An Introduction to High-Throughput Computing

Filip Křikava
I3S Laboratory, CNRS, France

The Goal

- Get both the **theoretical** and **practical** knowledge in operating Condor
- Understand what is **High-Throughput Computing** (HTC) and how does it differs from **High-Performance Computing** (HPC)
- Have some fun!

Outline

- An Introduction to High-Throughput Computing
- An Introduction to Condor

- Practical Condor hands-on session in the lab

An Introduction to High-Throughput Computing

A Warm Up

Concurrency

- Doing many things at once
- Typically on one host

- Multiple processes
- Threads
- Cooperative multitasking
- Coroutines
- Asynchronous programming

Parallelism

- Doing many things simultaneously

- Multiple processes
- Threads
- Distribute systems

Distributed Computing

- Doing many things across multiple machines, simultaneously
- Many cores on many machines

- Many ways!

Concurrency vs Parallelism vs Distributed Computing

Concurrency

- Doing **many** things at **once**
- Typically on one host

- Multiple processes
- Threads
- Cooperative multitasking
- Coroutines
- Asynchronous programming

Parallelism

- Doing **many** things **simultaneously**

- Multiple processes
- Threads
- Distribute systems

Distributed Computing

- Doing **many** things across **multiple** machines, **simultaneously**
- Many cores on many machines

- Many ways!

Why do we need this?

- The problem is:
  - How to maximize performance
    - throughput
    - utilization
    - response time
  - of a given system

- How to maximize the value of an investment in hardware for a given workload?

- Those were the problem of the 60's

Distributed Processing System

P. H. Enslow, "*What is a Distributed Data Processing System?*, Computer, January **1978**

What is a "Distributed" Data Processing System?

Philip H. Enslow, Jr.
Georgia Institute of Technology

Introduction

Words have only one purpose in a technical context—the transmission of information. When they fail to do that, they lead to confusion and misunderstanding. "Distributed data processing" and "distributed processing" are two phrases which illustrate that axiom. Like many other words in the lexicon of the computer professional, these have become cliches through over-use, losing much of their original meaning in the process. This paper is an attempt to reverse that trend.

A great many claims of improved performance (the table below presents only a partial list) are being made for "distributed data processing" systems by vendors as well as authors.[1-4] Hardly anyone reasonably knowledgeable about the current state of the art in multiple-processor data processing systems would claim that major advances toward any significant number of these goals are possible with present technology. To fulfill such a claim, distributed data processing must be something *new* and must represent a large research area. However, what is usually presented today as distributed data processing is just a warmed-over version of an old concept with a new sales word attached.

This paper discusses some of the essential characteristics for a new class of systems, still in the research stage, that will provide some of the benefits given in the table. We do not deprecate present work in improving systems for increased performance or reliability, even if it has been labeled "distributed data processing." Rather, we hope to introduce some precision of terminology and evaluation for this new area.

Claims made for "distributed" data processing systems.

HIGH SYSTEM PERFORMANCE, FAST RESPONSE, HIGH THROUGHPUT
HIGH AVAILABILITY
HIGH RELIABILITY
REDUCED NETWORK COSTS
GRACEFUL DEGRADATION (FAIL-SOFT CAPABILITY)
EASE OF MODULAR, INCREMENTAL GROWTH AND CONFIGURATION FLEXIBILITY
RESOURCE SHARING
AUTOMATIC LOAD SHARING
HIGH ADAPTABILITY TO CHANGES IN WORK LOAD
INCREMENTAL REPLACEMENT AND/OR UPGRADING OF COMPONENTS (BOTH HARDWARE AND SOFTWARE)
EASY EXPANSION IN BOTH CAPACITY AND FUNCTION
EASY ADAPTATION TO NEW FUNCTIONS
GOOD RESPONSE TO TEMPORARY OVERLOADS

What is distributed?

At least four physical components of a system might be distributed: hardware or processing logic, data, the processing itself, and the control (such as the operating system). Some speak of a system that has any *one* of these components distributed as being a "distributed data processing system."

However, a definition that is based solely on the physical distribution of some components of the system is doomed to failure. A proper definition must also cover the concepts under which the distributed components interact. Trivial examples abound of system organizations that exhibit physical distribution but are not considered distributed data processing systems—for example, physical distribution of input/output processing hardware and function. Similarly, there is certainly no distribution of processing functions if there is no distribution of processing hardware, and, conversely, a system that distributes hardware without distributing processing is difficult to imagine.

- High Availability and Reliability
- High System Performance
- Ease of Modular and Incremental Growth
- Autonomic Load and Resource Sharing
- Good Response to Temporary Overloads
- Easy Expansion in Capacity and/or Function

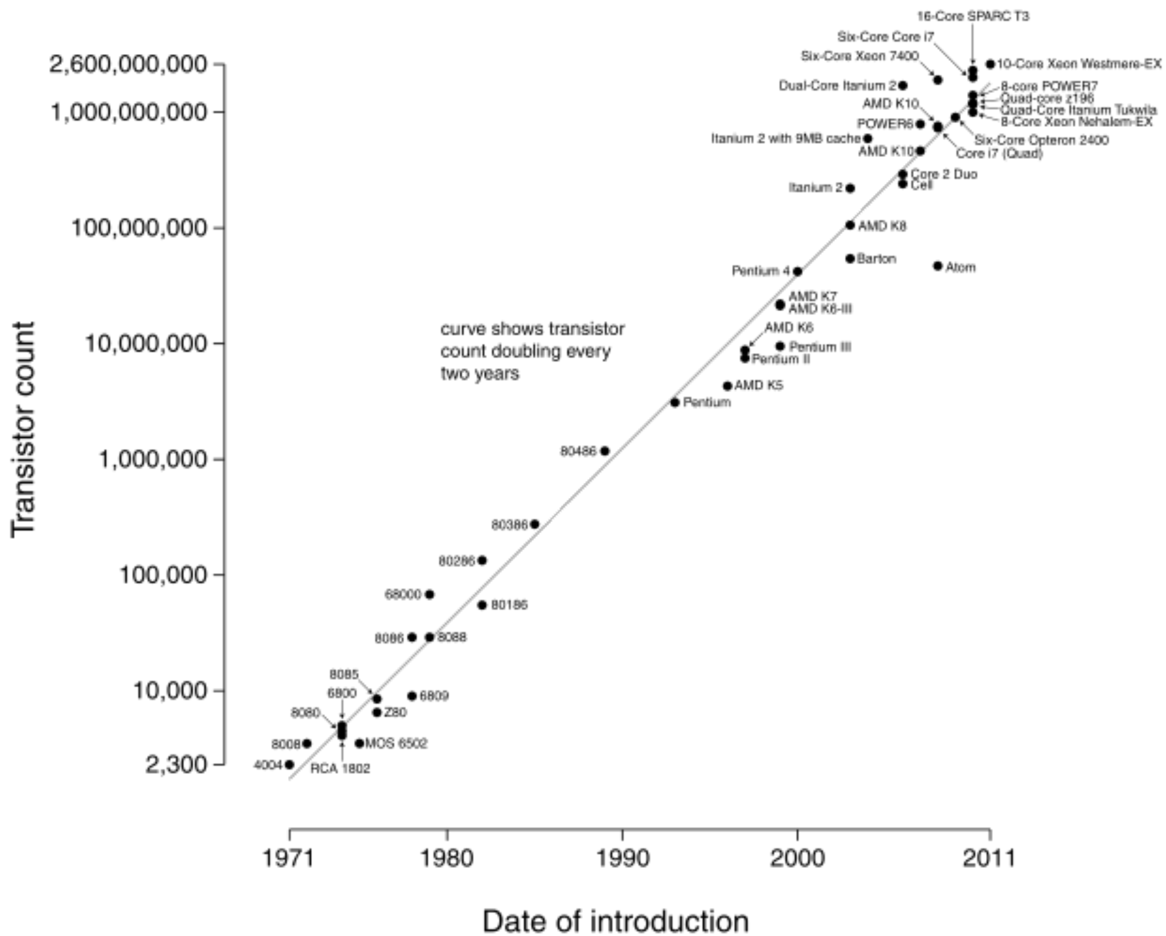
Definitional Criteria for a Distributed Processing System

- From P. H, Enslow and T. G. Saponas "*Distributed and Decentralized Control in Fully Distributed Processing Systems*", Technical Report, **1981**

  - Multiplicity of Resources
  - Component Interconnection
  - Unity of Control
  - System Transparency
  - Component Autonomy

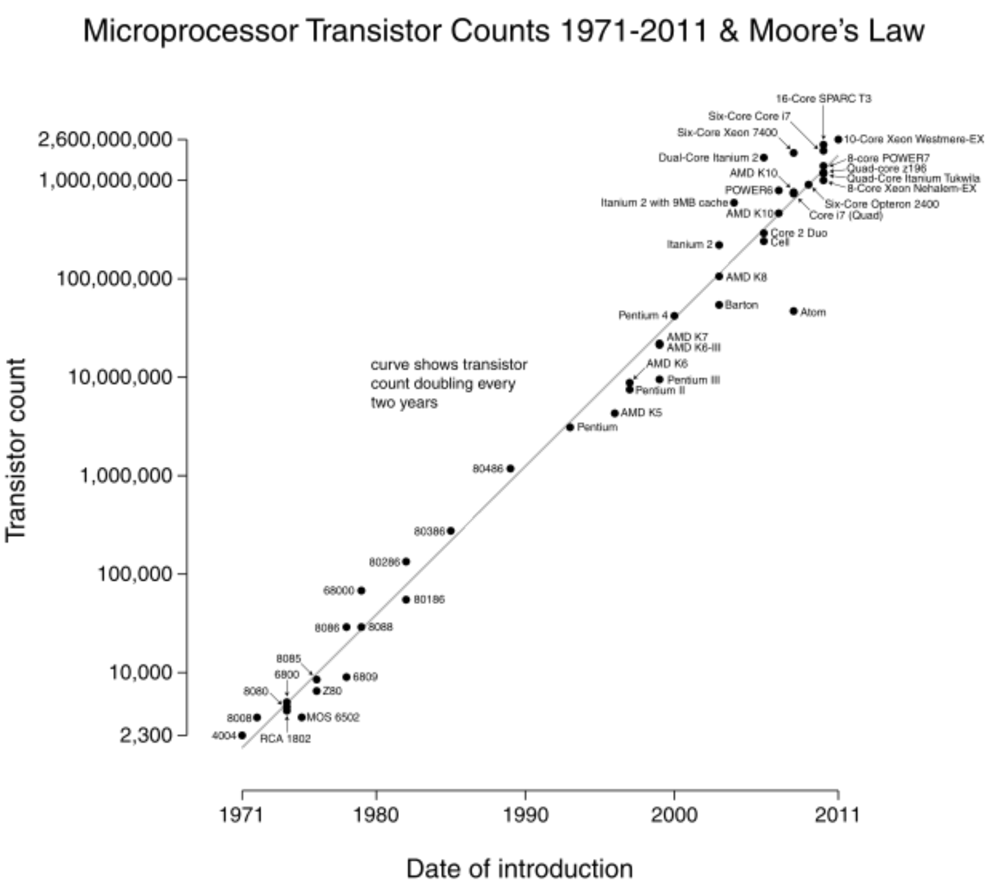
Moore's Law

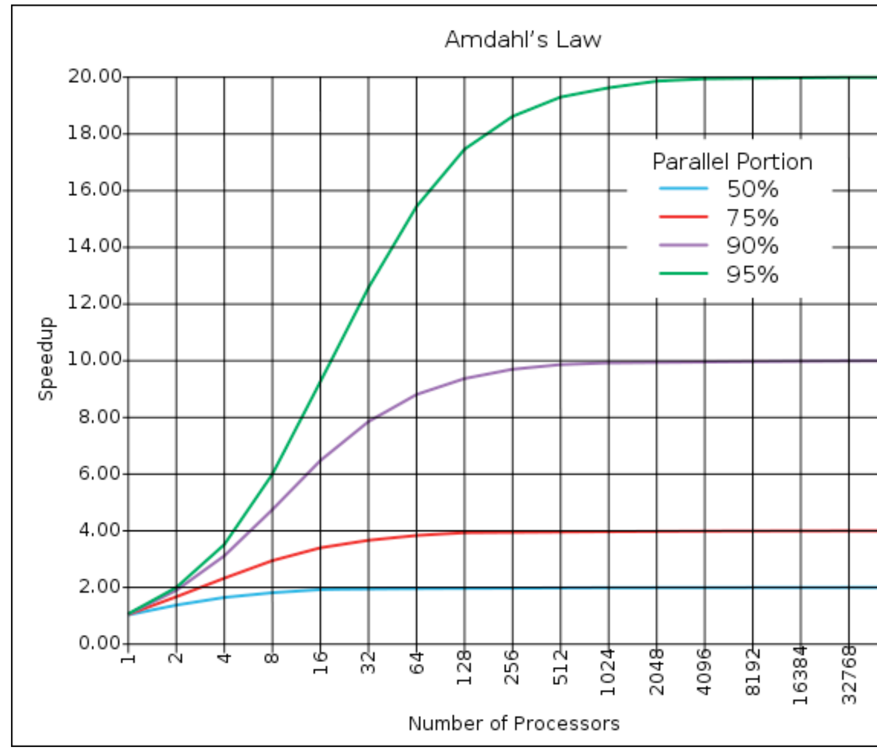
Microprocessor Transistor Counts 1971-2011 & Moore's Law



