Metacomputing/ Grid Computing

CSE 791

Fall, 1999

What is a Metacomputer?

A seamless assembly of computers working together to provide the the illusion of a single virtual system.

- Relieve the programmer of worrying about the irksome details.
- Provide access to remote resources.
- User concentrates on solving problem.

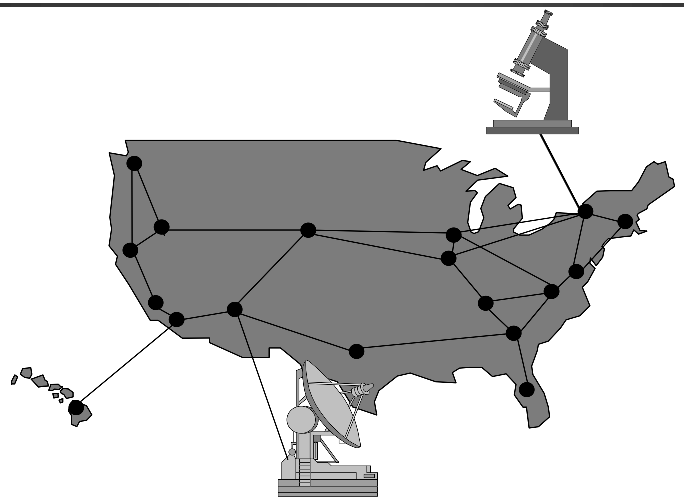
But...

- Isn't that Buyya's Single System Image?
  - Yes, in many ways.
  - But we're quite clear that we're talking about middleware, not OS SSI.

Metacomputing Properties

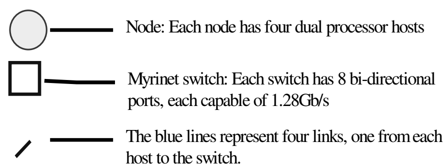
- Probably:
  - heterogeneous
  - distributed over a WAN
  - encompassing resources that you don't own
  - most useful for large applications
- But not necessarily...

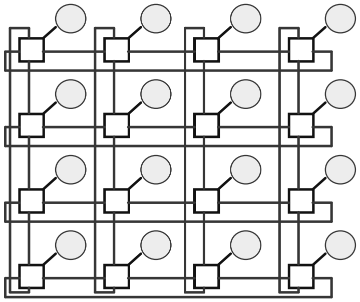
Sample Metacomputer I



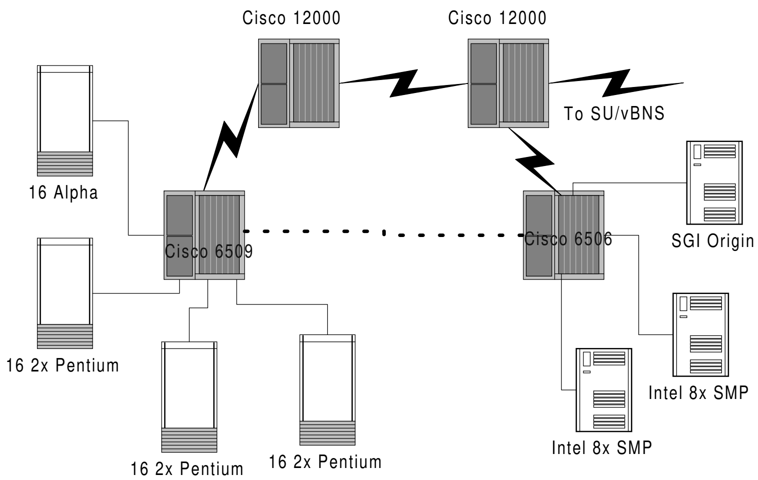
Sample Metacomputer II

- A single cluster
- cf. Orange Grove
  - heterogeneous
  - but we own all of it





Sample Metacomputer III



Gigabit Networks: *The* Key Technology

- Any non “embarrassingly parallel” program will do communication.
- Network bandwidth and latency dictate granularity of programs that can be run effectively (we can do some latency hiding).
- Fast networks allow for the distribution of communicating applications.

Software

Hardware is all well and good, but what do we need from the middleware to make this fly?

