Adaptive Grid Scheduling of a High-Throughput Bioinformatics Application

Eduardo Huedo Cuesta (huedoce@inta.es)
Rubén Santiago Montero
Ignacio Martín Llorente

Advanced Computing Laboratory
Centro de Astrobiología
Associated to *NASA Astrobiology Institute*
CSIC – INTA

Distributed Systems Architecture and Security Group
Dpto. de Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Applications Grid Workshop 2003

Introduction

Bioinformatics relies on the management and analysis of huge amounts of biological data.

Bioinformatics could enormously benefit from the **suitability** of the **Grid** to execute **high-throughpu**t applications.

Moreover, collections of biological data are **growing** very fast, so the analysis of this data will only be possible through Grid computing.

We will show the benefits of **adaptive scheduling** in the execution of an existing Bioinformatics application to provide both:

- **fault tolerance** and
- **performance improvement**

using the **Grid*W*ay** tool.

Grid Scheduling

Globus toolkit:

Enables **flexible** and **secure multiple domain** operation with different resource management systems and access policies (**site autonomy**)

Globus components:

- **Security infrastructure** (**GSI**)
- **Resource management** (**GRAM**)
- **Information services** (**MDS**)
- **Data management** (**GridFTP** & **Replica Management**)

Scheduling steps:

•Where does it execute?	**Resource selection**
•What does it need?	**Job preparation**
•How does it start?	**Job submission**
•How is it performing?	**Job monitoring**
•Could it perform better?	**Job migration**
•What does it produce?	**Job termination**

Dynamic Grid Characteristics

High fault rate

- **Resource**
- **Network**

Dynamic resource cost

- **Time** of the day (working / non working)
- Resource **demand**

Grid

Dynamic resource availability

- Job **cancellation**
- Resources **added** and **removed**

Dynamic resource load

- **Shared** resources
- **Idle** resources become **saturated**, and vice-versa

In order to obtain a reasonable degree of both application **performance** and **fault tolerance**, a job must be able to **migrate** among the Grid resources, **adapting** itself to their **characteristics**, **availability**, **performance** and **cost**