- "For the past 30 years, computer performance have been driven by Moore's Law

Amdahl's law





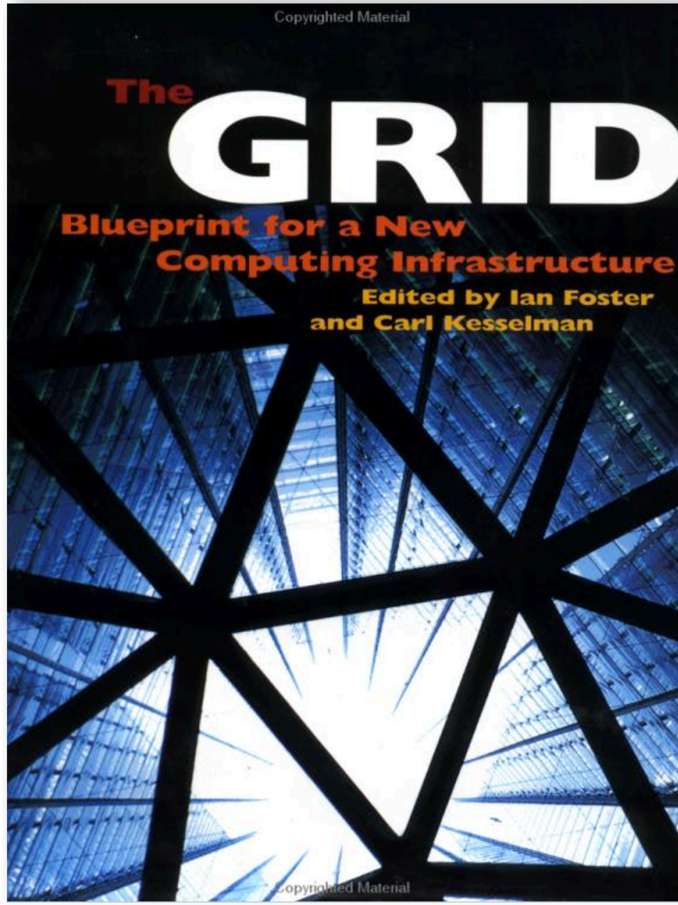
$$\frac{1}{(1-P)+\frac{P}{S}}$$

- "For the past 30 years, computer performance have been driven by Moore's Law **now it will be driven by Amdahl's law.**"
  - Doron Rajwan, Intel Corp.

The GRID

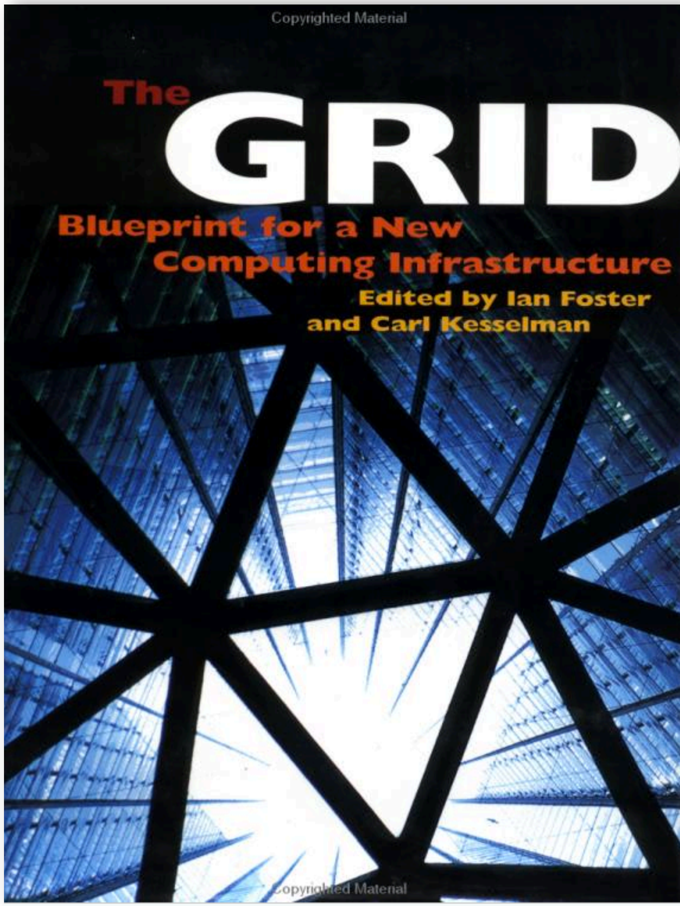
What it is to **you**?

The Grid

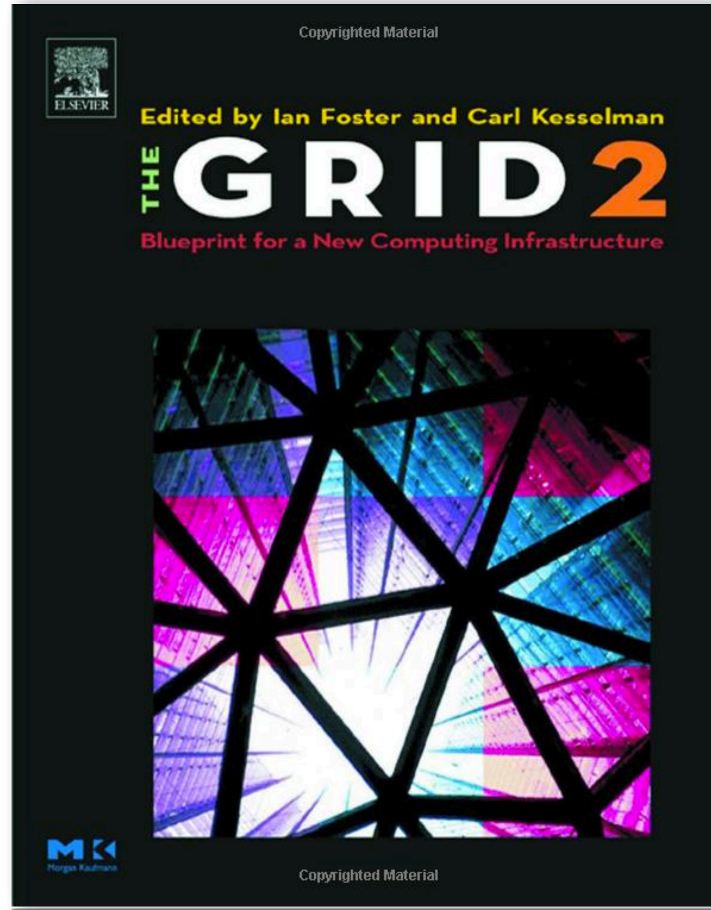


July 1998, 701 pages

The Grid



July 1998, 701 pages



December 2003, 748 pages

The Anatomy of the Grid - Enabling Scalable Virtual Organizations

- *The Anatomy of the Grid - Enabling Scalable Virtual Organizations*, Ian Foster, Carl Kesselman and Steven Tuecke 2001
- "We have provided in this article a concise statement of the *Grid problem*, which we define as controlled resource sharing and coordinated resource use in dynamic, scalable virtual organizations. We have also presented both requirements and a framework for a Grid architecture, identifying the principal functions required to enable sharing within VOs and defining key relationships among these different functions."

The Grid

- Promised to *fundamentally* change the way we thing about and use computing
- Connecting multiple regional, national and international computational grids
- Universal source of
  - pervasive computing
  - dependable computing
- Clear vision of
  - what
  - why
  - who
  - how

Grid Computing

- Partnership between clients and servers
- Clients must have more responsibilities
  - powerful mechanisms for dealing with recovering from failures
    - remote execution
    - work management
    - data output
  - clients have to be *smart*!
- Servers provide careful protocols

Douglas Thain and Miron Livny, "Building Reliable Clients and Servers", The Grid, 2nd ed, 2003

The Grid

- From an interview with *Vittorio Severino*, CIO of Hartford Life
  - The Hartford Financial Services Group, Inc. (NYSE: HIG)
    - fortune 100 Company,
    - one of America's largest investment and insurance company

- **Q**: "What do you expect to gain from *grid computing*? What are your main goals?"
- **Severino**: "Well number one was scalability.

  ...

  Second, we obviously wanted *scalability* with *stability*. As we brought more servers and desktops onto the grid we didn't make it any *less* stable by having a *bigger environment*."

Challenges

- Race Conditions
- Name spaces
- Distributed Ownership
- Heterogeneity
- Object Addressing
- Data Caching
- Object Identity
- Trouble Shooting
- ... any many others

