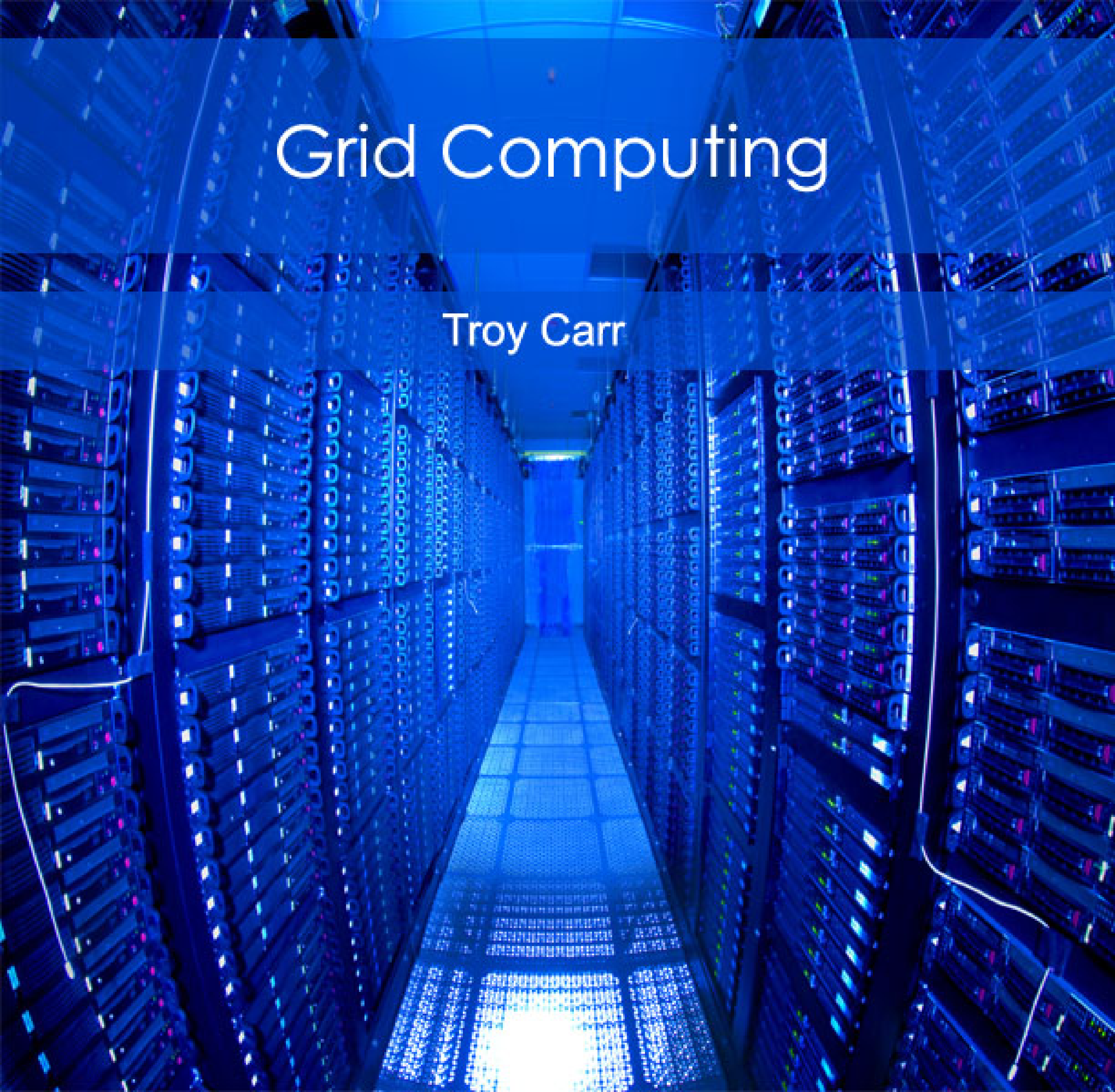
Grid Computing

Troy Carr



First Edition, 2012

ISBN 978-81-323-3059-2

Published by:
Research World
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Grid Computing

Grid computing is a term referring to the combination of computer resources from multiple administrative domains to reach a common goal. The **grid** can be thought of as a distributed system with non-interactive workloads that involve a large number of files. What distinguishes grid computing from conventional high performance computing systems such as cluster computing is that grids tend to be more loosely coupled, heterogeneous, and geographically dispersed. Although a grid can be dedicated to a specialized application, it is more common that a single grid will be used for a variety of different purposes. Grids are often constructed with the aid of general-purpose grid software libraries known as middleware.

Grid size can vary by a considerable amount. Grids are a form of distributed computing whereby a "super virtual computer" is composed of many networked loosely coupled computers acting together to perform very large tasks. Furthermore, "distributed" or "grid" computing, in general, is a special type of parallel computing that relies on complete computers (with onboard CPUs, storage, power supplies, network interfaces, etc.) connected to a network (private, public or the Internet) by a conventional network interface, such as Ethernet. This is in contrast to the traditional notion of a supercomputer, which has many processors connected by a local high-speed computer bus.

Overview

Grid computing combines computers from multiple administrative domains to reach a common goal, to solve a single task, and may then disappear just as quickly.

One of the main strategies of grid computing is to use middleware to divide and apportion pieces of a program among several computers, sometimes up to many thousands. Grid computing involves computation in a distributed fashion, which may also involve the aggregation of large-scale cluster computing-based systems.

The size of a grid may vary from small—confined to a network of computer workstations within a corporation, for example—to large, public collaborations across many companies and networks. "The notion of a confined grid may also be known as an intra-nodes cooperation whilst the notion of a larger, wider grid may thus refer to an inter-nodes cooperation".

Grids are a form of distributed computing whereby a "super virtual computer" is composed of many networked loosely coupled computers acting together to perform very large tasks. This technology has been applied to computationally intensive scientific, mathematical, and academic problems through volunteer computing, and it is used in commercial enterprises for such diverse applications as drug discovery, economic forecasting, seismic analysis, and back office data processing in support for e-commerce and Web services.

Comparison of grids and conventional supercomputers

"Distributed" or "grid" computing in general is a special type of parallel computing that relies on complete computers (with onboard CPUs, storage, power supplies, network interfaces, etc.) connected to a network (private, public or the Internet) by a conventional network interface, such as Ethernet. This is in contrast to the traditional notion of a supercomputer, which has many processors connected by a local high-speed computer bus.

The primary advantage of distributed computing is that each node can be purchased as commodity hardware, which, when combined, can produce a similar computing resource as multiprocessor supercomputer, but at a lower cost. This is due to the economies of scale of producing commodity hardware, compared to the lower efficiency of designing and constructing a small number of custom supercomputers. The primary performance disadvantage is that the various processors and local storage areas do not have high-speed connections. This arrangement is thus well-suited to applications in which multiple parallel computations can take place independently, without the need to communicate intermediate results between processors. The high-end scalability of geographically dispersed grids is generally favorable, due to the low need for connectivity between nodes relative to the capacity of the public Internet.

There are also some differences in programming and deployment. It can be costly and difficult to write programs that can run in the environment of a supercomputer, which may have a custom operating system, or require the program to address concurrency issues. If a problem can be adequately parallelized, a "thin" layer of "grid" infrastructure can allow conventional, standalone programs, given a different part of the same problem, to run on multiple machines. This makes it possible to write and debug on a single conventional machine, and eliminates complications due to multiple instances of the same program running in the same shared memory and storage space at the same time.

Design considerations and variations

One feature of distributed grids is that they can be formed from computing resources belonging to multiple individuals or organizations (known as multiple administrative domains). This can facilitate commercial transactions, as in utility computing, or make it easier to assemble volunteer computing networks.

One disadvantage of this feature is that the computers which are actually performing the calculations might not be entirely trustworthy. The designers of the system must thus introduce measures to prevent malfunctions or malicious participants from producing false, misleading, or erroneous results, and from using the system as an attack vector. This often involves assigning work randomly to different nodes (presumably with different owners) and checking that at least two different nodes report the same answer for a given work unit. Discrepancies would identify malfunctioning and malicious nodes.

Due to the lack of central control over the hardware, there is no way to guarantee that nodes will not drop out of the network at random times. Some nodes (like laptops or dialup Internet customers) may also be available for computation but not network communications for unpredictable periods. These variations can be accommodated by assigning large work units (thus reducing the need for continuous network connectivity) and reassigning work units when a given node fails to report its results in expected time.

The impacts of trust and availability on performance and development difficulty can influence the choice of whether to deploy onto a dedicated computer cluster, to idle machines internal to the developing organization, or to an open external network of volunteers or contractors. In many cases, the participating nodes must trust the central system not to abuse the access that is being granted, by interfering with the operation of other programs, mangling stored information, transmitting private data, or creating new security holes. Other systems employ measures to reduce the amount of trust “client” nodes must place in the central system such as placing applications in virtual machines.
___ Public systems or those crossing administrative domains (including different departments in the same organization) often result in the need to run on heterogeneous systems, using different operating systems and hardware architectures. With many languages, there is a trade off between investment in software development and the number of platforms that can be supported (and thus the size of the resulting network). Cross-platform languages can reduce the need to make this trade off, though potentially at the expense of high performance on any given node (due to run-time interpretation or lack of optimization for the particular platform).

Various middleware projects have created generic infrastructure to allow diverse scientific and commercial projects to harness a particular associated grid or for the purpose of setting up new grids.

In fact, the middleware can be seen as a layer between the hardware and the software. On top of the middleware, a number of technical areas have to be considered, and these may or may not be middleware independent. Example areas include SLA management, Trust and Security, Virtual organization management, License Management, Portals and Data Management. These technical areas may be taken care of in a commercial solution, though the cutting edge of each area is often found within specific research projects examining the field.

Market segmentation of the grid computing market

For the segmentation of the grid computing market, two perspectives need to be considered: the provider side and the user side:

The provider side

The overall grid market comprises several specific markets. These are the grid middleware market, the market for grid-enabled applications, the utility computing market, and the software-as-a-service (SaaS) market.

Grid middleware is a specific software product, which enables the sharing of heterogeneous resources, and Virtual Organizations. It is installed and integrated into the existing infrastructure of the involved company or companies, and provides a special layer placed among the heterogeneous infrastructure and the specific user applications. Major grid middlewares are Globus Toolkit, gLite, and UNICORE.

Utility computing is referred to as the provision of grid computing and applications as service either as an open grid utility or as a hosting solution for one organization or a VO. Major players in the utility computing market are Sun Microsystems, IBM, and HP.

Grid-enabled applications are specific software applications that can utilize grid infrastructure. This is made possible by the use of grid middleware, as pointed out above.

Software as a service (SaaS) is “software that is owned, delivered and managed remotely by one or more providers.” (Gartner 2007) Additionally, SaaS applications are based on a single set of common code and data definitions. They are consumed in a one-to-many model, and SaaS uses a Pay As You Go (PAYG) model or a subscription model that is based on usage. Providers of SaaS do not necessarily own the computing resources themselves, which are required to run their SaaS. Therefore, SaaS providers may draw upon the utility computing market. The utility computing market provides computing resources for SaaS providers.

The user side

For companies on the demand or user side of the grid computing market, the different segments have significant implications for their IT deployment strategy. The IT deployment strategy as well as the type of IT investments made are relevant aspects for potential grid users and play an important role for grid adoption.

CPU scavenging

CPU-scavenging, **cycle-scavenging**, **cycle stealing**, or **shared computing** creates a "grid" from the unused resources in a network of participants (whether worldwide or internal to an organization). Typically this technique uses desktop computer instruction cycles that would otherwise be wasted at night, during lunch, or even in the scattered seconds throughout the day when the computer is waiting for user input or slow devices.

Many Volunteer computing projects, such as BOINC, use the CPU scavenging model.

In practice, participating computers also donate some supporting amount of disk storage space, RAM, and network bandwidth, in addition to raw CPU power. Heat produced by CPU power in rooms with many computers can be used for fine heating premises. Since nodes are likely to go "offline" from time to time, as their owners use their resources for their primary purpose, this model must be designed to handle such contingencies.

History

The term *grid computing* originated in the early 1990s as a metaphor for making computer power as easy to access as an electric power grid in Ian Foster's and Carl Kesselman's seminal work, "The Grid: Blueprint for a new computing infrastructure" (2004).

CPU scavenging and volunteer computing were popularized beginning in 1997 by distributed.net and later in 1999 by SETI@home to harness the power of networked PCs worldwide, in order to solve CPU-intensive research problems.

The ideas of the grid (including those from distributed computing, object-oriented programming, and Web services) were brought together by Ian Foster, Carl Kesselman, and Steve Tuecke, widely regarded as the "fathers of the grid". They led the effort to create the Globus Toolkit incorporating not just computation management but also storage management, security provisioning, data movement, monitoring, and a toolkit for developing additional services based on the same infrastructure, including agreement negotiation, notification mechanisms, trigger services, and information aggregation. While the Globus Toolkit remains the de facto standard for building grid solutions, a number of other tools have been built that answer some subset of services needed to create an enterprise or global grid.

In 2007 the term cloud computing came into popularity, which is conceptually similar to the canonical Foster definition of grid computing (in terms of computing resources being consumed as electricity is from the power grid). Indeed, grid computing is often (but not always) associated with the delivery of cloud computing systems as exemplified by the AppLogic system from 3tera.

Fastest virtual supercomputers

- BOINC – 5.634 PFLOPS as of April 4th 2011.
- Folding@Home – 5 PFLOPS, as of March 17, 2009
- As of April 2010, MilkyWay@Home computes at over 1.6 PFLOPS, with a large amount of this work coming from GPUs.
- As of April 2010, SETI@Home computes data averages more than 730 TFLOPS.
- As of April 2010, Einstein@Home is crunching more than 210 TFLOPS.
- As of April 2010, GIMPS is sustaining 44 TFLOPS.

Current projects and applications

Grids computing offer a way to solve Grand Challenge problems such as protein folding, financial modeling, earthquake simulation, and climate/weather modeling. Grids offer a way of using the information technology resources optimally inside an organization. They also provide a means for offering information technology as a utility for commercial and noncommercial clients, with those clients paying only for what they use, as with electricity or water.

Grid computing is being applied by the National Science Foundation's National Technology Grid, NASA's Information Power Grid, Pratt & Whitney, Bristol-Myers Squibb Co., and American Express.

One of the most famous cycle-scavenging networks is SETI@home, which was using more than 3 million computers to achieve 23.37 sustained teraflops (979 lifetime teraflops) as of September 2001.

As of August 2009 Folding@home achieves more than 4 petaflops on over 350,000 machines.

The European Union has been a major proponent of grid computing. Many projects have been funded through the framework programme of the European Commission. Many of the projects are highlighted below, but two deserve special mention: BEinGRID and Enabling Grids for E-sciencE.

BEinGRID (Business Experiments in Grid) is a research project partly funded by the European commission as an Integrated Project under the Sixth Framework Programme (FP6) sponsorship program. Started on June 1, 2006, the project will run 42 months, until November 2009. The project is coordinated by Atos Origin. According to the project fact sheet, their mission is "to establish effective routes to foster the adoption of grid computing across the EU and to stimulate research into innovative business models using Grid technologies". To extract best practice and common themes from the experimental implementations, two groups of consultants are analyzing a series of pilots, one technical, one business. The project is significant not only for its long duration, but also for its budget, which at 24.8 million Euros, is the largest of any FP6 integrated project. Of this, 15.7 million is provided by the European commission and the remainder by its 98 contributing partner companies.

The Enabling Grids for E-sciencE project, which is based in the European Union and includes sites in Asia and the United States, is a follow-up project to the European DataGrid (EDG) and is arguably the largest computing grid on the planet. This, along with the LHC Computing Grid (LCG), has been developed to support the experiments using the CERN Large Hadron Collider. The LCG project is driven by CERN's need to handle huge amounts of data, where storage rates of several gigabytes per second (10 petabytes per year) are required. A list of active sites participating within LCG can be found online as can real time monitoring of the EGEE infrastructure. The relevant software and documentation is also publicly accessible. There is speculation that dedicated fiber optic links, such as those installed by CERN to address the LCG's data-intensive needs, may one day be available to home users thereby providing internet services at speeds up to 10,000 times faster than a traditional broadband connection.

Another well-known project is distributed.net, which was started in 1997 and has run a number of successful projects in its history.

The NASA Advanced Supercomputing facility (NAS) has run genetic algorithms using the Condor cycle scavenger running on about 350 Sun and SGI workstations.

In 2001, United Devices operated the United Devices Cancer Research Project based on its Grid MP product, which cycle-scavenges on volunteer PCs connected to the Internet. The project ran on about 3.1 million machines before its close in 2007.

As of 2011, over 6.2 million machines running the open-source Berkeley Open Infrastructure for Network Computing (BOINC) platform are members of the World Community Grid, which tops the processing power of the current fastest supercomputer system (China's Tianhe-I).

Definitions

Today there are many definitions of *grid computing*:

- In his article “What is the Grid? A Three Point Checklist”, Ian Foster lists these primary attributes:
    - Computing resources are not administered centrally.
    - Open standards are used.
    - Nontrivial quality of service is achieved.

- Plaszczak/Wellner define grid technology as "the technology that enables resource virtualization, on-demand provisioning, and service (resource) sharing between organizations."
- IBM defines grid computing as “the ability, using a set of open standards and protocols, to gain access to applications and data, processing power, storage

capacity and a vast array of other computing resources over the Internet. A grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of resources distributed across 'multiple' administrative domains based on their (resources) availability, capacity, performance, cost and users' quality-of-service requirements".

- An earlier example of the notion of computing as utility was in 1965 by MIT's Fernando Corbató. Corbató and the other designers of the Multics operating system envisioned a computer facility operating "like a power company or water company".
- Buyya/Venugopal define grid as "a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed autonomous resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements".
- CERN, one of the largest users of grid technology, talk of **The Grid**: "a service for sharing computer power and data storage capacity over the Internet."

Grids can be categorized with a three stage model of departmental grids, enterprise grids and global grids. These correspond to a firm initially utilising resources within a single group i.e. an engineering department connecting desktop machines, clusters and equipment. This progresses to enterprise grids where nontechnical staff's computing resources can be used for cycle-stealing and storage. A global grid is a connection of enterprise and departmental grids that can be used in a commercial or collaborative manner.

Chapter 2

D-Grid

The **D-Grid Initiative** (German Grid Initiative) builds a sustainable grid infrastructure for education and research (e-Science) in Germany. The D-Grid infrastructure will help to establish methods of e-Science in three core areas:

- Grid Computing
- Knowledge Management
- e-Learning

D-Grid started September 1, 2005 and consists of six Community Projects and an integration project (DGI) as well as several partner projects.

Integration Project

The D-Grid integration project is a part of the D-Grid Initiative and has to integrate all developments from the different community projects in one common D-Grid platform. The D-Grid integration project will act as a Grid resource and service provider for the science community in Germany. The project office is located at the Institute for Scientific Computing (IWR) at Forschungszentrum Karlsruhe. The resources to ensure a sustainable Grid infrastructure are provided by four work packages:

- Work Package 1: **D-Grid Base-Software**. The major task of this work package is to provide several different middleware packages. These are the Globus Toolkit, UNICORE, LCG/gLite, GridSphere and the Grid Application Toolkit (GAT). The community projects linked together in the D-Grid integration project are supported during the installation, operation and if needed and possible the customisation of the Base-Software.
- Work Package 2: **Deployment and operation of the D-Grid infrastructure**. Work package 2 builds up a Core-D-Grid. This will be used as a prototype to test the operational functionality of the system. This work package also deals with the challenges of Monitoring, Accounting and Billing of the Grid resources.

- Work Package 3: **Networks and Security**. The network infrastructure in D-Grid is based on the DFN Wissenschaftsnetz X-WiN. Work package 3 will provide extensions to the existing network infrastructure according to the needs of Grid middleware used in D-Grid. Further tasks are to build an AA-Infrastructure in D-Grid, develop firewall concepts for Grid environments and set up Grid specific CERT services.
- Work Package 4: **D-Grid project office**. The work package is responsible for the integration of the deliverables from the Grid integration project and the deliverables from the different community projects in one common D-Grid platform. Work package 4 also deals with the challenge of archiving sustainability in D-Grid and grid-based e-Science systems generally.

Communities

Currently there are six community projects participating the D-Grid Initiative:

AstroGrid-D

AstroGrid-D, also referred to as the German Astronomy Community Grid (GACG), is a joint research project of thirteen astronomical institutes and grid-oriented computer science groups, supported by supercomputing centers. The main objective of AstroGrid-D is the integration of German research facilities into a unified nationwide research infrastructure in the field of astronomy. The goal is to improve the efficiency and usability of hardware and software resources including computer clusters, astronomical data archives, and observational facilities such as robotic telescopes. AstroGrid-D supports the standards of the International Virtual Observatory Alliance (IVOA) and cooperates closely with international projects on grid development.

AstroGrid-D is managed by the Astrophysical Institute Potsdam (AIP).

C3-Grid

At the Collaborative Climate Community Data and Processing Grid (C3-Grid) scientific researchers are trying to understand the earth system including their subsystems like oceans, atmosphere and biosphere. For the last decades the amount of data has increased enormously in the field of climate research. On the one hand, due to rapid rise in computing power scientists are now able to use models with higher resolution and perform long term simulations. The scientists are able to couple models for the mentioned subsystems in complex cumulative simulations producing petabytes of output which is collected in distributed data archives. On the other hand, monitoring the earth with satellites results in a second huge data stream for climate research. Up to now, no uniform access to these distributed data is available what creates a bottleneck for the scientific research. The C3-Grid proposes to link these distributed data archives.

The management of C3-Grid has the Alfred Wegener Institute for Polar and Marine Research (AWI) in Bremerhaven.

GDI-Grid

The GDI-Grid ("Geodateninfrastrukturen-Grid" - "Spatial Data Infrastructure Grid") project focuses on solutions for efficient integration and processing of geodata based on GIS and SDI technologies. The project will integrate GDI and Grid technologies in a working GDI-Grid infrastructure and thus demonstrate the complementarity of both fields of science. Distributed geospatial data—currently accessed via standardized GIS and SDI services—will build the basis for this endeavour. This data basis can be put to more use by processing it and merging it with other data, creating standards-based, multi-functional generic SDI services.

The project focuses on data, models, services and workflows for spatial data infrastructures. Services for integration, processing and management of spatial data are to be developed and implemented within the D-Grid infrastructure. A proof of concept will be given using a number of representative scenarios such as emergency routing for disaster management, flood simulation and sound propagation simulation.

The project is managed by the University of Hanover at the Regional Computing Center for Lower Saxony (RRZN).

HEP-Grid

The HEP-Grid Project has its focus on High Energy, Nuclear as well as Astroparticle Physics. The main task of HEP-Grid is to optimize the data analysis using distributed computing and storage resources. The project developments are important extensions to the grid middleware from the Enabling Grids for E-scienceE (EGEE) and LHC Computing Grid (LCG) projects. They provide significant improvements for data analysis of experiments currently taking data and for those planned in the future at the Large Hadron Collider (LHC) or the proposed International Linear Collider (ILC).

For the HEP-Grid Deutsches Elektronen Synchrotron (DESY), "German Electron Synchrotron") in Hamburg collaborates with eight German research facilities and universities and a set of associate partners.

InGrid

InGrid is a community project in the field of Grid computing in engineering sciences. InGrid aims to enable engineering projects for grid-based applications and allow for the common, efficient use of common compute and software resources. The flexible use of grid technologies will combine the competences in modeling, simulation and optimization.