Technical Requirements

- Extensible
- Site autonomy
- Secure
- Fault tolerant
- Resource management
- Multi-language
- Scaleable
- Simple to use
- Single namespace
- Provide high performance

•We cannot replace host operating systems
•We cannot legislate changes to the internet
•We cannot require that we run as "root"

Metacomputing Systems and Middleware Layers

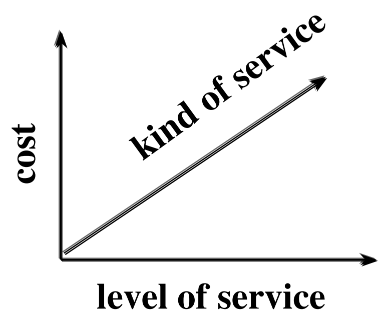
- Legion
- Globus
- Globe
- Harness
- MPI
- PVM
- SNIPE
- WebFlow

This is just a representative sample; there are more...

Metacomputing Case Study: Legion

Legion philosophy

- Provide flexible mechanism
- Allow users/application designers to choose a point in the space



Legion achieves its goals via a flexible object model, user-replaceable "system" components, and a "soft" layering scheme

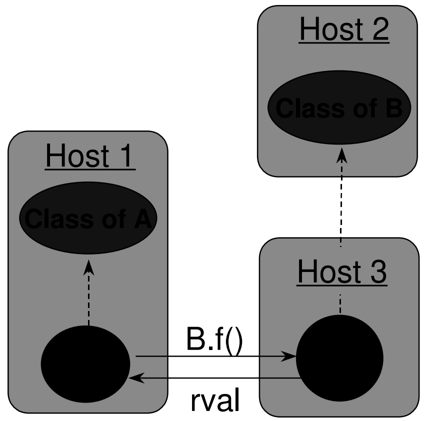
Architecture and Object Model

Legion is object-based

- Everything is an object
- Legion's core "system" objects
  - Hosts
  - Vaults
  - Binding Agents
- All system objects can be replaced
- Classes are objects

The object model

- Legion **objects**
  - belong to classes
  - are logically independent, address-space-disjoint
  - communicate via non-blocking method calls
  - are *Active* or *Inert*
  - export *object-mandatory* member functions
  - are named using LOID's

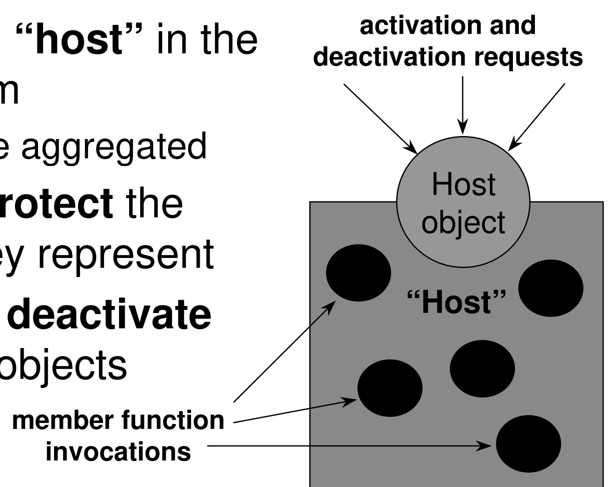


Legion Object Identifiers (LOID)

- LOID's name objects and are location independent strings of bits
- LOIDs have many fields (arbitrary)
- Basic LOID's have
  - type
  - "domain"
  - class id (integer)
  - instance id (integer)
  - public key (RSA)

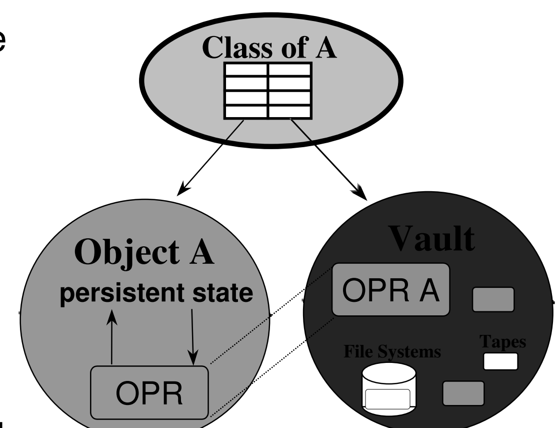
Host objects

- **Run on each "host"** in the Legion system
  - hosts may be aggregated
- **Guard and protect** the resources they represent
- **Activate** and **deactivate** other Legion objects



Vaults

- Objects will migrate
  - State information must persist
- Three objects involved
  - the object writes to its OPR
  - the Vault manages OPR's
  - classes select and know about Vaults