High Throughput Computing

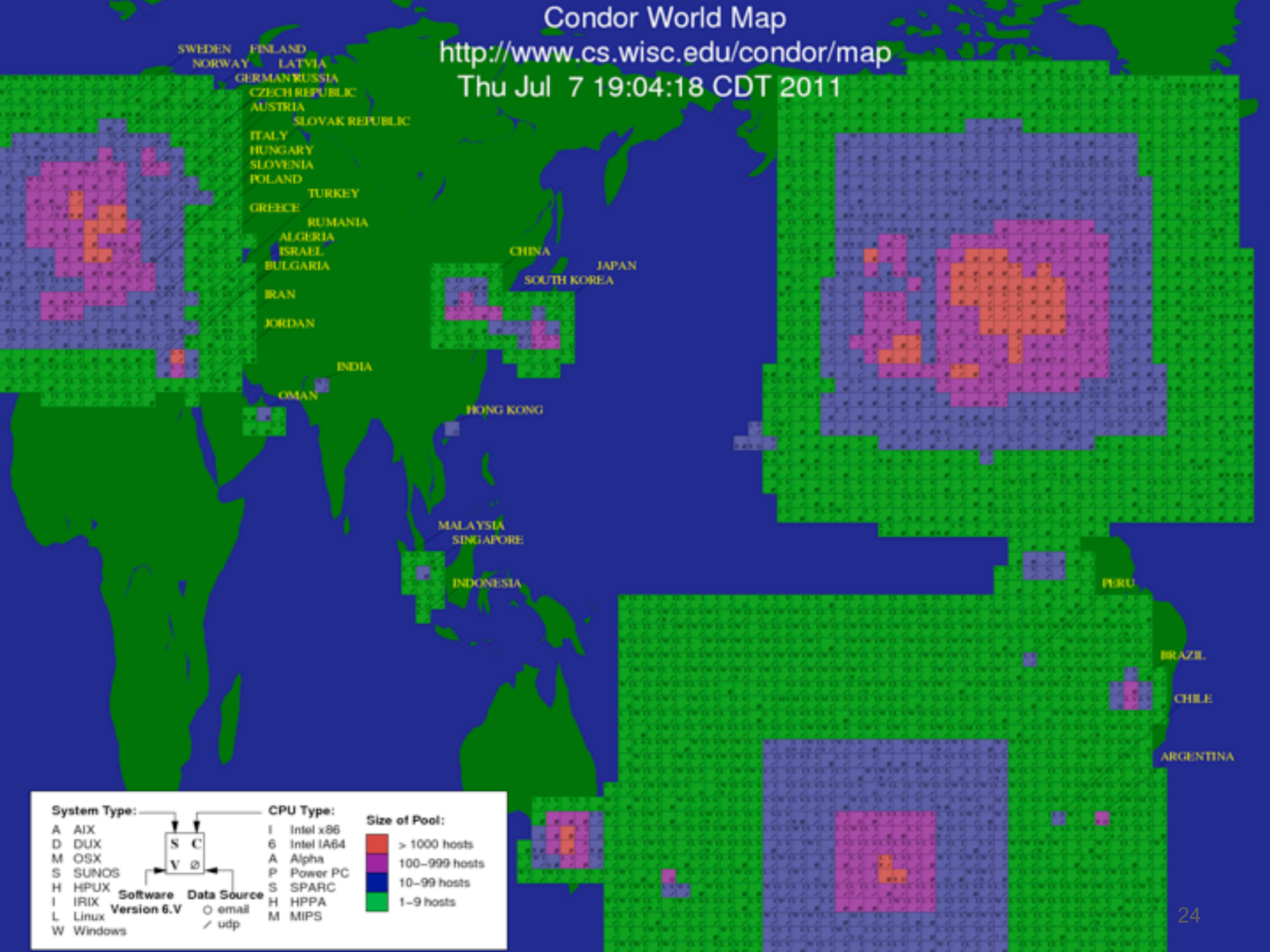
- First introduced in 1996 at a seminar at the NASA Goddard Flight Center, 1997 appeared in HPCWire
- Scientific progress and quality of research are strongly linked to computing *throughput*
  - Less concern of instantaneous computing power
  - More concern of the amount of computing they can use over longer period of time
- HPC is all about FLOPS instantaneously
- HTC is about delivering high perfomanceover a long period of time
- **HTC != 60*60*24*7*52*FLOPS**

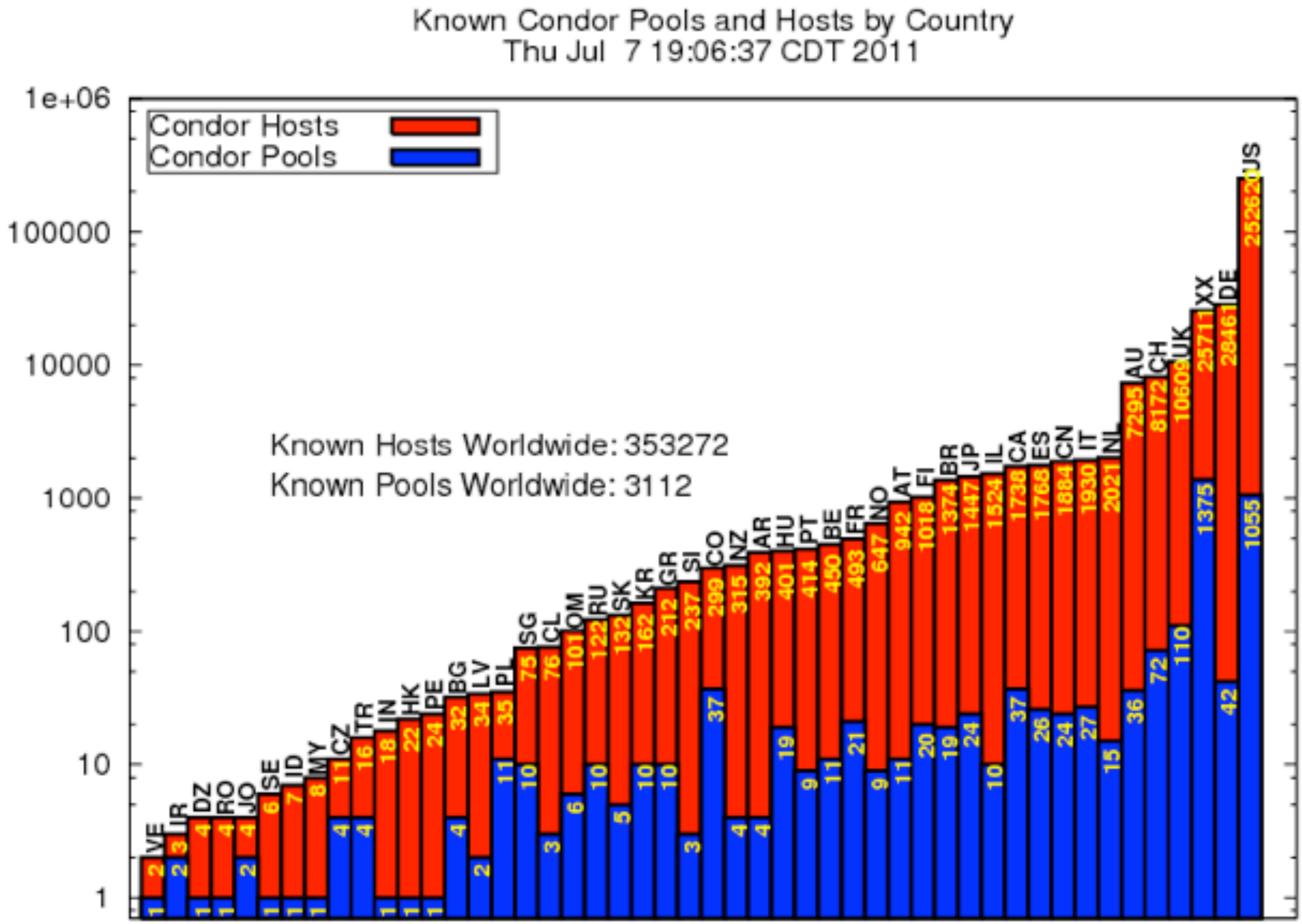
Grids and Clouds

- Grid focus has been remote job delegation
- Cloud focus is about resource allocation
- If you want to do real work in distributed computing you need both!

An Introduction to Condor

- Distributed computing research project in Computer Sciences, est. 1985
- Essentially: a workload management system for compute-intensive jobs
- Researched, Developed and Maintained by the Condor Team at the University of Wisconsin-Madison in US together with RedHat and others
- Large open source code base C/C++ ~680,000 LOC
- An option to use as a scheduler on Amazon EC2 (CycleCloud)
- Widely used in both Academia and Industry
- Condor is a hunter of idle workstations
- http://www.cs.wisc.edu/condor/


Condor World Map
http://www.cs.wisc.edu/condor/map
Thu Jul 7 19:04:18 CDT 2011
SWEDEN
FINLAND
NORWAY
LATVIA
GERMANY
RUSSIA
CZECH REPUBLIC
AUSTRIA
SLOVAK REPUBLIC
ITALY
HUNGARY
SLOVENIA
POLAND
TURKEY
GREECE
RUMANIA
ALGERIA
ISRAEL
BULGARIA
IRAN
JORDAN
INDIA
OMAN
CHINA
JAPAN
SOUTH KOREA
HONG KONG
MALAYSIA
SINGAPORE
INDONESIA
PERU
BRAZIL
CHILE
ARGENTINA
System Type:
A AIX
D DUX
M OSX
S SUNOS
H HPUX
I IRIX
L Linux
W Windows
S C
V ∅
Software Version 6.V
Data Source
○ email
⁄ udp
CPU Type:
I Intel x86
6 Intel IA64
A Alpha
P Power PC
S SPARC
H HPPA
M MIPS
Size of Pool:
> 1000 hosts
100–999 hosts
10–99 hosts
1–9 hosts



http://www.cs.wisc.edu/condor/map

Condor Topics

- Matchmaking
- Running a job
- Workflows
- MPI on Condor wings

Matchmaking

- Matchmaking is a fundamental to Condor
- It is a two process
  - Job describes what it needs
    - “I need Linux with 2GB of RAM”
  - Machine describes what it requires
    - “I’m Linux and I will only run jobs from Physics department”
- It allows preferences
  - “I need Linux with more memory, but any machine you provide will do”

ClassAds

- Is a way to express these preferences together with facts
  - The executable is "*myapplication*"
  - Available memory should "*> 2GB*"
- Almost schema-free
- User-extensible
- Name-value pairs with expression support

Example

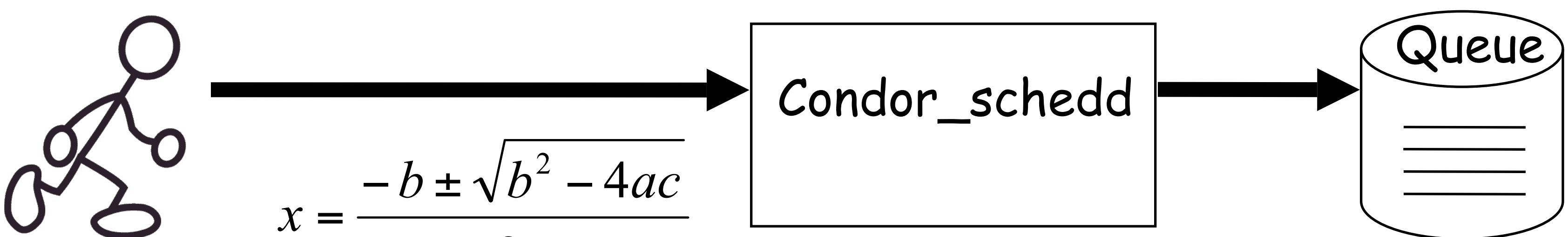
```
MyType       = "Job"          String
TargetType   = "Machine"
ClusterId    = 1377           Number
Owner        = "roy"
Cmd          = "analysis.exe"
Requirements =
    (Arch == "INTEL")         Boolean
&& (OpSys == "LINUX")
&& (Disk >= DiskUsage)
&& ((Memory * 1024)>=ImageSize)
…
```

Schema-free ClassAds

- There is a minimal schema imposed by Condor
  - `Owner` is a string
  - `JobId` is a number
- User can easily extend it, however they like, for both jobs and machines
  - `AnalysisJobType = “simulation”`
  - `HasJava_1_4 = TRUE`
  - `ShoeLength = 7`
- Matchmaking the uses these attributes
  - `Requirements = OpSys == "LINUX"`
    `&& HasJava_1_4 == TRUE`

