Jádrem technologie CORBA je ORB (Object Request Broker), což je softwarová vrstva (tzv. middleware) zprostředkovávající komunikaci mezi aplikací s jejími objekty, a vzdálenými objekty na síti. ORB se stará o :

- Získání referencí na objekty
- Přenos požadavku na volání metody po síti (a marshalling parametrů)
- Přenos návratové hodnoty

ORB by měl před uživatelem co nejlépe skrýt detaily distribuovaného systému a měl by umožnit programátorovi pracovat se vzdálenými objekty téměř stejně jako s místními.

Pojem ORB není specifický pouze pro technologii CORBA, používá se i v jiných podobných technologiích. Někdy se používá ve významu distribuované objektově orientované technologie obecně.

Implementace technologie CORBA probíhá v těchto krocích :

1. Napsání definice rozhraní v jazyce IDL.

Obrázek 3: Oficiální logo CORBA.

Obrázek 4: Logo konsorcia OMG.

2. Vygenerování zástupných tříd pomocí IDL compileru.
3. Implementace servantů, objektů implementujících definované rozhraní.
4. Napsání klientského kódu, který spustí ORB, inicializuje a aktivuje příslušné servant objekty pro prostředí CORBA.

V dalších podkapitolách blíže popíši některé technické aspekty, konkrétně :

- Abstraktní jazyk IDL, popisující rozhraní vzdálených objektů, jejich metod a parametrů.
- Skeleton, Stub, Servant - systém objektů, které zastupují volané objekty na straně serveru, případně klienta a zajišťující propojení s konkrétní instancí..
- IIOP, protokol, který používá CORBA pro komunikaci mezi ORBy po síti TCP/IP.
- POA, rozhraní pro registraci a správu objektů v ORB.
- Některé aspekty C++ mapování.

- Vybrané implementace ORB.

5.3 Interface Definition Language

Interface Definition Language, nebo také Interface Description Language, je abstraktní jazyk, sloužící pouze k definování rozhraní mezi objektem a ORB. U objektů distribuovaných technologií CORBA můžeme používat pouze rozhraní předem definované v IDL. [3]

Jazyk IDL je nezávislý na platformě a programovacím jazyku, ve kterém budou objekty implementovány. Pokud používáme více jazyků, je toto výhodou, neboť definice rozhraní v souboru IDL je společná pro implementace v různých jazycích.

Pro každý programovací jazyk, který má používat CORBA, musí existovat mapování mezi tímto jazykem a IDL. Toto mapování musí případně napodobit některé konstrukce, které jsou obsaženy v IDL, ale daný jazyk je nepodporuje (například výjimky v jazyce C pomocí struktur).

Ze souboru IDL se pak generují pomocí IDL compileru skeleton a stub třídy (zástupné třídy) a třídy slouží jako základ pro implementaci servant objektů (objektů implementující IDL rozhraní v konkrétním jazyce). IDL compiler je specifický pro daný jazyk a implementaci CORBy.

Jiný přístup používá například technologie RMI, která je vázána na jazyk Java. Distribuované rozhraní se v RMI píše přímo v Javě (jako `interface` dědící z `Remote`). Odpadá tak nutnost použít jazyk IDL, jistá duplicita v deklaracích rozhraní v IDL a nativním jazyce a použití IDL compileru, ovšem za cenu omezení se na jazyk Java.

Syntaxe IDL je specifikována v kapitole OMG IDL Syntax and Semantics specifikace CORBA. Zde popíši pouze nejdůležitější prvky.

Soubory IDL se ukládají do souborů s příponou `'.idl'`. Syntaxe IDL je na první pohled podobná jazyku C++ : deklarace se ukončují středníkem, bloky jsou ve složených závorkách. Komentáře je také možné používat řádkové `//` nebo blokové `/** ... */`. Příklad jednoduchého IDL souboru pro řídící systém fiktivní vesmírné lodi je ve výpisu 1.

5.3.1 Module

Základní blok **module** deklaruje jmenný prostor. Je to obdoba `namespace` v C++. V rámci modulu musí být všechny identifikátory jedinečné. Implicitně jsou deklarace identifikátorů přístupné v rámci modulu, explicitně lze použít deklaraci z jiného modulu syntaxí `NazevModulu::Identifikator`. Moduly mohou být do sebe zanořeny.

5.3.2 Základní datové typy

V tabulce 1 uvádím základní datové typy definované v IDL a odpovídající mapovaní pro jazyk Java. Pro jazyk C++ se mapování odlišuje v závislosti na platformě.

[3] Pomocí DII a DSI lze používat i dynamické rozhraní, ovšem docela komplikovaně.

IDL	Poznámka	Java mapping
`boolean`	TRUE/FALSE	boolean
`char`	8-bitový znak v kódování ISO Latin-1	char
`octet`	8-bitové celé číslo	byte
`short`	16-bitové celé číslo	short
`long`	32-bitové celé číslo	int
`long long`	64-bitové celé číslo	long
`float`	IEEE 754 single precision	float
`double`	IEEE 754 double precision	double
`string`	znaky ISO Latin-1 proměnné délky	String
`fixed<d,s>`	desetinné číslo s pevnou přesností	BigDecimal

Tabulka 1: Základní datové typy IDL.

Celočíselné datové typy : `short, long, long long`, jsou implicitně se znaménkem. Varianty bez znaménka jsou označeny `unsigned short, unsigned long, unsigned long long`.

Typy `char, string` mají své mezinárodní varianty `wchar, wstring`, kde znaky mohou být v libovolné znakové sadě. ORB je zodpovědný za převod znakových sad mezi různými klienty. Obvykle se používá kódování UTF-16.

Novější verze specifikace přidává typ `any`, který umožňuje proměnné nabývat libovolného typu. Použití typu `any` ovšem není podporováno ve všech implementacích a přináší jisté komplikace.

5.3.3 Komplexní datové typy

Kromě základních typů podporuje IDL i složitější datové typy. Příklady těchto typů jsou ve výpisu 2. Jedná se o :

- `enum` : klasický výčtový typ.
- `struct` : datová struktura obsahující proměnné.
- `union` : podobně jako struct, uložena je však vždy jen jedna proměnná. Switch v závorkách udává, která proměnná je aktivní.
- `sequence<type>` : sekvence, jednodimenzionální pole proměnné délky, obdoba `Vector` v Javě. `type` určuje datový typ prvků. Prvky `sequence` mohou být opět `sequence`, tímto lze vytvořit n-rozměrné dynamické pole. Typ sekvence musí být předtím, než je použit v deklaraci operace, deklarován pomocí `typedef`, jak je uvedeno ve výpisu 2.
- `array` : obdoba `sequence`, rozměr je ovšem pevně definován předem.

5.3.4 Interface

Rozhraní pro distribuované objekty deklarujeme blokem `interface`. Uvnitř rozhraní můžeme definovat atributy klíčovým slovem `attribute`. Atribut je proměnná, kterou lze číst a zapisovat. V mapovaní pro konkrétní jazyky je atribut realizován dvěma metodami `getIdentifikator` a `setIdentifikator`, kde `identifikator` je název proměnné. Klíčovým slovem `readonly` nastavíme atribut pouze pro čtení. [4]

Metody, respektive členské metody, jsou v IDL nazývány operace. Operace má parametry a může vracet hodnotu libovolného typu definovaného v IDL. Operace nevracející hodnotu je označena `void`.

Každý parametr operace musí mít určen směr, ve kterém je předáván, jedním z těchto klíčových slov před typem parametru :

- `in`. Parametr je vstupní - předáván volajícím volanému objektu.
- `out`. Parametr je výstupní - předáván volaným objektem volajícímu.
- `inout`. Parametr je vstupně-výstupní - předáván volajícím volanému objektu, poté zpět volajícímu.

Operaci lze označit jako jednosměrnou klíčovým slovem `oneway`, tím pádem nemůže vracet hodnotu a její provedení není potvrzeno.

Operace může vyvolat výjimku, což se zapisuje pomocí `raises(NazevTypuVyjimky)`.

Rozhraní v IDL podporují vícenásobnou dědičnost, zapisuje se podobně jako v C++ dvojtečkou za názvem rozhraní. Všechna uživatelská rozhraní implicitně dědí z rozhraní `Object`.

5.4 Skeleton, stub, servant a objektová reference

V následujícím textu si objasníme pojmy : objektová reference, skeleton, stub a servant, neboť jsou důležité pro implementaci distribuovaných struktur tak, aby byly pro programátora transparentní (tedy aby mohl vzdálené objekty používat téměř stejně jako místní).