Five typical applications (foundry technologies, metal forming technologies, groundwater flow and transport, turbine simulation and fluid-structure interaction) are considered as showcases in order to cover the three central areas of computationally intensive engineering applications, that are coupled multi-scale problems, coupled multi-discipline problems, and distributed simulation-based optimization. In particular adaptive and scalable process models and Grid based runtime environments for these tasks are developed.

Engineering research is inherently application and industry oriented. The support of virtual prototyping and the optimization of scientific engineering operational sequences is therefore an emphasis of the project. The project management for InGrid is provided by the High Performance Computing Center (Höchstleistungsrechenzentrum) Stuttgart (HLRS) of the University of Stuttgart.

MediGRID

The joint project MediGRID unifies well known research institutes in the area of **medicine, biomedical informatics and life sciences** into a consortium. Numerous associated partners from industry, healthcare and research facilities ensure a broad representation of these communities.

The main goal of MediGRID is the development of a Grid middleware integration platform enabling eScience services for biomedical life science. Therefore the consortium allocated the tasks in different modules. The four methodological modules (**middleware, ontology, resource fusion and eScience**) plan to incrementally develop and provide a Grid infrastructure while taking into account the need of the biomedical users. The user communities are represented in three research modules for **biomedical informatics, image processing and clinical research**.

SuGI

SuGI - Sustainable Grid Infrastructure - is a gap project of the German Grid Initiative. Its major task is to disseminate the knowledge of grid technology and to enhance its use. Thus, SuGI addresses all academic computing centers as well as enterprises, which still have not adopted grid technology. They will be supported in providing grid resources and services.

During this project, research experiences gained in the D-Grid projects will be made available to these institutions. Thus, SuGI offers own training courses; attends to external courses, create video and audio recordings and provide these online to the D-Grid communities via a scaling training infrastructure (SuGI-Portal). Further on, SuGI develops training systems for grid middleware, contributes to a simplification of installation and servicing procedures, and works on the development and evaluation of legal and organizational structures.

TextGrid

While grid technologies have first been developed for the natural and the life sciences, there are exciting opportunities for deploying grid technologies and e-Science concepts in other areas as well. TextGrid is a grid project in the humanities, and thereby contributes to the emerging e-Humanities.

It aims to create a grid-based infrastructure for the collaborative editing, annotation, analysis and publication of specialist texts for researchers in philology, linguistics, and related fields. In addition to providing a comprehensive toolset, the project establishes an open platform for other projects to plug into the TextGrid.

ValueGrids

ValueGrids develops an integrated concept and a set of tools for service level management in service value networks. This will enable providers of Software-as-a-Service solutions to utilize grid infrastructures and leverage the German national grid infrastructure.

The ValueGrids project partners are: SAP AG (Coordinator), Conemis AG, IBM Deutschland Research & Development GmbH, University of Freiburg and the Karlsruhe Institute of Technology.

Partner Projects

Several Partner Projects are involved in the D-Grid Initiative.

WISENT

The e-Science project WISENT ("Wissensnetz Energiemeteorologie") has the aim to optimize the cooperation of scientific organizations in the field of Energy Meteorology employing Grid technologies. The main focus of scientific research within Energy Meteorology is the investigation of the influence of weather and climate on transformation, transport, and utilisation of energy.

Funding

On the basis of the D-Grid Initiative more than 100 German research facilities are funded with about 20 billion Euros for a period of 3 years by the German Federal Ministry of Education and Research.

Chapter 3

Dynamic Infrastructure

Dynamic Infrastructure is an information technology paradigm concerning the design of data centers so that the underlying hardware and software can respond dynamically to changing levels of demand in more fundamental and efficient ways than before. The paradigm is also known as *Infrastructure 2.0* and *Next Generation Data Center*.

Top tier vendors promoting dynamic infrastructures include IBM, Microsoft, Sun, Fujitsu, HP and Dell.

The basic premise of Dynamic Infrastructures is to leverage pooled IT resources to provide flexible IT capacity, enabling the seamless, real-time allocation of IT resources in line with demand from business processes. This is achieved by using server virtualization technology to pool computing resources wherever possible, and allocating these resources on-demand using automated tools. This allows for load balancing and is a more efficient approach than keeping massive computing resources in reserve to run tasks that take place, for example, once a month, but are otherwise under-utilized.

Early examples of server-level Dynamic Infrastructures are the FlexFrame for SAP and FlexFrame for Oracle solutions introduced by Fujitsu Siemens Computers (now Fujitsu) in 2003. The FlexFrame approach is to dynamically assign servers to applications on demand, leveling peaks and enabling organizations to maximize the benefit from their IT investments.

Enterprises switching to Dynamic Infrastructures can also reduce costs, improve quality-of-service and make more efficient use of energy through reducing the number of standby or under-utilized machines in their data centers. Instead of the hot spare principle of keeping second servers on standby to replace all production machines in contingencies for hardware- and software-related failures, Dynamic Infrastructures provide for failover from a smaller pool of spare machines. By reducing redundant capacity, organizations are enabled to make more efficient use of their IT budgets and devote greater proportions of their budget to physical and virtual production servers.

Dynamic Infrastructures may also be used to provide security and data protection when workloads are moved during migrations, provisioning, enhancing performance or building co-location facilities.

Potential benefits of Dynamic Infrastructures include enhancing performance, scalability, system availability and uptime, increasing server utilization and the ability to perform routine maintenance on either physical or virtual systems all while minimizing interruption to business operations and reducing cost for IT. Dynamic Infrastructures also provide the fundamental business continuity and high availability requirements to facilitate cloud or grid computing.

Fujitsu's definition: "Dynamic Infrastructures enable customers to assign IT resources dynamically to services as required and to choose sourcing models which best fit their businesses. This brings IT flexibility and efficiency to the next level."

IBM's definition: “A dynamic infrastructure integrates business and IT assets and aligns them with the overall goals of the business while taking a smarter, new and more streamlined approach to helping improve service, reduce cost, and manage risk.”

For networking companies, Infrastructure 2.0 refers to the ability of networks to keep up with the movement and scale requirements of new enterprise IT initiatives, especially virtualization and cloud computing. According to companies like Cisco, F5 Networks and Infoblox, network automation and connectivity intelligence between networks, applications and endpoints will be required to reap the full benefits of virtualization and many types of cloud computing. This will require network management and infrastructure to be consolidated, enabling higher levels of dynamic control and connectivity between networks, systems and endpoints.

Need for a holistic approach

Even in the face of global uncertainty, it is the infrastructure that continues to enable commerce and communications – the roads, networks, utilities, and technologies connecting and differentiating organizations, competitors and customers. The need therefore, is for a new type of infrastructure that:

- Enables visibility, control and automation across all business and IT assets
- Is highly optimized to achieve more with less
- Addresses the information challenge
- Leverages flexible sourcing like clouds
- Manages and mitigates risks

Organizations need an infrastructure that can propel them forward — not hold them back. Until now, many organizations have thought of physical infrastructure and IT infrastructure as separate. This meant, for example, that airports, roadways, buildings, power plants, and oil wells were managed in one way, while datacenters, PCs, cell phones, routers, and broadband devices were managed quite differently.

To succeed in today's world of instrumented, interconnected, and intelligent assets, a new approach is needed. Now, the infrastructure of atoms and the infrastructure of bits are merging into an intelligent, global, dynamic infrastructure. This convergence of business and IT assets requires an infrastructure that can measure and manage the lifecycle of assets that exist beyond the data center, throughout an organization's entire facilities as well as between one organization and another. The range of this approach is broader than ever before, and its effect on organizations is equally far-reaching.

Benefits of having dynamic infrastructures

Dynamic infrastructures take advantage of intelligence gained across the network. By design, every dynamic infrastructure is service-oriented and focused on supporting and enabling the end users in a highly responsive way. It can utilize alternative sourcing approaches, like cloud computing to deliver new services with agility and speed.

Global organizations already have the foundation for a dynamic infrastructure that will bring together the business and IT infrastructure to create new possibilities. For example:

- Transportation companies can optimize their vehicles' routes leveraging GPS and traffic information.
- Facilities organizations can secure access to locations and track the movement of assets by leveraging RFID technology.
- Production environments can monitor and manage presses, valves and assembly equipment through embedded electronics.
- Technology systems can be optimized for energy efficiency, managing spikes in demand, and ensuring disaster recovery readiness.
- Communications companies can better monitor usage by location, user or function, and optimize routing to enhance user experience.
- Utility companies can reduce energy usage with a "smart grid."

Virtualized applications can reduce the cost of testing, packaging and supporting an application by 60%, and they reduced overall TCO by 5% to 7% in our model. – Source: Gartner – "TCO of Traditional Software Distribution vs. Application Virtualization" / Michael A Silver, Terrence Cosgrove, Mark A Margevicious, Brian Gammage / 16 April 2008

While green issues are a primary driver in 10% of current data center outsourcing and hosting initiatives, cost reductions initiatives are a driver 47% of the time and are now aligned well with green goals. Combining the two means that at least 57% of data center outsourcing and hosting initiatives are driven by green. – Source: Gartner – "Green IT Services as a Catalyst for Cost Optimization." / Kurt Potter / 4 December 2008

"By 2013, more than 50% of midsize organizations and more than 75% of large enterprises will implement layered recovery architectures." – Source: Gartner – "Predicts 2009: Business Continuity Management Juggles Standardization, Cost and Outsourcing Risk"). / Roberta J Witty, John P Morency, Dave Russell, Donna Scott, Rober Desisto / 28 January 2009

The key to a business and IT infrastructure that is "dynamic" is leveraging technologies, service delivery and acquisition models that optimize the infrastructure for efficiency and flexibility while transforming management to an automated service delivery and management model.

Chapter 4

Data Format Description Language

Data Format Description Language (DFDL, often pronounced *daff-o-dil*) is a modeling language from the Open Grid Forum for describing general text and binary data. A DFDL model or schema allows any text or binary data to be read (or "parsed") from its native format and to be presented as an instance of an information set. The same DFDL schema also allows data to be taken from an instance of an information set and written out (or "serialized") to its native format.

DFDL achieves this by building upon the facilities of W3C XML Schema 1.0. A subset of XML Schema is used, enough to enable the modeling of non-XML data. One of the results of this is that is very easy to use DFDL to convert general text and binary data, via a DFDL information set, into a corresponding XML document.

It is important to note that DFDL is *descriptive* and not *prescriptive*. DFDL is not a data format, nor does it impose the use of any particular data format. DFDL allows an application to design an appropriate data representation according to its requirements, and for that format to be described in a standard way so that multiple programs can directly interchange the data.

History

DFDL was created in response to a need for grid APIs to be able to understand data regardless of source. A language was needed capable of modeling a wide variety of existing text and binary data formats. A working group was established at the Global Grid Forum (which later became the Open Grid Forum) in 2003 to create a specification for such a language.

A decision was made early on to base the language on a subset of W3C XML Schema, using <xs:appinfo> annotations to carry the extra information necessary to describe non-XML physical representations. This is an established approach that is already being used today in commercial systems. DFDL takes this approach and evolves it into an open standard capable of describing many text or binary data formats.

Work continued on the specification, culminating in the publication of DFDL 1.0 as an OGF Proposed Recommendation in January 2011. A summary of DFDL and its features is available at the OGF site.

Implementations of DFDL processors that can parse and serialize data using DFDL schemas are in progress.

Example

Take as an example the following text data stream which gives the name, age and location of a person:

```
Joe Bloggs,46,Hampshire,England
```

The logical model for this data can be described by the following fragment of an XML Schema document. The order, names, types and cardinality of the fields are expressed by the XML schema model.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" ...>

<xs:complexType name="person_type">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="age" type="xs:short"/>
    <xs:element name="county" type="xs:string"/>
    <xs:element name="country" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

</xs:schema>
```

To additionally model the physical representation of the data stream, DFDL augments the XML schema fragment with annotations on the xs:element and xs:sequence objects, as follows:

```
<xs:schema xmlns:dfdl="http://www.ogf.org/dfdl/dfdl-1.0/"
xmlns:xs="http://www.w3.org/2001/XMLSchema" ...>

<xs:complexType name="person_type">
  <xs:sequence>
    <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
        <dfdl:sequence encoding="ASCII" sequenceKind="ordered"
                       separator="," separatorType="infix"
separatorPolicy="required"/>
    </xs:appinfo></xs:annotation>
    <xs:element name="name" type="xs:string">
      <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
        <dfdl:element lengthKind="delimited" encoding="ASCII"/>
      </xs:appinfo></xs:annotation>
    </xs:element>
    <xs:element name="age" type="xs:short">
      <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
        <dfdl:element representation="text" lengthKind="delimited"
encoding="ASCII"
                      textNumberRep="standard" textNumberPattern="#0"
textNumberBase="10"/>
      </xs:appinfo></xs:annotation>
    </xs:element>
    <xs:element name="county" type="xs:string">
      <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
        <dfdl:element lengthKind="delimited" encoding="ASCII"/>
      </xs:appinfo></xs:annotation>
    </xs:element>
    <xs:element name="country" type="xs:string">
      <xs:annotation><xs:appinfo source="http://www.ogf.org/dfdl/">
        <dfdl:element lengthKind="delimited" encoding="ASCII"/>
      </xs:appinfo></xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>

</xs:schema>
```

The property attributes on these DFDL annotations express that the data are represented in an ASCII text format with fields being of variable length and delimited by commas.

An alternative, more compact syntax is also provided, where DFDL properties are carried as non-native attributes on the XML Schema objects themselves.

```
<xs:schema xmlns:dfdl="http://www.ogf.org/dfdl/dfdl-1.0/"
xmlns:xs="http://www.w3.org/2001/XMLSchema" ...>

<xs:complexType name="person_type">
  <xs:sequence dfdl:encoding="ASCII" dfdl:sequenceKind="ordered"
               dfdl:separator="," dfdl:separatorType="infix"
dfdl:separatorPolicy="required">
    <xs:element name="name" type="xs:string"
                dfdl:lengthKind="delimited" dfdl:encoding="ASCII"/>
    <xs:element name="age" type="xs:short"
                dfdl:representation="text" dfdl:lengthKind="delimited"
dfdl:encoding="ASCII"
                dfdl:textNumberRep="standard"
dfdl:textNumberPattern="##0" dfdl:textNumberBase="10"/>
    <xs:element name="county" type="xs:string"
                dfdl:lengthKind="delimited" dfdl:encoding="ASCII"/>
    <xs:element name="country" type="xs:string"
                dfdl:lengthKind="delimited" dfdl:encoding="ASCII"/>
  </xs:sequence>
</xs:complexType>