Submitting Jobs

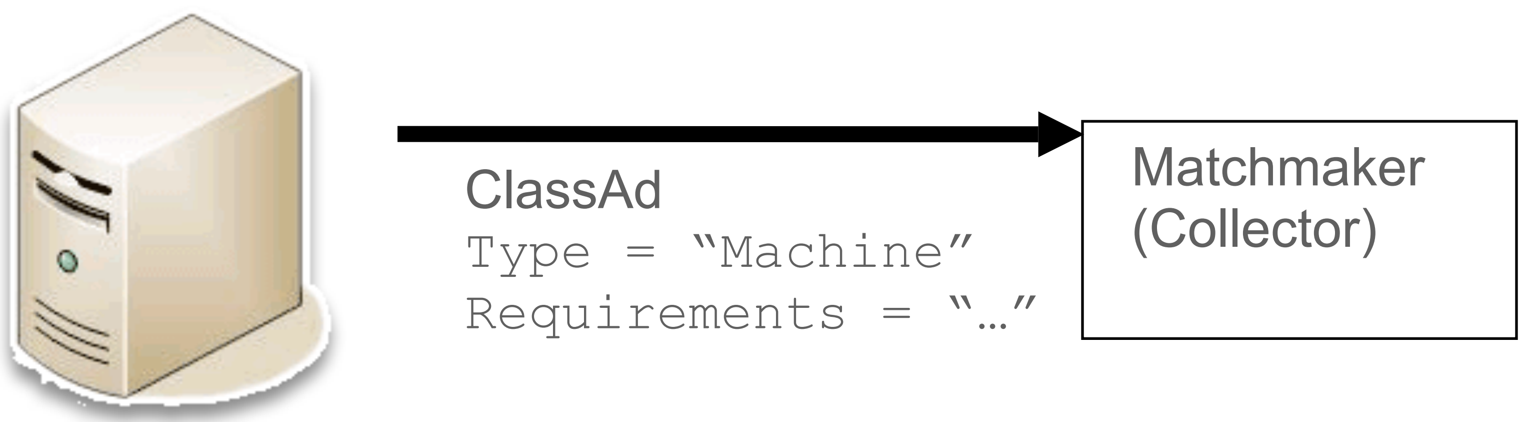
- Users submit jobs from a computer
  - Jobs described as ClassAds
  - Each submission computer has a queue
  - Queues are not centralized
  - Submission computer watches over queue
  - Can have multiple submission computers
  - Submission handled by condor_schedd



$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Advertising Computers

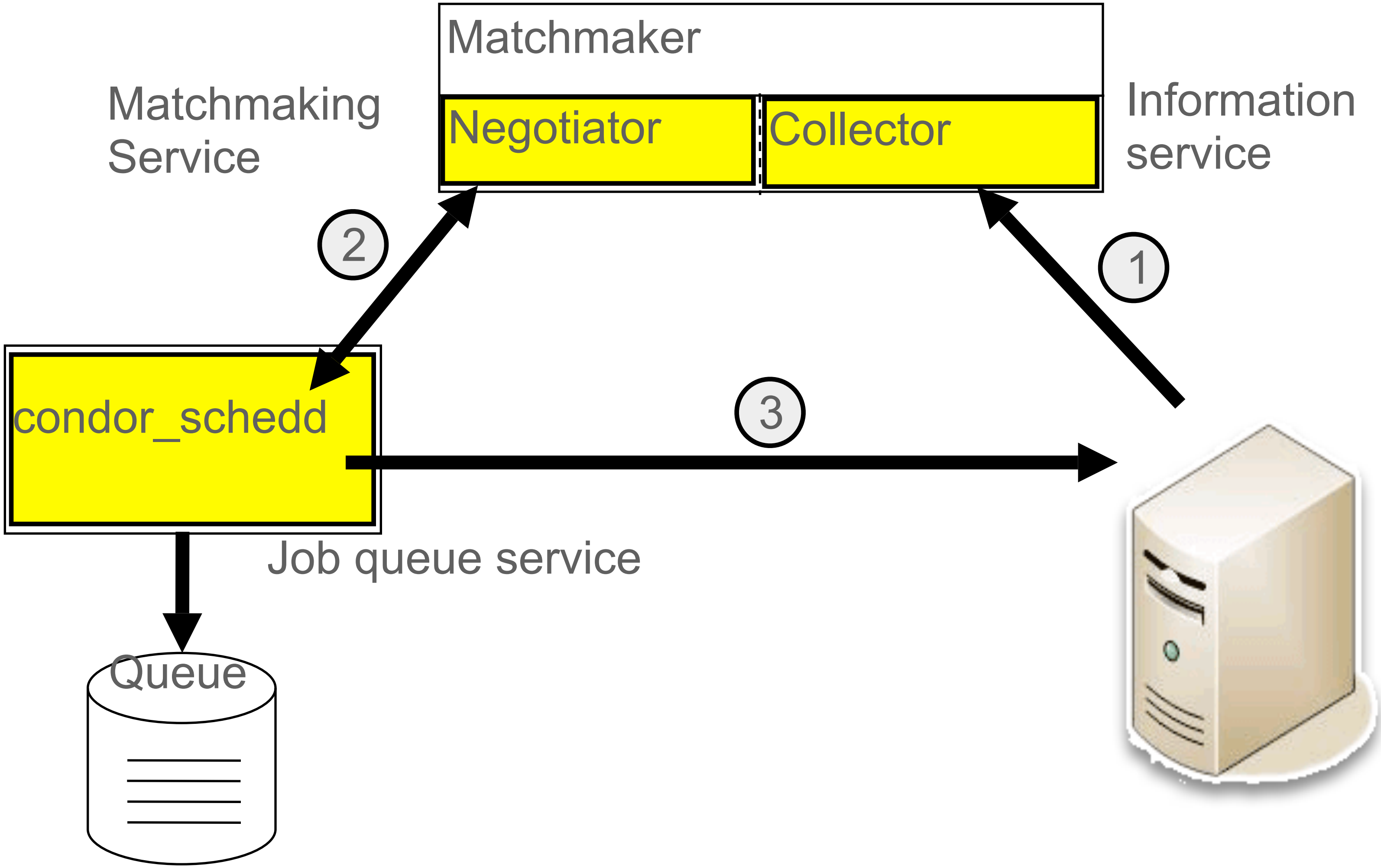
- Machine owners describe computers
  - Configuration file extends ClassAd
  - ClassAd has dynamic features
    - Load Average
    - Free Memory
    - …
  - ClassAds are sent to Matchmaker



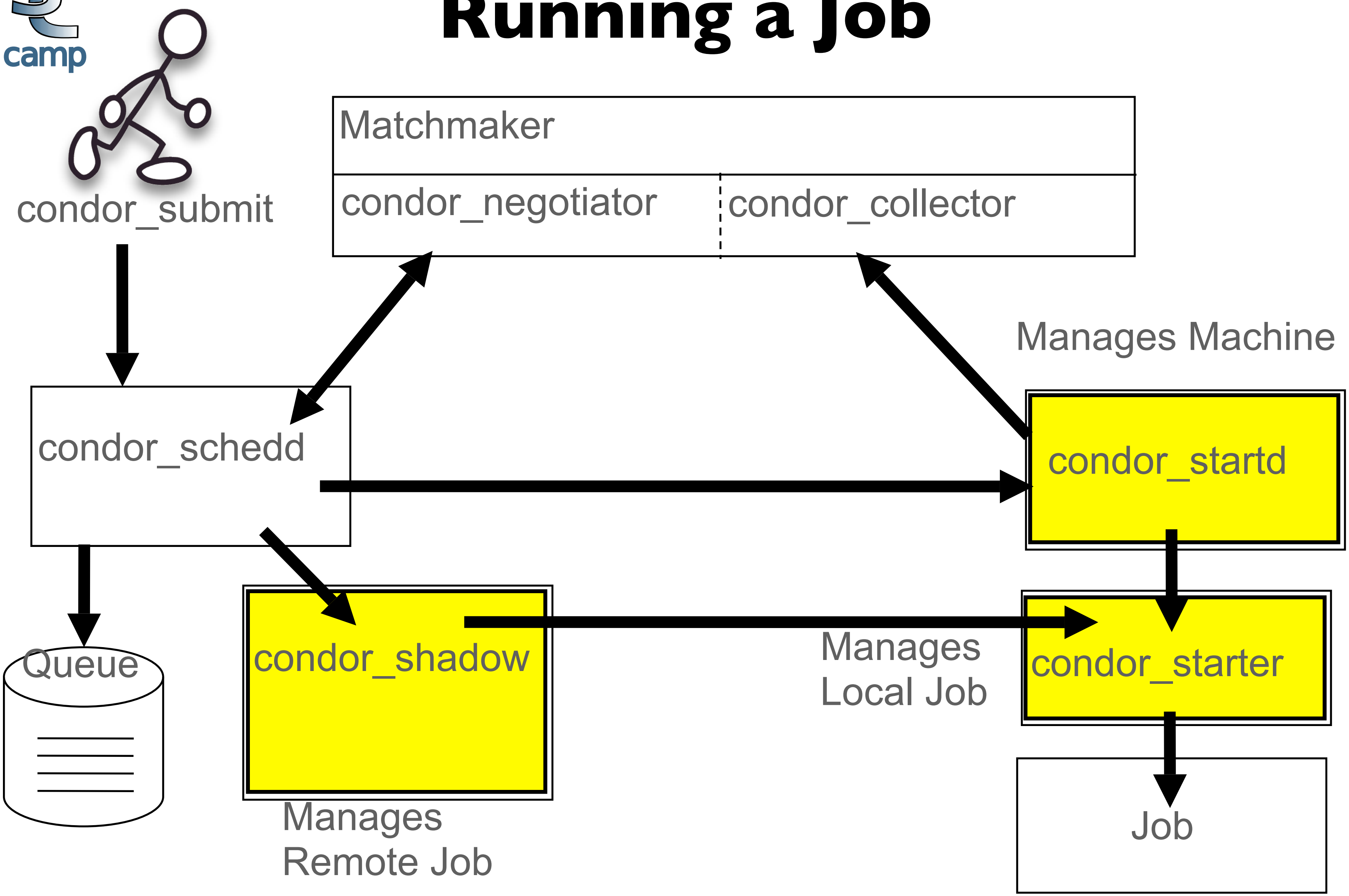
Matchmaking

- Negotiator collects list of computers
- Negotiator contacts each schedd
  - What jobs do you have to run?
- Negotiator compares each job to each computer
  - Evaluate requirements of job & machine
  - Evaluate in context of both ClassAds
  - If both evaluate to true, there is a match
- Upon match, schedd contacts execution computer

Matchmaking Diagram



Running a Job



Condor Daemons

- **Master** - Takes care of other processes
- **Collector** - Stores ClassAds
- **Negotiator** - Performs matchmaking
- **Schedd** - Manages job queue
- **Shadow** - Manages job (submit side)
- **Startd** - Manages computer
- **Starter** - Manages job (execution side)