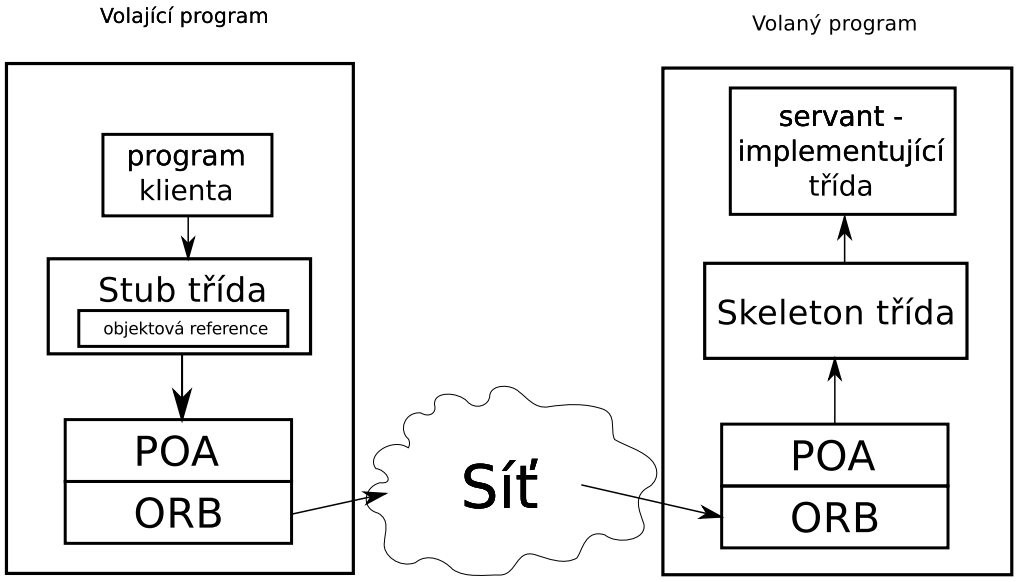
- *Objektová reference* jednoznačně identifikuje objekt v prostředí CORBA. Klient, který chce volat vzdálený objekt, musí nejprve získat jeho objektovou referenci. Přesnou podobu objektové reference specifikuje IOR (Interoperable Object Reference).
- *Stub* objekt zastupuje na straně volajícího programu vzdálený objekt určený objektovou referencí. Programátor může volat jeho metody, stub je předá pomocí ORB na příslušný vzdálený počítač [5] a vrátí přijatou návratovou hodnotu a výstupní parametry.

[4] Ekvivalentně lze v IDL deklarovat přímo `get, set` operace, koncept atributu je tedy ve specifikaci redudantní.

[5] případně i lokální, některé ORBy mají tento případ optimalizovaný a pokud je volající i volaný objekt ve stejném paměťovém prostoru, použije se přímé volání místo loopback síťového rozhraní.

- *Skeleton* zastupuje volaný objekt na straně volaného počítače. Skeleton předá volání konkrétní implementaci - servant objektu. Vazba skeleton-servant je obvykle 1:1, CORBA umožňuje ale i dynamický *servant location*, kdy se servant určuje dynamicky - takto lze implementovat load-balancing (rozdělování zátěže).
- *Servant* je objekt implementující dané rozhraní s výkonným kódem.
- *Objektové ID* je identifikátor objektu, platný pouze v rámci jednoho POA (Portable Object Adapter, viz následující podkapitola 5.6). Slouží k interním účelům POA.

Schématicky je tento proces znázorněn na obrázku 5.



Obrázek 5: Znázornění procesu vzdáleného volání.

5.5 Získání objektové reference, Naming Service

Abychom mohli pracovat se vzdáleným objektem, musíme nejprve získat jeho objektovou referenci. Object reference se dá získat těmito způsoby :

1. Získat ji od jiného objektu, jehož referenci už máme, v návratové hodnotě operace.
2. Vygenerovat IOR v podobě textového řetězce a přenést ho pomocí nějakého jiného média, například uložením do souboru na disk.
3. Použít jmennou službu, kterou CORBA poskytuje.

Krátce popíši poslední bod : CORBA Naming Service. Naming Service je podobná internetové službě DNS. Zatímco DNS překládá jména na IP adresy, CORBA Naming

Service překládá jména na IOR . Jména v Naming Service jsou také hierarchická, jednotlivé úrovně se nazývají *naming context*. Každý naming context se skládá ze dvou řetězců - name a kind. Tento systém jmen je poněkud komplikovaný, později byla přidána částečná podpora pro jména zapisovaná jako řetězec, kde lomítko odděluje jednotlivé úrovně, například `"organizace/sluzby/DatovyServer"`.

Interface pro Naming Service je definován v IDL pod názvem `NamingContext`. Jeho nejdůležitější operace jsou :

- `bind`, která zaregistruje objekt pod daným jménem
- `resolve`, která vrací referenci na objekt daného jména

5.6 Portable Object Adapter

V původní verzi specifikace CORBA nebylo příliš jasně určeno, jak mají být implementovány servat třídy. Díky tomu nebyl kód přenosný mezi různými implementacemi ORB. Proto byl specifikován *Portable Object Adapter*. Deklarace rozhraní POA je napsána v IDL.

POA splňuje tyto úkoly :

- Generování objektových referencí.
- Aktivace a deaktivace servantů. POA může volitelně automaticky vytvářet a dealokovat servant objekty.
- Předávání požadavků příslušným objektům.

Portable Object Adaptéry mohou tvořit stromovou hierarchii, kde si jednotlivé POA postupně předávají požadavky. Pro jednoduchost si vystačíme s používáním kořenového POA, `RootPOA`. V CORBA programu si musíme nejprve získat objektovou referenci na `RootPOA` voláním `resolve_initial_references("RootPOA")`, poté s POA pracujeme stejně jako s jakýmkoliv distribuovaným objektem.

POA umožňuje různé možnosti nastavení pravidel přidělování objektových identifikátorů, přídělování servantů, životnosti objektů atd., opět si většinou vystačíme s výchozím nastavením :

- Thread Policy: ORB_CTRL_MODEL, ORB zpracovává požadavky ve vláknech.
- Lifespan Policy: TRANSIENT, objektová reference je platná do ukončení POA.
- Object Id Uniqueness Policy: UNIQUE_ID, každá instance servanta je přiřazena jedinečnému Object ID.
- Id Assignment Policy: SYSTEM_ID, Object ID jsou jedinečné.
- Servant Retention Policy: RETAIN, POA si udržuje seznam aktivních servantů v Active Object Map.
- Request Processing Policy: USE_ACTIVE_OBJECT_MAP_ONLY, povol požadavky pouze na servanty v Active Object Map.

- Implicit Activation Policy: NO_IMPLICIT_ACTIVATION, servanti nejsou implicitně aktivováni.

Základní operace pro práci s POA jsou :

- `activate_object` aktivuje instanci servant objektu v POA.
- `deactivate_object` deaktivuje servant objekt.
- `the_POAManager` vrací referenci na POA Manager, na němž pak zavoláním operace :
- `activate` spustíme POA.

5.7 Implementace servant tříd

Základní kostra třídy pro POA je generována IDL compilerem, pro implementaci servant třídy máme dvě možnosti :

1. Použití dědičnosti. Rozšířit třídy generované IDL compilerem, které již obsahují kód pro POA, o implementace jednotlivých metod.
2. „Tie“ implementace. IDL compiler vygeneruje třídu pro POA, která se pak naváže (v C++ pomocí šablony) na třídu implementující servanta. Není třeba použít dědičnosti, proto se tento přístup používá v Javě, kde není povolena vícenásobná dědičnost. Tento způsob je také ve specifikaci uveden jako metoda, jak umožnit distribuovatelnost již existujících tříd, které nebyly psány pro CORBA. V praxi je to ovšem těžko proveditelné, protože je třeba zajistit, aby metody v existující implementaci měly stejnou signaturu jako generované Tie skeletony, což snadno naruší už jenom datové typy.

5.8 GIOP, IIOP

General Inter-ORB Protocol (GIOP) je abstraktní protokol pro komunikaci mezi ORBy.
Specifikace GIOP se skládá ze tří částí, které dále popíši detailněji :

5.8.1 Common Data Representation

Common Data Representation (CDR) určuje, v jakém formátu se přenášejí jednotlivé datové typy. CDR také zajišťuje :

- *Byte ordering (jiné pořadí bajtů na na různých platformách)* je řešen tak, že každá GIOP zpráva obsahuje příznak, který indikuje, v jakém pořadí jsou bajty. Pokud spolu komunikují dva počítače se stejným byte-orderingem (což je u výpočetního clusteru většinou zaručeno), není třeba provádět nic. Pokud je pořadí bajtů různé, musí příjemce zprávy provést přehození bajtů.

- Zarovnání primitivních datových typů. Primitivní datové typy jsou v GIOP zprávách zarovnány. To znamená, že každá hodnota primitivního datového typu začíná na bajtu, jehož index je násobkem délky příslušného datového typu (1,2,4 nebo 8 bajtů). Díky tomu je možné primitivní datové typy načíst přímo specifickými instrukcemi. Mnohé platformy mají totiž pomalejší načítání dat z paměti na nezarovnaných adresách, některé je neumožňují vůbec. Zarovnání tak umožňuje efektivnější zpracování zpráv. Bajty vložené do zprávy kvůli zarovnání (padding) mají nedefinovanou hodnotu.

- CDR definuje reprezentaci všech datových typů specifikovaných v IDL.

5.8.2 Formáty zpráv

GIOP definuje 8 typů zpráv, které jsou uvedeny v tabulce 5.8.2

Tabulka 2: Typy GIOP zpráv.