</xs:schema>
```

Features

The goal of DFDL is to provide a rich modeling language capable of representing any text or binary data format. The 1.0 release is a major step towards this goal. The capability includes support for:

- Language structures such as COBOL, C and PL/1
- Industry standards such as CSV, SWIFT, FIX, HL7, X12, HL7, HIPAA, EDIFACT, ISO8583
- Data delimited by text or binary markup
- Physical data types including text strings, text numbers, binary two's complement integers, BCD, mainframe zoned and packed decimals, IEEE and mainframe floats, text and binary calendars, text and binary Booleans
- Any encoding and endian-ness
- Bi-directional text
- Bit data of arbitrary length
- Pattern languages for text numbers and calendars
- Ordered and unordered content
- Default values on parsing and serializing
- Nil values capability for handling out-of-band data
- A built-in expression language including variables to model dynamic data
- Mechanisms to resolve choices and optionality
- Fixed and variable arrays
- Hiding elements in the data from the information set
- Calculating element values for the information set
- Validation to XML Schema 1.0 rules
- A scoping mechanism that allows common property values to be applied at multiple annotation points

Future releases are anticipated in which it is hoped to include support for:

- Direct access by offset
- True multi-dimensional arrays
- Embedded comments
- Custom language extensions

Chapter 5

Global Information Grid-Bandwidth Expansion

The Global Information Grid Bandwidth Expansion (GIG-BE) Program was a major United States Department of Defense (DOD) net-centric transformational initiative executed by DISA. GIG-BE created a ubiquitous "bandwidth-available" environment to improve national security intelligence, surveillance and reconnaissance, information assurance, as well as command and control. Through GIG-BE, DISA leveraged DOD's existing end-to-end information transport capabilities, significantly expanding capacity and reliability to select Joint Staff-approved locations worldwide. GIG-BE achieved Full Operational Capability (FOC) on Dec. 20, 2005.

Scope

This program provided increased bandwidth and diverse physical access to approximately 87 critical sites in the continental United States (CONUS), Pacific Theater, and European Theater. These locations are interconnected via an expanded GIG core.

Capabilities and Services

GIG-BE provides a secure, robust, optical terrestrial network that delivers very high-speed classified and unclassified Internet Protocol (IP) services to key operating locations worldwide. The Assistant Secretary of Defense for Networks and Information Integration's (ASD/NII) vision is a "color to every base," physically diverse network access, optical mesh upgrades for the backbone network, and regional/MAN upgrades, where needed. "A color to every base" implies that every site has an OC-192 (10 gigabits per second) of usable IP dedicated to that site.

Implementation

After extensive component integration and operational testing, implementation began in the middle of the 2004 fiscal year and extended through calendar year 2005. The initial implementation concentrated on six sites used during the proof of Initial Operational Capability (IOC), achieved on September 30, 2004. The GIG-BE Program Office conducted detailed site surveys at all of the approximately 87 Joint Staff-approved locations and parallel implementation in CONUS and overseas. The GIG-BE Program Office completed the Final Operational Test and Evaluation (FOT&E) at 54 operational sites on October 7, 2005. On December 20, 2005, the GIG-BE program achieved the milestone of Full Operational Capability.

Contract

GIG-BE was awarded to SAIC in 2001 for $877 million. This contract was for the development, instantiation, and maintenance of the GIG-BE network. SAIC instantly divided the equipment and tasks into subcontracts. These subcontracts are as follows:

- Ciena Corporation- optical transport segment worth $200-$300 million over two years
- Sycamore Networks- optical cross-connect segment worth $100-$150 million
- Cisco Systems- multiservice provisioning platform segment worth $150-$200 million
- Juniper Networks- core IP router portion worth $150-$200 million
- By Light- installation and maintenance worth $100-$150 million

Congressional critic

Representatives Marty Meehan (D-Mass) and Jim Saxton (R-NJ) expressed concern in the selection process for a contractor to lead the GIG-BE effort. They say it was handled "irresponsibly." The reason for this as expressed by Meehan's spokesperson Kimberly Abbott is the "whole bidding process wasn't as fair and open as it could have been", because a contractor and not the government led the decision process. SAIC won the contract who eventually handed a large subset of the engineering to By Light. Meehan did stress that he is **not** questioning the competence of the winners. These two congressman delivered a letter to the DoD regarding the contract but the issue was slowly forgotten.

Chapter 6

Web Services Resource Framework and WEB2GRID

Web Services Resource Framework

Web Services Resource Framework (WSRF) is a family of OASIS-published specifications for web services. Major contributors include the Globus Alliance and IBM.

A web service by itself is nominally stateless, i.e., it retains no data between invocations. This limits the things that can be done with web services, although workarounds exist – such as having the web service read from a database, for example, or using session state by way of cookies or WS-Session.

WSRF provides a set of operations that web services may implement to become stateful; web service clients communicate with *resource* services which allow data to be stored and retrieved. When clients talk to the web service they include the identifier of the specific resource that should be used inside the request, encapsulated within the WS-Addressing endpoint reference. This may be a simple URI address, or it may be complex XML content that helps identify or even fully describe the specific resource in question.

Alongside the notion of an explicit resource reference comes a standardized set of web service operations to get/set resource properties. These can be used to read and perhaps write resource state, in a manner somewhat similar to having member variables of an object alongside its methods. The primary beneficiary of such a model are management tools, which can enumerate and view resources, even if they have no other knowledge of them. This is the basis for WSDM.

Issues with WSRF

WSRF is not without controversy. Most fundamental is architectural: are distributed objects with state and operations the best way to represent remote resources? It is almost a port into XML of the **distributed objects** pattern, of which CORBA and DCOM are examples. A WSRF resource may be a stateful entity to which multiple clients have resource references and the WSRF specification itself does not deal with concerns such as isolation and availability, deferring to the composable nature of web service specifications to deal with these. Many WSRF stacks appear to avoid these concerns by being low-availability, mapping 1:1 from a WSRF resource reference to a local object instance, which in C++ and Java is usually not at all persistent (with the exception of those bound to a database through some persistence mechanism). There are, however, implementations of WSRF that support persistence, clustering and high-availability of resources (for example, in WebSphere Application Server).

With a distributed objects view of the network, WSRF is also at loggerheads with the REST model of the network, in which everything is a resource, but in which all actions are enabled through a limited and standardized set of operations. In some ways, the two models are closer than pure SOAP and REST, because they both have stateful resources at the far end. However, REST, as implemented on HTTP, assumes that the URL is all that is needed to address the resource – there is no need for the complexity of the WS-Addressing ReferenceParameters. The idea of managing the lifetime of remote content through renewable leasing comes in for particular criticism. The other issue with the architecture from the REST community is that callbacks/notifications, as described in WS-Notification, do not go through firewalls. This is why REST designs prefer polling, such as in RSS and Atom (standard) feeds. WSRF has done nothing to make SOAP more acceptable to the REST community.

The introduction of WSRF also caused splits in the WS-* world. It was first announced to the World at a Global Grid Forum event in February 2004, as a successor to the Open Grid Services Infrastructure. Its limited compatibility with the mainstream WS-I architecture created dissent from the UK grid community. The Global Grid Forum ultimately isolated their dependencies on WSRF in a **WSRF profile** for their Open Grid Services Architecture. WSRF protocols were also used by WSDM as the means to interacts with **manageable resources** described in WSDM. The WS-* world, however, was not united on a single standard for Web services management with Microsoft, Sun and others choosing to pursue WS-Management, with its dependency on WS-Transfer as the means to describe manageable resources.

Eventually, in spring 2006, an announcement was made of a planned future convergence between WSDM and WS-Management. This may or may not include all of WSRF. Most likely, many of the more controversial aspects of the technology will either be omitted or made optional.

Component specifications

- **WS-Resource** defines a *WS-Resource* as the composition of a resource and a Web service through which the resource can be accessed.
- **WS-ResourceProperties** describes an interface to associate a set of typed values with a WS-Resource that may be read and manipulated in a standard way.
- **WS-ResourceLifetime** describes an interface to manage the lifetime of a WS-Resource.
- **WS-BaseFaults** describes an extensible mechanism for rich SOAPFaults.
- **WS-ServiceGroup** describes an interface for operating on collections of WS-Resources.

Also of relevance is WS-Notification which says how to push information to other web-services about what is going on.

Implementations

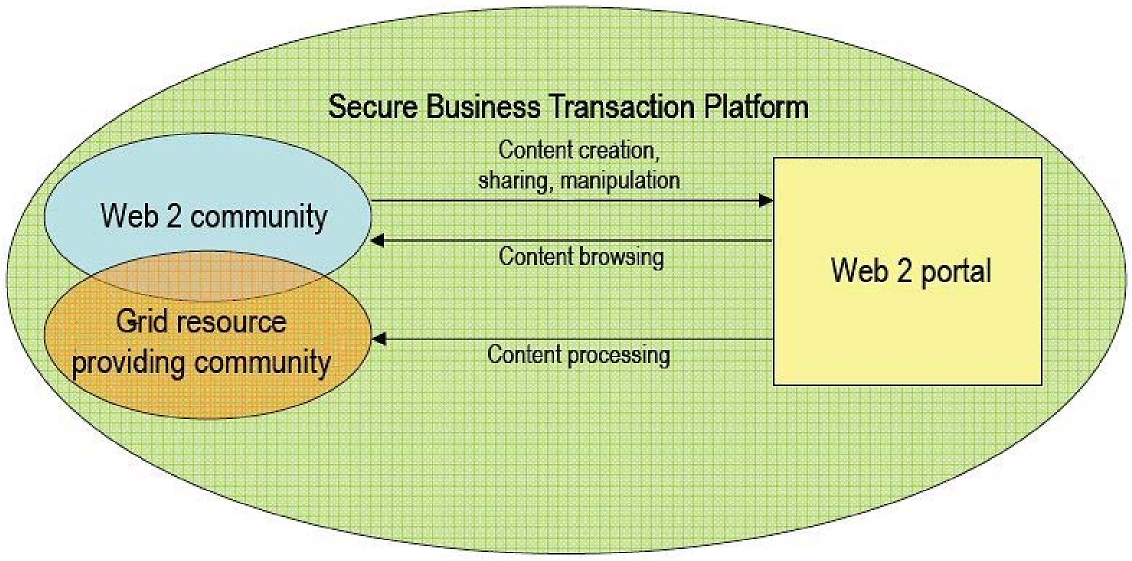
Implementing the basic property get/set semantics of WSRF resources is relatively simple. The hardest problem is probably returning faults as WSRF Base Faults where the specification requires it, because SOAP stacks themselves prefer to raise SOAPFault faults. Managing resource lifetimes is harder, but this is optional, as is WS-Notification, which is the hardest to test.

- The Globus Toolkit version 4 contains Java and C implementations of WSRF; many other Globus tools have been rebuilt around WSRF.
- WebSphere Application Server version 6.1 provides a WSRF environment which supports both simple and clustered, highly available WSRF endpoints.
- The Apache Foundation have a Muse 2.0 project which is a Java-based implementation of the WSRF, WS-Notification, and WSDM specifications.
- WSRF::Lite is a perl-based implementation that makes exclusive use of the *Address* element of the endpoint reference, thus making WS-Resources identifiable via URIs. In addition, WSRF::Lite provides a mapping of HTTP verbs to WSRF operations, making it possible to use WS-Resources in a REST architectural style.
- WSRF.NET is a .NET based project about WSRF specs from a research team of the University of Virginia.
- The latest version 6.0 of UNICORE is built on a Java implementation of the WSRF 1.2 standard including WS-ResourceLifetime and a partial implementation of WS-Notification.

WEB2GRID

The **WEB2GRID** project aims to facilitate the commercial and non-profit exploitation of the EU FP7 EDGeS (Enabling Desktop Grids for e-Science) project and the HAGRID project focusing on the Desktop Grid and WEB2 technologies.

The project, started in 2009, is coordinated by the Laboratory of Parallel and Distributed Systems (LPDS) at MTA-SZTAKI, Hungary.

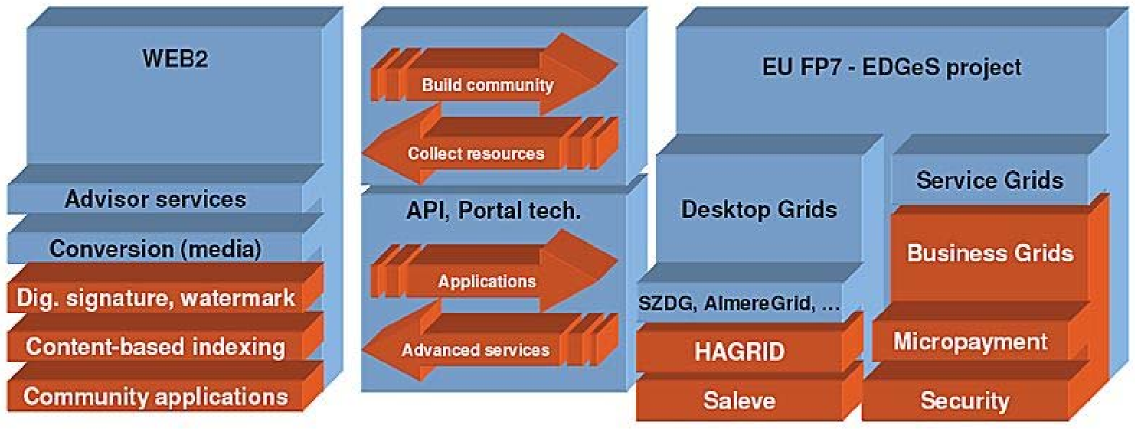

Secure Business Transaction Platform
Content creation, sharing, manipulation
Web 2 community
Content browsing
Web 2 portal
Grid resource providing community
Content processing

WEB2 and Grid relations

Overview

While WEB2 technologies assist to assure the resources in the desktop grid community on voluntary or settlement bases, the developed platform extended with desktop grid systems offers back-end infrastructure for the operational requirements of WEB2 portals and systems. By combining WEB2 with Desktop Grid technologies, WEB2 can extend its capabilities from the community contents towards to shared services with the help of grid technologies. The project aims to develop the tools, interfaces and methodologies through which the above mentioned targeted services can be established both in closed (local desktop grid), and in open (global desktop grid) environments.

Structure



Connections between the main units of the project

The three main blocks of the project are the Grid, the WEB2 and the connections between the two technologies.

In the last decades Grid systems offered large-scale distributed computing and storage services mainly for academic and university research. In this area the focus is shifting towards services and knowledge sharing, and the extension of the application domain with standardized access methods eliminating system heterogeneity. From Grid point of view, the project relies on the results of the EU FP7 supported EDGeS (Enabling Desktop Grids for e-Science) infrastructure project. In Desktop Grids the system is often based on community offerings, which is very similar to the WEB2 content building solution. While the EDGeS project is important in Grid/WEB2 integration, the WEB2GRID project also extends the EDGeS technologies towards the direction of commercial Grids.

In WEB2 services the owner of the server only provides the service framework, and the real content is shared and maintained by the users. Generally due to the large number of connections and users, WEB2 systems should handle heavy data traffic and complex relations, which may needs large computational power in many cases. From the WEB2 side, numerous application topics with large IT capacity can be solved by Grid technologies.

The connection between WEB2 and Grid is realized with well defined, low-level application development interface (API), and high-level graphical application development solutions of the project partners. WEB2 applications come from various fields, which justifies the need for the development of multi-level interfaces, and the project benefits from numerous EU FP6 and EU FP7 project achievements.

Social and Economical Goals

By connecting WEB2 communities with Grid technologies, the project aims to achieve various social and economical goals including:

- To strengthen the role and recognition of the national research community and knowledge base on international fields
- To demonstrate the wide range of possibilities of the industrial and business usage of Grid and WEB
- To increase the intensity of the connection between state financed research institutes and businesses
- To facilitate the permeation of knowledge-intensive technologies
- The application of methods being developed during the projects contributes to the balanced distribution of research tasks between regions
- The project encourages the cooperation between research institutes and provides academic opportunities for young researchers and PhD students.

Project Partners

The following partners participate in the project:

- econet.hu Media, Telecommunications and Holding Plc.
- Budapest University of Technology and Economics
- E-Group ICT Software Informatics Inc.

Chapter 7

Xgrid and TeraGrid

Xgrid

Xgrid is a proprietary software program and distributed computing protocol developed by the Advanced Computation Group subdivision of Apple Inc that allows networked computers to contribute to a single task.

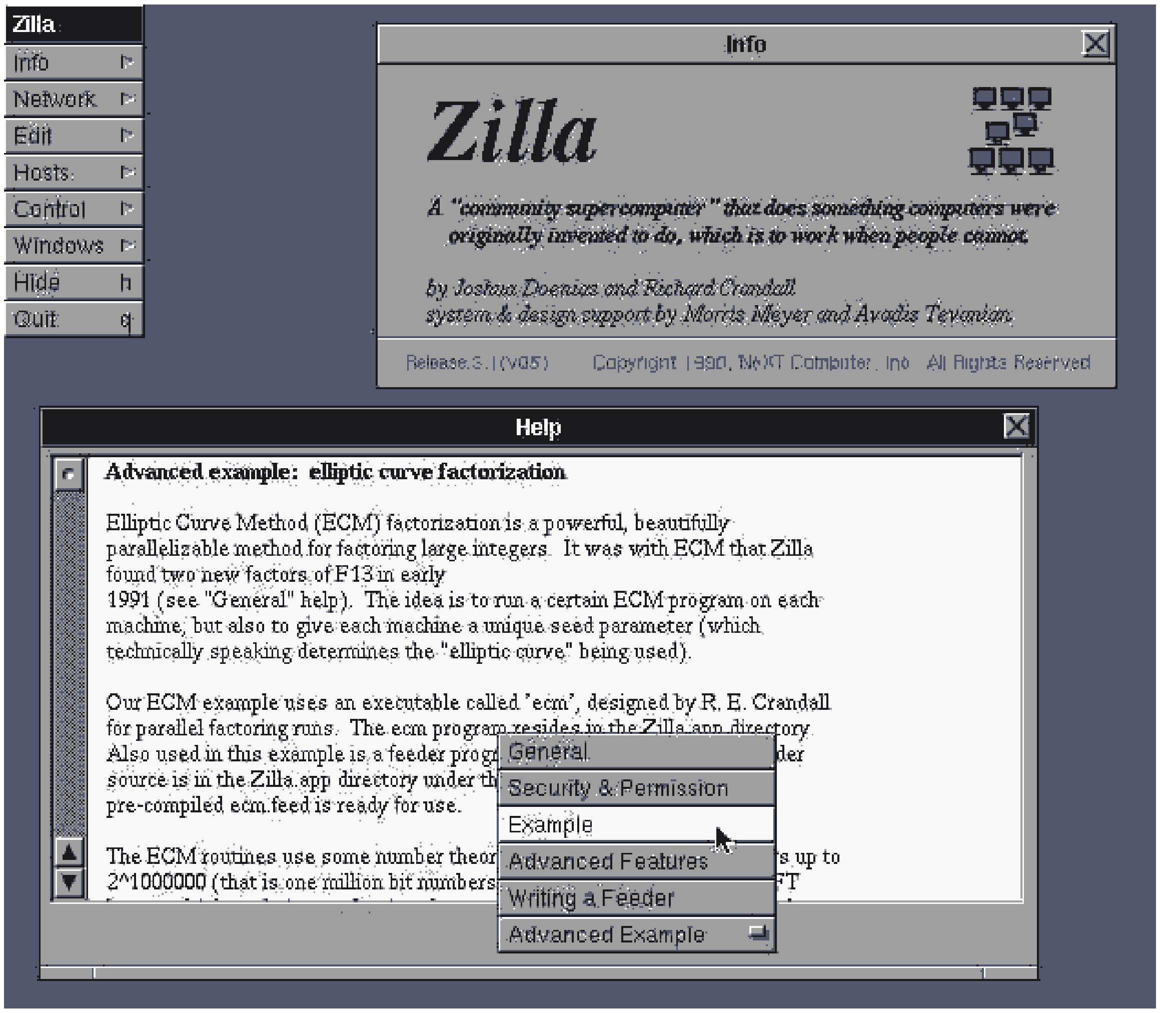
It provides network administrators a method of creating a computing cluster, which allows them to exploit previously unused computational power for calculations that can be divided easily into smaller operations, such as Mandelbrot maps. The setup of an Xgrid cluster can be achieved at next to no cost, as Xgrid client is pre-installed on all computers running Mac OS X 10.4 or later. The Xgrid controller, the job scheduler of the Xgrid operation, is also included within Mac OS X Server and as a free download from Apple. Apple has kept the command-line job control mechanism minimalist while providing an API to develop more sophisticated tools built around it.

The program employs its own communication protocol layered on top of a schema to communicate to other nodes. This communication protocol interfaces with the BEEP infrastructure, a network application protocol framework. Computers discovered by the Xgrid system, that is computers with Mac OS X's Xgrid service enabled, are automatically added to the list of available computers to use for processing tasks.

When the initiating computer sends the complete instructions, or job, for processing to the controller, the controller splits the task up into these small instruction packets, known as tasks. The design of the Xgrid system consists of these small packets being transferred to all the Xgrid-enabled computers on the network. These computers, or nodes, execute the instructions provided by the controller and then return the results. The controller assembles the individual task results into the whole job results and returns them to the initiating computer.

Apple modeled the design of Xgrid on the Zilla program, distributed with NeXT's OPENSTEP operating system application programming interface (API), which Apple owned the rights to. The company also opted to provide the client version of Mac OS X with only command-line functions and little flexibility, while giving the Mac OS X Server version of Xgrid a GUI control panel and a full set of features.

History



Zilla

A "community supercomputer" that does something computers were originally invented to do, which is to work when people cannot.

by Joshua Doenias and Richard Crandall
system & design support by Morris Meyer and Avadis Tevanian

Release 3.1 (v05) Copyright 1990, NeXT Computer, Inc. All Rights Reserved

Advanced example: elliptic curve factorization

Elliptic Curve Method (ECM) factorization is a powerful, beautifully parallelizable method for factoring large integers. It was with ECM that Zilla found two new factors of F13 in early
1991 (see "General" help). The idea is to run a certain ECM program on each machine, but also to give each machine a unique seed parameter (which technically speaking determines the "elliptic curve" being used).

Our ECM example uses an executable called 'ecm', designed by R. E. Crandall for parallel factoring runs. The ecm program resides in the Zilla.app directory
Also used in this example is a feeder progr der
source is in the Zilla.app directory under th
pre-compiled ecm.feed is ready for use.

The ECM routines use some number theor s up to
2^1000000 (that is one million bit numbers FT

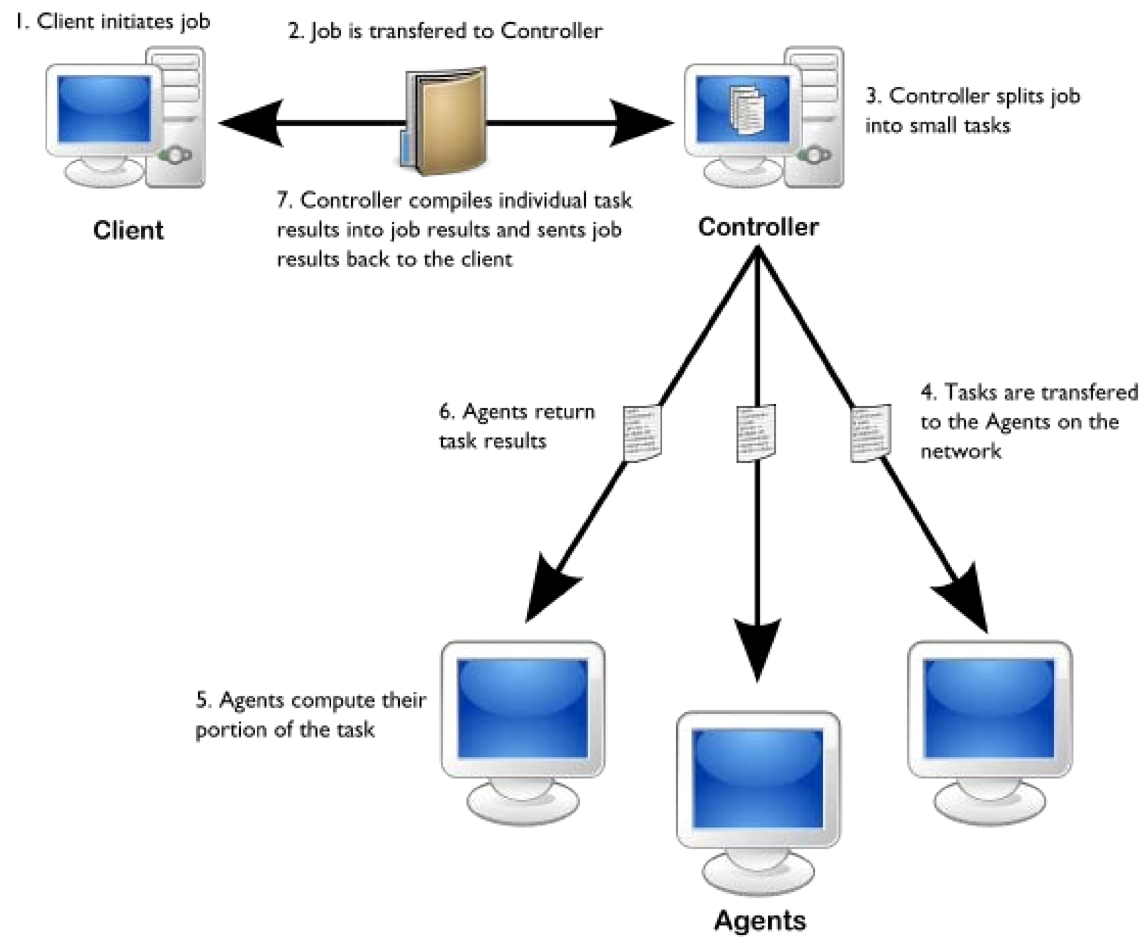
Zilla

Xgrid's original concept can be traced back to Zilla.app, found in the OPENSTEP operating system API, created by NeXT in the late 1980s. Zilla was the first distributed computing program released on an end-user operating system and which used the idle screen-saver motif, a design feature found in widely used projects such as Seti@Home and Distributed.net. Zilla won the national ComputerWorld Smithsonian Award (Science Category) in 1991 for ease of use and good design. Apple acquired Zilla, along with the rest of NeXT, in 1997 and later used Zilla as inspiration for Xgrid. The first beta version of Xgrid was released in January 2004.

Several organizations have adopted Xgrid in large international computing networks. One example of an Xgrid cluster is MacResearch's OpenMacGrid, where scientists can request access to large amounts of processing power to run tasks related to their research. Another was the now defunct Xgrid@Stanford project, which used a range of computers on the Stanford University campus and around the world to perform biochemical research.

In a pre-release promotional piece, *MacWorld* cited Xgrid among the Unix features in "10 Things to Know about TIGER", calling it "handy if you work with huge amounts of experimental data or render complex animations". After Xgrid's introduction in 2004, *InfoWorld* noted that it was a "'preview' grade technology" which would directly benefit from the Xserve G5's launch later that year. *InfoWorld* commentator Ephraim Schwartz also predicted that Xgrid was an opening move in Apple's entry into the enterprise computing market.

Protocol


1. Client initiates job
2. Job is transfered to Controller
3. Controller splits job into small tasks
Client
Controller
7. Controller compiles individual task results into job results and sents job results back to the client
6. Agents return task results
4. Tasks are transfered to the Agents on the network
5. Agents compute their portion of the task
Agents

Xgrid Protocol

The Xgrid protocol uses the BEEP network framework to communicate with nodes on the network. The system's infrastructure includes three types of computers which communicate over the protocol. One is the client, which communicates the calculation. Next is the controller, which starts and segregates the calculation. Finally, the agents process their own allocated part of the calculation.

A computer can act as one or all three of these components at the same time. The Xgrid protocol provides the basic infrastructure for computers to communicate, but is not involved in the processing of the specified calculation. Xgrid is targeted towards time consuming computations that can be easily segregated into smaller tasks, sometimes called *embarrassingly parallel* tasks. This includes Monte Carlo calculations, 3D rendering and Mandelbrot maps.

Within the Xgrid protocol, three types of messages can be passed to other computers on the same cluster: requests, notifications and replies. Requests must be responded to by the recipient with a reply, notifications do not require a reply, and replies are responses to sent messages. They are identified by their name, type (request/notification/reply) and contents. Each message is encapsulated in a BEEP message (BEEP MSG) and is acknowledged on receipt by an empty reply (RPY). Xgrid does not leverage BEEPs message/reply infrastructure. Any received message which requires a response merely generates an independent BEEP message containing the reply. The Xgrid messages are encoded as dictionaries of key/value pairs which are converted to XML before being sent across the BEEP network.

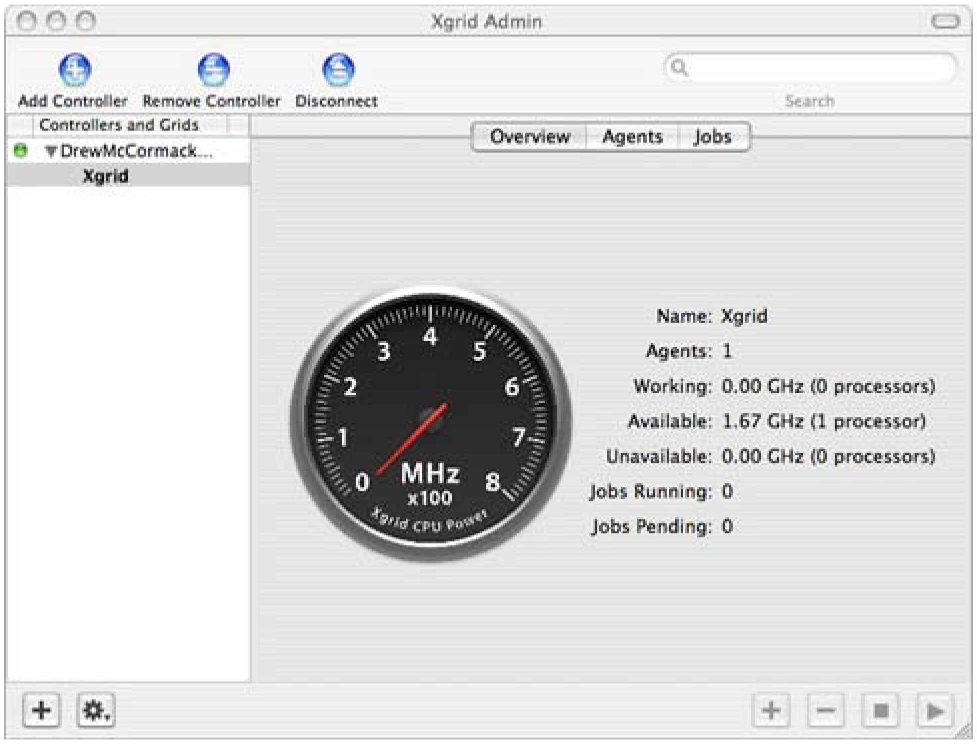
Architecture

The architecture of the Xgrid system is designed around a job based system; the controller sends agents jobs, and the agents return the responses. The actual computation that the controller executes in an Xgrid system is known as a job. The job contains all the files required to complete the task successfully, such as the input parameters, data files, directories, executables and/or shell scripts, the files included in an Xgrid job must be able to be executed either simultaneously or asynchronously, or any benefits of running such a job on an Xgrid is lost. Once the job completes, the controller can be set to notify the client of the task's completion or failure, for example by email. The client can leave the network while the tasks are running. It can also monitor the job status on demand by querying the controller, although it cannot track the ongoing progress of individual tasks.

The controller is central to the correct function of an Xgrid, as this node is responsible for the distribution, supervision and coordination of tasks on agents. The program running on the controller can assign and reassign tasks to handle individual agent failures on demand. The number of tasks assigned to an agent depend on two factors: the number of agents on an Xgrid and the number of processors in each node. The number of agents on an Xgrid determines how the controller will assign tasks. The tasks may be assigned simultaneously for a large number of agents, or queued for a small number of agents. When a node with more than one processor is detected on an Xgrid, the controller may assign one task per processor; this only occurs if the number of agents on the network is lower than the number of tasks the controller has to complete.

Xgrid is layered upon the Blocks Extensible Exchange Protocol (BEEP), an IETF standard comparable to HTTP, but with a focus on two-way multiplexed communication, such as that found in peer-to-peer networks. BEEP, in turn, uses XML to define profiles for communicating between multiple agents over a single network or internet connection.

Interface



Name: Xgrid
Agents: 1
Working: 0.00 GHz (0 processors)
Available: 1.67 GHz (1 processor)
Unavailable: 0.00 GHz (0 processors)
Jobs Running: 0
Jobs Pending: 0

Xgrid administration tool

While it is possible to access Xgrid from the command line, the Xgrid graphical user interface, a program bundled with Mac OS X Server and, as of March 2009, available online, is a much more efficient way of administering an Xgrid system. Originally, the Xgrid agent was included in all Mac OS X version 10.4 installations but the GUI was reserved for users of Mac OS X Server. This decision limited the efforts of the computer community to embrace the platform. Eventually, Apple released the Mac OS X Server Administration Tools to the public, which included the Xgrid administration application bundled with Mac OS X Server.

Despite the lack of a graphical controller interface in the standard (non-server) Mac OS X distribution, it is possible to set up an Xgrid controller via the command line tools `xgridctl` and `xgrid`. Once the Xgrid controller daemon is running, administration of the grid with Apple's Xgrid Admin tool is possible. Some applications, such as VisualHub, provided Xgrid controller capability through their user interfaces.

TeraGrid

TeraGrid is an open scientific discovery grid-computing infrastructure combining leadership class resources at eleven partner sites to create an integrated, persistent computational resource.

Using high-performance network connections, the TeraGrid integrates high-performance computers, data resources and tools, and high-end experimental facilities around the country. Currently, TeraGrid resources include more than a petaflop of computing capability and more than 30 petabytes of online and archival data storage, with rapid access and retrieval over high-performance networks. Researchers can also access more than 100 discipline-specific databases. With this combination of resources, the TeraGrid is the world's largest, most comprehensive distributed cyberinfrastructure for open scientific research.

TeraGrid is coordinated through the Grid Infrastructure Group (GIG) at the University of Chicago, working in partnership with the Resource Provider sites: Indiana University, the Louisiana Optical Network Initiative, National Center for Supercomputing Applications, the National Institute for Computational Sciences, Oak Ridge National Laboratory, Pittsburgh Supercomputing Center, Purdue University, San Diego Supercomputer Center, Texas Advanced Computing Center, and University of Chicago/Argonne National Laboratory, and the National Center for Atmospheric Research.

History

The TeraGrid project was launched by the National Science Foundation in August 2001 with $53 million in funding to four sites: the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign, the San Diego Supercomputer Center (SDSC) at the University of California, San Diego, University of Chicago Argonne National Laboratory, and Center for Advanced Computing Research (CACR) at the California Institute of Technology in Pasadena, California.

UCSD TeraGrid

In October 2002, the Pittsburgh Supercomputing Center (PSC) at Carnegie Mellon University and the University of Pittsburgh joined the TeraGrid as major new partners when NSF announced $35 million in supplementary funding. The TeraGrid network was transformed through the ETF project from a 4-site mesh to a dual-hub backbone network with connection points in Los Angeles and at the Starlight facilities in Chicago.

In October 2003, NSF awarded $10 million to add four sites to TeraGrid as well as to establish a third network hub, in Atlanta. These new sites were Oak Ridge National Laboratory (ORNL), Purdue University, Indiana University, and the Texas Advanced Computing Center (TACC) at The University of Texas at Austin.

TeraGrid construction was also made possible through key corporate partnerships with Sun Microsystems, IBM, Intel Corporation, Qwest Communications, Juniper, Myricom, Hewlett-Packard Company, and Oracle Corporation.

TeraGrid construction was completed in October 2004, at which time the TeraGrid facility began full production.

In August 2005, NSF awarded $148M for a five-year program to operate and enhance the TeraGrid facility, with eight resource provider awards and a system integration award (the Grid Infrastructure Group at the University of Chicago).

The TeraGrid: 2005-2011

In August 2005, NSF's newly created Office of Cyberinfrastructure extended support for the TeraGrid with a $150 million set of awards for operation, user support and enhancement of the TeraGrid facility over the next five years. Using high-performance network connections, the TeraGrid featured high-performance computers, data resources and tools, and high-end experimental facilities around the country. In May 2007, TeraGrid integrated resources included more than 250 teraflops of computing capability and more than 30 petabytes (quadrillions of bytes) of online and archival data storage with rapid access and retrieval over high-performance networks. Researchers could access more than 100 discipline-specific databases. In late 2009, The TeraGrid resources had grown to 2 petaflops of computing capability and more than 60 petabytes storage. In mid 2009, NSF extended the operation of TeraGrid to 2011. TeraGrid competes with EGEE to be the world's largest, most comprehensive distributed cyberinfrastructure for open scientific research.

Architecture

TeraGrid resources are integrated through a *service-oriented architecture* in that each resource provides a "service" that is defined in terms of interface and operation. Computational resources run a set of software packages called "Coordinated TeraGrid Software and Services" (CTSS). CTSS provides a familiar user environment on all TeraGrid systems, allowing scientists to more easily port code from one system to another. CTSS also provides integrative functions such as single-signon, remote job submission, workflow support, data movement tools, etc. CTSS includes the Globus Toolkit, Condor, distributed accounting and account management software, verification and validation software, and a set of compilers, programming tools, and environment variables.

TeraGrid uses a 10 Gigabits per second dedicated optical backbone network, with hubs in Chicago, Denver, and Los Angeles. All resource provider sites connect to a backbone node at 10 Gigabits per second. TeraGrid users access the facility through national research networks such as the Internet2 Abilene backbone and National LambdaRail.

Usage

TeraGrid users primarily come from U.S. universities. There are roughly 4,000 users at over 200 universities. Academic researchers in the United States can obtain exploratory, or *development* allocations (roughly, in "CPU hours") based on an abstract describing the work to be done. More extensive allocations involve a proposal that is reviewed during a quarterly peer-review process. All allocation proposals are handled through the TeraGrid website. Proposers select a scientific discipline that most closely describes their work, and this enables reporting on the allocation of, and use of, TeraGrid by scientific discipline. As of July 2006 the scientific profile of TeraGrid allocations and usage is shown in the following table.

Allocated (%)	**Used (%)**	**Scientific Discipline**
19	23	Molecular Biosciences
17	23	Physics
14	10	Astronomical Sciences
12	21	Chemistry
10	4	Materials Research
8	6	Chemical, Thermal Systems
7	7	Atmospheric Sciences
3	2	Advanced Scientific Computing
2	0.5	Earth Sciences
2	0.5	Biological and Critical Systems
1	0.5	Ocean Sciences
1	0.5	Cross-Disciplinary Activities
1	0.5	Computer and Computation Research
0.5	0.25	Integrative Biology and Neuroscience
0.5	0.25	Mechanical and Structural Systems
0.5	0.25	Mathematical Sciences
0.5	0.25	Electrical and Communication Systems
0.5	0.25	Design and Manufacturing Systems
0.5	0.25	Environmental Biology

Each of these discipline categories correspond to a specific program area of the National Science Foundation (thus more detail can be found at the NSF website).

During 2006, TeraGrid has begun to provide application-specific services to *Science Gateway* partners, who serve (generally via a web portal) discipline-specific scientific and education communities. Through the Science Gateways program TeraGrid aims to broaden access by at least an order of magnitude in terms of the number of scientists, students, and educators who are able to use TeraGrid.

TeraGrid Resource Providers

- Argonne National Laboratory (ANL) operated by the University of Chicago and the Department of Energy
- Indiana University
- Louisiana Optical Network Initiative (LONI)
- National Center for Atmospheric Research (NCAR)
- National Center for Supercomputing Applications (NCSA)

- National Institute for Computational Sciences (NICS) operated by University of Tennessee at Oak Ridge National Laboratory.
- Oak Ridge National Laboratory (ORNL)
- Pittsburgh Supercomputing Center (PSC) operated by University of Pittsburgh and Carnegie Mellon University.
- Purdue University
- San Diego Supercomputer Center (SDSC)
- Texas Advanced Computing Center (TACC)

Similar projects

- DEISA Distributed European Infrastructure for Supercomputing Applications, a facility integrating eleven European supercomputing centers.
- EGEE Enabling Grids for E-sciencE
- NAREGI Japanese NAtional REsearch Grid Initiative involving several supercomputer centers
- Open Science Grid - a distributed computing infrastructure for scientific research

Chapter 8

Space-Based Architecture and SAGA C++ Reference Implementation

Space-Based Architecture

Space-Based Architecture (**SBA**) is a software architecture pattern for achieving linear scalability of stateful, high-performance applications using the tuple space paradigm. It follows many of the principles of Representational State Transfer (REST), service-oriented architecture (SOA) and Event-driven architecture (EDA), as well as elements of grid computing. With a space-based architecture, applications are built out of a set of self-sufficient units, known as processing-units (PU). These units are independent of each other, so that the application can scale by adding more units.

The SBA model is closely related to other patterns that have been proved successful in addressing the application scalability challenge, such as Shared-Nothing Architecture, used by Google, Amazon.com and other well-known companies. The model has also been applied by many firms in the securities industry for implementing scalable electronic securities trading applications.

Components of Space-Based Architecture

An application built on the principles of space-based architecture typically has the following components:

- **Processing Unit** — the unit of scalability and fail-over. Normally, a processing unit is built out of a POJO (Plain Old Java Object) container, such as that provided by the Spring Framework.

- **Virtual Middleware** — a common runtime and clustering model, used across the entire middleware stack. The core middleware components in a typical SBA architecture are:

Component	Description
Messaging Grid	Handles the flow of incoming transaction as well as the communication between services
Data Grid	Manages the data in distributed memory with options for synchronizing that data with an underlying database
Processing Grid	Parallel processing component based on the master/worker pattern (also known as a blackboard pattern) that enables parallel processing of events among different services

- **POJO-Driven Services Model** — A lightweight services model that can take any standard Java implementation and turn it into a loosely coupled distributed service. The model is ideal for interaction with services that run within the same processing-unit.

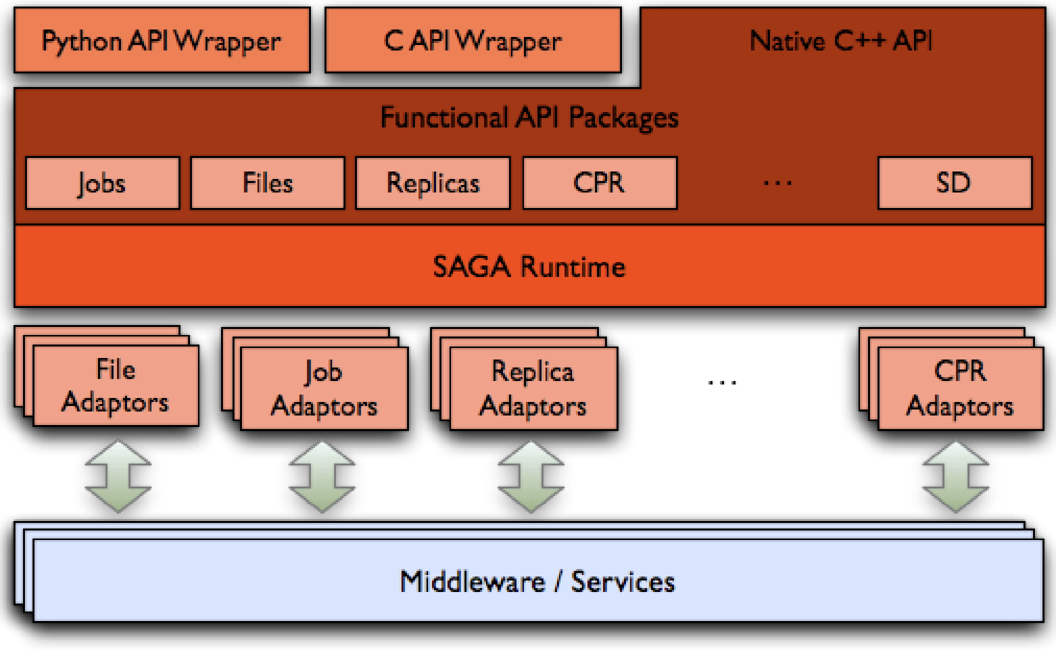
- **SLA-Driven Container** — The SLA-driven container enables the deployment of the application on a dynamic pool of machines based on Service Level Agreements. SLA definitions include the number of instances that need to run in order to comply with the application scaling and fail-over policies, as well as other policies.

SAGA C++ Reference Implementation

The **SAGA C++ Reference Implementation** is a set of free cross-platform libraries written in C++ and Python which provide a set of high-level interfaces and runtime components that allow the development of distributed computing and grid computing applications, frameworks and tools. SAGA is the first complete implementation of the Open Grid Forum Simple API for Grid Applications standard GFD-R-P.90. SAGA is available for all major operating systems, including Linux and other Unix-like systems, Microsoft Windows and Mac OS X. SAGA is open source and licensed under the Boost Software License.

SAGA can be used to develop scalable and portable large-scale distributed applications, frameworks and tools. SAGA supports many of the widely used distributed grid middleware systems, like Globus, Condor, UNICORE and gLite as well as cloud computing services like Amazon EC2 and Eucalyptus.

Architecture



The SAGA C++/Python architecture: a light-weight runtime system dispatches API calls from the application to the middleware through a set of plug-ins or *adaptors*.

SAGA is designed as an object oriented interface. It encapsulates related functionality in a set of objects, that are grouped in functional namespaces, which are called *packages* in SAGA. The SAGA core implementation defines the following packages:

- saga::advert - interface for advert service access
- saga::filesystem - interface for file and directory access
- saga::job - interface for job definition, management and control
- saga::namespace - abstract interface (used by advert, filesystem and replica interfaces)
- saga::replica - interface for replica management
- saga::rpc - interface for remote procedure calls client and servers
- saga::sd- interface for service discovery in distributed environments
- saga::stream - interface for data stream client and servers

The overal architecture of SAGA follows the adaptor pattern, a software design pattern which is used for translating one interface into another. In SAGA it translates the calls from the API packages to the interfaces of the underlying middleware. The SAGA run-time system uses late-binding to decide at run-time which plug-in (*middleware adaptor*) to load and bind.

Supported Middleware

The following table lists the distributed middleware systems that are currently supported by SAGA. The column labeled *Adaptor Suite* names the collection (release package) of the (set of) middleware adaptors that provides support for the middleware system.

Middleware System	SAGA Adaptor Suite	SAGA API Namespace
Amazon EC2	saga-adaptors-aws	saga::job
Condor	saga-adaptors-condor	saga::job
Eucalyptus	saga-adaptors-aws	saga::job
Globus GRAM (2 and 5)	saga-adaptors-globus	saga::job
Globus GridFTP	saga-adaptors-globus	saga::filesystem
Globus RLS	saga-adaptors-globus	saga::replica
HDFS	saga-adaptors-hdfs	saga::file
Local File system	*part of saga-core*	saga::file
Local Fork	*part of saga-core*	saga::job
Nimbus	saga-adaptors-aws	saga::job
PBS (Pro)	saga-adaptors-pbs	saga::job
Platform LSF	saga-adaptors-lsf	saga::job
SQL Advert Service	*part of saga-core*	saga::advert
SQL Replica Service	*part of saga-core*	saga::replica
SSHFS	saga-adaptors-ssh	saga::file
SSH	saga-adaptors-ssh	saga::job
TORQUE	saga-adaptors-torque	saga::job

Examples

Job Submission

A typical task in a distributed application is to submit a *job* to a local or remote distributed resource manager. SAGA provides a high-level API called the *job package* for this. The following two simple examples show how the SAGA job package API can be used to submit an MPI job to a remote Globus GRAM resource manager.

C++:

```
#include <saga/saga.hpp>

int main (int argc, char** argv)
{
  namespace sa  = saga::attributes;
  namespace sja = saga::job::attributes;

  try
  {
    saga::job::description jd;

    jd.set_attribute (sja::description_executable, "/home/user/hello-
mpi");
    jd.set_attribute (sja::description_output, "/home/user/hello.out");
    jd.set_attribute (sja::description_error, "/home/user/hello.err");

    // Declare this as an MPI-style job
    jd.set_attribute (sja::description_spmd_variation, "mpi");

    // Name of the queue we want to use
    jd.set_attribute (sja::description_queue, "checkpt");
    jd.set_attribute (sja::description_spmd_variation, "mpi");
    // Number of processors to request
    jd.set_attribute (sja::description_number_of_processes, "32");

    saga::job::service js("gram://my.globus.host/jobmanager-pbs");
    saga::job::job j = js.create_job(jd);

    j.run()
  }
  catch(saga::exception const & e)
  {
    std::cerr << "SAGA exception caught: " << e.what() << std::endl;
  }
}
```

Python:

```
import saga

try:
  jd = saga.job.description()

  jd.executable = "/home/user/hello-mpi"
  jd.error  = "/home/user/hello.err"
  jd.output = "/home/user/hello.out"

  # Declar this as an MPI-style job
  jd.spmd_variation = "mpi"

  # Name of the queue we want to use
  jd.queue = "checkpt"
  # Number of processors to request
  jd.number_of_processes = "32"

  # URL of the resource manager. In this case Globus GRAM
  js = saga.job.service("gram://my.globus.host/jobmanager-pbs")

  job = js.create_job(jd)
  job.run()