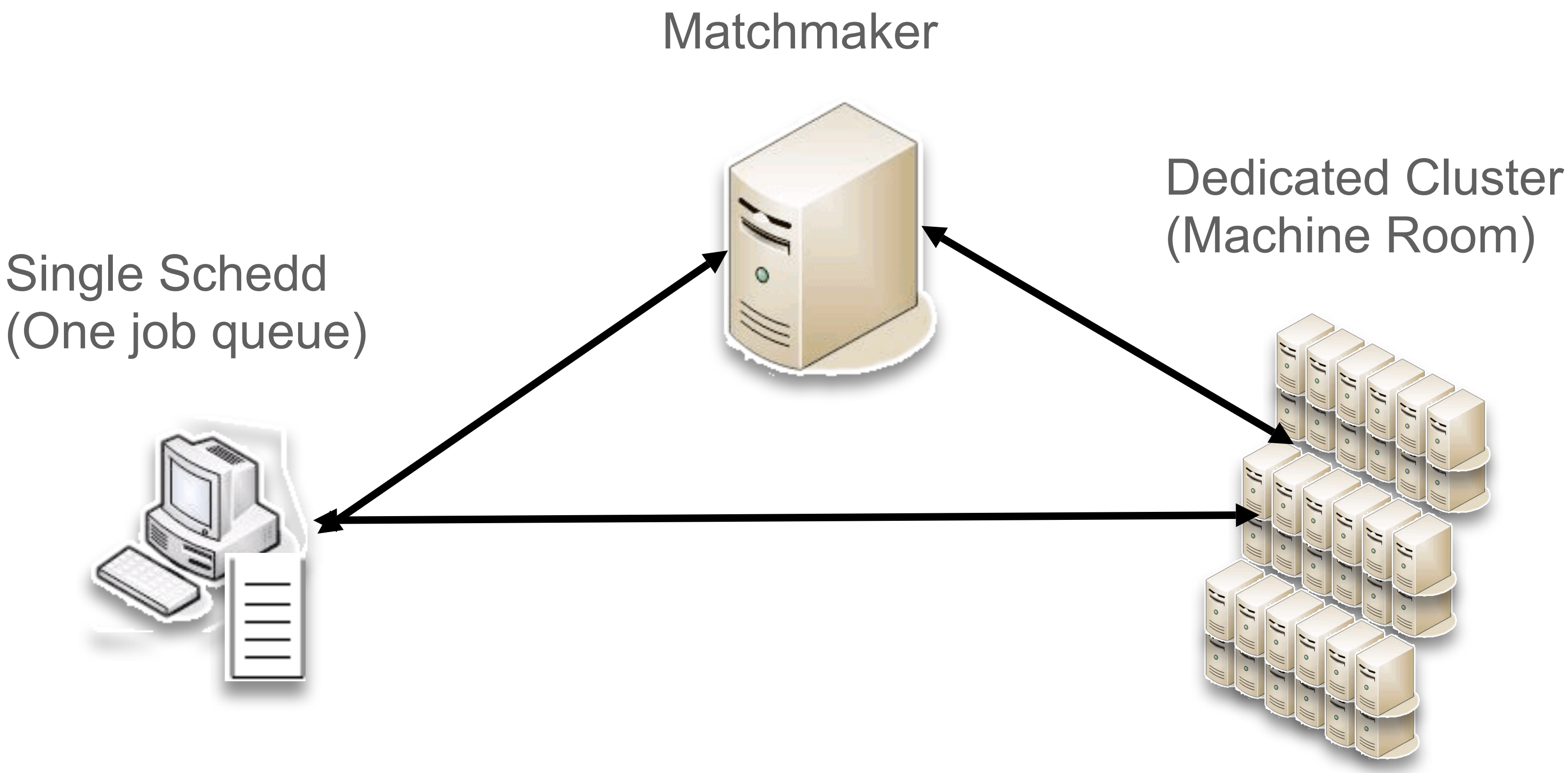
Condor Daemons

- **Master** - Takes care of other processes
- **Collector** - Stores ClassAds
- **Negotiator** - Performs matchmaking
- **Schedd** - Manages job queue
- **Shadow** - Manages job (submit side)
- **Startd** - Manages computer
- **Starter** - Manages job (execution side)

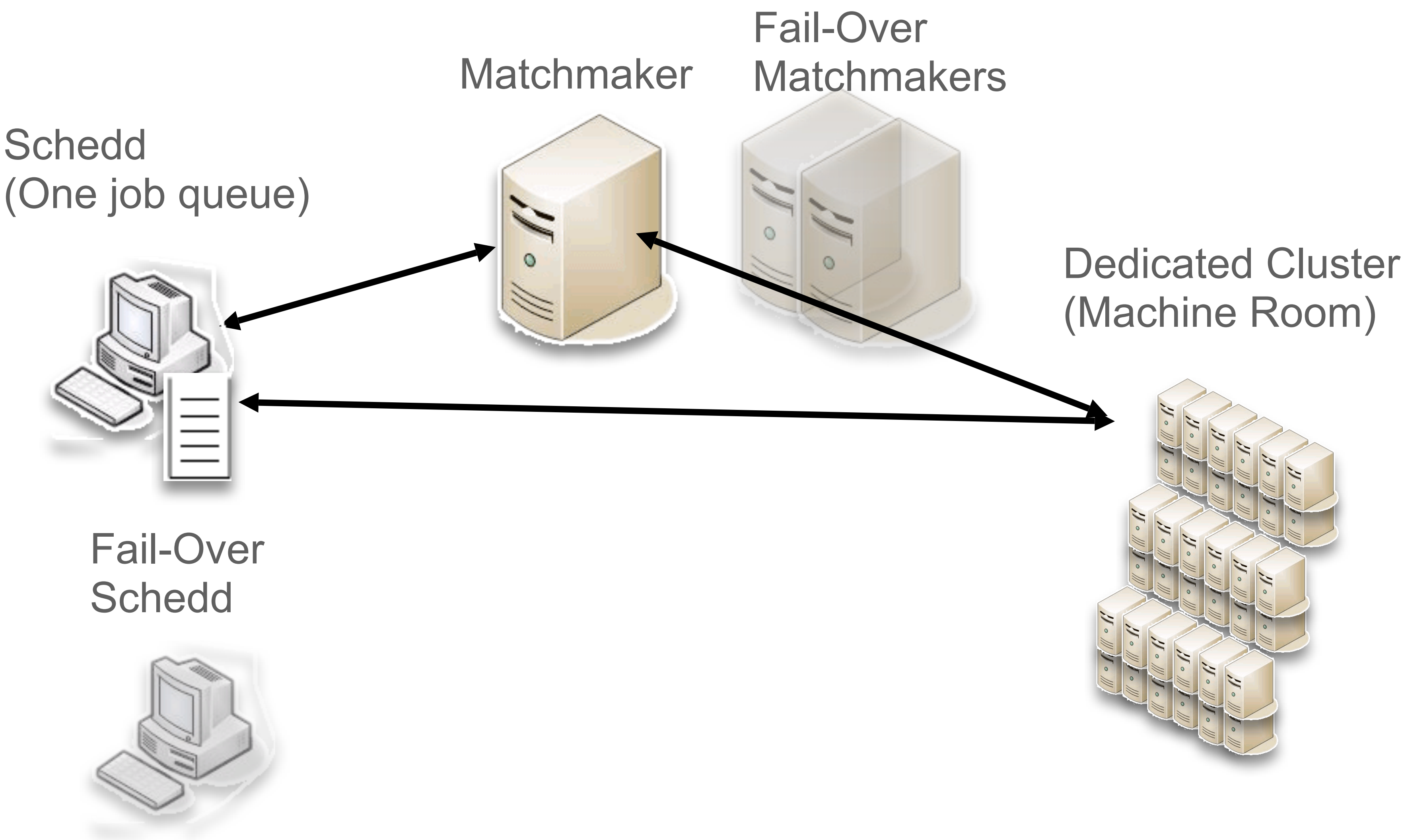
Some Notes

- One negotiator/collector per pool
- Can have many schedds (submitters)
- Can have many startds (computers)
- A machine can have any combination of:
  - Just a startd (typical for a dedicated cluster)
  - schedd + startd (perhaps a desktop)
  - Personal Condor: everything

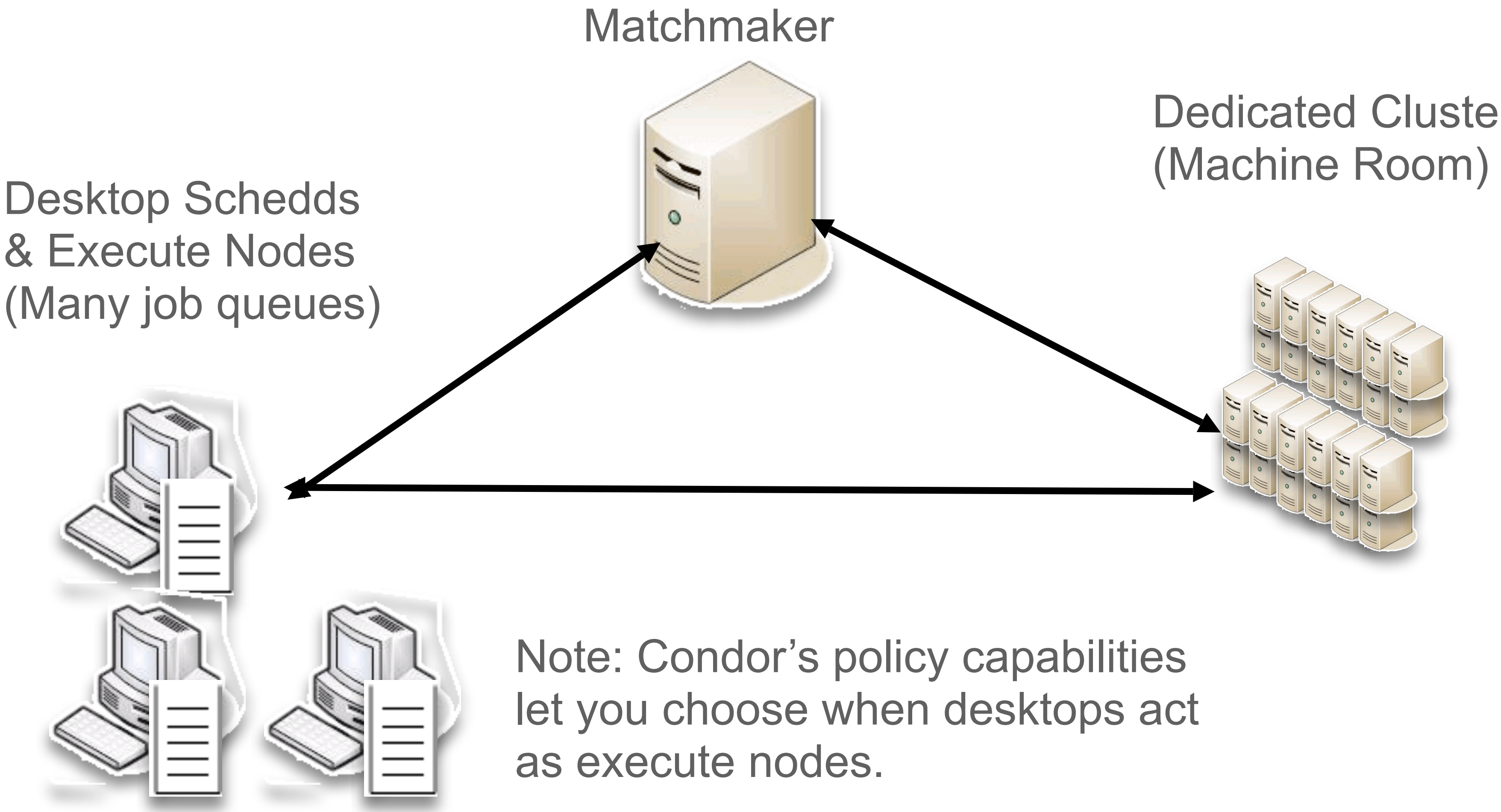
Example Pool 1



Example Pool 1a



Example Pool 2



Note: Condor's policy capabilities let you choose when desktops act as execute nodes.

Sample Condor Pool

```
Name               OpSys      Arch   State     Activity LoadAv Mem   ActvtyTime

slot1@paradent-29. LINUX      X86_64 Unclaimed Idle     0.080  4031  0+00:00:04
slot2@paradent-29. LINUX      X86_64 Unclaimed Idle     0.000  4031  0+00:00:05
slot3@paradent-29. LINUX      X86_64 Unclaimed Idle     0.000  4031  0+00:00:06
slot4@paradent-29. LINUX      X86_64 Unclaimed Idle     0.000  4031  0+00:00:07
slot5@paradent-29. LINUX      X86_64 Unclaimed Idle     0.000  4031  0+00:00:08
slot6@paradent-29. LINUX      X86_64 Unclaimed Idle     0.000  4031  0+00:00:09

...
                     Total Owner Claimed Unclaimed Matched Preempting Backfill

        X86_64/LINUX   200     0       0       200       0          0        0

               Total   200     0       0       200       0          0        0
```

This output comes from the `condor_status`

Summary

- Condor uses ClassAd to represent state of jobs and machines
- Matchmaking operates on ClassAds to find matches
- Users and machine owners can specify their preferences

Four Steps to Run a Job

1. Choose a Universe for your job
2. Make your job batch-ready
3. Create a submit description file
4. Run `condor_submit`

I. Choose a Universe

- There are many choices
  - Vanilla: any old job
  - Standard: checkpointing & remote I/O
  - Java: better for Java jobs
  - MPI: Run parallel MPI jobs

2. Make your job batch-ready

- Must be able to run in the background: no interactive input, windows, GUI, etc.
- Can still use `STDIN`, `STDOUT`, and `STDERR` (the keyboard and the screen), but files are used for these instead of the actual devices
- Organize data files

3. Create a Submit Description File

- A plain ASCII text file (any file extension)
  - Not a ClassAd (although looks very similar)
  - But condor_submit will make a ClassAd from it

- Tells Condor about your job:
  - Which executable,
  - Which universe,
  - Input, output and error files to use,
  - Command-line arguments,
  - Environment variables,
  - Any special requirements or preferences

Simple Submit Description File

```
# Simple condor_submit input file
# (Lines beginning with # are comments)
# NOTE: the words on the left side are not
#       case sensitive, but filenames are!
Universe   = vanilla
Executable = analysis
Log        = my_job.log
Queue
```

4. Run condor_submit

- You give `condor_submit` the name of the submit file you have created:

```
condor_submit my_job.submit
```

- `condor_submit` parses the submit file, checks for it errors, and creates a ClassAd that describes your job.