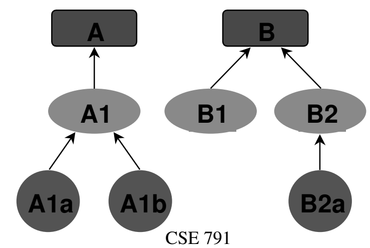


Vaults

- Are the **mechanism** for object persistence
  - “contain” inert Legion objects
  - keep the OPR’s safe while an object is inert
  - known by the class of the object
- Are **policy makers** (selected by the class)
  - how is the OPR maintained?
  - how is it kept “safe”?
  - many different implementations are possible
    - **file system, DBMS, even shared memory**
    - **trust anyone, trust only the class, etc.**

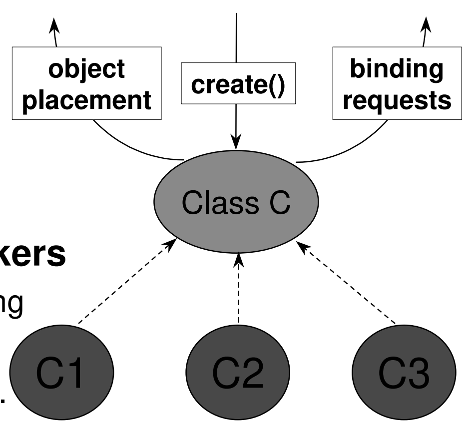
Legion class system

- Classes
  - define the characteristics of their instances
- Meta-classes
  - classes whose instances are classes
  - define the characteristics of classes



Class roles and responsibilities

- Classes are **managers** that provide system-level services
  - creation
  - deletion
  - binding
- Classes are **policy makers**
  - placement and scheduling
  - fault-tolerance
  - instance implementation.



For example...

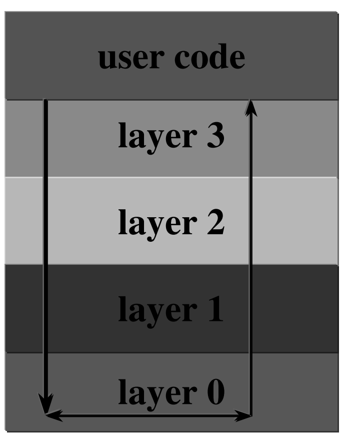
- Classes may desire
  - application-specific scheduling algorithms
  - to be part of a mandatory security protocol for all instances
  - to multiplex multiple instances within a single address space
  - to hand out different aliases for the same instance
  - to define and use special inheritance semantics
  - to manage changing instances

Flexible layers

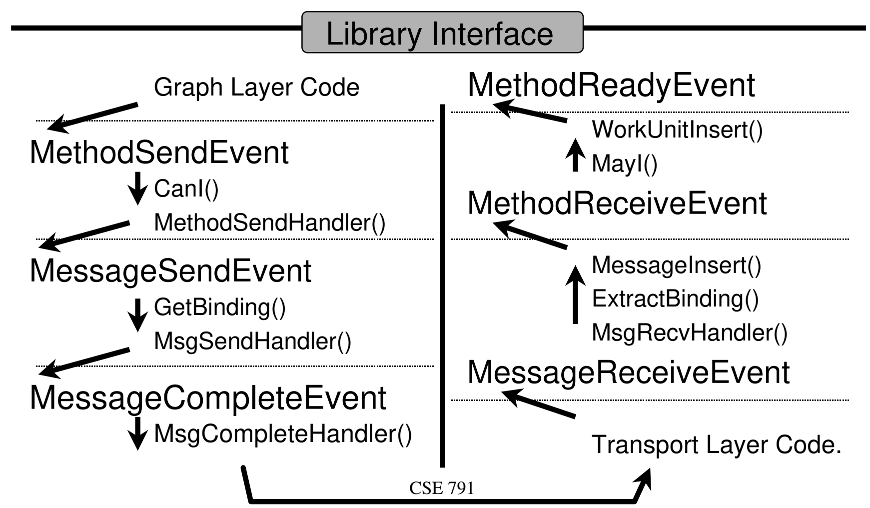
Legion layers are “soft”

- We don’t want to constrain implementations
- We want to be able to support many different policies and mechanisms
  - different communication libraries
  - different security policies
  - different language implementations and styles
  - different fault-tolerance techniques

Traditional layering scheme



Event lists for MPL



Receiving side flexibility - active messages

call the method

MethodReadyEvent

WorkUnitInsert()
MayI()

MethodReceiveEvent

MessageInsert()
ExtractBinding()
MsgRecvHandler()

MessageReceiveEvent

Transport Layer Code.