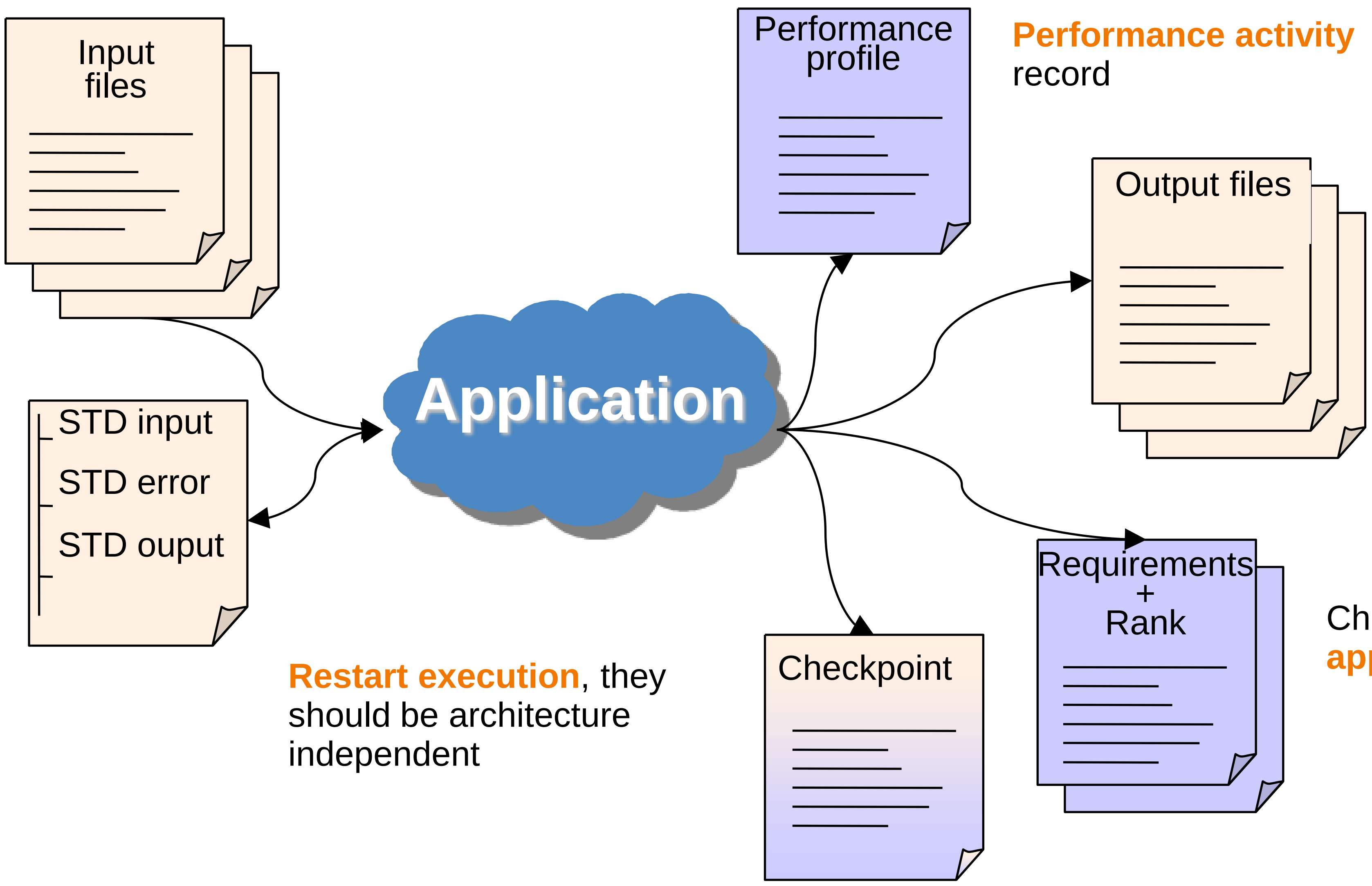
The Grid*W*ay Framework

Provides an **easier** and more **efficient** execution (***submit** & **forget***) on **heterogeneous** and **dynamic** Grids

Design guidelines:

- Easily **adaptable** (modular design)
- Easily **scalable** (decentralized architecture)
- Easily **deployable** (user, standard services)
- Easily **applicable** (wide range of applications)