except saga.exception, e:
  print e.get_all_messages()
```

Chapter 9

Oracle Grid Engine and OurGrid

Oracle Grid Engine

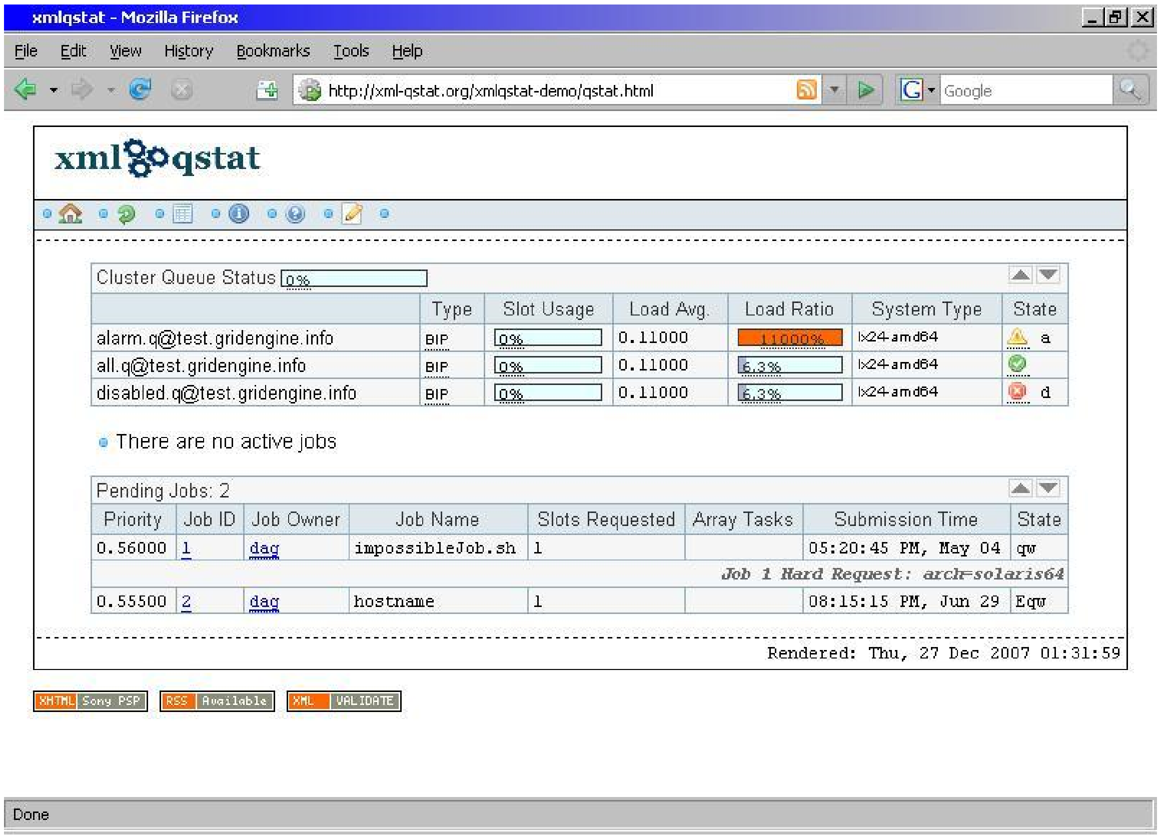
Oracle Grid Engine, previously known as *Sun Grid Engine* (**SGE**), previously known as **CODINE** (COmputing in DIstributed Networked Environments) or **GRD** (Global Resource Director), is an open source batch-queuing system, developed and supported by Sun Microsystems. Sun once also sold a commercial product based on SGE, known as **N1 Grid Engine** (**N1GE**).

Grid Engine is open source and free to use from the project website under the Sun Industry Standards Source License. There is a commercial version available from the Oracle site. It appears that all further versions, starting from 6.2u6, will be commercial (with a 90-day free trial). Licenses cost 500 USD per processor).

SGE is typically used on a computer farm or high-performance computing (HPC) cluster and is responsible for accepting, scheduling, dispatching, and managing the remote and distributed execution of large numbers of standalone, parallel or interactive user jobs. It also manages and schedules the allocation of distributed resources such as processors, memory, disk space, and software licenses.

SGE is the foundation of the Sun Grid utility computing system, made available over the Internet in the United States in 2006, later becoming available in many other countries.

Features


xmlqstat - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://xml-qstat.org/xmlqstat-demo/qstat.html
Google
xmlqstat
XHTML Sony PSP
RSS Available
XML VALIDATE
Done

Cluster Queue Status 0%

	Type	Slot Usage	Load Avg.	Load Ratio	System Type	State
alarm.q@test.gridengine.info	BIP	0%	0.11000	11000%	lx24-amd64	a
all.q@test.gridengine.info	BIP	0%	0.11000	6.3%	lx24-amd64	
disabled.q@test.gridengine.info	BIP	0%	0.11000	6.3%	lx24-amd64	d

There are no active jobs

Pending Jobs: 2

Priority	Job ID	Job Owner	Job Name	Slots Requested	Array Tasks	Submission Time	State
0.56000	1	dag	impossibleJob.sh	1		05:20:45 PM, May 04	qw
Job 1 Hard Request: arch=solaris64							
0.55500	2	dag	hostname	1		08:15:15 PM, Jun 29	Eqw

Rendered: Thu, 27 Dec 2007 01:31:59

A screenshot of the xml-qstat web interface.

Features new in version 6.2

- Advance reservation
- Array job interdependencies
- Rule-based *Resource Quota* control
- Enhanced remote execution (without using external rshd/rlogind/sshd processes)
- Multi-clustering
- Daemons managed by the Service Management Facility on Solaris
- Pseudo TTY (pty) support for interactive jobs
- Job Submission Verifier (client-side and server-side job verification)
- GUI Installer and SGE Inspect
- Topology-aware scheduling and thread binding
- Hadoop integration, Amazon EC2 integration for cloud computing

Other features of SGE include:

- Multiple advanced scheduling algorithms allow powerful policy-based resource allocation
- Cluster queues

- Job and scheduler fault tolerance - Grid Engine continues to operate as long as there is one or more hosts available
- Job checkpointing
- Job arrays and job tasks
- DRMAA (Job API)
- Resource reservation
- XML status reporting (*qstat* and *qhost*), and the *xml-qstat* web interface
- Parallel jobs (MPI, PVM, OpenMP), and scalable parallel job startup with qrsh
- Usage accounting
- Accounting and Reporting COnsole (ARCO)
- parallel make: distmake, dmake (Sun Studio), and SGE's own qmake
- FLEXlm integration and multi-cluster software license management with *LicenseJuggler*

Platforms

SGE runs on multiple platforms, including:

- AIX
- BSD - FreeBSD, NetBSD, OpenBSD
- HP-UX
- IRIX
- Linux
- Mac OS X
- Solaris
- SUPER-UX
- Tru64
- Windows via SFU (Interix) or SUA (Microsoft Windows Services for UNIX) (as execution hosts only)
- Z/OS (in progress)

Cluster architecture

A typical Grid Engine cluster consists of a master host, and one or more execution hosts. Moreover, multiple *shadow masters* can be configured as hot spares, which take over the role of the master when the original master host crashes.

Support and training

Sun provides support contracts for the commercial version of Grid Engine on most UNIX platforms and Windows. Professional services, consulting, training, and support are also provided by Sun Partners. Sun partners with Georgetown University to deliver Grid Engine administration classes. *The Bioteam* runs short SGE training workshops that are 1 or 2 days long.

Users can obtain community support on the Grid Engine mailing lists.

Grid Engine Workshops were held in 2002, 2003, 2007, and 2009 in Regensburg, Germany.

Prominent users

Notable deployments of SGE include:

- Sun Grid
- the TSUBAME supercomputer at the Tokyo Institute of Technology, which was number 7 on June 2006 TOP500 list.
- Ranger at the Texas Advanced Computing Center (TACC). Ranger has 62,976 processor cores in 3,936 nodes and a peak performance of 504TFlops. Ranger was the 4th most powerful TOP500 supercomputer in 2008.
- San Diego Supercomputer Center (SDSC)
- Geophysical Fluid Dynamics Laboratory (NOAA GFDL)

History

In 2000, Sun acquired Gridware, Inc. a privately owned commercial vendor of advanced computing resource management software with offices in San Jose, Calif., and Regensburg, Germany. Later that year, Sun offered a free version of Gridware for Solaris and Linux, and renamed the product Sun Grid Engine.

In 2001, Sun made the source code available, and adopted the open source development model. Ports for Mac OS X and *BSD were contributed by the non-Sun open source developers.

In 2010, after the purchase of Sun by Oracle, the Grid Engine 6.2 update 6 source code was not included with the binaries, and changes were not put back to the project's source repository. In response to this, the Grid Engine community started the *Open Grid Scheduler* project to continue to develop and maintain a free implementation of Grid Engine.

On January 18, 2011, it was announced that Univa had recruited several principal engineers from the former Sun Grid Engine team and that Univa would be developing their own forked version of Grid Engine. The newly announced Univa Grid Engine will include commercial support and would compete with the official version of Oracle Grid Engine.

Other Grid Engine based products

- Sun Constellation System
- Sun Visualization System
- Sun Compute Cluster
- ClusterVisionOS Distribution
- Rocks Cluster Distribution

- EGEE
- Univa's UniCluster Express
- BioTeam's iNquiry
- Nimbus - uses Grid Engine as a virtual machine scheduler in a cloud computing environment

Add-on software

A number of SGE add-ons are available:

- Solaris Cluster integration
- *Service Domain Management* module in order to meet service level objectives
- Transfer-queue Over Globus (TOG). Globus added support for Grid Engine in Globus Toolkit 5.0.0
- JOb Scheduling Hierarchically (JOSH)

OurGrid

OurGrid is an opensource grid middleware based on a peer-to-peer architecture.

OurGrid is mainly develop at the Federal University of Campina Grande (Brazil), which also run an OurGrid instance named "OurGrid" too, in production since December 2004. Anyone can freely and easily join it to gain access to large amount of computational power and run parallel applications. This computational power is provided by the idle resources of all participants, and is shared in a way that makes those who contribute more get more when they need. Currently, the platform can be used to run any application whose tasks (ie, parts that run on a single machine) do not communicate among themselves during execution, like most simulations, data mining and searching.

OurGrid Vision

The recent advances in computing and networking are changing the way we do scientific research, a trend that has been dubbed eScience. Thanks to the power of computer-based communication, research is now a much more collaborative endeavor. Moreover, computers play an ever-increasing role in the process of scientific discovery. Data analysis without computers sounds antediluvian. Simulation has joined theory and experimentation as the third scientific methodology. As a result, many research labs now demand non-trivial computing capabilities.

As a solution to this demand, Grid Computing has appeared with the enticing promise of turning computing into utility. The vision is plug in the grid and solve your problem. However, turning the grid vision into reality is no trivial matter. In particular, grid solutions available today require highly specialized computer skills to install, set-up and operate, as well as an off-line negotiation to decide how the resources are going to be shared among the nodes that compose the grid. Therefore, current grid solutions only make sense for large labs. They do have the resources (both human and computer) to undertake the effort of putting a grid into production, as well as staff to negotiate with the other sites that compose the grid.

However, the vast majority of the research labs are small (a dozen people or so), and thus cannot afford deploying grid technology. Yet, these labs increasingly demand large amounts of computational power. OurGrid was designed to fill this gap, catering for small and medium-sized labs around the world that have unserved computational demands.

OurGrid is an open, free-to-join, cooperative grid in which labs donate their idle computational resources in exchange for accessing other labs' idle resources when needed. It uses a peer-to-peer technology that makes it in each lab's best interest to collaborate with the system by donating its idle resources. OurGrid leverages from the fact that people do not use their computers all the time. Even when actively using computers as research tools, researchers alternate between job execution (when they demand computational power) and result analysis (when their computational resources go mostly idle).

For OurGrid to be useful, it must be fast, simple, scalable, and secure. These were the four major goals that guided the design of OurGrid. Putting it in more detail:

- OurGrid must be fast (i.e. the turnaround time of a job must be much better than what is possible using only local resources) otherwise there will be no point in using it.

- Simplicity is also a fundamental requirement for OurGrid. After all, labs want to spend the minimum possible effort on the computer technology that will solve their problems. They want to focus on whatever research they do. Computers are just tools for them.

- OurGrid must scale well; otherwise it will not tap the huge amount of computational power that goes idle in the labs around the world. Note that scalability is not just a technical issue. It also has administrative ramifications. In particular, it is not acceptable to have to go through a human negotiation to define who can access what, when and how (something that is needed to set up current grids). Therefore, OurGrid must be a free-to-join open grid.

- OurGrid must be secure, because its peer-to-peer automatic granting of access will allow unknown foreign code to access one's machine. Nevertheless one's machine must remain safe.

Achieving these goals was a very challenging task. In order to simplify somewhat the problem, at least for now, we reduce OurGrid's scope to supporting Bag-of-Tasks (BoT) applications. BoT applications are those parallel applications whose tasks are independent. Despite their simplicity, BoT applications are used in a variety of scenarios, including data mining, massive searches (such as key breaking), parameter sweeps, simulations, fractal calculations, computational biology, and computer imaging. Assuming applications to be BoT simplifies our requirements in a few important ways. In particular, we can deliver fast execution of applications without demanding any QoS guarantees. It also makes it easier to provide a secure environment, since network access is not necessary during the execution of a foreign task.

Inside OurGrid

OurGrid 4.0 Main Components

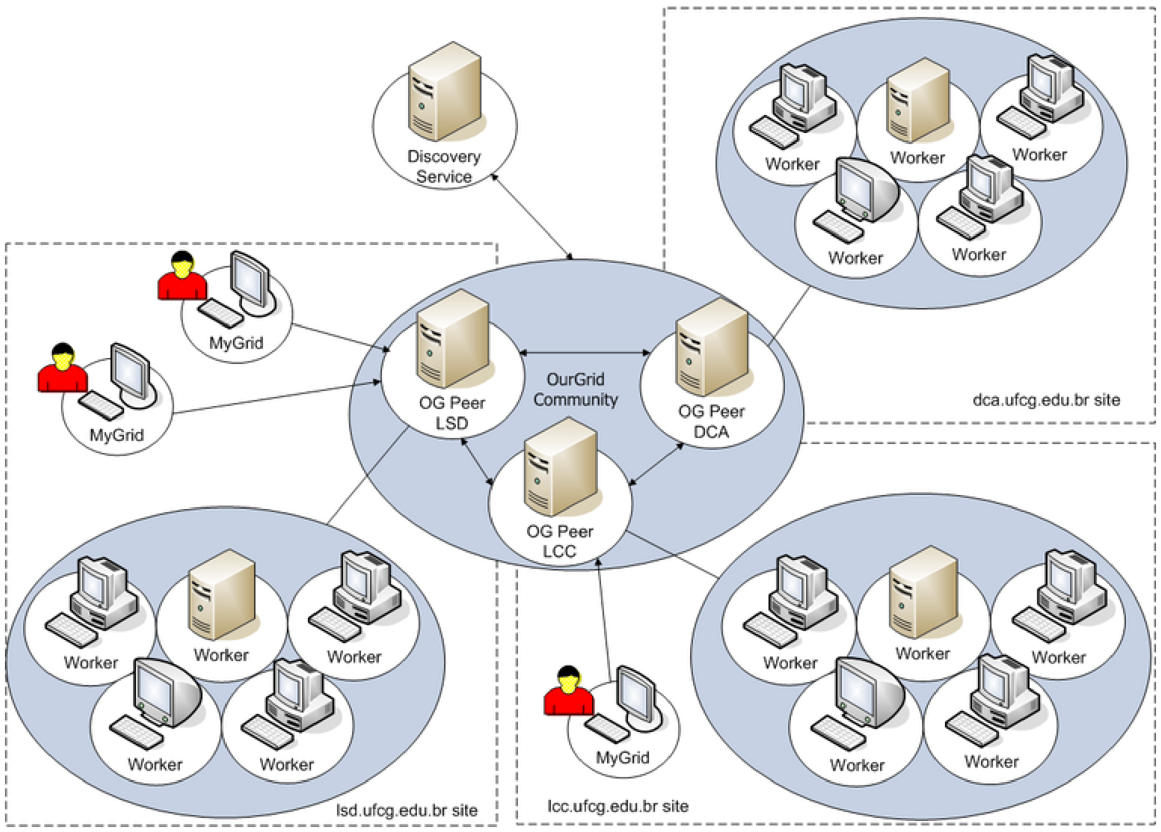


Figure 1.1 OurGrid Main Components

- OurGrid Broker

The OurGrid Broker (originally called MyGrid) is the scheduling component of the OurGrid solution. A machine running the Broker is called the home machine, which is the central point of a grid. During the processing of jobs, it acts as the grid coordinator, scheduling the execution of tasks and doing all the necessary data transfer to and from grid machines. Due to its central role, grid configuration and management, as well as job specification, is done on the home machine. For these reasons, you will likely use your desktop as the home machine for your grid. This approach decentralizes access to the grid allowing multiple users, each using their own installation of the Broker, to do concurrent processing.

The Broker is OurGrid's user frontend. It provides all the necessary support to describe, execute, and monitor jobs. Job processing is done by machines running OurGrid Workers. During the execution of a job, the Broker gets Workers on-demand from its associated Peer. It is the Broker's role to schedule the tasks to run on the Workers and to deploy and retrieve all data to/from Workers before and after the execution of tasks.

- Peers

An OurGrid Peer runs on a machine called the peer machine. The main role of a Peer is to organize and provide worker machines that belong to the same administrative domain. From the user's perspective, a Peer is a Worker provider, i.e., a network service that dynamically provides Workers for task execution. From an administrative point of view, a Peer determines how and which machines can be used as workers.

In Figure 1.1, there is one Peer for each administrative domain. This architecture allows different site administrators to enforce their own policies regarding the use of their Workers.

- Workers

The OurGrid Worker component runs on each machine that will be available for task execution. The Worker provides necessary access functionality to the home machine. It also provides some basic support for instrumentation and fault handling. Furthermore, combined with the OurGrid Peer, it allows for the use of machines in private networks.

In practice, any computer connected to the Internet can be used as a worker machine, even if it lies in a different administrative domain or behind a firewall. In Figure 1.1, administrative domains, possibly using their own intranets, are illustrated as rectangles containing Workers.

Network of Favours

To encourage resource contribution to the network, OurGrid uses a resource allocation mechanism called Network of Favours. The Network of Favours is an autonomous reputation scheme that rewards Peers that contribute more. This way, there is an incentive for each Peer to contribute as much as possible to the system.

The OurGrid Community is a peer-to-peer resource sharing system focused on providing resources to BoT applications. The central mission of OurGrid Community is that the sharing be done using the network of favours model. In this model, each Peer offers access to its idle resources to the community. In return, when there is work that exceeds local capacity, a Peer expects to gain access to the idle resources of other participants. The system aims to allow users of BoT applications to easily obtain access and use the community's computational resources, dynamically forming an on-demand, large-scale, grid.

However, the OurGrid solution provides more. Peers may belong to a world wide peer-to-peer community that share resources in a network of favours. This turns your Broker into a world wide out-of-the-box enabled grid. Once your Broker is connected to the community, tasks will be farmed out to machines that belong to different administrative domains all over the world without any further configuration.

Each Peer in the community is an entity that owns a number of resources and occasionally needs more computing power than these resources can provide. Whenever a Peer needs more power, it requests resources to the community. Whenever it has idle resources, it allocates them to one of the requesters. As there are no guarantees about the quality of service obtained from the idle resources donated to the community, not all applications are suitable for OurGrid.

Chapter 10

MTA SZTAKI Laboratory of Parallel and Distributed Systems & National Grid Service

MTA SZTAKI Laboratory of Parallel and Distributed Systems

The **Laboratory of Parallel and Distributed Systems** (LPDS) focuses its research on cluster and grid technologies, applying its research results in products such as the P-GRADE Grid Portal, gUSE and the SZTAKI Desktop Grid. The Laboratory is an active participant in European Grid projects including Enabling Grids for E-sciencE (EGEE) and Enabling Desktop Grids for e-Science (EDGeS).

LPDS and MTA SZTAKI: main building

Products

- The P-GRADE Grid Portal is a Grid portal solution capable of granting access to multiple Grids. It allows users to manage the whole life-cycle of executing a parallel application, enabling the creation, execution and monitoring of workflows through high-level Web interfaces.

- gUSE provides a scalable set of high-level Grid services by which interoperation between Grids and user communities can be achieved. Incorporating a more flexible workflow concept and enabling its distribution on clusters and different Grid sites, gUSE is aimed at extending the objectives and features of the P-GRADE Portal.

- The goal of SZTAKI Desktop Grid is providing an enterprise solution to exploit PCs and clusters located at different sites of a company or institute, solving large scale distributed programs via an easy-to-use application programming interface. It is extended to include clusters as single powerful PCs and to hierarchically propagate work from one desktop grid to the other.

Research

The main research areas of the LPDS are parallel distributed and concurrent programming, graphical programming environments, supercomputing, cluster computing and Grid computing.

Major projects

The Laboratory participated in the CoreGRID Network of Excellence and has been working as a project member in all phases of the grid infrastructure project Enabling Grids for E-sciencE (EGEE), serving as a regional training centre from 2004, as the assistant leader in training from 2007 and as the coordinator of application porting centers from 2008. The LPDS is the coordinator of the EU FP7 EDGeS (Enabling Desktop Grids for e-Science) project with the aim of bridging service grid and desktop grid infrastructures and supporting their user communities from both academic and industry environments. The Laboratory also participates in other large international projects such as SEE-GRID-SCI, S-Cube, ETICS-2 and CancerGRID.

WEB2GRID

The WEB2GRID project aims to facilitate the commercial and non-profit exploitation of the EDGeS EU FP7 Desktop Grid project and the HAGRID project focusing on the Desktop Grid and WEB2 technologies. While WEB2 technologies assist to assure the resources in the desktop grid community on voluntary or settlement bases, the developed platform extended with desktop grid systems offers back-end infrastructure for the operational requirements of WEB2 portals and systems. By combining WEB2 with Desktop Grid technologies, WEB2 can extend its capabilities from the community contents towards to shared services with the help of grid technologies. The project aims to develop the tools, interfaces and methodologies through which the above mentioned targeted services can be established both in closed (local desktop grid), and in open (global desktop grid) environments.

A comprehensive list of other projects can be found on the LPDS homepage.

Services

GASuC

The Grid Application Support Centre is an EGEE-supported project established in 2008. In close collaboration with application owners, GASuC provides assistance in gridification (i.e. porting legacy applications onto Grid infrastructures). The GASuC team identifies the suitable approaches and tools for the porting process, sets up realistic porting scenarios and organizes workshops and personalized training events for application owners.

Portals for Grids/VOs

The LPDS sets up Grid portal installations serving user communities of international multi-institutional grids and grid based virtual organizations. The list of actual P-GRADE Portal installations can be found on the P-Grade Portal homepage. The list of gUSE Portal installations can be found on the gUSE homepage.

OMNeT++

The LPDS carried out the gridification of OMNeT++, a public-source, component-based, modular, discrete event simulation environment. OMNeT++ is frequently used in a wide area of simulation applications due to its strong GUI support and embeddable simulation kernel. The P-GRADE Portal environment was successfully integrated with the OMNeT++ simulation framework to enable large-scale grid resources to the simulation user community, providing significant performance increase for OMNeT++-based simulations.

GSSVA (Security Assessment Tool)

Grid Site Software Vulnerability Analyzer is a monitoring tool which collects important information about the status of the grid machines, analyzes the information gathered and compares the results using an external information repository to find the security problems of machines. The tool can be set up and maintained easily, as administrators do not need to install the client side or configure the firewalls, and the information is visualized through a graphical user interface.

LPDS Training Room

Training

Since 2004 the LPDS has been operating the Central-European Regional Training Centre of EGEE and plays active role in providing grid trainings in Europe and worldwide. With the national and international trainings the Laboratory provides knowledge transfer and targets new users from industry as well as from science. The LPDS has organized and hosted grid summer schools in 2005, 2006, and 2007. In year 2008 the LPDS organized the world's largest grid training event, the ISSGC08 Grid Summer School, and also hosted several grid related training events.

Personnel

The Head of the LPDS is Prof. Dr. Péter Kacsuk. The Deputy Head of the LPDS is Dr. Robert Lovas. 1 DSc, 5 PhDs, and over 20 full or part-time members work in the laboratory.

National Grid Service

The **National Grid Service (NGS)** aims to help UK academics and researchers carry out their research by providing easy to use access to computational, data and other resources. It is funded by two governmental bodies, Engineering and Physical Sciences Research Council (EPSRC) and the Joint Information Systems Committee (JISC).

The NGS provides compute and data resources that are accessed through a standard common set of services, the Minimum Software Stack. NGS services are based on the Globus Toolkit for job submission and storage resource broker (SRB) for data management. NGS resources host a large number of scientific software packages, such as SIESTA and GAUSSIAN. As well as providing access to compute and data resources, the NGS also offers training (through the National e-Science Centre in Edinburgh) and Grid support to all UK academics and researchers in grid computing.

The NGS has entered its fourth year of production and its third phase with funding of £3M from EPSRC and JISC . This follows an upgrade of the resources at the core sites in September 2007 .

With over 500 users and twenty five sites, the NGS is rapidly expanding, with a mission to provide coherent electronic access for UK researchers to all computational and data based resources and facilities required to carry out their research, independent of resource or researcher location.

The NGS provides a link for UK researches into the EGEE international e-infrastructure .

Background

The NGS grew out of requirements within the UK to develop a production quality Grid Computing service for use by academic researchers. Prior to the establishment of the NGS the UK e-Science program funded the development of the Grid Support Centre and later the Grid Operations Support Centre (GOSC). These were both focussed on providing support for end users in the use of and development of grid computing.

Following initial successes by the UK Engineering Task Force in developing the so called 'Level 2 Grid', which was an ad-hoc collection of individual institutes committing a variety of compute resources for use within a Grid Computing infrastructure, the NGS was funded to develop this into a full service. Initially called the ETFp Grid, or Engineering Task Force Production Grid, this soon became known as the National Grid Service (NGS). During phase 1, from October 2004 to September 2006, the NGS worked closely with the separately funded Grid Operations Support Centre (GOSC). The success of this closer collaboration led to a natural evolution of the Grid Operations Support Centre and the NGS becoming a single entity. By the start of phase 2 of the NGS, which commenced in October 2006, the activities of the NGS and the GOSC were harmonised under the single project name of the NGS. The NGS is currently in phase 3 which will run until the end of September 2011.

Services

The NGS provides the following free-to-use services to UK academic researchers:

- The UK e-Science Certification Authority, which provides digital certificates to identify UK Grid users.

- A web portal, the applications repository, which provides a graphical method for submitting jobs to the NGS.
- A resource broker, which can determine the best site or cluster for a submitted job based on its requirements and the current workloads.
- The GSI-SSHTerm, a Java-based terminal used to give command-line access NGS resources using GSI-enabled SSH.
- Oracle databases.
- The NGS P-GRADE Portal, based on P-GRADE Portal technology, which enables the creation, execution and monitoring of workflows – composed of sequential and parallel jobs – on NGS and EGEE resources.

Sites

All NGS sites provide common interfaces for users to access resources, but they are differentiated on the access they provide to NGS users.

Partner Sites

The NGS consists of the following partner sites:

- Cardiff University
- University of Glasgow Computing Service
- University of Glasgow ScotGrid
- Lancaster University
- University of Manchester
- University of Oxford
- Queen's University Belfast
- Rutherford Appleton Laboratory (STFC/RAL)
- University of Westminster
- White Rose Grid (University of Leeds)

Affiliate Sites

Affiliate sites also provide resources to the NGS but with more conditions than partner sites. Sometimes they will not provide any resources to the general NGS pool, but simply provide access to their own users through the common NGS interfaces. Affiliate sites currently consist of:

- University of Birmingham (GridPP)
- University of Bristol
- Brunel University (GridPP)
- Durham University (GridPP)
- University of Edinburgh (ECDF)
- Imperial College London (GridPP)
- Keele University
- University of Liverpool(GridPP)

- University of Manchester (GridPP)
- University of Oxford (GridPP)
- Queen Mary, University of London (GridPP)
- University of Reading
- Royal Holloway, University of London (GridPP)
- University of Sheffield
- University of Southampton
- STFC SCARF
- STFC Ceramics and Minerals Consortium (Mott) VO
- University of York (White Rose Grid at York)

More affiliate sites are currently under discussion.

Research using the NGS

All academics in the UK are eligible to apply for a free account on the NGS. The science and research achieved using the NGS covers a diverse range of subjects, from the permeation of drugs through a membrane to the welfare of ethnic minority groups in the United Kingdom. Other applications include cases for modelling the coastal oceans, modelling of HIV mutations, research into cameras for imaging patients during cancer treatments and simulating galaxy formation.

As an example use of NGS, "The motivation, methodology and implementation of an e-Social Science pilot demonstrator project entitled: Grid Enabled Micro-econometric Data Analysis (GEMEDA). This used the NGS to investigate a policy relevant social science issue: the welfare of ethnic minority groups in the United Kingdom. The underlying problem is that of a statistical analysis that uses quantitative data from more than one source. The application of grid technology to this problem allows one to integrate elements of the required empirical modelling process: data extraction, data transfer, statistical computation and results presentation, in a manner that is transparent to a casual user".

Chapter 11

Grid File System and DRMAA

Grid File System

A **Grid File System** is a computer file system whose goal is improved reliability and availability by taking advantage of many smaller file storage areas.

Components

Current file systems contain up to three components: -File Table (FAT table, MFT, etc) -File Data -MetaData (user permissions, etc)

A Grid File System would have similar needs: -File Table (or search index) -File Data -MetaData

Comparisons

Because current File Systems are designed to appear as a single disk for a single computer to manage (entirely), many new challenges arise in a grid scenario whereby any single disk within the grid should be capable of handling requests for any data contained in the grid.

Features

Most file storage utilizes layers of redundancy to achieve a high level of data protection (inability to lose data). Current means of redundancy include replication and parity checks. Such redundancy can be implemented via a RAID array (whereby multiple physical disks appear to a local computer as a single disk, which may include data replication, and/or disk partitioning). Similarly, a Grid File System would consist of some level of redundancy (either at the logical file level, or at the block level, possibly including some sort of parity check) across the various disks present in the "Grid".

Framework

First and foremost, a File Table mechanism is necessary. Additionally, the file table must include a mechanism for locating the (target/destination) file within the grid. Secondly, a mechanism for working with File Data must exist. This mechanism is responsible for making File Data available to requests.

Implementation

With the recent advent of Torrent technology, a parallel can be drawn to a Grid File System, in that a torrent tracker (and search engine) would be the "File Table", and the torrent applications (transmitting the files) would be the "File Data" component. An RSS-Feed like mechanism could be utilized by File Table nodes to indicate when new files are added to the table, to instigate replication and other similar components.

A File system which incorporates Torrent technology (distributed replication, distributed data request/fulfillment) would likely be a good start for such a technology.

If both such systems (file table, and file data) were capable of being addressed as a single entity (ie: using virtual nodes in a cluster), then growth into such a system could be easily controlled simply by deciding which uses the grid member would be responsible (File Table and file lookups, and/or File Data).

Availability

Assuming there exists some method of managing data replication (assigning quotas, etc) autonomously within the grid, data could be configured for high availability, regardless of loss or outage.

Challenges

The largest problem currently revolves around distributing data updates. Torrents support minimal hierarchy (currently implemented either as metaData in the torrent tracker, or strictly as UI and basic categorization). Updating multiple nodes concurrently (assuming atomic transactions are required) presents latency during updates and additions, usually to the point of not being feasible. Additionally, a grid (network based) file system breaks traditional TCP/IP paradigms in that a File System (generally low level, ring 0 type of operations) require complicated TCP/IP implementations, introducing layers of abstraction and complication to the process of creating such a grid file system.

Examples

Current examples of high available data include: Network Load Balancing / CARP - splitting incoming requests to multiple computers, usually configured identically or as one whole Shared Storage Clustering / SANs - a single disk (one or more physical disks acting as a single logical disk) is presented to multiple computers which split incoming requests. This is usually used when more computing power is required than disk access. Data Replication / Mirroring - multiple computers may attempt to synchronize data (usually point-in-time or snapshot based). Used more often for either Reporting (based on last snapshot) or backup purposes. Data Partitioning - splitting data among multiple computers. In databases, data is often partitioned based on tables (certain tables exist on certain computers, or a table is split among multiple computers at certain "break points")... general files tend to be partitioned either by category (category based folders), or location (geographically separated).

Grid computing would bring the benefits from many such solutions, if it were widely adopted.

DRMAA

DRMAA or **Distributed Resource Management Application API** is a high-level Open Grid Forum API specification for the submission and control of jobs to one or more Distributed Resource Management Systems (DRMS) within a Grid architecture. The scope of the API covers all the high level functionality required for Grid applications to submit, control, and monitor jobs to local Grid DRM systems.

In 2007, DRMAA was one of the first two (the other one was GridRPC) specifications that reached the *full recommendation* status in the Open Grid Forum.

Development Model

The development of this API was done through the Global Grid Forum, in the model of IETF standard development, and it was originally co-authored by:

- Roger Brobst from Cadence Design Systems
- Waiman Chan from IBM
- Fritz Ferstl from Sun Microsystems
- Jeff Gardiner from John P. Robarts Research Institute
- Andreas Haas from Sun Microsystems (Co-Chair)
- Bill Nitzberg from Altair Engineering
- Hrabri Rajic from Intel (Maintainer & Co-Chair)
- John Tollefsrud from Sun Microsystems Founding (Chair)

This specification was first proposed at Global Grid Forum 3 (GGF3) in Frascati, Italy, but gained most of its momentum at Global Grid Forum 4 in Toronto, Ontario. The development of the specification was first proposed with the objective to facilitate direct interfacing of applications to existing DRM systems by application's builders, portal builders, and Independent Software Vendors (ISVs). Because the API was co-authored by participants from a wide-selection of companies and included participants from industries and education, its development resulted in an open standard that received a relatively good reception from a wide audience quickly.

Significance

Without DRMAA, no standard model existed to submit jobs to component regions of a Grid, assuming each region was running local DRMSs. The first version of DRMAA API has been implemented in Sun's Grid Engine and also in the University of Wisconsin–Madison's program Condor. Furthermore C, Java, and IDL binding documents have been made available.

Implementations

- Sun Grid Engine
- Condor
- Torque/PBS
- GridWay
- Xgrid
- EGEE (LCG2 / gLite)
- Platform LSF
- UNICORE
- Kerrighed Cluster Framework
- IBM Tivoli Workload Scheduler LoadLeveler
- SLURM

Language Bindings

- C/C++
- Java/JavaScript
- Perl
- Python
- Ruby

Other language bindings can be generated easily from SWIG, which was first used by the Perl binding.

DRMAA applications

A number of software solutions use DRMAA to interface with different resource management systems:

- tigr-workflow
- eXludus RepliCator
- GridwiseTech Grid Engine-Globus Toolkit adapter

Chapter 12

Computer Cluster

The Silicon Graphics Cluster-SGI; an example of a cluster computer

A **computer cluster** is a group of linked computers, working together closely thus in many respects forming a single computer. The components of a cluster are commonly, but not always, connected to each other through fast local area networks. Clusters are usually deployed to improve performance and availability over that of a single computer, while typically being much more cost-effective than single computers of comparable speed or availability.

Cluster categorizations

High-availability (HA) clusters