Název	Hodnota	Přenáší
Request	0	invokace operací, atributů
Reply	1	návratové hodnoty, výstupní parametry, výjimky
Cancel Request	2	indikuje, že klient již neočekává odpověď na zprávu Request
Locate request	3	dotazy na objektové reference
Locate reply	4	odpověď na Locate request
Close connection	5	uzavření spojení
Message error	6	chyba ve formátu zprávy
Fragment	7	rozdělení zprávy na menší části (fragmentace)

Každá GIOP zpráva má definovanou hlavičku. Obsah hlavičky je uveden v tabulce 3.

Tabulka 3: Hlavička GIOP zprávy.

Pole	Délka (bajty)	Význam
magic	4	4 ASCII znaky "GIOP", indikují GIOP protokol
version	2	verze protokolu
flags	1	příznaky určující pořadí bajtů a zda je zpráva fragmentovaná
message_type	1	typ zprávy, viz tabulka výše
size	4	délka zprávy

Za hlavičkou pak ve zprávě následují hlavičky specifické pro jednotlivé typy zpráv a jejich data.

Od verze 1.1 podporuje GIOP rozdělení zpráv na fragmenty, díky tomu je možné posílat i dlouhé zprávy. GIOP verze 1.2 přidává obousměrnou komunikaci (což umožňuje obejít některé problémy s firewally).

5.8.3 Transport Assumptions

Definuje požadavky na transportní vrstvu. GIOP vyžaduje transportní protokol, který :

- navazuje spojení, GIOP zprávy jsou platné v kontextu spojení.
- je spolehlivý, tedy zaručuje doručení bajtů ve stejném pořadí jako byly odeslány a poskytuje způsob, jak potvrdit doručení zprávy.

V praxi tyto požadavky splňuje hlavní transportní protokol internetu - TCP/IP. Implementace GIOP nad TCP/IP se nazývá IIOP a je prakticky jediným protokolem používaným pro komunikaci mezi ORB.

Protokol IIOP používá také RMI-IIOP, implementace Java RMI nad IIOP. RMI-IIOP je používáno v Enterprise JavaBeans.

5.9 Další funkce

Do předchozího přehledu hlavních vlastností CORBy se nevešly některé pokročilejší funkce. Krátce je zmíním nyní :

- *Portable interceptors* umožňují modifikovat zpracování objektových referencí a volání v POA.
- *CORBA Component Model*, aplikační framework podobný Enterprise Java Beans.
- *Security Service* přidává bezpečnostní prvky.
- *Event Service*, asynchronní notifikační služba formou push/pull kanálů.
- *Transaction Service*, rozhraní pro transakce.

5.10 C++ mapování

Mapování mezi CORBA a C++ je detailně popsáno ve specifikaci C++ Language Mapping.

Při implementaci je třeba mít na paměti tyto aspekty C++ mapování :

- `module` se překládá jako `namespace`.
- `interface` se překládá jako abstraktní třída.
- Pro objektové reference jsou definovány dva typy. Při použití typu s příponou `_var` je reference automaticky uvolněna, při použití přípony `_ptr` není.
- Metoda `_duplicate` kopíruje objektovou reference, `_release` ji uvolňuje. Voláním `is_nil` se zjišťuje, zda je reference nulová. Zúžení na konkrétnější interface se provádí pomocí `_narrow`.

- Správa paměti. Vstupní parametry typu `in` jsou uvolněny po skončení operace, pokud mají být používány i poté, musí být vytvořena kopie. Výstupní parametry typu `out` musí byt alokovány uvnitř operace.
- Vlákna. Pokud daný ORB podporuje vlákna, mohou být operace vyvolávány současně. Pokud jsou používána sdílená data, je třeba zajistit synchronizaci vláken (pomocí zámků atd.)
- Před dealokováním servanta je třeba ho deaktivovat v POA.
- Objektovou referenci servanta získáme metodou `_this()`.

5.11 Dostupné implementace ORB

Existuje mnoho dostupných implementací CORBy, lišících se v podporovaných verzích specifikace, podporovaných programovacích jazycích, atd. Přehled implementací CORBy lze najít v [16] (není ovšem zcela aktuální).

5.11.1 Komerční

- **VisiBroker** firmy Borland, kompatibilita s CORBA 3.0. Podporuje jazyky Java, C++ a C#.
- **ORBacus, Orbix** : produkty společnosti IONA Technologies. ORBacus je jednodušší, Orbix je určen pro enterprise nasazení.

5.11.2 Volně dostupné

- **OmniORB** byl původně vyvíjen v laboratořích Olivetti Research, později AT&T. OmniORB je volně k použití pod licencí LGPL. OmniORB je kompatibilní se specifikací CORBA 2.6, IIOP 1.2, poskytuje mapování pro jazyky C++ a Pyhon. Runtime plně podporuje vlákna. OmniORB lze používat na široké škále platforem, od Windows, Linux až po komerční UNIX systémy. Lze zakoupit komerční podporu.
- **ACE ORB (TAO)** Kompatibilní se specifikací CORBA 3.0, mapování pro jazyk C++. TAO je zaměřen na real-time aplikace.
- **ORBit** je kompatibilní s CORBA 2.4. Byl určen jako middleware pro desktopové prostředí GNOME. Později byl nahrazen řešením postaveným na DBUS (distribuovaná technologie speciálně určená pro komunikaci desktop aplikací na jednom stroji). ORBit již není aktivně vyvíjen.
- **MICO** je implementace původně vyvinutá na Frankfurtské univerzitě. Nyní je vyvíjena společností ObjectSecurity Ltd. Předností MICO je silná podpora bezpečnostních mechanizmů.

Díky rozšířenosti, dobrým výsledků ve výkonnostních testech [11] a přehledné dokumentaci jsem se rozhodl pro svou implementaci v knihovně OOSol použít implementaci omniORB.

```
// definice modulu (namespace) pro systemy vesmirne lodi
module Starship {
  // priklad deklarace struktury
  struct SystemInformation {
    double avaiableEnergy;
    boolean emergencyMode;
  };

  // deklarace vyjimky
  exception NotEnoughEnergyException {double energyNeeded};

  // interface pro lodni systemy
  interface ShipSystem {
    void setEmergencyMode(in boolean mode);
    oneway void shutdown();
    // metoda ktera vraci strukturu
    SystemInformation getSystemInformation();
    string getStatusReport();
  };

  // interface pro system podporu zivota dedi z
  // obecneho rozhrani lodniho systemu
  interface LifeSupportSystem : ShipSystem {
    // priklad atributu
    attribute double temperature;
    attribute double oxygenLevel;
    // atribut pouze pro cteni
    readonly attribute double moisture;
  };

  interface WarpDrive : ShipSystem {
    // priklad metody, ktera muze vyvolat vyjimku
    void setSpeed(in double speed) raises (
        NotEnoughEnergyException);
    double getSpeed();
    void stop();
    void start();
    void ejectCore();
  };

  interface PhaserWeapon : ShipSystem {
    long getIntesity();
    void setIntesity(in long intensity);
    // priklad vstupnevystupniho parametru
    void fire(in double x, in double y, in double z, inout
        double targetDamage) raises (
        NotEnoughEnergyException);
  };

};
```

Výpis 1: Příklad interface v IDL.

```
1 // priklady komplexnich datovych typu
2 module ComplexTypes {
3   enum Barva {zelena, zluta, cervena};
4
5   struct Zaznam {
6     string jmeno;
7     string prijmeni;
8   }
9
10  union Union switch (short) {
11    case 1: string stringFormat;
12    case 2: long digitalFormat;
13    default: Zaznam structFormat;
14  };
15
16  typedef sequence<String> StringSequence;
17  typedef double[4][4] TransfMatrix;
18
19  interface Demo {
20    void writeStringList(in StringSequence list);
21    void applyTransformationMatrix(in TransfMatrix mat);
22  };
23 };
```

Výpis 2: Komplexní datové typy v IDL.

6 Implementace v knihovně OOSol

6.1 Zařazení do třídní hierarchie

Systém distribuovaných objektů jsem se snažil co nejpřirozeněji zařadit do existující třídní hierarchie knihovny OOSol. Vzhledem k tomu, že zároveň s touto implementací byla vyvíjena MPI verze, snažil jsem se třídy pro CORBA implementaci vytvářet podobně jako MPI třídy, aby bylo možné snadněji zaměňovat CORBA a MPI implementace.

CORBA i MPI používá společnou třídu `OOSblockPattern` pro uložení informací o blokové matici.

Funkce specifické pro technologii CORBA jsou umístěny ve třídě `OOScorba` (obrázek 7). Jsou to tyto statické metody :

- `init` : spustí ORB, získá referenci na RootPOA a NameService a aktivuje POA.
- `finish` : ukončí ORB.
- `activateObject` : aktivace servanta v POA.
- `deactivateObject` : deaktivace servanta v POA.
- `bindObjectToName` : zaregistruje objekt v jmenné službě pod daným jménem.
- `resolveObjectName` : získá z jmenné služby objektovou referenci.