The Job Queue

- `condor_submit` sends your job's ClassAd to the schedd
  - Manages the local job queue
  - Stores the job in the job queue
    - Atomic operation, two-phase commit
    - "Like money in the bank"
- View the queue with `condor_q`

An example submission

```
% condor_submit my_job.submit
Submitting job(s).
1 job(s) submitted to cluster 1.
```

```
% condor_q
-- Submitter: perdita.cs.wisc.edu :
<128.105.165.34:1027> :
ID    OWNER SUBMITTED RUN_TIME    ST PRI SIZE CMD
1.0   roy   7/6 06:52 0+00:00:00 I  0   0.0  foo

1 jobs; 1 idle, 0 running, 0 held
```

Some details

- Condor sends you email about events
  - Turn it off: `Notification = Never`
  - Only on errors: `Notification = Error`

- Condor creates a log file (user log)
  - "The Life Story of a Job"
  - Shows all events in the life of a job
  - Always have a log file
  - Specified with: `Log = filename`

Sample Condor User Log

```
Job submitted from host: <128.105.146.14:1816>
```

```
Job executing on host: <128.105.146.14:1026>
```

```
Job terminated.
(1) Normal termination (return value 0)
 Usr 00:00:37, Sys 00:00:00 - Run Remote Usage
 Usr 00:00:00, Sys 00:00:05 - Run Local Usage
 Usr 00:00:37, Sys 00:00:00 - Total Remote Usage
 Usr 00:00:00, Sys 00:00:05 - Total Local Usage
9624      -  Run Bytes Sent By Job
7146159   -  Run Bytes Received By Job
9624      -  Total Bytes Sent By Job
7146159   -  Total Bytes Received By Job
```

More Submit Features

```
Universe   = vanilla
Executable = /home/krikava/condor/my_job.condor
Log        = my_job.log
Input      = my_job.stdin
Output     = my_job.stdout
Error      = my_job.stderr
Arguments  = -arg1 -arg2
InitialDir = /home/krikava/condor/run_1
Queue
```

Removing Jobs condor_rm

- If you want to remove a job from the Condor queue, you use `condor_rm`
- You can only remove jobs that you own (you can't run `condor_rm` on someone else's jobs unless you are root)
- You can give specific job ID's, or you can remove all of your jobs with the "*-all*" option.
  - **`condor_rm 21`** · Removes job 21
  - **`condor_rm -all`** · Removes all of your jobs
  - **`condor_rm filip`** · Removes all filip's jobs

How can my jobs access their data files?

Access to Data in Condor

- Use shared filesystem if available
  - Not available for today's exercises
- No shared filesystem?
  - Condor can transfer files
    - Can automatically send back changed files
    - Atomic transfer of multiple files
    - Can be encrypted over the wire
    - This is what we'll do in the exercises
  - Remote I/O Socket
  - Standard Universe can use remote system calls (more on this later)

Condor File Transfer

- `ShouldTransferFiles = YES`
  - Always transfer files to execution site
- `ShouldTransferFiles = NO`
  - Rely on a shared filesystem
- `ShouldTransferFiles = IF_NEEDED`
  - Will automatically transfer the files if the submit and execute machine are not in the same FileSystemDomain

```
Universe   = vanilla
Executable = my_job
Log        = my_job.log
ShouldTransferFiles   = IF_NEEDED
Transfer_input_files  = dataset$(Process), common.data
Queue 600
```

Some of the machines in the Pool do not have enough memory or scratch disk space to run my job!

Specify Requirements

- An expression (syntax similar to C or Java)
- Must evaluate to True for a match to be made

```
Universe      = vanilla
Executable    = my_job
Log           = my_job.log
InitialDir    = run_$(Process)
Requirements = Memory >= 256 && Disk > 10000
Queue 600
```

Specify Rank

- All matches which meet the requirements can be sorted by preference with a Rank expression.
- Higher the Rank, the better the match

```
Universe    = vanilla
Executable = my_job
Log         = my_job.log
Arguments  = -arg1 -arg2
InitialDir = run_$(Process)
Requirements = Memory >= 256 && Disk > 10000
Rank = (KFLOPS*10000) + Memory
Queue 600
```

My jobs run for very long time

- What happens when they get pre-empted?
- How can I add fault tolerance to my jobs?

Condor’s Standard Universe to the rescue!

- Condor can support various combinations of features/environments in different “Universes”
- Different Universes provide different functionality for your job:
  - Vanilla: Run any serial job
  - Scheduler: Plug in a scheduler
  - Standard: Support for transparent process checkpoint and restart

Process Checkpointing

- Condor's process checkpointing mechanism saves the entire state of a process into a checkpoint file
  - Memory, CPU, I/O, etc.
- The process can then be restarted from right where it left off
- Typically no changes to your job's source code needed - however, your job must be relinked with Condor's Standard Universe support library

Relinking Your Job for Standard Universe

To do this, just place "condor_compile" in front of the command you normally use to link your job:

```
% condor_compile gcc -o myjob myjob.c
```

- OR -

```
% condor_compile f77 -o myjob filea.f
fileb.f
```

Limitations of the Standard Universe

- Condor's checkpointing is not at the kernel level. Thus in the Standard Universe the job may not:
  - fork()
  - Use kernel threads
  - Use some forms of IPC, such as pipes and shared memory
- Many typical scientific jobs are OK
- Must be same gcc as Condor was built with

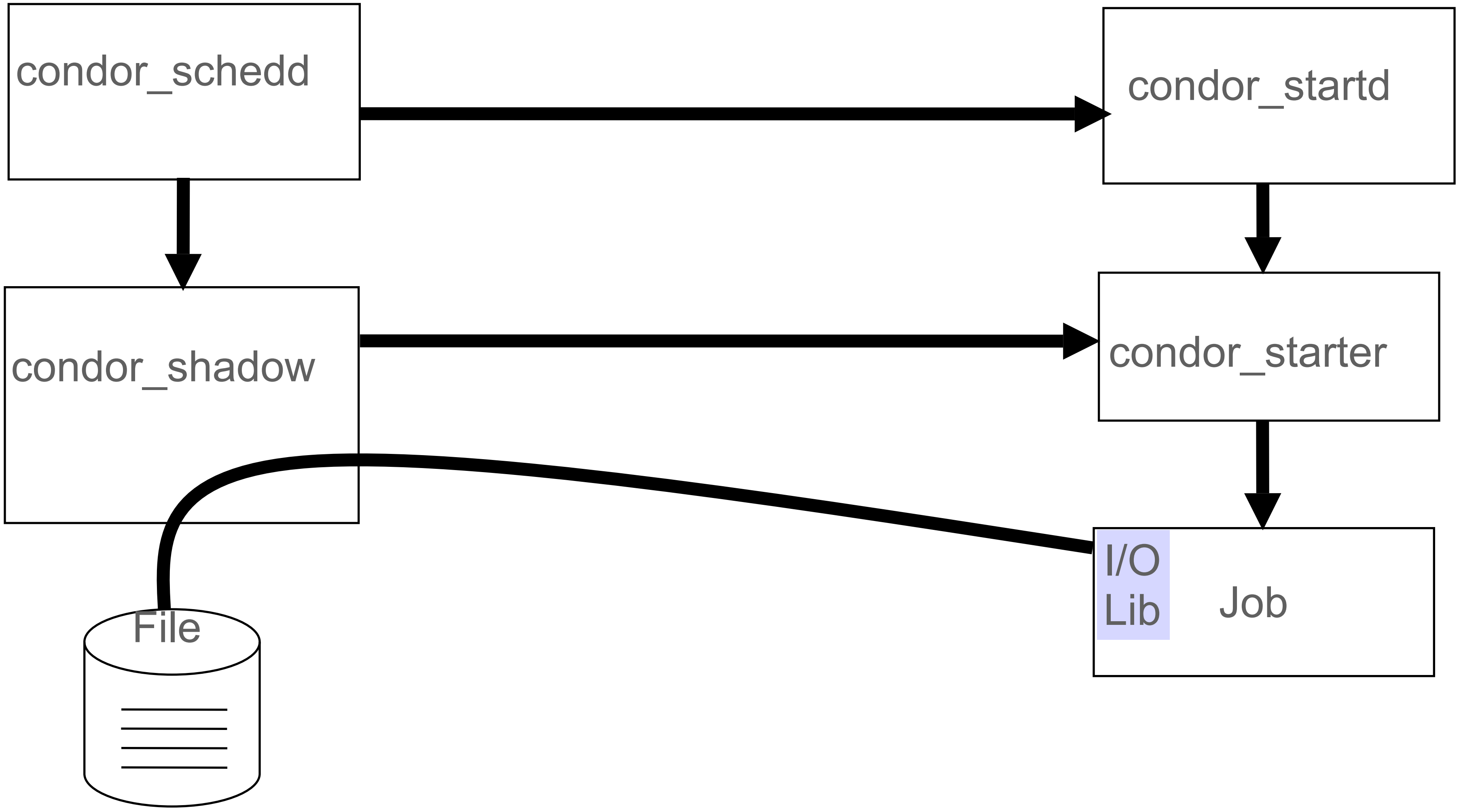
When will Condor checkpoint your job?

- Periodically, if desired (for fault tolerance)
- When your job is preempted by a higher priority job
- When your job is vacated because the execution machine becomes busy
- When you explicitly run:
  - condor_checkpoint
  - condor_vacate
  - condor_off
  - condor_restart

Remote System Calls

- I/O system calls are trapped and sent back to submit machine
- Allows transparent migration across administrative domains
  - Checkpoint on machine A, restart on B
- No source code changes required
- Language independent
- Opportunities for application steering

Remote I/O



condor_schedd
condor_startd
condor_shadow
condor_starter
I/O
Lib
Job
File

Clusters and Processes

- If your submit file describes multiple jobs, we call this a "cluster"
- Each cluster has a unique "cluster number"
- Each job in a cluster is called a "process"
  - Process numbers always start at zero
- A Condor "Job ID" is the cluster number, a period, and the process number ("20.1")
- A cluster is allowed to have one or more processes.
  - There is always a cluster for every job

Example Submit Description File for a Cluster

```
# Example submit description file that defines a
# cluster of 2 jobs with separate working directories
Universe   = vanilla
Executable = my_job
log        = my_job.log
Arguments  = -arg1 -arg2
Input      = my_job.stdin
Output     = my_job.stdout
Error      = my_job.stderr
InitialDir = run_0
Queue                    · Becomes job 2.0
InitialDir = run_1
Queue                    · Becomes job 2.1
```

Submitting The Job

```
% condor_submit my_job.submit-file
Submitting job(s).
2 job(s) submitted to cluster 2.

% condor_q
-- Submitter: perdita.cs.wisc.edu : <128.105.165.34:1027> :
 ID      OWNER              SUBMITTED     RUN_TIME ST PRI SIZE CMD
  2.0     frieda            4/15 06:56  0+00:00:00 I  0   0.0  my_job
  2.1     frieda            4/15 06:56  0+00:00:00 I  0   0.0  my_job
2 jobs; 2 idle, 0 running, 0 held
```

Submit Description File for a BIG Cluster of Jobs

- The initial directory for each job can be specified as run_$(Process), and instead of submitting a single job, we use "Queue 600" to submit 600 jobs at once
- The $(Process) macro will be expanded to the process number for each job in the cluster (0 - 599), so we'll have "run_0", "run_1", … "run_599" directories
- All the input/output files will be in different directories!