Receiving side flexibility - message logging

MethodReadyEvent

WorkUnitInsert()
MayI()

MethodReceiveEvent

MessageInsert()

Store_message_in_log_file()

ExtractBinding()
MsgRecvHandler()

MessageReceiveEvent

Transport Layer Code.

Modules

- Graph management
- Data Delivery
- Implicit parameters
- Encryption
- Authentication
- Protocol layer
- MayI - ACL
- Method matching
- Naming and binding
- Exception management

Program Graphs

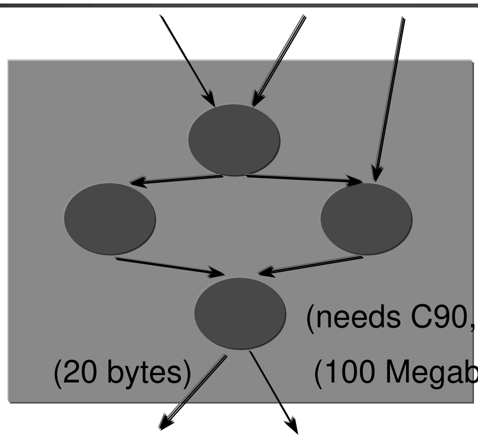
Use graphs as basis for many forms of parallelism

- Support compiler writers and tool-builders by giving them powerful graph manipulation tools
- Tools can be used to generate both compile-time static graphs and at run-time to construct graphs dynamically

Program Graphs

- Nodes are member function calls or sub-graphs
- Arcs model dependencies, one for each parameter or result
- Graph vertices and edges may be annotated, e.g., given identifiers, labeled with computation & communication requirements

Graph example



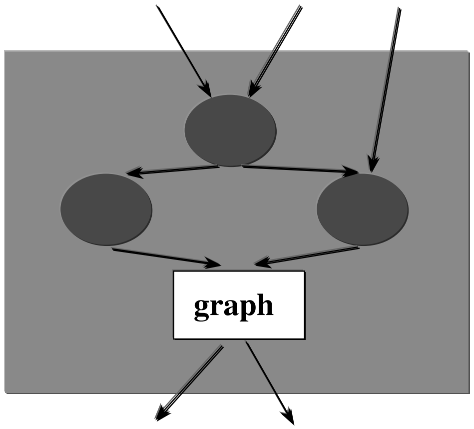
Function of
•three arguments
•two outputs

Annotate the graph with resource information

Program graphs are first class

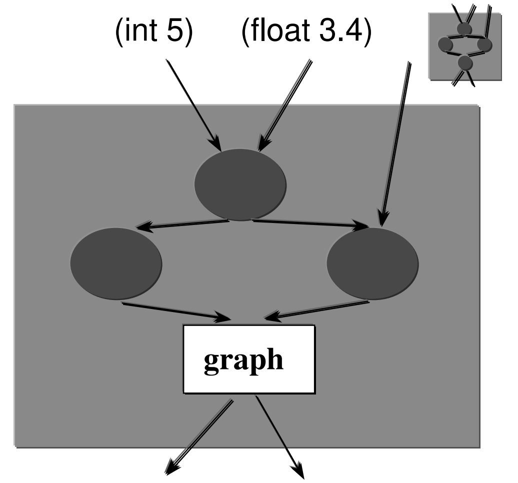
- Graphs are *first-class* and *recursive*
  - they can be copied and used as parameters and return values
  - they can be written and read
  - graphs can contain other other graphs
- Graph composition == function composition

A contained graph



Function of
•three arguments
•two outputs

A graph as a parameter



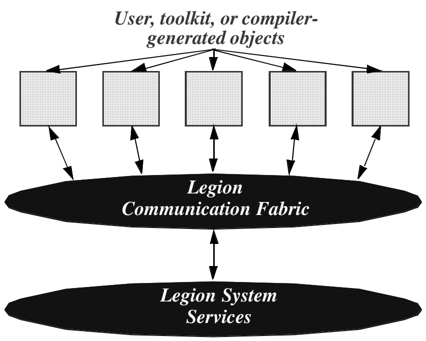
Parallel Computing

Parallel Computing in Legion comes in many forms

- Support for parallel libraries
- Parallel method invocation in MPL, BFS, Java
- Expose run-time system to compiler writers and toolkit builders
  - low level message system
  - program graphs