Třídy, implementující distribuované datové struktury blokových matic, `OOSmatrixBlockCorbaDist` a `OOSmatrixBlockDiagCorbaDist`, dědí z abstraktní třídy pro matice `OOSmatrix` (obrázek 7). Díky tomu lze distribuované matice použít v libovolném algoritmu, který pracuje s obecnou maticí `OOSmatrix`. Blokové CORBA matice navíc dědí z abstraktní třídy `OOSIcorbaDistributable`. Pro srovnání je na obrázku 9 třídní diagram MPI implementace.

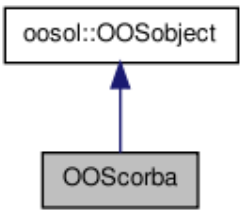
Třídy, jež jsou používany interně pro potřeby CORBy, tedy servant třídy dědící z abstraktní třídy `OOSIcorbaDistributable` (viz obrázek 8), jsou umístěny v samostatném jmenném prostoru `oosolCorba`.

6.2 Architektura manager/worker

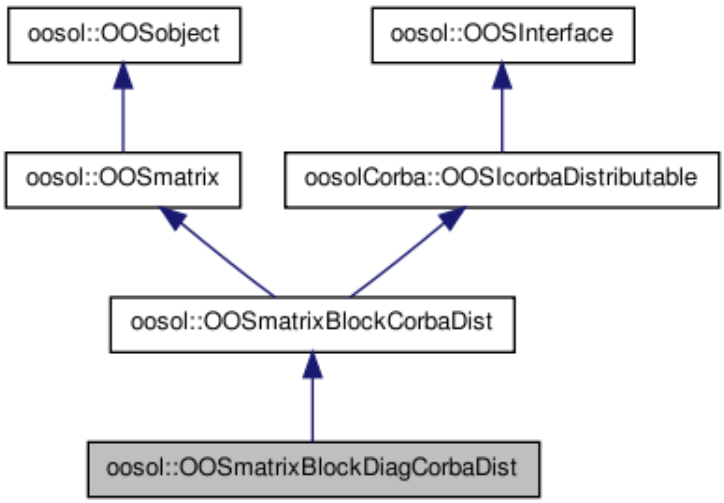
Celý výpočetní systém je navržen v architektuře manager/worker, také nazývané master/slave atp. V tomto textu používám české pojmy správce a dělník.

Úkolem správce je udržovat reference na dělníky, zajišťovat jejich registraci a případné odpojení. Správce také zajišťuje koordinaci asynchronních operací. Správce je spouštěn jako instance v rámci hlavního programu, obvykle na řídícím uzlu daného výpočetního clusteru. Hlavní program obsahuje všechny algoritmy.

Distribuované datové struktury blokových matic jsou také instanciovány v hlavním programu a obsahují objektové reference na jednotlivé bloky, které jsou instanciovány na jednotlivých dělnících.



Obrázek 6: Třídní diagram třídy `OOScorba`.



Obrázek 7: Třídní diagram třídy blokových distribuovaných matic.

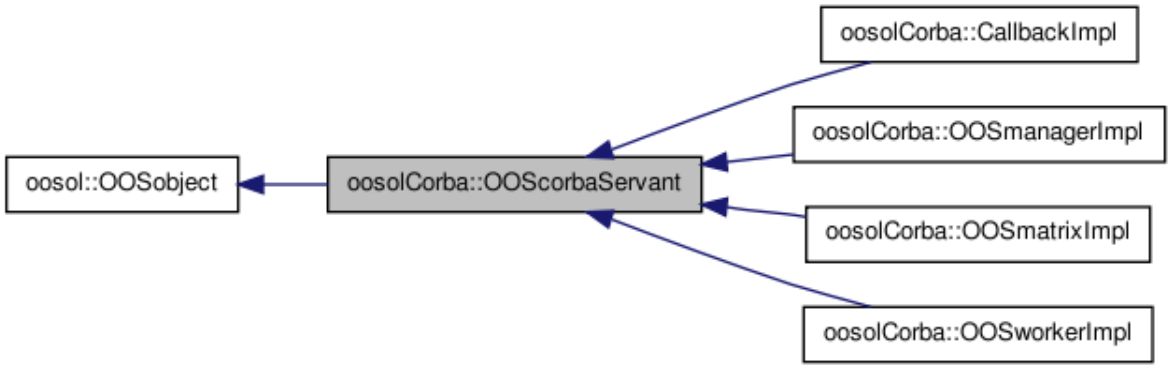
Dělník je základní výpočetní jednotka, je vždy zaregistrován u správce. Dělník sám o sobě neprovádí žádnou činnost, operace je vždy zahájena vzdáleným voláním správce. Dělník umí vytvářet základní distribuované objekty (v současné implementaci pouze matice) a zpřístupňovat je správci.

Správce i dělník mají své IDL rozhraní `OOSmanager` a `OOSworker`.

Dělníci jsou pak spouštěni na jednotlivých výpočetních uzlech, obvykle jeden dělník na jeden procesor. Celé prostředí je znázorněno na obrázku 10.

Distribuovaný výpočet probíhá v těchto krocích :

1. Spustíme hlavní program, který má provádět distribuovaný výpočet.
2. Tento hlavní program si vytvoří instanci objektu `OOSmanagerImpl`, tedy implementaci správce, zaregistruje ji v POA a v Naming Service pod názvem "`OOSmanager`".
3. Určitým spouštěcím mechanismem jsou spuštěny programy s instancemi dělníků. Dělníci si přes CORBA Naming Service získají referenci na správce a zaregistrují se u něho.



Obrázek 8: Třídní diagram pro servant třídy v knihovně OOSol.

4. Správce počká, dokud není zaregistrován dostatečný počet dělníků, voláním operace `waitForWorkers`.

5. V hlavním programu vytvoříme distribuovanou datovou strukturu, například blokovou matici.

6. Distribuovaná datová struktura (například bloková matice) je používána v algoritmu. Algoritmus volá její metody, ty jsou implementovány tak, že operaci (například násobení matice vektorem) rozdělí na operace s prvky distribuované datové struktury (například jednotlivé bloky matice), které jsou spuštěny v rámci CORBY na jednotlivých procesorech.

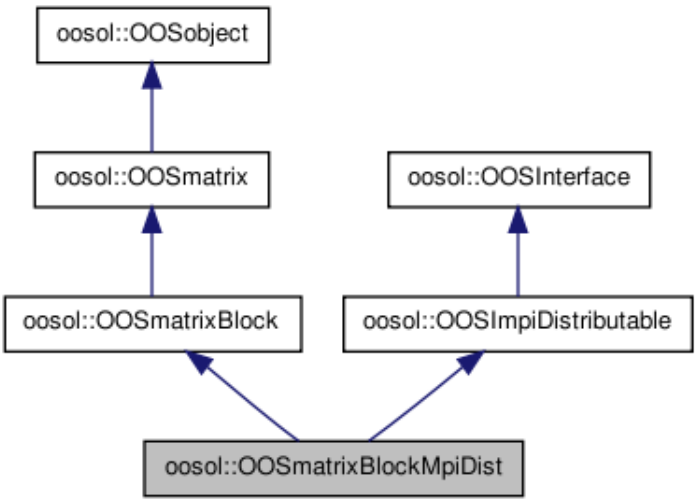
7. Hlavní program končí, správce zavolá na dělnících operaci `shutdown()`, čímž jim nařídí deaktivaci rozhraní a ukončení procesu.

Tento model je možné dále rozšiřovat, například zavést více správců, vytvořit různé typy dělníků, nebo dělníky dynamicky přidělovat během výpočtu.

6.3 IDL rozhraní

V souboru `OOSol.idl` jsem deklaroval namespace `oosolCorba` a v něm tato rozhraní :

- Datový typ `TIND` typu `long` pro indexy.
- Datovy typ `DoubleSeq` typu `sequence<double>` pro pole double čísel.
- Rozhraní `Callback` pro synchronizaci asynchronních operací.
- Rozhraní `OOSmatrix` pro obecné matice. Maticové operace jsou deklarovány podle rozhraní `oosol::OOSmatrix`, definované v OOSolu jako abstraktní třída.



Obrázek 9: Třídní diagram třídy implementující blokové matice v MPI.

- Rozhraní `OOSworker` pro dělníky. Momentálně obsahuje dvě operace : `createMatrix` vytvoří matici daného typu, `shutdown` ukončí dělníka.
- Rozhraní `OOSmanager` pro správce. Dělníci se registrují voláním operace `registerWorker`. Operacemi `getWorkerCount, getWorkers, getWorker` lze zjišťovat počet, seznam a objektové reference dělníků. Operace `shutdownWorkers` způsobí ukončení procesů všech dělníků.