GridWay: Application Model

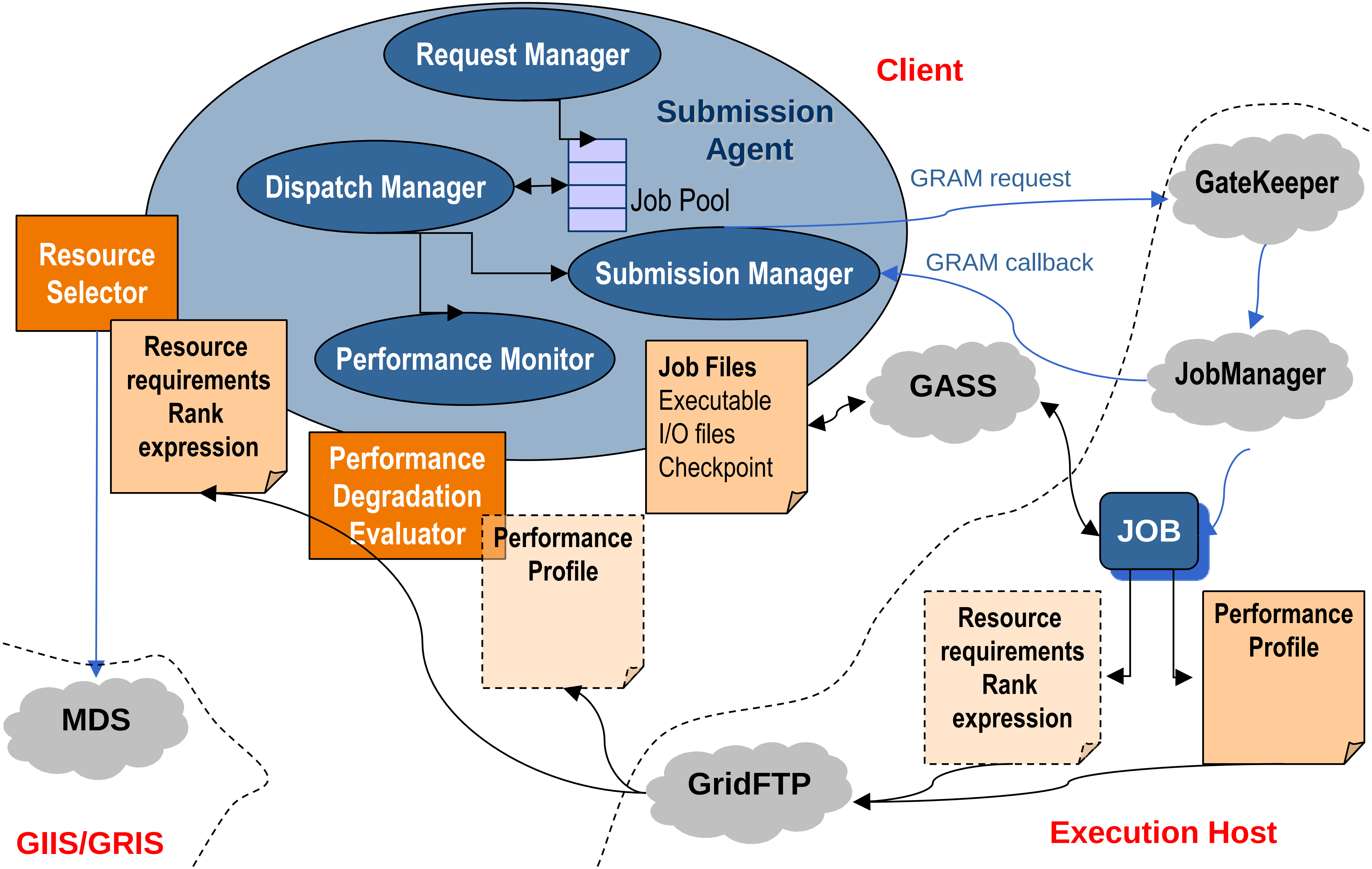


Performance activity record

Restart execution, they should be architecture independent

Characterization of **application needs**

GridWay: Architecture



GridWay: Resource Discovery and Selection

It is maybe the most important step in Grid scheduling and in turns relies completely in the information gathered from the Grid.

Due to the **heterogeneous** and **dynamic** nature of the Grid, users must establish:

- The **requirements** which must be met by the target resources:
  - Characteristics: operating system, architecture, specific software…
  - Implicit requirements: authorization and availability.
- The **preferences** to classify the matching resources:
  - Status: load, free memory, free storage…
  - They can include **performance models**, in terms of application-specific metrics

Static and **dynamic information** gathered from the information services available:

- Predefined list of resources and probe scripts (`uptime, pbsnodes`…)
- Globus MDS
- Network Weather Service
- Replica Location Service

GridWay: Job Execution

Job execution in three steps by the following modules:

- ***Prolog***, which prepares the remote system and stages the input files.
- ***Wrapper***, which executes the actual job and obtains its exit code.
- ***Epilog***, which stages the output files and cleans up the remote system.

Transfer strategies: Use of the **fork *jobmanager*** and a **reverse server** model (file server is started on the submission client).