Submit Description File for a BIG Cluster of Jobs

```
# Example condor_submit input file that defines
# a cluster of 600 jobs with different directories
Universe    = vanilla
Executable = my_job
Log         = my_job.log
Arguments  = -arg1 -arg2
Input       = my_job.stdin
Output      = my_job.stdout
Error       = my_job.stderr
InitialDir = run_$(Process)
Queue 600
```

- run_0 … run_599
- Becomes job 3.0 … 3.599

More $(Process)

- You can use $(Process) anywhere.

```
Universe   = vanilla
Executable = my_job
Log        = my_job.$(Process).log
Arguments  = -randomseed $(Process)
Input      = my_job.stdin
Output     = my_job.stdout
Error      = my_job.stderr
InitialDir = run_$(Process)
Queue 600
```

- run_0 … run_599
- Becomes job 3.0 … 3.599

Sharing a directory

- You don't have to use separate directories.
- $(Cluster) will help distinguish runs

```
Universe   = vanilla
Executable = my_job
Arguments  = -randomseed $(Process)
Input      = my_job.input.$(Process)
Output     = my_job.stdout.$(Cluster).$(Process)
Error      = my_job.stderr.$(Cluster).$(Process)
Log        = my_job.$(Cluster).$(Process).log
Queue 600
```

Job Priorities

- Are some of the jobs in your sweep more interesting than others?
- `condor_prio` lets you set the job priority
  - Priority relative to your jobs, not other peoples
  - Priority can be any integer
- Can be set in submit file:
  - `Priority = 14`

DAGMan

Directed
Acyclic Graph

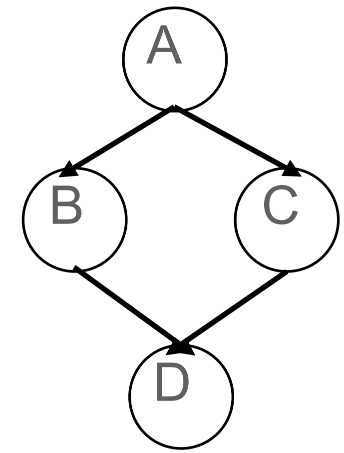
Manager

- DAGMan allows you to specify the dependencies between your Condor jobs, so it can manage them automatically for you.
- Example: "Don't run job B until job A has completed successfully."
- Recall LIGO DAG from Miron's talk?
  - 250,000+ jobs in a DAG

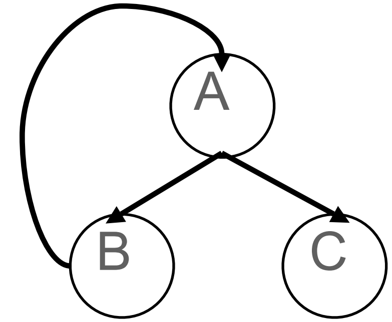
What is a DAG?

- A DAG is the data structure used by DAGMan to represent these dependencies.
- Each job is a node in the DAG.
- Each node can have any number of "parent" or "children" nodes – as long as there are no loops!

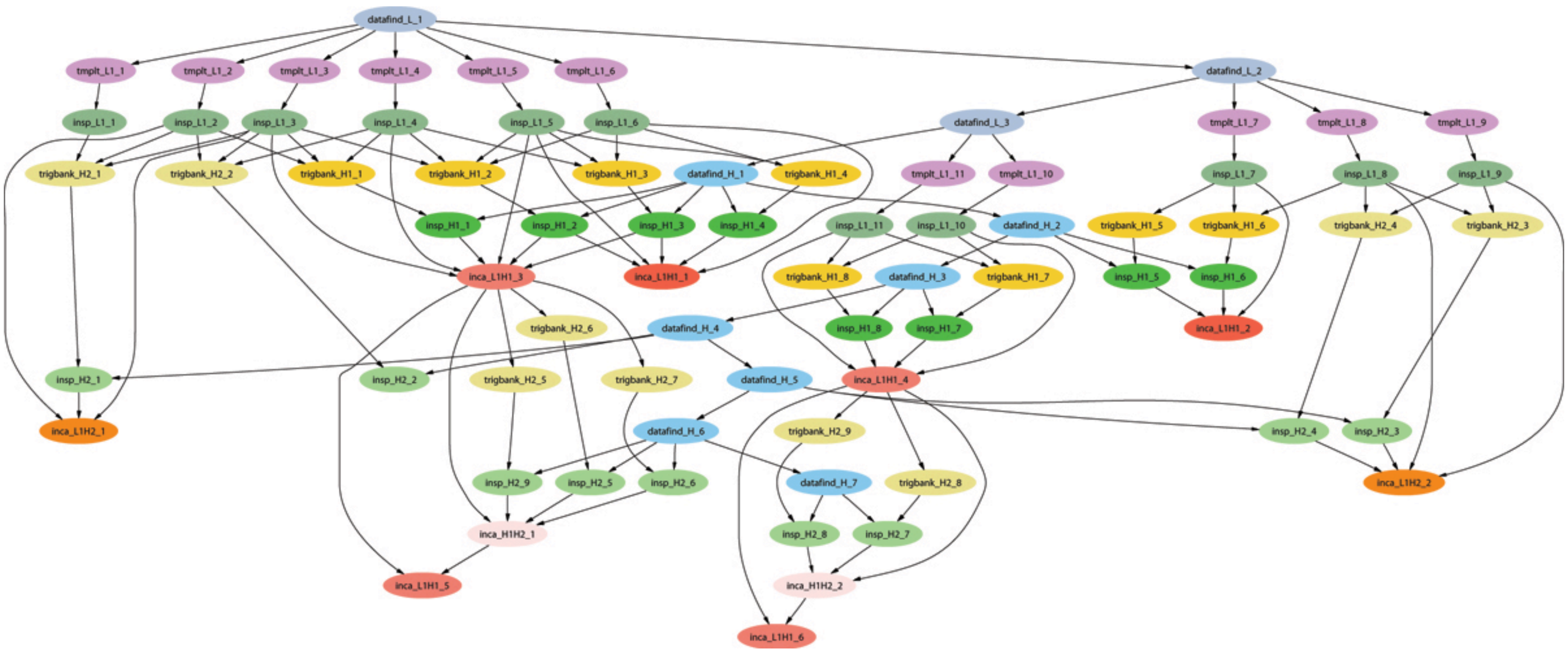
OK:



Not OK:



Example of a LIGO Inspiral DAG

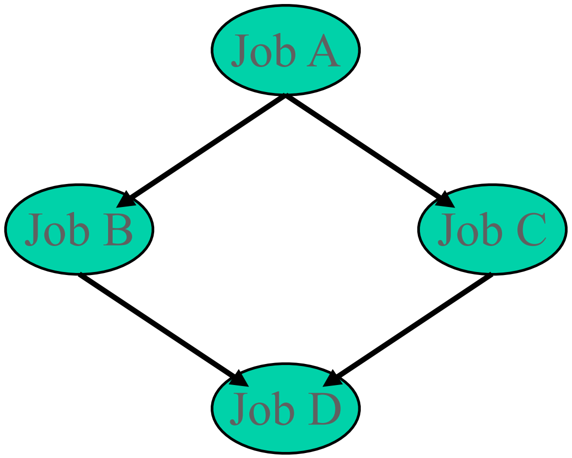


Defining a DAG

- A DAG is defined by a .dag file, listing each of its nodes and their dependencies:

  - `Job A a.sub`
  - `Job B b.sub`
  - `Job C c.sub`
  - `Job D d.sub`

  - `Parent A Child B C`
  - `Parent B C Child D`



DAG Files….

- The complete DAG is five files

One DAG File:

```
Job A a.sub

Job B b.sub

Job C c.sub

Job D d.sub

Parent A Child B C
Parent B C Child D
```

Four Submit Files:

```
Universe = Vanilla
Executable = analysis…
```

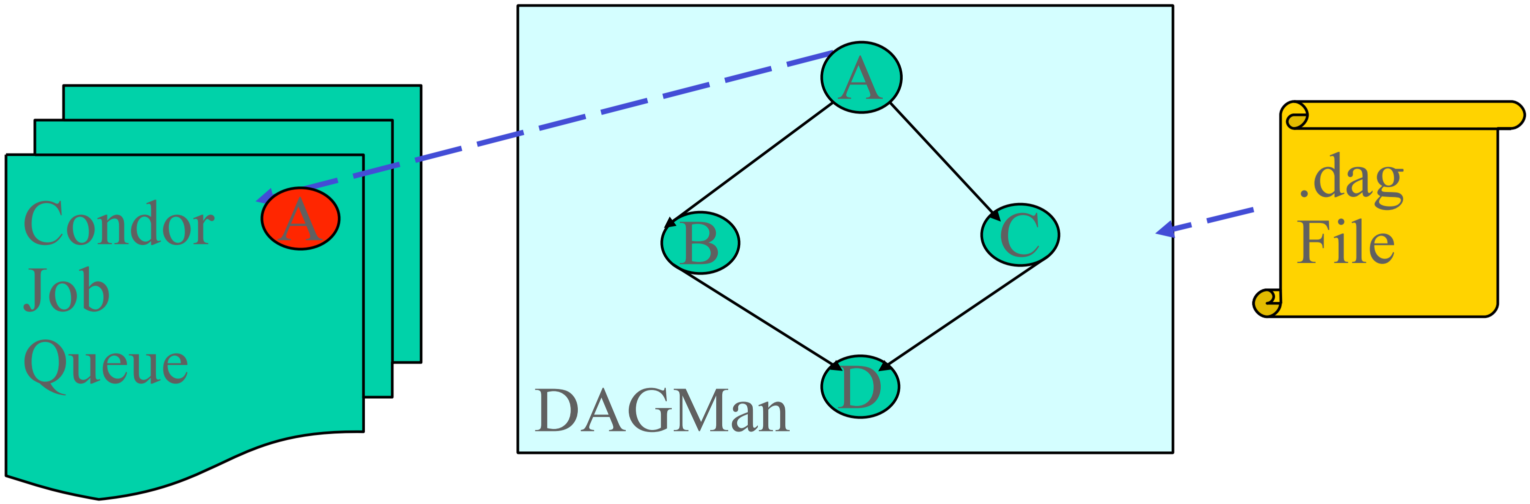
Submitting a DAG

- To start your DAG, just run **`condor_submit_dag`** with your .dag file, and Condor will start a personal DAGMan process which to begin running your jobs:
  - `% condor_submit_dag diamond.dag`

- condor_submit_dag submits a Scheduler Universe job with DAGMan as the executable.
- Thus the DAGMan daemon itself runs as a Condor job, so you don't have to baby-sit it.