6.4 Implementace servantů

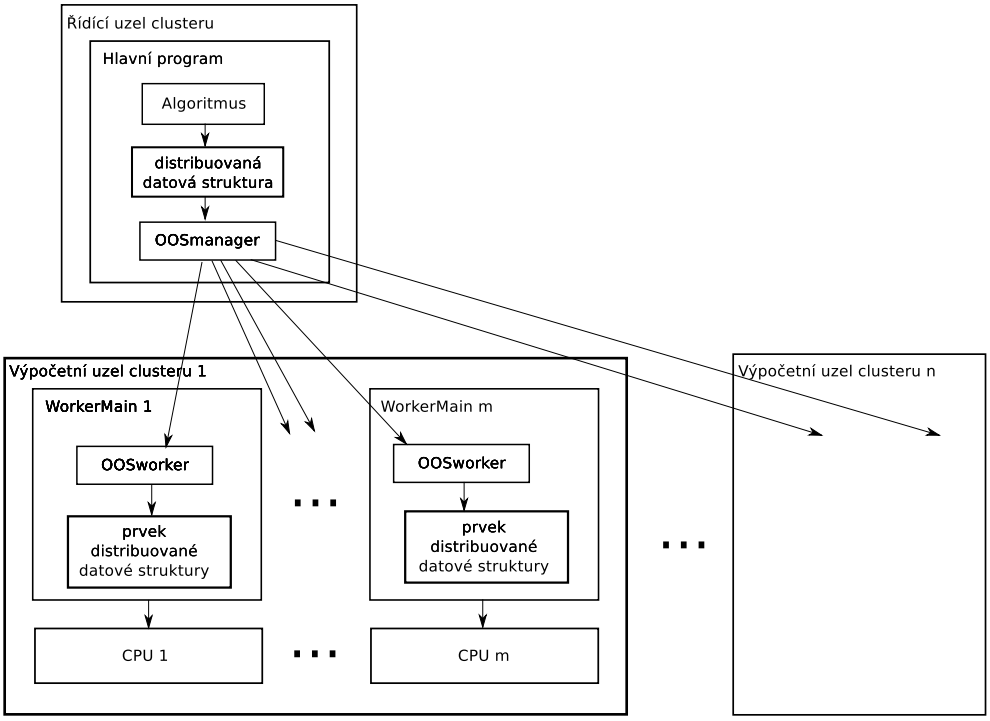
Servant třídy jsou implementovány pomocí dědičnosti rozšířením skeleton tříd generovaných IDL kompilerem. Implementují jednotlivé operace deklarované v IDL rozhraní. Aktivace a deaktivace objektů v POA je řešena v konstruktorech, respektive destruktorech.

6.5 Asynchronní volání

Jistým problémem v implementaci je potřeba asynchronních volání. Standardní volání metod v CORBA pomocí SII jsou synchronní. To znamená, že vlákno volající vzdálenou metodu je blokováno, dokud vzdálený objekt nevrátí výsledek. Volání metod více objektů tak probíhá sekvenčně.

V praxi ale často potřebujeme vyslat požadavek na nějakou operaci na mnoho objektů paralelně a pak shromáždit výsledky. Příkladem je operace mv, kde chceme všem blokům matice paralelně poslat příslušnou část vektoru, nechat je paralelně počítat operaci mv a pak sestavit výsledek.

Implementace asynchronních volání v CORBA je možná několika způsoby :



Obrázek 10: Použití modelu manager/worker v knihovně OOSol.

- DII (Dynamic Invocation Interface) je CORBA interface, který umožňuje dynamicky volat metody a také i asynchronní volání pomocí `send_deferred`. Vzhledem k tomu, že je primárně určeno k dynamické invokaci metod (tedy i takových, které nejsou známy v době překladu), použití DII vyžaduje ručně sestavovat požadavky volání metod a dosazovat jednotlivé parametry. To značně prodlužuje a znepřehledňuje kód, navíc otevírá prostor k chybám, neboť obchází typovou kontrolu.

  Pro příklad uvedu jednoduché volání pomocí SII :

```
1  obj->echoString(”Hello!”);
```

  a pomocí DII :

```
1  CORBA::String_var arg = (const char*) ”Hello!”;
2  CORBA::Request_var req = obj->_request(”echoString”);
3  req->add_in_arg() <<= arg;
4  req->set_return_type(CORBA::_tc_string);
5
6  req->send_deferred();
7
8  if( req->env()->exception() ) {
9    return;
```

```
10    }
```

- Client thread. Další možností je na straně klienta (programu, který má využívat asynchronních volání) vytvořit samostatné vlákno pro každé volání. Hlavní vlákno tak může pokračovat nezávisle a případně si zjišťovat status jednotlivých vláken. Nevýhodou tohoto přístupu je omezená škálovatelnost. Náklady na spuštění a běh vlákna nejsou zanedbatelné a u rozsáhlého distribuovaného systému by správce musel mít spuštěno tisíce vláken.
- Server thread. Podobně jako předchozí možnost, jen je situace opačná. Na straně objektu, který chceme asynchronně volat, vytvoříme metodu, která vytvoří vlákno provádějící výpočet, a původní metoda se ukončí. Po skončení výpočtu je volající objekt notifikován pomocí zpětného volání (callback). Oproti předchozímu řešení není problém se škálovatelností, je však třeba na straně dělníka vytvořit systém pro správu těchto vláken a implementaci vláken provádějící různé typy operací.
- AMI (Asynchronous Messaging Interface). AMI je model asynchronního volání specifikovaný v novějších verzích CORBA. IDL compiler automaticky vytvoří ke každé synchronní metodě dvě další asynchronní varianty, s předponami sendc_ a sendp_. První varianta umožňuje asynchronní volání s callback notifikací, druhá s pomocí polling objektu. Bohužel, ne všechny implementace CORBA podporují AMI (nepodporuje ho například omniORB).
- Oneway calls. Standardní IDL umožňuje definovat metody jako *oneway*, to znamená, že nemají návratovou hodnotu a volající nečeká na její dokončení. Notifikaci o dokončení výpočtu pak provedeme voláním na určený Callback objekt. Dle specifikace nemusí být oneway volání vždy úspešné, protože volající nedostane potvrzení o jeho doručení. Teoreticky se může stát, že požadavek na oneway volání bude ztracen. Vzhledem k tomu, že pro komunikaci mezi ORBy se používá protokol IIOP nad TCP/IP, o doručení se postará transportní síťová vrstva.

Ve své implementaci, která používá omniORB, jsem z důvodu absence AMI použil metodu oneway volání. Funkcionalita je podobná jako u AMI, ale tato metoda funguje i u implementací ORB, které AMI nepodporují.

6.6 Callback

Pro synchronizaci asynchronních oneway operací jsem vytvořil interface `Callback`. `Callback` obsahuje tyto operace :

- `startOperation` značí zahájení asynchronní operace pro daný počet dělníků.
- `operationDone` volají dělníci, když dokončí asynchronní operaci.
- `waitForOperationsDone` čeká, dokud nejsou dokončeny všechny asynchronní operace.

Použití `Callback` pro asynchronní operace je následující :

1. Vytvořím v hlavním programu `Callback` servanta, zavolám na něm `startOperation` s požadovaným počtem operací jako parametrem.

2. Volám na rozhraní jednotlivých dělníků `oneway` asynchronní operace, objektovou referenci na `Callback` objektu uvedu v parametrech volání.

3. Jakmile dělník dokončí operaci, zavolá `operationDone` na `Callback` objektové referenci.

4. V hlavním programu zavolám `waitForOperationsDone`, tím počkám až požadovaný počet dělníků dokončí operaci.

Vnitřní čítač počtu dokončených operací je synchronizován mutexem, aby nedošlo ke kolizi vláken, které do něho současně zapisují.

Toto schéma asynchronních operací je použito například v metodách `mv, load` distribuované blokové matice.

6.7 Distribuované blokové matice

Třída `OOSblockCorbaDist` implementuje distribuovanou blokovou matici pomocí technologie CORBA. Třída dědí z abstraktního rozhraní `OOSmatrix`. Díky tomu je možné ji v rámci OOSolu použít v libovolné metodě, která pracuje nad obecnou maticí `OOSmatrix`.

Datová struktura odpovídá blokové matici

$$A = \begin{bmatrix} B_{1,1} & \dots & B_{1,l} \\ B_{2,1} & & \\ \vdots & \ddots & \vdots \\ & & B_{k-1,l} \\ B_{k,1} & \dots & B_{k,l} \end{bmatrix}$$

Kde k je počet bloků ve sloupci a l počet bloků v řádku. $B_{i,j}$ je blok matice na pozici i, j, který je uložený na $i + jk$-tém dělníkovi.

`OOSblockCorbaDist` získává v parametru konstruktoru odkaz na rozhraní `OOSmanager`, z něj si získá reference na kl dělníků, kde kl je počet bloků matice. Dále se v konstruktoru pomocí třídy `OOSblockPattern` předává informace o velikosti jednotlivých bloků. `OOSblockCorbaDist` v konstruktoru zavolá na každém dělníkovi `createMatrix`, čímž vytvoří jednotlivé bloky. `OOSblockCorbaDist` si udržuje seznam referencí na jednotlivé bloky.

`OOSblockCorbaDist` je v podstatě wrapper mezi OOSolem a CORBOu. Samotná třída neobsahuje žádná data matice, pouze reference na jednotlivé bloky uložené na jiných počítačích. Pokud je zavolána metoda této třídy, například `mv` pro násobení vektorem, operace se provede na jednotlivých blocích. Správce vrací sestavený výsledek.

6.8 Distribuované blokové diagonální matice

`OOSblockCorbaDiagDist`, dědící z `OOSblockCorbaDist` je speciální případ blokové matice, u níž jsou bloky pouze na diagonále :

$$A = \begin{bmatrix} B_{1,1} & & & \\ & B_{2,2} & & \\ & & \ddots & \\ & & & B_{n,n} \end{bmatrix}$$

`OOSblockCorbaDiagDist` se od `OOSblockCorbaDist` odlišuje pouze jiným indexováním bloků.