- **Direct** transfers (files stored in the client or in a remote server) ✓
- Use of **GASS-cache** (critical for parameter sweep applications) ✓
- Data **compression** ✓
- **Replica** management (selection and dissemination) and **3rd party** transfers
- Access to **data bases** (e.g. *Protein Data Bank*)

Advantages versus transfers and execution at once (Nimrod/G, Condor-G):

- Valid for closed systems
- Better adjustment of RSL parameters (**`maxtime`**)
- Possibility to separately schedule transfers and executions
- Easy and efficient way to implement job migration

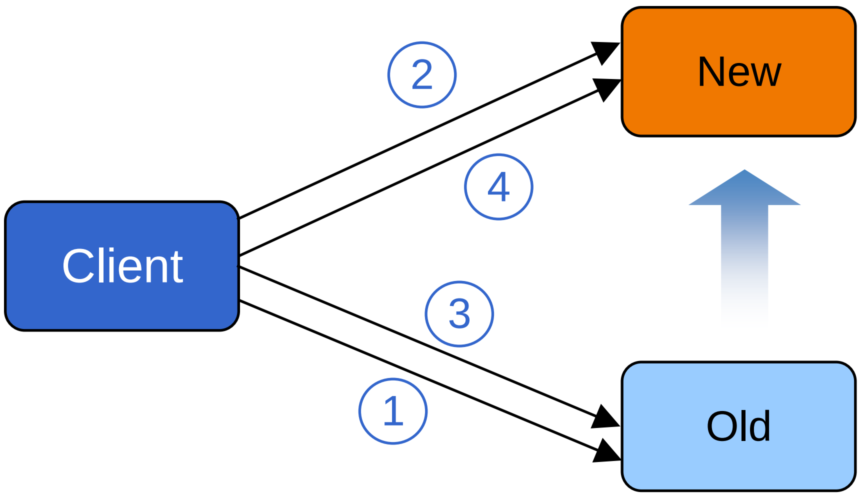
Grid*W*ay: Adaptive Job Execution

Adaptation to changing conditions is achieved through **dynamic rescheduling** of jobs:

- Periodically, to discover ***better* resources** (**opportunistic migration**) ✓
- When a resource or its network connection **fail** ✓
- When the job is **canceled**
- When the job remains **suspended** (*PENDING* state) too much time
- When a **performance degradation** is detected
- When the application **demands** change (**self-migration**)

Grid (first four cases) — **Application** (last two cases)

Job rescheduling can lead to its **migration** to a more suitable resource.



Migration process:

1) Job **cancellation** (if it is still running)

2) **Prolog** submission to the new host (transferring *checkpoint files*)

3) **Epilog** submission to the old host (if it is still available)

4) **Wrapper** submission to the new host

Grid*W*ay: Job Monitoring

Correct job execution:

- The *jobmanager* notifies submission failures as GRAM **callbacks**.
- *The jobmanager* is probed periodically (each **polling** interval). If the *jobmanager* does not respond, the *gatekeeper* is probed. If the *gatekeeper* responds, a new *jobmanager* is started to resume watching the job. If the *gatekeeper* fails to respond, a resource or network failure occurred.
- The **standard output** of *prolog*, *wrapper* and *epilog* is parsed to detect execution failures. In the case of *wrapper*, this is useful to capture the job **exit code**.

Efficient job execution:

- A performance evaluator is periodically executed (each **monitoring** interval) to detect performance slowdown based on system state (accessing the Grid information systems) or application performance (parsing the performance profile).
- The tool keeps count of the overall job **suspension time**.