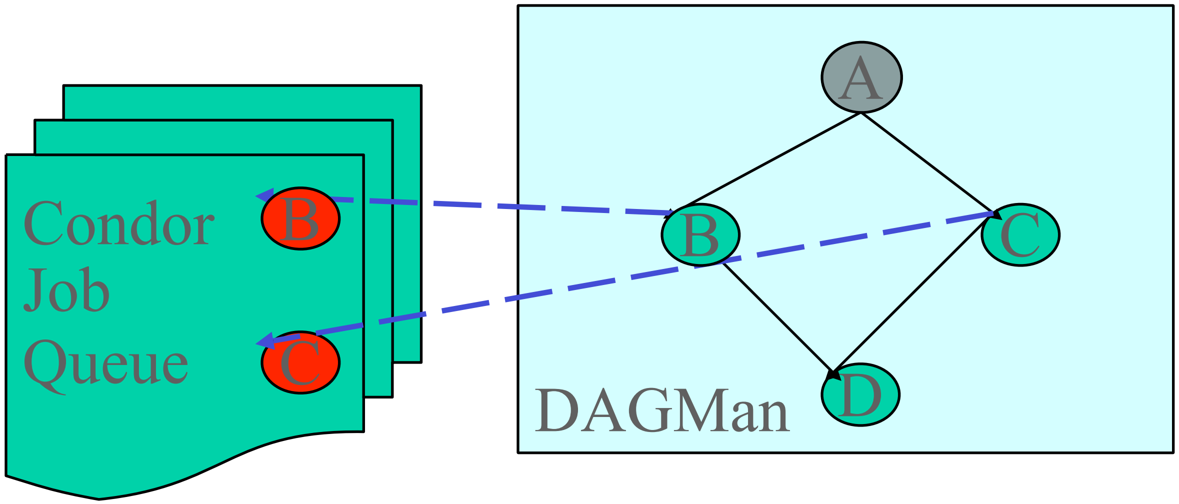
Running a DAG

- DAGMan acts as a scheduler, managing the submission of your jobs to Condor based on the DAG dependencies.



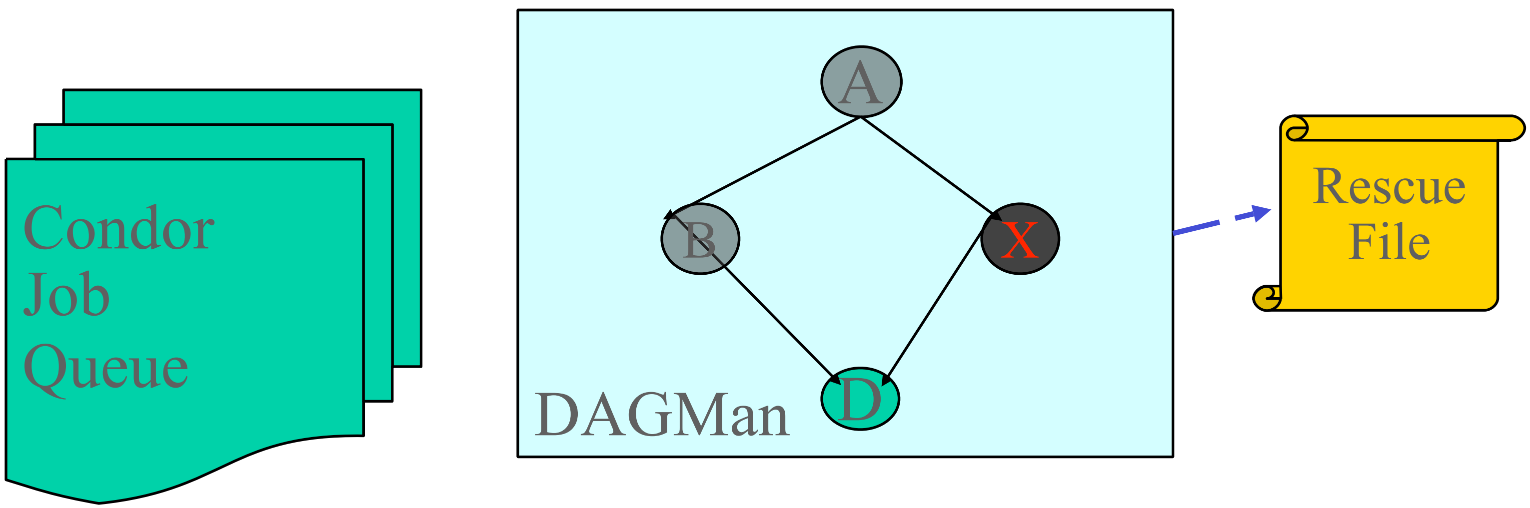
Running a DAG (cont'd)

- DAGMan holds & submits jobs to the Condor queue at the appropriate times.



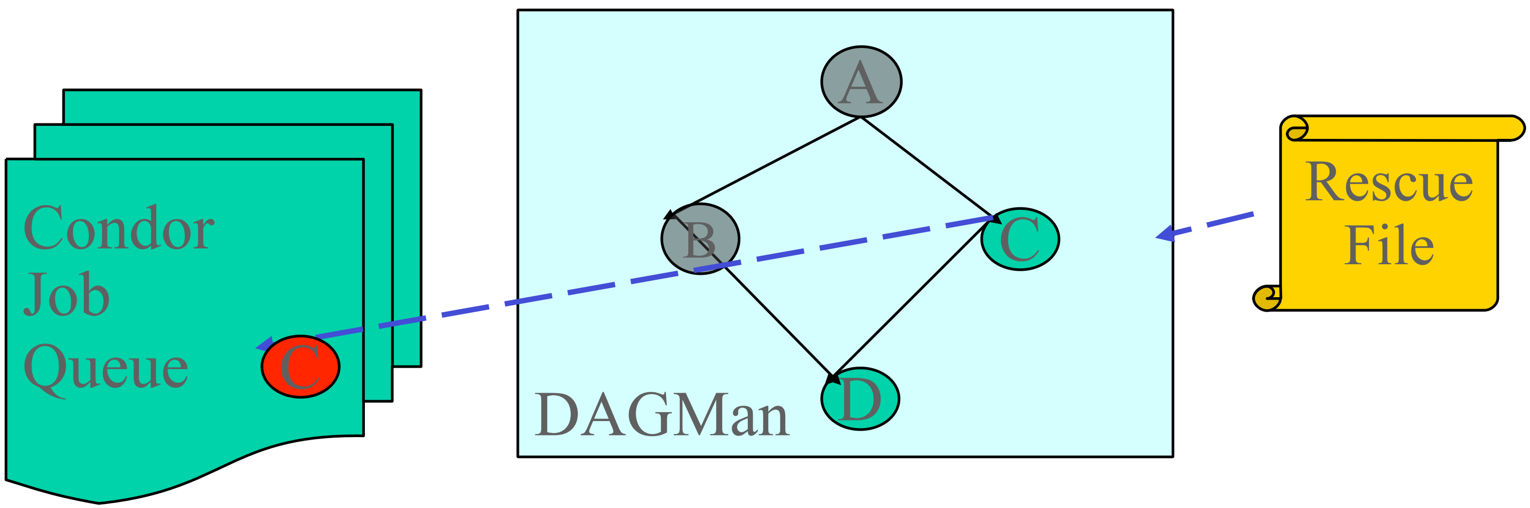
Running a DAG (cont'd)

- In case of a job failure, DAGMan continues until it can no longer make progress, and then creates a "rescue" file with the current state of the DAG.



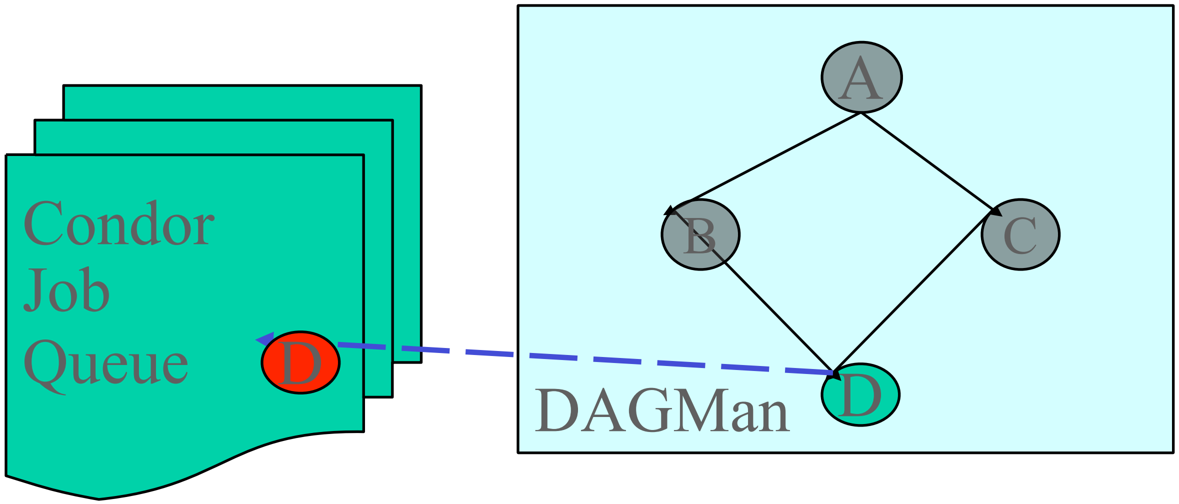
Recovering a DAG

- Once the failed job is ready to be re-run, the rescue file can be used to restore the prior state of the DAG.



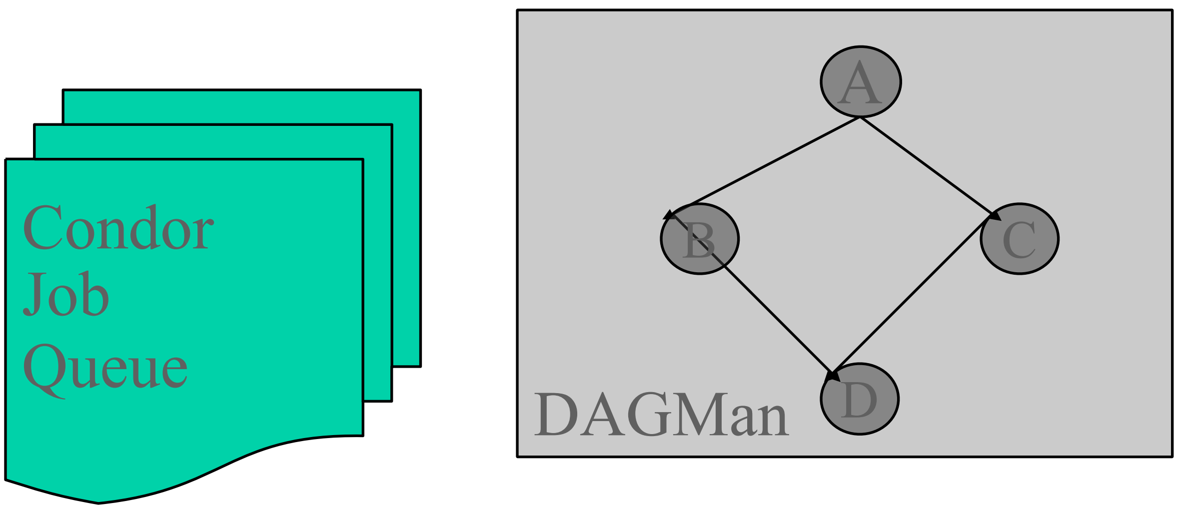
Recovering a DAG (cont'd)

- Once that job completes, DAGMan will continue the DAG as if the failure never happened.



Finishing a DAG

- Once the DAG is complete, the DAGMan job itself is finished, and exits.



DAGMan & Log Files

- For each job, Condor generates a log file
- DAGMan reads this log to see what has happened
- If DAGMan dies (crash, power failure, etc…)
  - Condor will restart DAGMan
  - DAGMan re-reads log file
  - DAGMan knows everything it needs to know

Opportunistic Computing with Condor

- Computing power is everywhere and Condor is trying to make it usable by anyone.
- Every laptop/workstation can have Condor running
- Define rules when it can be used to run a job
- Let share the computing power by building cheap Grids

Topics Not Covered

- Condor Flocking
  - allows a job to run in a different pool when it cannot immediately run it the submitted pool
- Condor-C
  - allows jobs in one machine's job queue to be moved to another machine's job queue
  - highly resistant to network disconnections and machine failures on both the submission and remote sides
- Condor-G
  - using Condor to submit to different Grid middlewares
- *and many others...*

Acknowledgement

- A big thanks to Condor Team at University of Wisconsin-Madison
  - especially
    - prof. Miron Livny
    - dr. Alain Roy
      - the original author of the Condor presentation and exercises at IWSGC'10 which was the main source for this session

http://www.cs.wisc.edu/condor/