Struktura diagonální blokové matice je dobře škálovatelná, neboť operace se provádějí na n blocích, zatímco obecná bloková matice musí provést n^2 operací s jednotlivými bloky.

6.9 Operace mv, mtv

Operace `mv, mtv` jsou standardní BLAS Level 2 rutiny, které provádí násobení matice vektorem podle předpisu

$$y = \beta y + \alpha A x$$

respektive

$$y = \beta y + \alpha A^T x$$

kde A je matice o r řádcích a c sloupcích, x je vektor délky c, y je vektor délky r.

Pro provedení operace nad blokovou maticí musíme rozdělit vektory x, y na bloky tak, aby bylo možné násobení provést blokově :

$$y = \beta \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ \\ y_k \end{bmatrix} + \alpha \begin{bmatrix} B_{1,1} & \dots & B_{1,l} \\ B_{2,1} & & \\ \vdots & \ddots & \vdots \\ & & B_{k-1,l} \\ B_{k,1} & \dots & B_{k,l} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \\ x_l \end{bmatrix} = \begin{bmatrix} \beta y_1 + \alpha(B_{1,1}x_1 + B_{1,2}x_2 + \ldots + B_{1,l}x_l) \\ \beta y_2 + \alpha(B_{2,1}x_1 + B_{2,2}x_2 + \ldots + B_{2,l}x_l) \\ \\ \dots \\ \\ \beta y_k + \alpha(B_{k,1}x_1 + B_{k,2}x_2 + \ldots + B_{k,l}x_l) \end{bmatrix}$$

kde x_i je i-tý blok vektoru x, jeho dimenze je stejná jako počet sloupců v bloku matice $B_{m,i}$. y_i je i-tý blok vektoru y, jeho dimenze je stejná jako počet řádků v bloku matice $B_{i,m}$.

Tímto jsme úlohu násobení matice rozdělili na kl operací nad jednotlivými bloky.

Ve třídě `OOSblockCorbaDist` je operace `mv` implementována takto :

- Je vytvořen Callback servant.
- Na každém bloku matice je zavolána operace `mv` s příslušnými částmi vektoru.
- Čeká se, dokud nejsou v Callbacku všechny operace označeny jako hotové.

- Správce operací `getResultVect` stáhne výsledky jednotlivých dělníku, po řádcích je sečte a vrátí výsledek.

Operace `mtv` je implementována stejně, pouze je provedena transpozice : blok $B_{i,j}$ je nahrazen blokem $B_{j,i}$ a místo operace `mv` se na jednotlivých blocích provádí operace `mtv`.

Obdobná je implementace pro diagonální matici `OOSblockDiagCorbaDist`, kde stačí násobit pouze diagonální bloky.

6.10 Metoda sdružených gradientů

Knihovna OOSol již obsahovala běžně známý algoritmus sdružených algoritmů (CG) v sekvenční verzi pro jeden procesor ve třídě `OOScg`. Algoritmus sdružených algoritmů řeší úlohu :

$$\min \frac{1}{2} x^T A x - x^T b$$

pro dané A, b kde A je symetrická pozitivně definitní a b je vektor.

V konstruktoru `OOScg` je matice A typu `OOSmatrix &`. Díky tomu, jak byla provedena implementace v CORBě, je možné jako matici A použít instanci `OOSblockCorbaDist` nebo `OOSblockCorbaDist`. Výpočet pak probíhá nad distribuovanou maticí stejně jako nad lokální bez jakékoliv změny.

6.11 Spouštěcí systém

Spouštěcí systém implementace v CORBě může být libovolný. Jediný požadavek v současné implementaci je, aby dělníci byli spuštěni až tehdy, kdy je správce zaregistrován v Naming Service.

Pro účely testování jsem vytvořil jednoduchý spouštěcí script `corbarun`, který se používá podobně jako `mpirun` obvykle používaný v MPI. Tento skript má syntaxi :

```
corbarun -np N [--hostfile FILE] PROGRAM
```

Kde :

- N je počet dělníku plus jedna,
- FILE je soubor s adresami počítačů, na kterých se mají spouštět dělníci. Musí to být textový soubor, na každém řádku jedna adresa. Jedna adresa může být uvedena vícekrát (pokud má počítač více procesorů.) Pokud není hostfile určen, corbarun se pokusí použít `$PBS_NODEFILE` plánovače PBS.
- PROGRAM je hlavní program se správcem.

`corbarun` nejprve spustí na současném stroji hlavní program se správcem a poté pomocí SSH (nástroje pro vzdálený příkazový řádek) spustí dělníky na strojích uvedených v hostfile.

Pro vynucené ukončení jsem vytvořil skript `corbakill`, který má stejnou syntaxi jako `corbarun`. `corbakill` ukončí hlavní program i dělníky na všech uzlech.

Bylo by možné použít i jiné spouštěcí systémy, například nechat dělníky bežet neustále, nezávisle na hlavním programu - jako služby.

7 Testování

7.1 Způsob testování

7.1.1 Metodika testování

Čas každého testu byl měřen pro deset spuštění. Jako směrodatnou hodnotu jsem použil minimum z těchto deseti časů, abych potlačil vnější vlivy jako například procesy jiných uživatelů, systémové procesy spouštěné na pozadí atd.

Testy probíhaly na nevytížených systémech, v případě clusterů SPC VŠB-TU Ostrava byly výpočty spouštěny pomocí plánovače úloh PBS.

Čas je měřen jako reálný čas pomocí funkce `gettimeofday()`, měřen je pouze čas volání metody řešiče `solve()` - nezahrnuje načtení matice atp.

Všechny programy byly zkompilovány s úrovní optimalizace `-O2`. Jako ORB byl ve všech testech použit omniORB 4.1.

7.1.2 Použité výpočetní prostředky

Testování probíhalo především na výpočetních clusterech Superpočítačového centra VŠB-TU Ostrava.

Cluster COMSIO je linuxový cluster skládající se z 8 výpočetních uzlů a jednoho řídícího. Každý výpočetní uzel má :

- 4 procesory AMD Opteron (2 $\times$ dual core)
- 8 GB RAM

Celkem je tedy možno použít až 32 procesorů. V testech bylo použito síťové rozhraní gigabitového ethernetu.

7.2 Úloha pro metodu sdružených gradientů

Algoritmus sdružených algoritmů byl testován na modelové poissonově úloze s laplaceovým operátorem a homogenní pravou stranou :

$$-\Delta u = -1$$

Tato úloha určuje průhyb 2D membrány na oblasti $\Omega = (0,1)\times(0,1)$. Na levé straně $x = 0$ je předepsaná dirichletova okrajová podmínka $u = 0$. Diskretizací metodou konečných prvků dostaneme úlohu :

$$\min \frac{1}{2}x^T Ax - x^T b$$

kde A je symetrická pozitivně definitní matice tuhosti, b vektor pravých stran. Tuto úlohu lze řešit metodou sdružených gradientů.

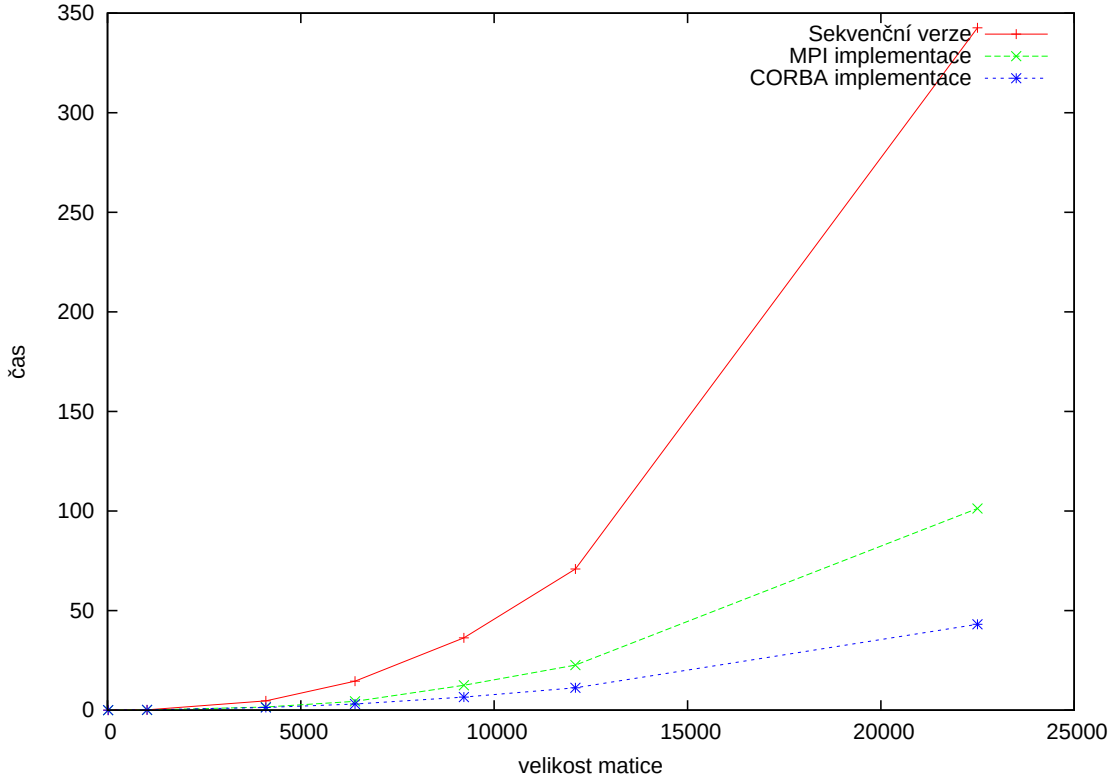
7.3 Testy CG

7.3.1 Test 1 - konstantní počet procesorů, plné matice

V tomto testu jsem měřil dobu výpočtu v závislosti na velikosti matice při konstantním počtu 16 procesorů. Test proběhl na clusteru COMSIO. Testované velikosti matic byly 16, 1024, 4096, 6400, 9216, 12100 a 22500. Jednotlivé bloky byly uloženy jako plné matice.