High-availability clusters (also known as Failover Clusters) are implemented primarily for the purpose of improving the availability of services that the cluster provides. They operate by having redundant nodes, which are then used to provide service when system components fail. The most common size for an HA cluster is two nodes, which is the minimum requirement to provide redundancy. HA cluster implementations attempt to use redundancy of cluster components to eliminate single points of failure.

There are commercial implementations of High-Availability clusters for many operating systems. The Linux-HA project is one commonly used free software HA package for the Linux operating system. The LanderCluster from Lander Software can run on Windows, Linux, and UNIX platforms.

Load-balancing clusters

Load-balancing is when multiple computers are linked together to share computational workload or function as a single virtual computer. Logically, from the user side, they are multiple machines, but function as a single virtual machine. Requests initiated from the user are managed by, and distributed among, all the standalone computers to form a cluster. This results in balanced computational work among different machines, improving the performance of the cluster systems.

Compute clusters

Often clusters are used primarily for computational purposes, rather than handling IO-oriented operations such as web service or databases. For instance, a cluster might support computational simulations of weather or vehicle crashes. The primary distinction within computer clusters is how tightly-coupled the individual nodes are. For instance, a single computer job may require frequent communication among nodes - this implies that the cluster shares a dedicated network, is densely located, and probably has homogenous nodes. This cluster design is usually referred to as Beowulf Cluster. The other extreme is where a computer job uses one or few nodes, and needs little or no inter-node communication. This latter category is sometimes called "Grid" computing. Tightly-coupled compute clusters are designed for work that might traditionally have been called "supercomputing". Middleware such as MPI (Message Passing Interface) or PVM (Parallel Virtual Machine) permits compute clustering programs to be portable to a wide variety of clusters.

Implementations

The TOP500 organization's semiannual list of the 500 fastest computers usually includes many clusters. TOP500 is a collaboration between the University of Mannheim, the University of Tennessee, and the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory. As of January 2011 the top supercomputer is the Tianhe-1A in Tianjin, China, with performance of 2566 TFlops measured with the High-Performance LINPACK benchmark.

Clustering can provide significant performance benefits versus price. The System X supercomputer at Virginia Tech, the 28th most powerful supercomputer on Earth as of June 2006, is a 12.25 TFlops computer cluster of 1100 Apple XServe G5 2.3 GHz dual-processor machines (4 GB RAM, 80 GB SATA HD) running Mac OS X and using InfiniBand interconnect. The cluster initially consisted of Power Mac G5s; the rack-mountable XServes are denser than desktop Macs, reducing the aggregate size of the cluster. The total cost of the previous Power Mac system was $5.2 million, a tenth of the cost of slower mainframe computer supercomputers. (The Power Mac G5s were sold off.)

The central concept of a Beowulf cluster is the use of commercial off-the-shelf (COTS) computers to produce a cost-effective alternative to a traditional supercomputer. One project that took this to an extreme was the Stone Soupercomputer.

However it is worth noting that Flops (floating point operations per second), aren't always the best metric for supercomputer speed. Clusters can have very high Flops, but they cannot access all data in the cluster as a whole at once. Therefore clusters are excellent for parallel computation, but much poorer than traditional supercomputers at non-parallel computation.

JavaSpaces is a specification from Sun Microsystems that enables clustering computers via a distributed shared memory.

Consumer game consoles

Due to the increasing computing power of each generation of game consoles, a novel use has emerged where they are repurposed into High-performance computing(HPC) clusters. Some examples of game console clusters are Sony PlayStation clusters and Microsoft Xbox clusters.

History

The history of cluster computing is best captured by a footnote in Greg Pfister's *In Search of Clusters*: “Virtually every press release from DEC mentioning clusters says ‘DEC, who invented clusters…’. IBM did not invent them either. *Customers* invented clusters, as soon as they could not fit all their work on one computer, or needed a backup. The date of the first is unknown, but it would be surprising if it was not in the 1960s, or even late 1950s.”

The formal *engineering* basis of cluster computing as a means of doing parallel work of any sort was arguably invented by Gene Amdahl of IBM, who in 1967 published what has come to be regarded as the seminal paper on parallel processing: Amdahl's Law. Amdahl's Law describes mathematically the speedup one can expect from parallelizing any given otherwise serially performed task on a parallel architecture. Here we defined the engineering basis for both multiprocessor computing and cluster computing, where the primary differentiator is whether or not the interprocessor communications are supported "inside" the computer (on for example a customized internal communications bus or network) or "outside" the computer on a *commodity* network.

Consequently the history of early computer clusters is more or less directly tied into the history of early networks, as one of the primary motivation for the development of a network was to link computing resources, creating a de facto computer cluster. Packet switching networks were conceptually invented by the RAND corporation in 1962. Using the concept of a packet switched network, the ARPANET project succeeded in creating in 1969 what was arguably the world's first commodity-network based computer cluster by linking four different computer centers (each of which was something of a "cluster" in its own right, but probably not a *commodity* cluster). The ARPANET project grew into the Internet—which can be thought of as "the mother of all computer clusters" (as the union of nearly all of the compute resources, including clusters, that happen to be connected). It also established the paradigm in use by *all* computer clusters in the world today—the use of packet-switched networks to perform interprocessor communications between processor (sets) located in otherwise disconnected frames.

The development of customer-built and research clusters proceeded hand in hand with that of both networks and the Unix operating system from the early 1970s, as both TCP/IP and the Xerox PARC project created and formalized protocols for network-based communications. The Hydra operating system was built for a cluster of DEC PDP-11 minicomputers called C.mmp at Carnegie Mellon University in 1971. However, it was not until circa 1983 that the protocols and tools for *easily* doing remote job distribution and file sharing were defined (largely within the context of BSD Unix, as implemented by Sun Microsystems) and hence became generally available commercially, along with a shared filesystem.

The *first* commercial clustering product was ARCnet, developed by Datapoint in 1977. ARCnet was not a commercial success and clustering per se did not really take off until DEC released their VAXcluster product in 1984 for the VAX/VMS operating system. The ARCnet and VAXcluster products not only supported parallel computing, but also shared file systems and peripheral devices. The idea was to provide the advantages of parallel processing, while maintaining data reliability and uniqueness. VAXcluster, now VMScluster, is still available on OpenVMS systems from HP running on Alpha and Itanium systems.

Two other noteworthy early commercial clusters were the *Tandem Himalaya* (a circa 1994 high-availability product) and the *IBM S/390 Parallel Sysplex* (also circa 1994, primarily for business use).

No history of commodity computer clusters would be complete without noting the pivotal role played by the development of Parallel Virtual Machine (PVM) software in 1989. This open source software based on TCP/IP communications enabled the *instant* creation of a virtual supercomputer—a high performance compute cluster—made out of any TCP/IP connected systems. Free form heterogeneous clusters built on top of this model rapidly achieved total throughput in FLOPS that greatly exceeded that available even with the most expensive "big iron" supercomputers. PVM and the advent of inexpensive networked PCs led, in 1993, to a NASA project to build supercomputers out of commodity clusters. In 1995 the invention of the "beowulf"-style cluster—a compute cluster built on top of a commodity network for the specific purpose of "being a supercomputer" capable of performing tightly coupled parallel HPC computations. This in turn spurred the independent development of Grid computing as a named entity, although Grid-style clustering had been around at least as long as the Unix operating system and the Arpanet, whether or not it, or the clusters that used it, were named.

Technologies

MPI is a widely-available communications library that enables parallel programs to be written in C, Fortran, Python, OCaml, and many other programming languages.

The GNU/Linux world supports various cluster software; for application clustering, there is Beowulf, distcc, and MPICH. Linux Virtual Server, Linux-HA - director-based clusters that allow incoming requests for services to be distributed across multiple cluster nodes. MOSIX, openMosix, Kerrighed, OpenSSI are full-blown clusters integrated into the kernel that provide for automatic process migration among homogeneous nodes. OpenSSI, openMosix and Kerrighed are single-system image implementations.

Microsoft Windows Compute Cluster Server 2003 based on the Windows Server platform provides pieces for High Performance Computing like the Job Scheduler, MSMPI library and management tools. NCSA's recently installed Lincoln is a cluster of 450 Dell PowerEdge 1855 blade servers running Windows Compute Cluster Server 2003. This cluster debuted at #130 on the Top500 list in June 2006.

gridMathematica provides distributed computations over clusters including data analysis, computer algebra and 3D visualization. It can make use of other technologies such as Altair PBS Professional, Microsoft Windows Compute Cluster Server, Platform LSF and Sun Grid Engine.

gLite is a set of middleware technologies created by the Enabling Grids for E-sciencE (EGEE) project.

Another example of consumer game products being added to high-performance computing is the Nvidia Tesla Personal Supercomputer workstation, which gets its processing power by harnessing the power of multiple graphics accelerator processor chips.

Algorithmic Skeletons are a high-level parallel programming model for parallel and distributed computing which take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by combining the basic ones.

Global Storage Architecture (GSA) – a highly scalable cloud based NAS solution - combines proprietary IBM HPC technology (storage and server hardware and IBM's high-performance shared-disk clustered file system - GPFS) with open source components like Linux, Samba and CTDB to deliver distributed storage solutions. GSA exports the clustered file system through industry standard protocols like CIFS, NFS, FTP and HTTP. All of the GSA nodes in the grid export all files of all file systems simultaneously.

Chapter 13

Distributed Computing

Distributed computing is a field of computer science that studies distributed systems. A **distributed system** consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal. A computer program that runs in a distributed system is called a **distributed program**, and **distributed programming** is the process of writing such programs.

Distributed computing also refers to the use of distributed systems to solve computational problems. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers.

Introduction

The word *distributed* in terms such as "distributed system", "distributed programming", and "distributed algorithm" originally referred to computer networks where individual computers were physically distributed within some geographical area. The terms are nowadays used in a much wider sense, even referring to autonomous processes that run on the same physical computer and interact with each other by message passing.

While there is no single definition of a distributed system, the following defining properties are commonly used:

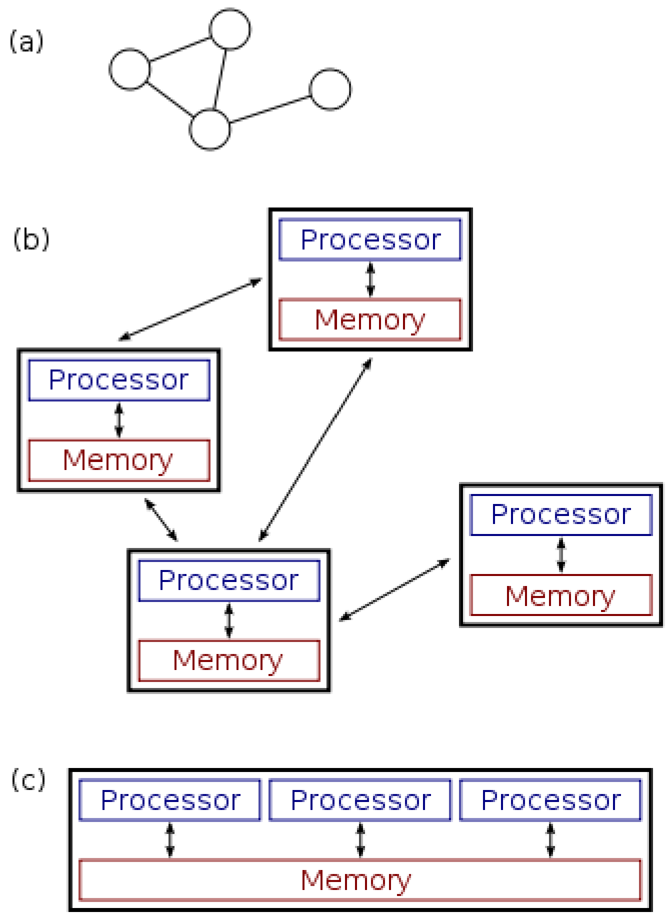
- There are several autonomous computational entities, each of which has its own local memory.
- The entities communicate with each other by message passing.

Here, the computational entities are called *computers* or *nodes*.

A distributed system may have a common goal, such as solving a large computational problem. Alternatively, each computer may have its own user with individual needs, and the purpose of the distributed system is to coordinate the use of shared resources or provide communication services to the users.

Other typical properties of distributed systems include the following:

- The system has to tolerate failures in individual computers.
- The structure of the system (network topology, network latency, number of computers) is not known in advance, the system may consist of different kinds of computers and network links, and the system may change during the execution of a distributed program.
- Each computer has only a limited, incomplete view of the system. Each computer may know only one part of the input.



(a)–(b) A distributed system.
(c) A parallel system.

Parallel and distributed computing?

Distributed systems are networked computers operating with same processors. The terms "concurrent computing", "parallel computing", and "distributed computing" have a lot of overlap, and no clear distinction exists between them. The same system may be characterised both as "parallel" and "distributed"; the processors in a typical distributed system run concurrently in parallel. Parallel computing may be seen as a particular tightly-coupled form of distributed computing, and distributed computing may be seen as a loosely-coupled form of parallel computing. Nevertheless, it is possible to roughly classify concurrent systems as "parallel" or "distributed" using the following criteria:

- In parallel computing, all processors have access to a shared memory. Shared memory can be used to exchange information between processors.
- In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.

The figure on the right illustrates the difference between distributed and parallel systems. Figure (a) is a schematic view of a typical distributed system; as usual, the system is represented as a graph in which each node (vertex) is a computer and each edge (line between two nodes) is a communication link. Figure (b) shows the same distributed system in more detail: each computer has its own local memory, and information can be exchanged only by passing messages from one node to another by using the available communication links. Figure (c) shows a parallel system in which each processor has a direct access to a shared memory.

The situation is further complicated by the traditional uses of the terms parallel and distributed *algorithm* that do not quite match the above definitions of parallel and distributed *systems*. Nevertheless, as a rule of thumb, high-performance parallel computation in a shared-memory multiprocessor uses parallel algorithms while the coordination of a large-scale distributed system uses distributed algorithms.

History

The use of concurrent processes that communicate by message-passing has its roots in operating system architectures studied in the 1960s. The first widespread distributed systems were local-area networks such as Ethernet that was invented in the 1970s.

ARPANET, the predecessor of the Internet, was introduced in the late 1960s, and ARPANET e-mail was invented in the early 1970s. E-mail became the most successful application of ARPANET, and it is probably the earliest example of a large-scale distributed application. In addition to ARPANET and its successor Internet, other early worldwide computer networks included Usenet and FidoNet from 1980s, both of which were used to support distributed discussion systems.

The study of distributed computing became its own branch of computer science in the late 1970s and early 1980s. The first conference in the field, Symposium on Principles of

Distributed Computing (PODC), dates back to 1982, and its European counterpart International Symposium on Distributed Computing (DISC) was first held in 1985.

Applications

There are two main reasons for using distributed systems and distributed computing. First, the very nature of the application may *require* the use of a communication network that connects several computers. For example, data is produced in one physical location and it is needed in another location.

Second, there are many cases in which the use of a single computer would be possible in principle, but the use of a distributed system is *beneficial* for practical reasons. For example, it may be more cost-efficient to obtain the desired level of performance by using a cluster of several low-end computers, in comparison with a single high-end computer. A distributed system can be more reliable than a non-distributed system, as there is no single point of failure. Moreover, a distributed system may be easier to expand and manage than a monolithic uniprocessor system.

Examples of distributed systems and applications of distributed computing include the following:

- Telecommunication networks:
    - Telephone networks and cellular networks.
    - Computer networks such as the Internet.
    - Wireless sensor networks.
    - Routing algorithms.
- Network applications:
    - World wide web and peer-to-peer networks.
    - Massively multiplayer online games and virtual reality communities.
    - Distributed databases and distributed database management systems.
    - Network file systems.
    - Distributed information processing systems such as banking systems and airline reservation systems.
- Real-time process control:
    - Aircraft control systems.
    - Industrial control systems.
- Parallel computation:
    - Scientific computing, including cluster computing and grid computing and various volunteer computing projects.
    - Distributed rendering in computer graphics.

Theoretical foundations

Models

Many tasks that we would like to automate by using a computer are of question–answer type: we would like to ask a question and the computer should produce an answer. In theoretical computer science, such tasks are called computational problems. Formally, a computational problem consists of *instances* together with a *solution* for each instance. Instances are questions that we can ask, and solutions are desired answers to these questions.

Theoretical computer science seeks to understand which computational problems can be solved by using a computer (computability theory) and how efficiently (computational complexity theory). Traditionally, it is said that a problem can be solved by using a computer if we can design an algorithm that produces a correct solution for any given instance. Such an algorithm can be implemented as a computer program that runs on a general-purpose computer: the program reads a problem instance from input, performs some computation, and produces the solution as output. Formalisms such as random access machines or universal Turing machines can be used as abstract models of a sequential general-purpose computer executing such an algorithm.

The field of concurrent and distributed computing studies similar questions in the case of either multiple computers, or a computer that executes a network of interacting processes: which computational problems can be solved in such a network and how efficiently? However, it is not at all obvious what is meant by "solving a problem" in the case of a concurrent or distributed system: for example, what is the task of the algorithm designer, and what is the concurrent or distributed equivalent of a sequential general-purpose computer?

The discussion below focusses on the case of multiple computers, although many of the issues are the same for concurrent processes running on a single computer.

Three viewpoints are commonly used:

Parallel algorithms in shared-memory model

- All computers have access to a shared memory. The algorithm designer chooses the program executed by each computer.
- One theoretical model is the parallel random access machines (PRAM) that are used. However, the classical PRAM model assumes synchronous access to the shared memory.
- A model that is closer to the behavior of real-world multiprocessor machines and takes into account the use of machine instructions, such as Compare-and-swap (CAS), is that of *asynchronous shared memory*. There is a wide body of work on this model, a summary of which can be found in the literature.

Parallel algorithms in message-passing model

- The algorithm designer chooses the structure of the network, as well as the program executed by each computer.
- Models such as Boolean circuits and sorting networks are used. A Boolean circuit can be seen as a computer network: each gate is a computer that runs an extremely simple computer program. Similarly, a sorting network can be seen as a computer network: each comparator is a computer.

Distributed algorithms in message-passing model

- The algorithm designer only chooses the computer program. All computers run the same program. The system must work correctly regardless of the structure of the network.
- A commonly used model is a graph with one finite-state machine per node.

In the case of distributed algorithms, computational problems are typically related to graphs. Often the graph that describes the structure of the computer network *is* the problem instance. This is illustrated in the following example.

An example

Consider the computational problem of finding a coloring of a given graph *G*. Different fields might take the following approaches:

Centralized algorithms

- The graph *G* is encoded as a string, and the string is given as input to a computer. The computer program finds a coloring of the graph, encodes the coloring as a string, and outputs the result.

Parallel algorithms

- Again, the graph *G* is encoded as a string. However, multiple computers can access the same string in parallel. Each computer might focus on one part of the graph and produce a colouring for that part.
- The main focus is on high-performance computation that exploits the processing power of multiple computers in parallel.

Distributed algorithms

- The graph *G* is the structure of the computer network. There is one computer for each node of *G* and one communication link for each edge of *G*. Initially, each computer only knows about its immediate neighbours in the graph *G*; the computers must exchange messages with each other to discover more about the structure of *G*. Each computer must produce its own colour as output.

- The main focus is on coordinating the operation of an arbitrary distributed system.

While the field of parallel algorithms has a different focus than the field of distributed algorithms, there is a lot of interaction between the two fields. For example, the Cole–Vishkin algorithm for graph colouring was originally presented as a parallel algorithm, but the same technique can also be used directly as a distributed algorithm.

Moreover, a parallel algorithm can be implemented either in a parallel system (using shared memory) or in a distributed system (using message passing). The traditional boundary between parallel and distributed algorithms (choose a suitable network vs. run in any given network) does not lie in the same place as the boundary between parallel and distributed systems (shared memory vs. message passing).

Complexity measures

In parallel algorithms, yet another resource in addition to time and space is the number of computers. Indeed, often there is a trade-off between the running time and the number of computers: the problem can be solved faster if there are more computers running in parallel. If a decision problem can be solved in polylogarithmic time by using a polynomial number of processors, then the problem is said to be in the class NC. The class NC can be defined equally well by using the PRAM formalism or Boolean circuits – PRAM machines can simulate Boolean circuits efficiently and vice versa.

In the analysis of distributed algorithms, more attention is usually paid on communication operations than computational steps. Perhaps the simplest model of distributed computing is a synchronous system where all nodes operate in a lockstep fashion. During each *communication round*, all nodes in parallel (1) receive the latest messages from their neighbours, (2) perform arbitrary local computation, and (3) send new messages to their neighbours. In such systems, a central complexity measure is the number of synchronous communication rounds required to complete the task.

This complexity measure is closely related to the diameter of the network. Let D be the diameter of the network. On the one hand, any computable problem can be solved trivially in a synchronous distributed system in approximately $2D$ communication rounds: simply gather all information in one location (D rounds), solve the problem, and inform each node about the solution (D rounds).

On the other hand, if the running time of the algorithm is much smaller than D communication rounds, then the nodes in the network must produce their output without having the possibility to obtain information about distant parts of the network. In other words, the nodes must make globally consistent decisions based on information that is available in their *local neighbourhood*. Many distributed algorithms are known with the running time much smaller than D rounds, and understanding which problems can be solved by such algorithms is one of the central research questions of the field.

Other commonly used measures are the total number of bits transmitted in the network (cf. communication complexity).

Other problems

Traditional computational problems take the perspective that we ask a question, a computer (or a distributed system) processes the question for a while, and then produces an answer and stops. However, there are also problems where we do not want the system to ever stop. Examples of such problems include the dining philosophers problem and other similar mutual exclusion problems. In these problems, the distributed system is supposed to continuously coordinate the use of shared resources so that no conflicts or deadlocks occur.

There are also fundamental challenges that are unique to distributed computing. The first example is challenges that are related to *fault-tolerance*. Examples of related problems include consensus problems, Byzantine fault tolerance, and self-stabilisation.

A lot of research is also focused on understanding the *asynchronous* nature of distributed systems:

- Synchronizers can be used to run synchronous algorithms in asynchronous systems.
- Logical clocks provide a causal happened-before ordering of events.
- Clock synchronization algorithms provide globally consistent physical time stamps.

Properties of distributed systems

So far the focus has been on *designing* a distributed system that solves a given problem. A complementary research problem is *studying* the properties of a given distributed system.

The halting problem is an analogous example from the field of centralised computation: we are given a computer program and the task is to decide whether it halts or runs forever. The halting problem is undecidable in the general case, and naturally understanding the behaviour of a computer network is at least as hard as understanding the behaviour of one computer.

However, there are many interesting special cases that are decidable. In particular, it is possible to reason about the behaviour of a network of finite-state machines. One example is telling whether a given network of interacting (asynchronous and non-deterministic) finite-state machines can reach a deadlock. This problem is PSPACE-complete, i.e., it is decidable, but it is not likely that there is an efficient (centralised, parallel or distributed) algorithm that solves the problem in the case of large networks.

Architectures

Various hardware and software architectures are used for distributed computing. At a lower level, it is necessary to interconnect multiple CPUs with some sort of network, regardless of whether that network is printed onto a circuit board or made up of loosely-coupled devices and cables. At a higher level, it is necessary to interconnect processes running on those CPUs with some sort of communication system.

Distributed programming typically falls into one of several basic architectures or categories: client–server, 3-tier architecture, *n*-tier architecture, distributed objects, loose coupling, or tight coupling.

- Client–server: Smart client code contacts the server for data then formats and displays it to the user. Input at the client is committed back to the server when it represents a permanent change.
- 3-tier architecture: Three tier systems move the client intelligence to a middle tier so that stateless clients can be used. This simplifies application deployment. Most web applications are 3-Tier.
- *n*-tier architecture: *n*-tier refers typically to web applications which further forward their requests to other enterprise services. This type of application is the one most responsible for the success of application servers.
- Tightly coupled (clustered): refers typically to a cluster of machines that closely work together, running a shared process in parallel. The task is subdivided in parts that are made individually by each one and then put back together to make the final result.
- Peer-to-peer: an architecture where there is no special machine or machines that provide a service or manage the network resources. Instead all responsibilities are uniformly divided among all machines, known as peers. Peers can serve both as clients and servers.
- Space based: refers to an infrastructure that creates the illusion (virtualization) of one single address-space. Data are transparently replicated according to application needs. Decoupling in time, space and reference is achieved.

Another basic aspect of distributed computing architecture is the method of communicating and coordinating work among concurrent processes. Through various message passing protocols, processes may communicate directly with one another, typically in a master/slave relationship. Alternatively, a "database-centric" architecture can enable distributed computing to be done without any form of direct inter-process communication, by utilizing a shared database.

Chapter 14

Parallel Computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism—with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically one of the greatest obstacles to getting good parallel program performance.

The maximum possible speed-up of a program as a result of parallelization is observed as Amdahl's law.

Background

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions.

These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time—after that instruction is finished, the next is executed.

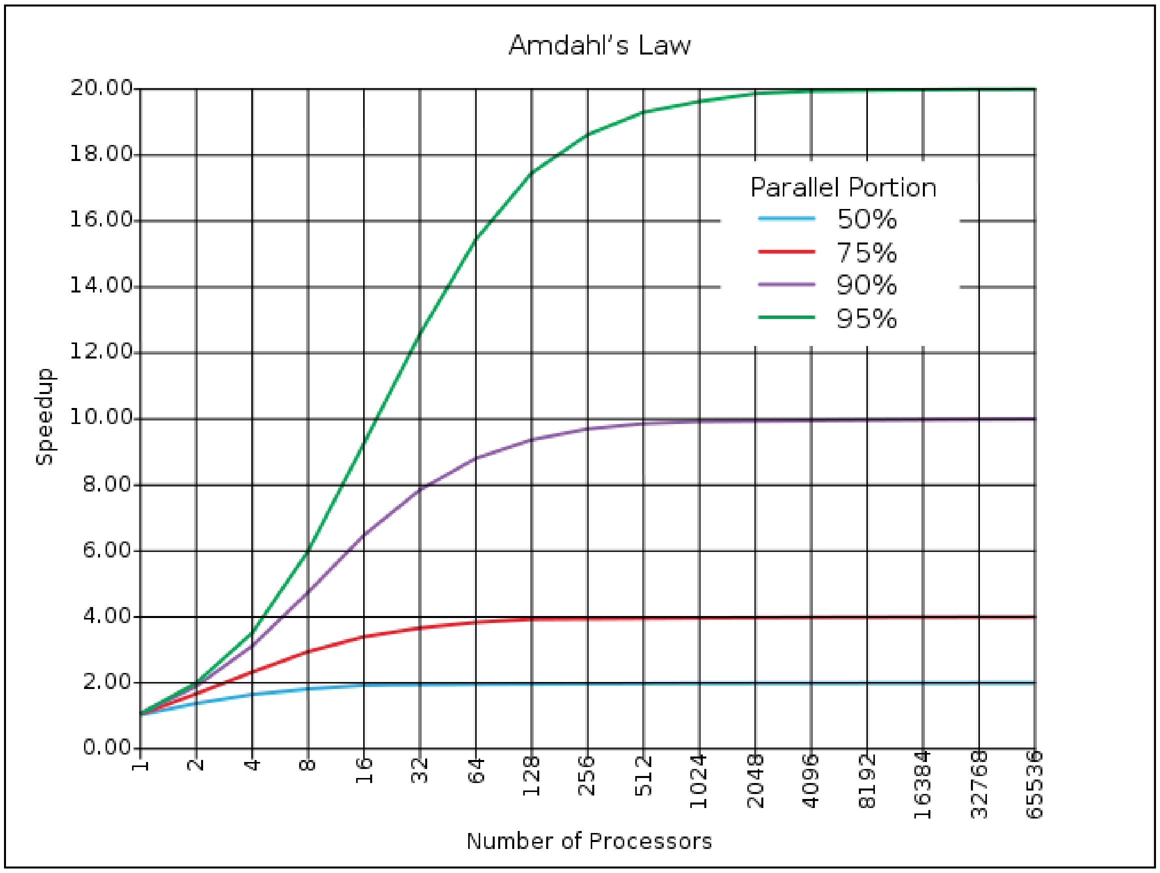
Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above.

Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. An increase in frequency thus decreases runtime for all computation-bounded programs.

However, power consumption by a chip is given by the equation $P = C \times V^2 \times F$, where P is power, C is the capacitance being switched per clock cycle (proportional to the number of transistors whose inputs change), V is voltage, and F is the processor frequency (cycles per second). Increases in frequency increase the amount of power used in a processor. Increasing processor power consumption led ultimately to Intel's May 2004 cancellation of its Tejas and Jayhawk processors, which is generally cited as the end of frequency scaling as the dominant computer architecture paradigm.

Moore's Law is the empirical observation that transistor density in a microprocessor doubles every 18 to 24 months. Despite power consumption issues, and repeated predictions of its end, Moore's law is still in effect. With the end of frequency scaling, these additional transistors (which are no longer used for frequency scaling) can be used to add extra hardware for parallel computing.

Amdahl's law and Gustafson's law



A graphical representation of Amdahl's law. The speed-up of a program from parallelization is limited by how much of the program can be parallelized. For example, if 90% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 10x no matter how many processors are used.

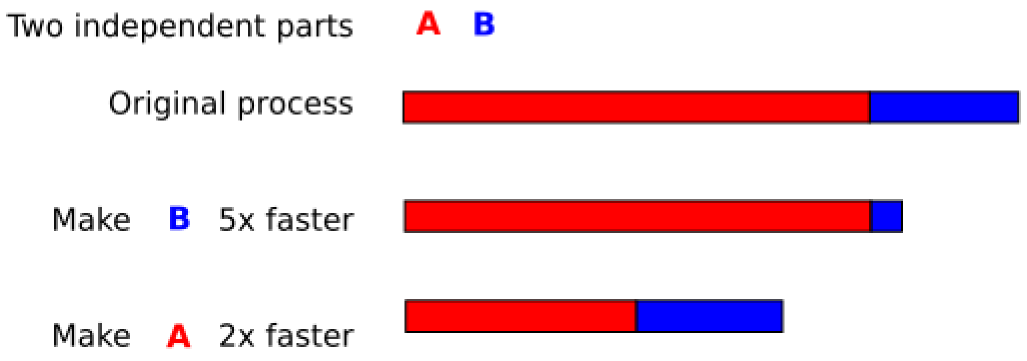
Optimally, the speed-up from parallelization would be linear—doubling the number of processing elements should halve the runtime, and doubling it a second time should again halve the runtime. However, very few parallel algorithms achieve optimal speed-up. Most of them have a near-linear speed-up for small numbers of processing elements, which flattens out into a constant value for large numbers of processing elements.

The potential speed-up of an algorithm on a parallel computing platform is given by Amdahl's law, originally formulated by Gene Amdahl in the 1960s. It states that a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization. A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. If α is the fraction of running time a sequential program spends on non-parallelizable parts, then

$$S = \frac{1}{\alpha}$$

is the maximum speed-up with parallelization of the program. If the sequential portion of a program accounts for 10% of the runtime, we can get no more than a 10× speed-up, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. "When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned."

Gustafson's law is another law in computing, closely related to Amdahl's law. It states that the speedup with *P* processors is



Assume that a task has two independent parts, A and B. B takes roughly 25% of the time of the whole computation. With effort, a programmer may be able to make this part five times faster, but this only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part A twice as fast. This will make the computation much faster than by optimizing part B, even though B got a greater speed-up (5× versus 2×).

$$S(P) = P - \alpha(P - 1).$$

Amdahl's law assumes a fixed problem size and that the running time of the sequential section of the program is independent of the number of processors, whereas Gustafson's law does not make these assumptions.

Dependencies

Understanding data dependencies is fundamental in implementing parallel algorithms. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.

Let P_i and P_j be two program fragments. Bernstein's conditions describe when the two are independent and can be executed in parallel. For P_i, let I_i be all of the input variables and O_i the output variables, and likewise for P_j. P_i and P_j are independent if they satisfy

- $I_j \cap O_i = \varnothing,$
- $I_i \cap O_j = \varnothing,$
- $O_i \cap O_j = \varnothing.$

Violation of the first condition introduces a flow dependency, corresponding to the first statement producing a result used by the second statement. The second condition represents an anti-dependency, when the second statement (P_j) would overwrite a variable needed by the first expression (P_i). The third and final condition represents an output dependency: When two statements write to the same location, the final result must come from the logically last executed statement.

Consider the following functions, which demonstrate several kinds of dependencies:

```
1: function Dep(a, b)
2:    c := a·b
3:    d := 2·c
4: end function
```

Operation 3 in Dep(a, b) cannot be executed before (or even in parallel with) operation 2, because operation 3 uses a result from operation 2. It violates condition 1, and thus introduces a flow dependency.

```
1: function NoDep(a, b)
2:      c := a·b
3:      d := 2·b
4:      e := a+b
5: end function
```

In this example, there are no dependencies between the instructions, so they can all be run in parallel.

Bernstein's conditions do not allow memory to be shared between different processes. For that, some means of enforcing an ordering between accesses is necessary, such as semaphores, barriers or some other synchronization method.

Race conditions, mutual exclusion, synchronization, and parallel slowdown

Subtasks in a parallel program are often called threads. Some parallel computer architectures use smaller, lightweight versions of threads known as fibers, while others use bigger versions known as processes. However, "threads" is generally accepted as a generic term for subtasks. Threads will often need to update some variable that is shared between them. The instructions between the two programs may be interleaved in any order. For example, consider the following program:

Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A Write back to variable V	3B: Write back to variable V

If instruction 1B is executed between 1A and 3A, or if instruction 1A is executed between 1B and 3B, the program will produce incorrect data. This is known as a race condition. The programmer must use a lock to provide mutual exclusion. A lock is a programming language construct that allows one thread to take control of a variable and prevent other threads from reading or writing it, until that variable is unlocked. The thread holding the lock is free to execute its critical section (the section of a program that requires exclusive access to some variable), and to unlock the data when it is finished. Therefore, to guarantee correct program execution, the above program can be rewritten to use locks:

Thread A	Thread B
1A: Lock variable V	1B: Lock variable V
2A: Read variable V	2B: Read variable V
3A: Add 1 to variable V	3B: Add 1 to variable V
4A Write back to variable V	4B: Write back to variable V
5A: Unlock variable V	5B: Unlock variable V

One thread will successfully lock variable V, while the other thread will be locked out—unable to proceed until V is unlocked again. This guarantees correct execution of the program. Locks, while necessary to ensure correct program execution, can greatly slow a program.

Locking multiple variables using non-atomic locks introduces the possibility of program deadlock. An atomic lock locks multiple variables all at once. If it cannot lock all of them, it does not lock any of them. If two threads each need to lock the same two variables using non-atomic locks, it is possible that one thread will lock one of them and the second thread will lock the second variable. In such a case, neither thread can complete, and deadlock results.

Many parallel programs require that their subtasks act in synchrony. This requires the use of a barrier. Barriers are typically implemented using a software lock. One class of algorithms, known as lock-free and wait-free algorithms, altogether avoids the use of locks and barriers. However, this approach is generally difficult to implement and requires correctly designed data structures.

Not all parallelization results in speed-up. Generally, as a task is split up into more and more threads, those threads spend an ever-increasing portion of their time communicating with each other. Eventually, the overhead from communication dominates the time spent solving the problem, and further parallelization (that is, splitting the workload over even more threads) increases rather than decreases the amount of time required to finish. This is known as parallel slowdown.

Fine-grained, coarse-grained, and embarrassing parallelism

Applications are often classified according to how often their subtasks need to synchronize or communicate with each other. An application exhibits fine-grained parallelism if its subtasks must communicate many times per second; it exhibits coarse-grained parallelism if they do not communicate many times per second, and it is embarrassingly parallel if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize.

Consistency models

Parallel programming languages and parallel computers must have a consistency model (also known as a memory model). The consistency model defines rules for how operations on computer memory occur and how results are produced.

One of the first consistency models was Leslie Lamport's sequential consistency model. Sequential consistency is the property of a parallel program that its parallel execution produces the same results as a sequential program. Specifically, a program is sequentially consistent if "... the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program".

Software transactional memory is a common type of consistency model. Software transactional memory borrows from database theory the concept of atomic transactions and applies them to memory accesses.

Mathematically, these models can be represented in several ways. Petri nets, which were introduced in Carl Adam Petri's 1962 doctoral thesis, were an early attempt to codify the rules of consistency models. Dataflow theory later built upon these, and Dataflow architectures were created to physically implement the ideas of dataflow theory. Beginning in the late 1970s, process calculi such as Calculus of Communicating Systems and Communicating Sequential Processes were developed to permit algebraic reasoning about systems composed of interacting components. More recent additions to the process calculus family, such as the π-calculus, have added the capability for reasoning about dynamic topologies. Logics such as Lamport's TLA+, and mathematical models such as traces and Actor event diagrams, have also been developed to describe the behavior of concurrent systems.

Flynn's taxonomy

Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy. Flynn classified programs and computers by whether they were operating using a single set or multiple sets of instructions, whether or not those instructions were using a single or multiple sets of data.

Flynn's taxonomy

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

The single-instruction-single-data (SISD) classification is equivalent to an entirely sequential program. The single-instruction-multiple-data (SIMD) classification is analogous to doing the same operation repeatedly over a large data set. This is commonly done in signal processing applications. Multiple-instruction-single-data (MISD) is a rarely used classification. While computer architectures to deal with this were devised (such as systolic arrays), few applications that fit this class materialized. Multiple-instruction-multiple-data (MIMD) programs are by far the most common type of parallel programs.

According to David A. Patterson and John L. Hennessy, "Some machines are hybrids of these categories, of course, but this classic model has survived because it is simple, easy to understand, and gives a good first approximation. It is also—perhaps because of its understandability—the most widely used scheme."

Types of parallelism

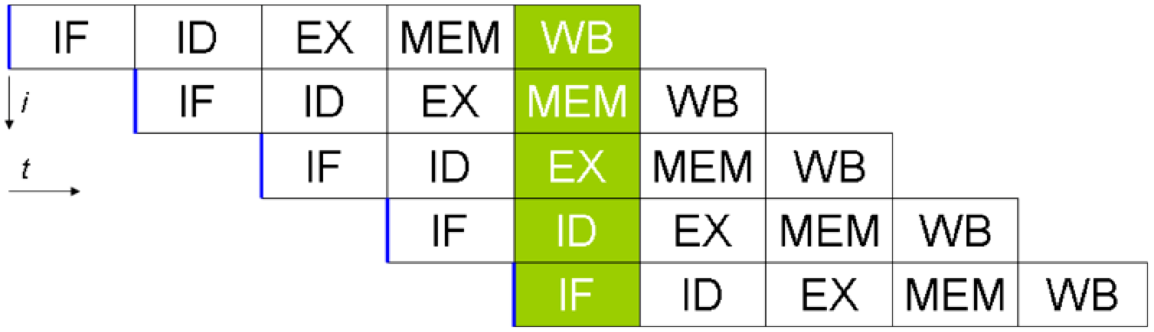
Bit-level parallelism

From the advent of very-large-scale integration (VLSI) computer-chip fabrication technology in the 1970s until about 1986, speed-up in computer architecture was driven by doubling computer word size—the amount of information the processor can manipulate per cycle. Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose sizes are greater than the length of the word. For example, where an 8-bit processor must add two 16-bit integers, the processor must first add the 8 lower-order bits from each integer using the standard addition instruction, then add the 8 higher-order bits using an add-with-carry instruction and the carry bit from the lower order addition; thus, an 8-bit processor requires two instructions to complete a single operation, where a 16-bit processor would be able to complete the operation with a single instruction.

Historically, 4-bit microprocessors were replaced with 8-bit, then 16-bit, then 32-bit microprocessors. This trend generally came to an end with the introduction of 32-bit processors, which has been a standard in general-purpose computing for two decades.

Not until recently (c. 2003–2004), with the advent of x86-64 architectures, have 64-bit processors become commonplace.

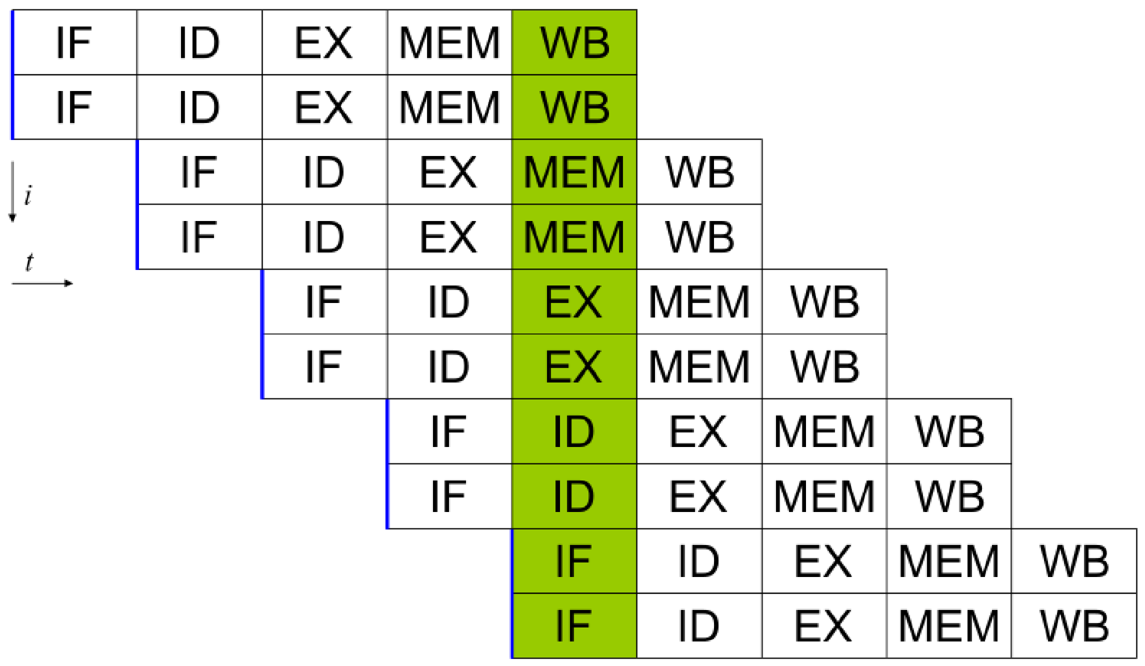
Instruction-level parallelism



A canonical five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

A computer program is, in essence, a stream of instructions executed by a processor. These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism. Advances in instruction-level parallelism dominated computer architecture from the mid-1980s until the mid-1990s.

Modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage; a processor with an N-stage pipeline can have up to N different instructions at different stages of completion. The canonical example of a pipelined processor is a RISC processor, with five stages: instruction fetch, decode, execute, memory access, and write back. The Pentium 4 processor had a 35-stage pipeline.



A five-stage pipelined superscalar processor, capable of issuing two instructions per cycle. It can have two instructions in each stage of the pipeline, for a total of up to 10 instructions (shown in green) being simultaneously executed.

In addition to instruction-level parallelism from pipelining, some processors can issue more than one instruction at a time. These are known as superscalar processors. Instructions can be grouped together only if there is no data dependency between them. Scoreboarding and the Tomasulo algorithm (which is similar to scoreboarding but makes use of register renaming) are two of the most common techniques for implementing out-of-order execution and instruction-level parallelism .

Data parallelism

Data parallelism is parallelism inherent in program loops, which focuses on distributing the data across different computing nodes to be processed in parallel. "Parallelizing loops often leads to similar (not necessarily identical) operation sequences or functions being performed on elements of a large data structure." Many scientific and engineering applications exhibit data parallelism.

A loop-carried dependency is the dependence of a loop iteration on the output of one or more previous iterations. Loop-carried dependencies prevent the parallelization of loops. For example, consider the following pseudocode that computes the first few Fibonacci numbers:

```
1:    PREV1 := 0
2:    PREV2 := 1
4:    do:
5:       CUR := PREV1 + PREV2
```

```
        PREV1 := PREV2
        PREV2 := CUR
    while (CUR < 10)
```

This loop cannot be parallelized because CUR depends on itself (PREV2) and PREV1, which are computed in each loop iteration. Since each iteration depends on the result of the previous one, they cannot be performed in parallel. As the size of a problem gets bigger, the amount of data-parallelism available usually does as well.

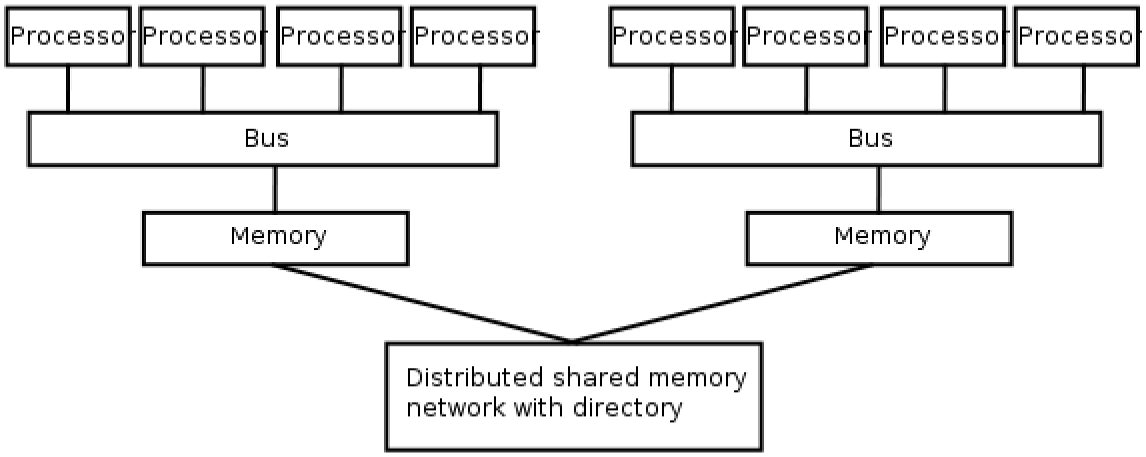
Task parallelism

Task parallelism is the characteristic of a parallel program that "entirely different calculations can be performed on either the same or different sets of data". This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data. Task parallelism does not usually scale with the size of a problem.

Hardware

Memory and communication

Main memory in a parallel computer is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space). Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well. Distributed shared memory and memory virtualization combine the two approaches, where the processing element has its own local memory and access to the memory on non-local processors. Accesses to local memory are typically faster than accesses to non-local memory.



A logical view of a Non-Uniform Memory Access (NUMA) architecture. Processors in one directory can access that directory's memory with less latency than they can access memory in the other directory's memory.

Computer architectures in which each element of main memory can be accessed with equal latency and bandwidth are known as Uniform Memory Access (UMA) systems. Typically, that can be achieved only by a shared memory system, in which the memory is not physically distributed. A system that does not have this property is known as a Non-Uniform Memory Access (NUMA) architecture. Distributed memory systems have non-uniform memory access.

Computer systems make use of caches—small, fast memories located close to the processor which store temporary copies of memory values (nearby in both the physical and logical sense). Parallel computer systems have difficulties with caches that may store the same value in more than one location, with the possibility of incorrect program execution. These computers require a cache coherency system, which keeps track of cached values and strategically purges them, thus ensuring correct program execution. Bus snooping is one of the most common methods for keeping track of which values are being accessed (and thus should be purged). Designing large, high-performance cache coherence systems is a very difficult problem in computer architecture. As a result, shared-memory computer architectures do not scale as well as distributed memory systems do.

Processor–processor and processor–memory communication can be implemented in hardware in several ways, including via shared (either multiported or multiplexed) memory, a crossbar switch, a shared bus or an interconnect network of a myriad of topologies including star, ring, tree, hypercube, fat hypercube (a hypercube with more than one processor at a node), or n-dimensional mesh.

Parallel computers based on interconnect networks need to have some kind of routing to enable the passing of messages between nodes that are not directly connected. The medium used for communication between the processors is likely to be hierarchical in large multiprocessor machines.

Classes of parallel computers

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes. These are not mutually exclusive; for example, clusters of symmetric multiprocessors are relatively common.

Multicore computing

A multicore processor is a processor that includes multiple execution units ("cores") on the same chip. These processors differ from superscalar processors, which can issue multiple instructions per cycle from one instruction stream (thread); by contrast, a multicore processor can issue multiple instructions per cycle from multiple instruction streams. Each core in a multicore processor can potentially be superscalar as well—that is, on every cycle, each core can issue multiple instructions from one instruction stream.

Simultaneous multithreading (of which Intel's HyperThreading is the best known) was an early form of pseudo-multicoreism. A processor capable of simultaneous multithreading has only one execution unit ("core"), but when that execution unit is idling (such as during a cache miss), it uses that execution unit to process a second thread. IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is another prominent multicore processor.

Symmetric multiprocessing

A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus. Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors. "Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists."

Distributed computing

A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable.

Cluster computing

A Beowulf cluster

A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. While machines in a cluster do not have to be symmetric, load balancing is more difficult if they are not. The most common type of cluster is the Beowulf cluster, which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network. Beowulf technology was originally developed by Thomas Sterling and Donald Becker. The vast majority of the TOP500 supercomputers are clusters.

Massive parallel processing

A massively parallel processor (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks (whereas clusters use commodity hardware for networking). MPPs also tend to be larger than clusters, typically having "far more" than 100 processors. In an MPP, "each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect."

A cabinet from Blue Gene/L, ranked as the fourth fastest supercomputer in the world according to the 11/2008 TOP500 rankings. Blue Gene/L is a massively parallel processor.

Blue Gene/L, the fifth fastest supercomputer in the world according to the June 2009 TOP500 ranking, is an MPP.

Grid computing

Grid computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the Internet, grid computing typically deals only with embarrassingly parallel problems. Many grid computing applications have been created, of which SETI@home and Folding@Home are the best-known examples.

Most grid computing applications use middleware, software that sits between the operating system and the application to manage network resources and standardize the software interface. The most common grid computing middleware is the Berkeley Open Infrastructure for Network Computing (BOINC). Often, grid computing software makes use of "spare cycles", performing computations at times when a computer is idling.

Specialized parallel computers

Within parallel computing, there are specialized parallel devices that remain niche areas of interest. While not domain-specific, they tend to be applicable to only a few classes of parallel problems.

Reconfigurable computing with field-programmable gate arrays

Reconfigurable computing is the use of a field-programmable gate array (FPGA) as a co-processor to a general-purpose computer. An FPGA is, in essence, a computer chip that can rewire itself for a given task.

FPGAs can be programmed with hardware description languages such as VHDL or Verilog. However, programming in these languages can be tedious. Several vendors have created C to HDL languages that attempt to emulate the syntax and/or semantics of the C programming language, with which most programmers are familiar. The best known C to HDL languages are Mitrion-C, Impulse C, DIME-C, and Handel-C. Specific subsets of SystemC based on C++ can also be used for this purpose.

AMD's decision to open its HyperTransport technology to third-party vendors has become the enabling technology for high-performance reconfigurable computing. According to Michael R. D'Amour, Chief Operating Officer of DRC Computer Corporation, "when we first walked into AMD, they called us 'the socket stealers.' Now they call us their partners."

General-purpose computing on graphics processing units (GPGPU)

Nvidia's Tesla GPGPU card

General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in computer engineering research. GPUs are co-processors that have been heavily optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations—particularly linear algebra matrix operations.

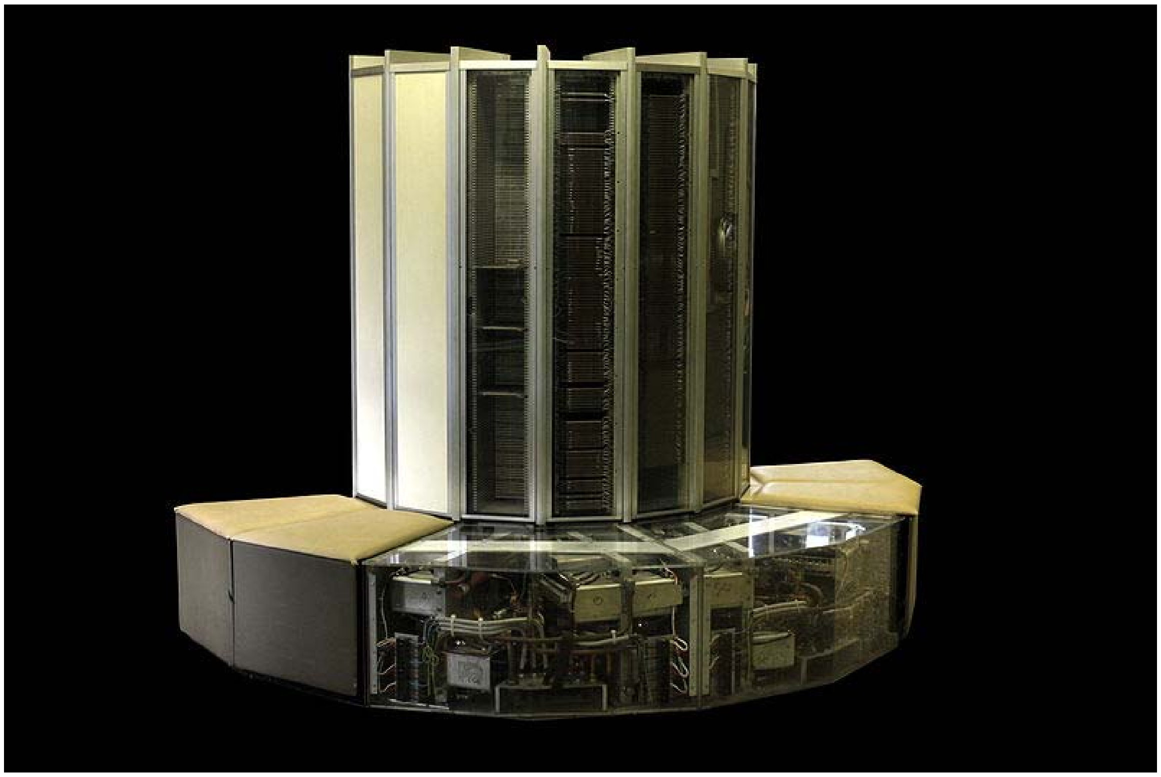
In the early days, GPGPU programs used the normal graphics APIs for executing programs. However, several new programming languages and platforms have been built to do general purpose computation on GPUs with both Nvidia and AMD releasing programming environments with CUDA and CTM respectively. Other GPU programming languages are BrookGPU, PeakStream, and RapidMind. Nvidia has also released specific products for computation in their Tesla series. The technology consortium Khronos Group has released the OpenCL specification, which is a framework for writing programs that execute across platforms consisting of CPUs and GPUs. Apple, Intel, Nvidia and others are supporting OpenCL.

Application-specific integrated circuits

Several application-specific integrated circuit (ASIC) approaches have been devised for dealing with parallel applications.

Because an ASIC is (by definition) specific to a given application, it can be fully optimized for that application. As a result, for a given application, an ASIC tends to outperform a general-purpose computer. However, ASICs are created by X-ray lithography. This process requires a mask, which can be extremely expensive. A single mask can cost over a million US dollars. (The smaller the transistors required for the chip, the more expensive the mask will be.) Meanwhile, performance increases in general-purpose computing over time (as described by Moore's Law) tend to wipe out these gains in only one or two chip generations. High initial cost, and the tendency to be overtaken by Moore's-law-driven general-purpose computing, has rendered ASICs unfeasible for most parallel computing applications. However, some have been built. One example is the peta-flop RIKEN MDGRAPE-3 machine which uses custom ASICs for molecular dynamics simulation.

Vector processors



The Cray-1 is the most famous vector processor.

A vector processor is a CPU or computer system that can execute the same instruction on large sets of data. "Vector processors have high-level operations that work on linear arrays of numbers or vectors. An example vector operation is $A = B \times C$, where A, B, and C are each 64-element vectors of 64-bit floating-point numbers." They are closely related to Flynn's SIMD classification.

Cray computers became famous for their vector-processing computers in the 1970s and 1980s. However, vector processors—both as CPUs and as full computer systems—have generally disappeared. Modern processor instruction sets do include some vector processing instructions, such as with AltiVec and Streaming SIMD Extensions (SSE).

Software

Parallel programming languages

Concurrent programming languages, libraries, APIs, and parallel programming models (such as Algorithmic Skeletons) have been created for programming parallel computers. These can generally be divided into classes based on the assumptions they make about the underlying memory architecture—shared memory, distributed memory, or shared distributed memory. Shared memory programming languages communicate by manipulating shared memory variables. Distributed memory uses message passing. POSIX Threads and OpenMP are two of most widely used shared memory APIs, whereas Message Passing Interface (MPI) is the most widely used message-passing system API. One concept used in programming parallel programs is the future concept, where one part of a program promises to deliver a required datum to another part of a program at some future time.

Automatic parallelization

Automatic parallelization of a sequential program by a compiler is the holy grail of parallel computing. Despite decades of work by compiler researchers, automatic parallelization has had only limited success.

Mainstream parallel programming languages remain either explicitly parallel or (at best) partially implicit, in which a programmer gives the compiler directives for parallelization. A few fully implicit parallel programming languages exist—SISAL, Parallel Haskell, and (for FPGAs) Mitrion-C.

Application checkpointing

The larger and more complex a computer is, the more that can go wrong and the shorter the mean time between failures. Application checkpointing is a technique whereby the computer system takes a "snapshot" of the application—a record of all current resource allocations and variable states, akin to a core dump; this information can be used to restore the program if the computer should fail. Application checkpointing means that the program has to restart from only its last checkpoint rather than the beginning. For an application that may run for months, that is critical. Application checkpointing may be used to facilitate process migration.

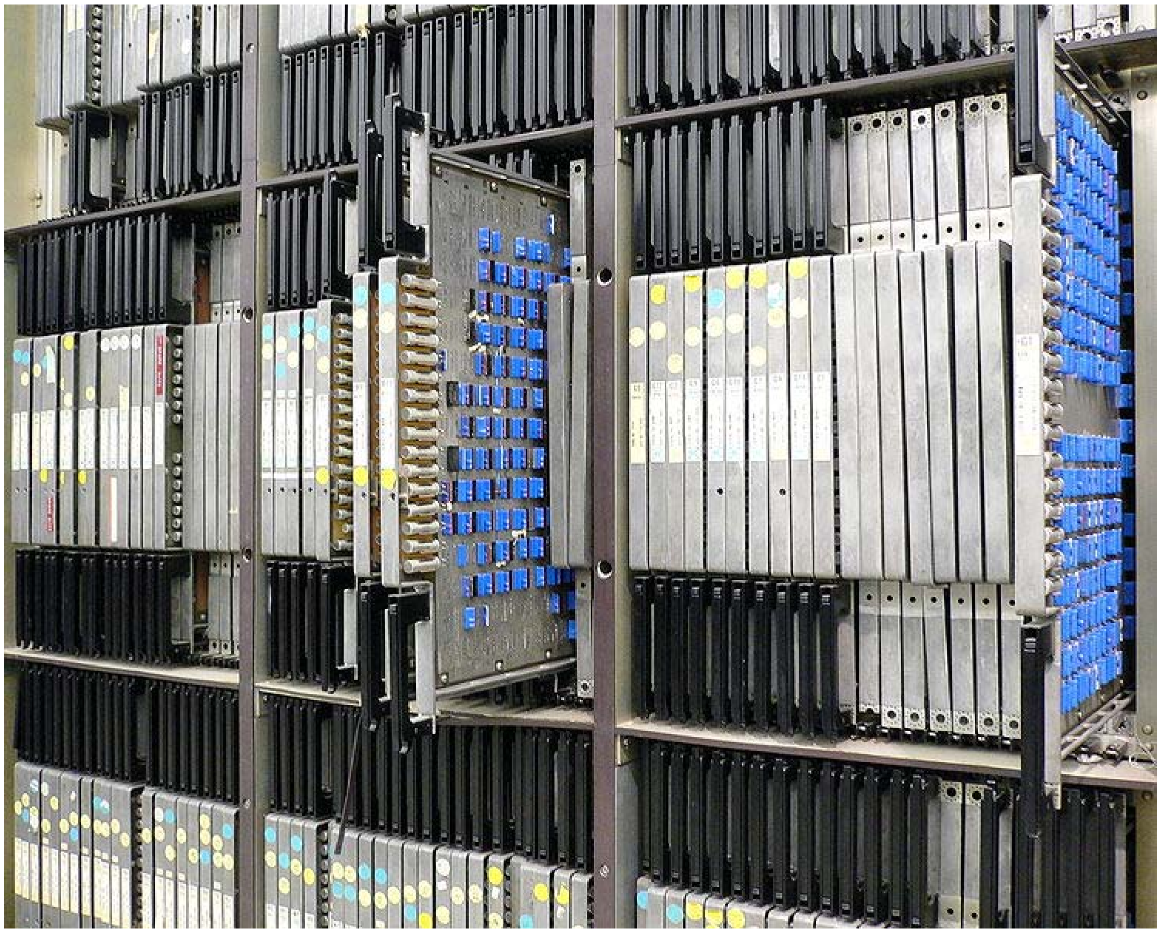
Algorithmic methods

As parallel computers become larger and faster, it becomes feasible to solve problems that previously took too long to run. Parallel computing is used in a wide range of fields, from bioinformatics (protein folding and sequence analysis) to economics (mathematical finance). Common types of problems found in parallel computing applications are:

- Dense linear algebra
- Sparse linear algebra
- Spectral methods (such as Cooley–Tukey fast Fourier transform)
- *n*-body problems (such as Barnes–Hut simulation)
- Structured grid problems (such as Lattice Boltzmann methods)
- Unstructured grid problems (such as found in finite element analysis)
- Monte Carlo simulation
- Combinational logic (such as brute-force cryptographic techniques)

- Graph traversal (such as sorting algorithms)
- Dynamic programming
- Branch and bound methods
- Graphical models (such as detecting hidden Markov models and constructing Bayesian networks)
- Finite-state machine simulation

History



ILLIAC IV, "perhaps the most infamous of Supercomputers"

The origins of true (MIMD) parallelism go back to Federico Luigi, Conte Menabrea and his "Sketch of the Analytic Engine Invented by Charles Babbage". IBM introduced the 704 in 1954, through a project in which Gene Amdahl was one of the principal architects. It became the first commercially available computer to use fully automatic floating point arithmetic commands.

In April 1958, S. Gill (Ferranti) discussed parallel programming and the need for branching and waiting. Also in 1958, IBM researchers John Cocke and Daniel Slotnick discussed the use of parallelism in numerical calculations for the first time. Burroughs Corporation introduced the D825 in 1962, a four-processor computer that accessed up to

16 memory modules through a crossbar switch. In 1967, Amdahl and Slotnick published a debate about the feasibility of parallel processing at American Federation of Information Processing Societies Conference. It was during this debate that Amdahl's Law was coined to define the limit of speed-up due to parallelism.

In 1969, US company Honeywell introduced its first Multics system, a symmetric multiprocessor system capable of running up to eight processors in parallel. C.mmp, a 1970s multi-processor project at Carnegie Mellon University, was "among the first multiprocessors with more than a few processors". "The first bus-connected multi-processor with snooping caches was the Synapse N+1 in 1984."

SIMD parallel computers can be traced back to the 1970s. The motivation behind early SIMD computers was to amortize the gate delay of the processor's control unit over multiple instructions. In 1964, Slotnick had proposed building a massively parallel computer for the Lawrence Livermore National Laboratory. His design was funded by the US Air Force, which was the earliest SIMD parallel-computing effort, ILLIAC IV. The key to its design was a fairly high parallelism, with up to 256 processors, which allowed the machine to work on large datasets in what would later be known as vector processing. However, ILLIAC IV was called "the most infamous of Supercomputers", because the project was only one fourth completed, but took 11 years and cost almost four times the original estimate. When it was finally ready to run its first real application in 1976, it was outperformed by existing commercial supercomputers such as the Cray-1.