Support Popular Libraries

- PVM and MPI support
- User codes run “on top of” Legion
- Retarget toolkits that use PVM or MPI
- Retarget compilers



MPL - An OO Parallel Language

- Philosophy
  - Programmers can best make decomposition decisions
  - Compilers can best manage dependencies, communication, synchronization
  - Exploit the comparative advantages of each

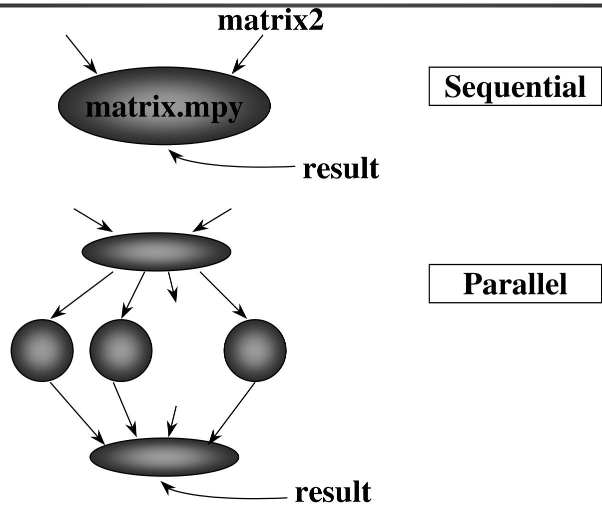
MPL Features

- Extended C++ - five new keywords
- Medium grain (user specified granularity)
- Data and task parallelism
- Exploits both intra-object and inter-object parallelism encapsulation
- Compiler and run-time system manage scheduling, communication and synchronization

MPL syntax

- Mentat classes
  - persistent - have state
  - stateless - pure functions
- Select/accept
- Invocation syntax the same

Intra-object encapsulation

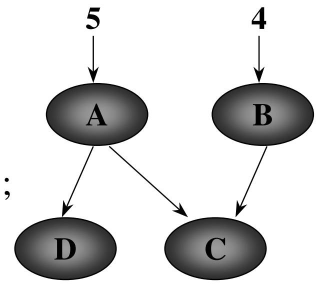


Inter-object parallelism

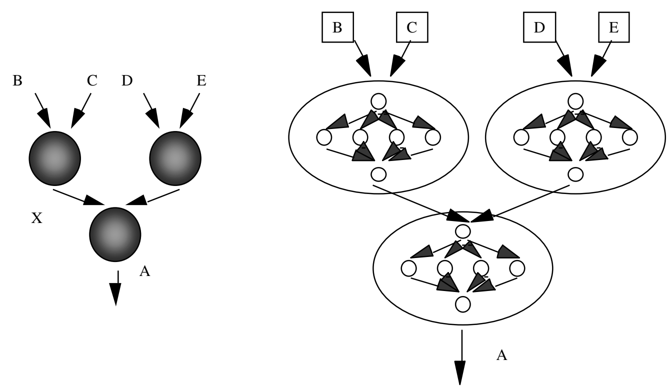
non blocking

```
i = A.op1(5);
j = B.op1(4);

k = C.op2(i,j);
l = D.op1(i);
```

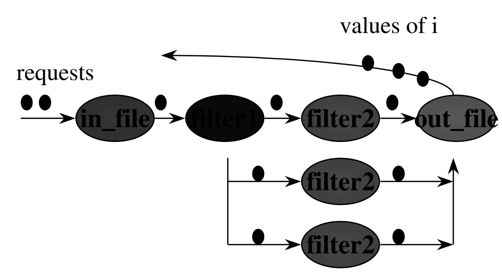


Nested parallelism



Pipeline example

```
m_file  in_file("file1"),
        out_file("file2");
data_processor dp;
block *res;
for (i=0; i <
   MAX_BLOCKS, i++) {
   res = in_file.read_block(i);
   res = dp.filter1(res);
   res = dp.filter2(res);
   out_file.write_block(i,res);
}
```



Basic Fortran Support - BFS

- Same basic idea as MPL but with Fortran-like syntax
- Define Legion classes - persistent and stateless
- Object wrappers automatically generated and linked with Fortran .o files
- Calls made via pseudo-comments