Graf času výpočtu v závislosti na velikosti matice pro sekvenční verzi, MPI implementaci a CORBA implementaci je na obrázku 11. Hodnoty jsou v tabulce 4.

Vidíme, že jak MPI, tak CORBA implementace, urychlují čas výpočtu oproti sekvenční verzi na jednom procesoru. Překvapivé je, že pro větší matice je CORBA implementace znatelně rychlejší.



Obrázek 11: Graf testu 1, čas výpočtu v závislosti na velikost matice pro sekvenční verzi, MPI implementaci a CORBA implementaci.

Velikost matice	Sekvenční	MPI	CORBA
16	0	0,01	0,01
1024	0,15	0,09	0,19
4096	4,64	1,56	1,34
6400	14,52	4,43	3,1
9216	36,33	12,52	6,57
12100	70,85	22,63	11,25
22500	342,64	101,23	43,08

Tabulka 4: Tabulka časů řešiče pro test 1.

Počet procesorů	CORBA implementace	Ideální škálovatelnost
1	156,9	156,9
2	100,93	78,45
4	52,3	39,23
8	32,39	19,61
16	21,21	9,81
32	16,01	4,9

Tabulka 5: Tabulka časů řešiče pro test 2.

7.3.2 Test 2 - test škálovatelnosti

Cílem tohoto testu bylo ověřit škálovatelnost distribuovaného algoritmu. Velikost matice byla konstantní, 16384×16384, jednotlivé bloky matice uložené jako plné matice, počet procesorů 1 až 32 na clusteru COMSIO.

Graf času v závislosti na počtu procesorů v logaritmickém měřítku je na obrázku 12, v lineárním měřítku na obrázku 13.

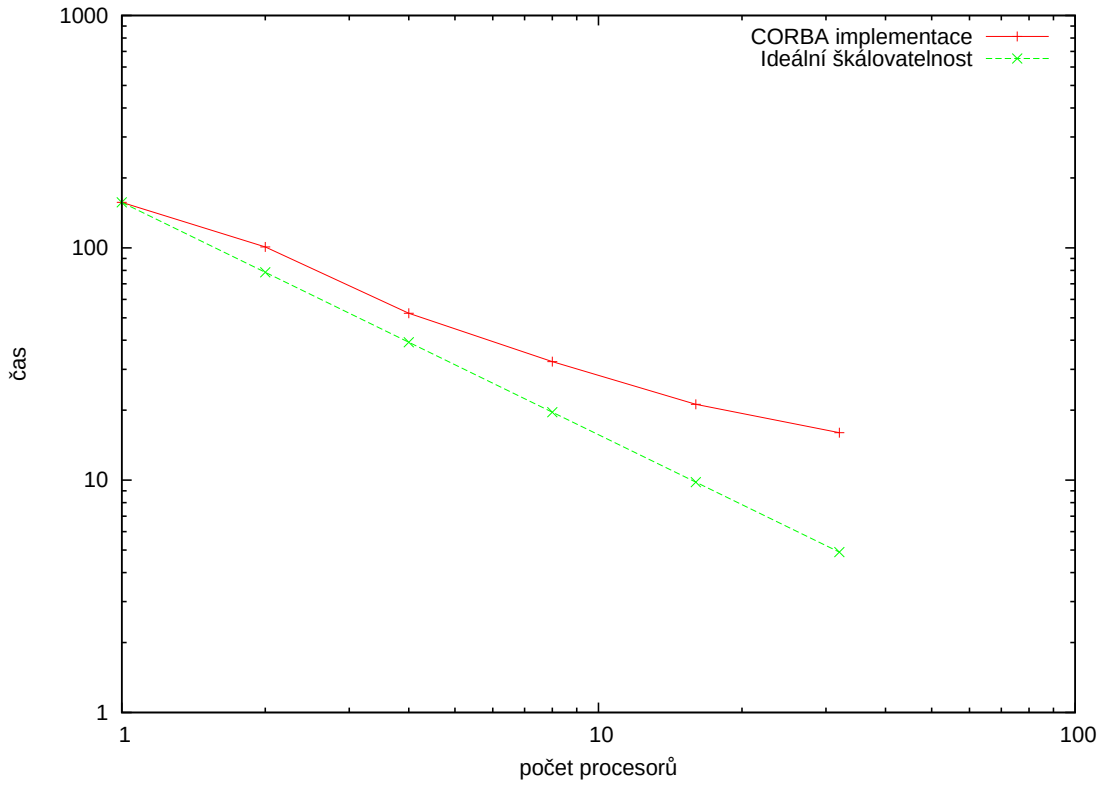
7.4 Porovnání s MPI implementací

Měl jsem možnost porovnat implementaci OOSolu pomocí CORBA, kterou popisuji v této práci, se samostatně vyvíjenou implementací využívající MPI. Obě implementace jsou funkčně téměř shodné, líší se ale způsobem naprogramování. Rozdíly mezi MPI a CORBA implementacemi OOSolu dávají náhled na rozdílnost programátorského přístupu při použití MPI a CORBy.

7.4.1 Přehlednost kódu

Model manager-worker se v MPI implementaci dodržuje spíše jenom „ideově“, program je identický pro správce i dělníky, pouze se v některých operacích kód rozvětvuje na operace pro správce a dělníka.

V CORBě je model manager-worker definován přímo pomocí rozhraní. Programy správce a dělníka jsou zcela oddělené a komunikují pouze přes toto rozhraní.

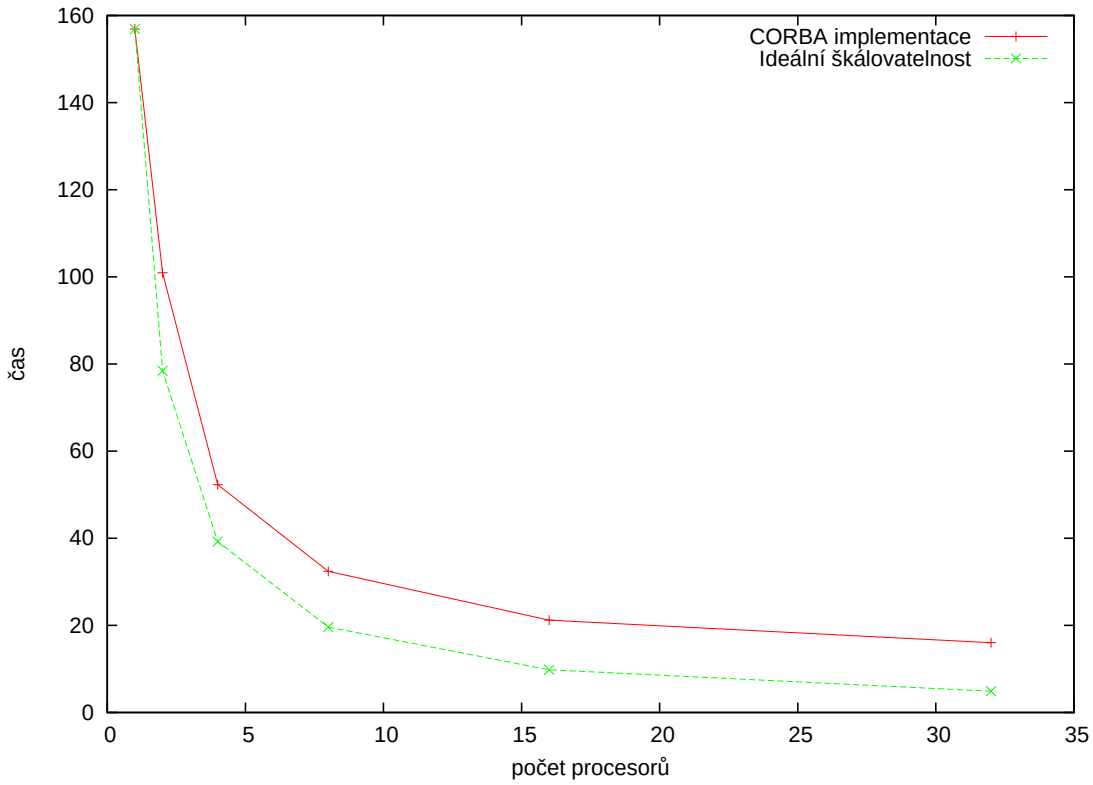


Obrázek 12: Graf testu 2, čas výpočtu v závislosti na počtu procesorů pro CORBA implementaci ve srovnání s ideální škálovatelností. Osy jsou v logaritmickém měřítku.

Programátor, který používá MPI verzi OOSolu, musí pracovat s tím, že jeho program se spouští na všech uzlech zároveň. Části kódu, které se spouští pouze na určitém uzlu, musí vždy oddělit větvením, například takto :

```
1 OOSmatrix* bigMatrix;
2 if (OOSmpi::Rank() == 0) {
3    // vytvorime matici pouze na ridicim uzlu
4   bigMatrix = new OOSfullMatrix(10000, 10000);
5 } else {
6    //
7    bigMatrix = NULL;
8 }
```

Kvůli rozsahu platnosti proměnných v C++ musí být proměnné deklarované ve společné části kódu, a zajistit větvením jejich inicializaci na určitých uzlech a na ostatních uzlech hodnotu NULL. Tyto proměnné se musí kontrolovat i na jiných místech v kódu, což vede k jeho nepřehlednosti.



Obrázek 13: Graf testu 2, čas výpočtu v závislosti na počtu procesorů pro CORBA implementaci ve srovnání s ideální škálovatelností. Osy jsou v lineárním měřítku.

Programátor využívající CORBA verzi OOSolu píše pouze hlavní program, a přes rozhraní správce dává úkoly dělníkům. Program dělníků se nemění, pro různé hlavní programy i algoritmy se používá stejný program dělníka.

V MPI se míchá dohromady kód pro komunikaci, výpočty i distribuci. V CORBě jsou jednotlivé části odděleny na úrovni rozhraní.

7.4.2 Možnosti rozšiřování a změn

Díky objektovému návrhu je možné CORBA implementaci snadno rozšiřovat o novou funkcionalitu přidáváním metod a rozhraní v IDL.

V MPI implementaci je velká provázanost komunikačního kódu v programu, implementace nové funkcionality je složitější.

CORBA implementaci lze snadněji používat v jiných programech a knihovnách, neboť pro její provoz stačí pouze dynamicky (případně staticky) linkovaná knihovna ORB.

Externí použití MPI implementace je složitější, neboť MPI programy se musí spouštět speciálně pomocí `mpiexec` v prostředí, kde je MPI nakonfigurováno.

7.4.3 Implementace algoritmů

V MPI implementaci není možné použít existující sekvenční verze algoritmů, neboť je nutné zajistit synchronizaci řídících částí algoritmu mezi všemi procesy. Pro implementaci algoritmů v MPI tak musely být vytvořeny jejich samostatné verze pro MPI do nových tříd `OOScgParMpi, OOSmprgpParMpi`.

V CORBě je možné použít existující algoritmy bez jakékoliv změny, neboť algoritmus běží pouze v hlavním programu a dělníci pouze vykonávají pouze operace zavolané správcem, bez jakékoliv informace o algoritmu.

7.4.4 Výkon

Přestože jsem očekával, že výkon CORBA implementace bude slabší než MPI, praktické testy ukázaly, že výkon CORBA implementace na testovaných úlohách je srovnatelný, dokonce lepší než implementace v MPI.

Test 2 ukazuje dobrou škálovatelnost na modelovém případě.

Domnívám se, že větší rychlost CORBA implementace oproti MPI je dána tím, že v MPI implementaci jsou použity synchronní blokující funkce `MPI_Send` a `MPI_Recv`, zatímco v CORBě jsou použity asynchronní oneway operace, které neblokují hlavní program.

Navíc, omniORB zpracovává požadavky ve vláknech, díky tomu může CORBA efektivně komunikovat s několika procesy zároveň a lépe tak využít síťová rozhraní. Například v clusteru COMSIO má každý uzel 4 síťová rozhraní.

8 Závěr

8.1 Dosažení stanovených cílů

Podařilo se mi splnit všechny cíle, které jsem si stanovil v podkapitole 1.3. Implementace distribuovaného OOSolu pomocí CORBy je :

- Zařazená v třídní hierarchii.
- Implementovaná v ANSI C++.
- Přenositelná na jakoukoliv platformu, kde je podporován omniORB (operační systémy Windows, Linux, Solaris, Mac OS X, HP-UX, Irix, AIX, Tru64, OpenVMS a další)
- Přehledná a snadno upravitelná.
- Objektově orientovaná.
- Používá otevřené technologie CORBA a implementaci omniORB, která je volně dostupná pod licencí LGPL.
- Paralelně škálovatelná.

8.2 Přínos této práce

Přínos této diplomové práce je shrnut v těchto bodech :

- Práce popisuje použití distribuované objektové technologie CORBA v rámci obecné knihovny pro numerické metody a algoritmy. Tento přístup k paralelnímu programovaní zatím není příliš obvyklý. V této práci popisuji výhody distribuovaně objektového přístupu oproti často používané technologii MPI.
- Distribuované datové struktury se podařilo implementovat tak, že je možné paralelizovat existující sekvenční verze algoritmů bez jejich změny, pouze použitím distribuované datové struktury.
- Implementace v knihovně OOSol je paralelně škálovatelná.
- Podařilo se vytvořit přenositelné, objektově orientované distribuované výpočetní rozhraní, které může být dále rozvíjeno.

9 Literatura

[1] VONDRÁK, Vít. *OOSol Website*, URL: `http://am.vsb.cz/oosol/` [cit. 2010-3-1], 2005.

[2] GRISBY, Duncan - LO, Sai-Lai - RIDDOCH, David. *The omniORB version 4.1 User's Guide*, URL: `http://omniorb.sourceforge.net/omni41/omniORB.pdf` [cit. 2010-3-1], 2006.

[3] HENNING, Michi. *The Rise and Fall of CORBA*, URL: `http://queue.acm.org/detail.cfm?id=1142044` [cit. 2010-3-1], 2006.

[4] IONA Technologies PLC. *Orbix Programmer's Guide C++ Edition*, URL: `http://www.iona.com/support/docs/orbix/gen3/33/html/orbix33cxx_pguide/index.html` [cit. 2010-3-1], 2000.

[5] MageLang Institute. *Introduction to CORBA*, URL: `http://java.sun.com/developer/onlineTraining/corba/corba.html` [cit. 2010-3-1], 1999.

[6] MCHALE, Ciaran. *CORBA EXPLAINED SIMPLY*, URL: `http://www.CiaranMcHale.com/corba-explained-simply` [cit. 2010-3-1], 2007.

[7] SCHMIDT, Douglas C. - VINOSKI, Steve. Object Interconnections : An Introduction to CORBA Messaging, *SIGS C++ Report Magazine*, 1998, November/December issue.

[8] TŮMA, Petr - BUBLE, Adam. *Technical Report on Open CORBA Benchmarking*. URL: `http://d3s.mff.cuni.cz/~bench/pkg/Benchmarking-Techreport-200101.pdf` [cit. 2010-3-1], 2001.

[9] GRYGÁREK, Petr. *Studijní materiály k předmětu Distribuované objektové systémy.* URL: `http://www.cs.vsb.cz/grygarek/dosys/` [cit. 2010-3-1], 2003.

[10] PYARALI, Irfan - SCHMIDT, Douglas C.. An Overview of the CORBA Portable Object Adapter, *ACM StandartView magazine*, 1998, Volume 6, Issue 1, Dostupné z: `http://www.cs.wustl.edu/~schmidt/PDF/POA.pdf` [cit. 2010-3-1], 1998.

[11] DENIS, Alexandre - PÉREZ, Christian - PRIOL, Thierry. Towards High Performance CORBA and MPI Middlewares for Grid Computing, *Proceedings of the Second International Workshop on Grid Computing*, 2001, s. 14-25, Dostupné z: http://www.cs.wustl.edu/ schmidt/PDF/POA.pdf. ISBN:3-540-42949-2.

[12] DENIS, Alexandre - PÉREZ, Christian - PRIOL, Thierry - RIBES, André. *Padico: A Component-Based Software Infrastructure for Grid Computing*. Research Report RR-4974, INRIA, 2003.

[13] GUDAITIS, Michael - NELSON, Curtis. *PERFORMANCE ASSESSMENT OF SOFTWARE INTERFACES FOR THE SOFTWARE COMMUNICATIONS ARCHITECTURE (SCA)*. Dostupné z: `http://groups.sdrforum.org/download.php?sid=1124` [cit. 2010-3-1], 2007.

[14] ES-SQUALI, T. - FLEURY, E., GUYHARD, J. BHIRI, S. Evaluating the Performance of CORBA for Distributed and Grid Computing Applications. *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, s. 288, 2001.

[15] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1*. Dostupné z: `http://www.omg.org/spec/CORBA/3.1` [cit. 2010-3-1], 2008.

[16] PUDER, Arno. *CORBA Product Profiles*. Dostupné z: `http://www.puder.org/corba/matrix/` [cit. 2010-3-1], 2004.