Experimental Testbed: UCM-CAB

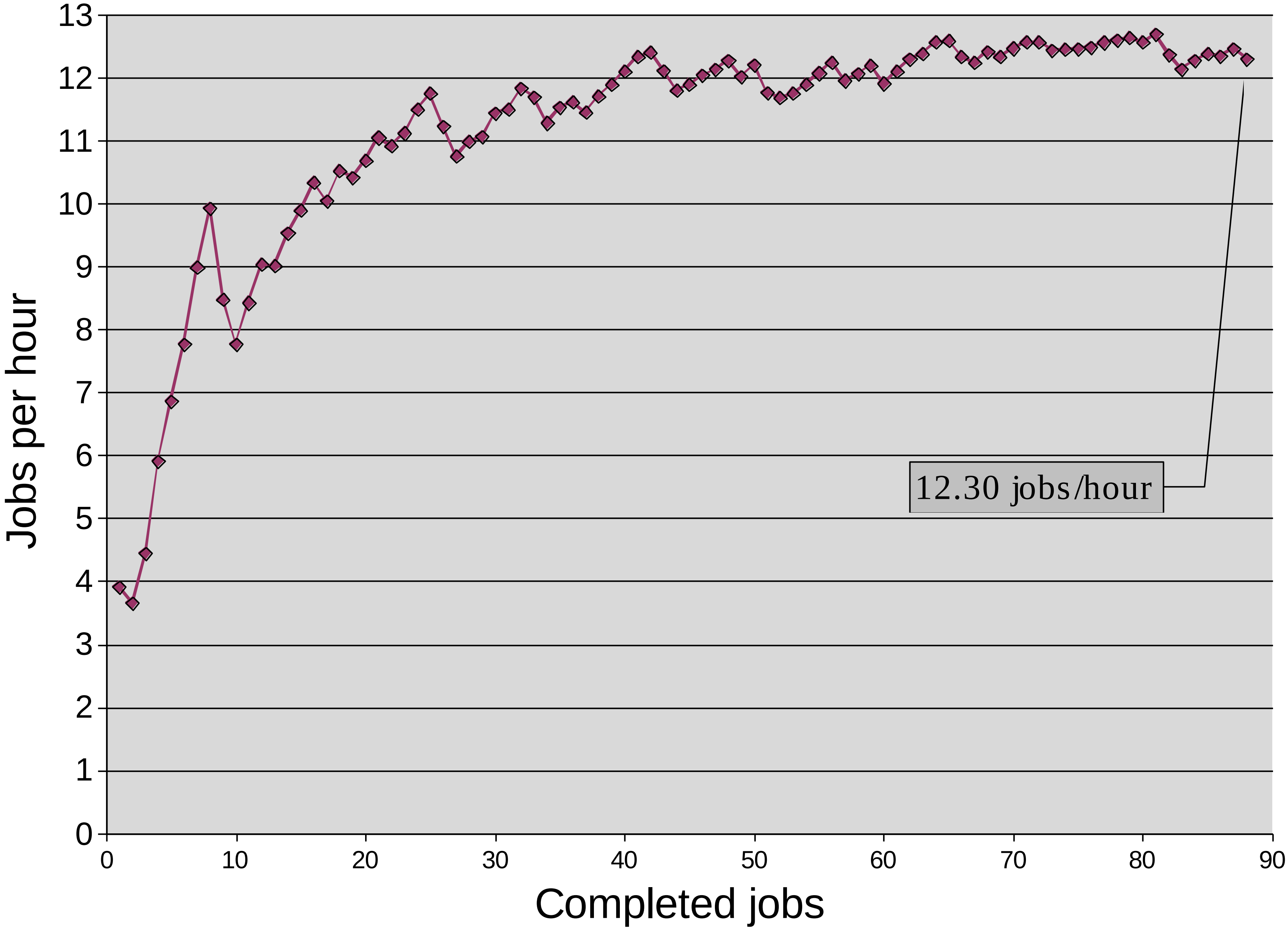
Testbed description:

Host	Nodes	Speed	OS	Memory	VO
ursa	1 x Sun UltraSPARC IIe	500Mhz	Solaris 8	256MB	DACYA
draco	1 x Sun UltraSPARC I	167Mhz	Solaris 8	128MB	DACYA
pegasus	1 x Intel Pentium 4	2.4MHz	Linux 2.4	1GB	DACYA
solea	2 x Sun UltraSPARC II	296MHz	Solaris 8	256MB	QUIM
babieca	5 x Alpha EV6	466MHz	Linux 2.2	1.25GB	CAB

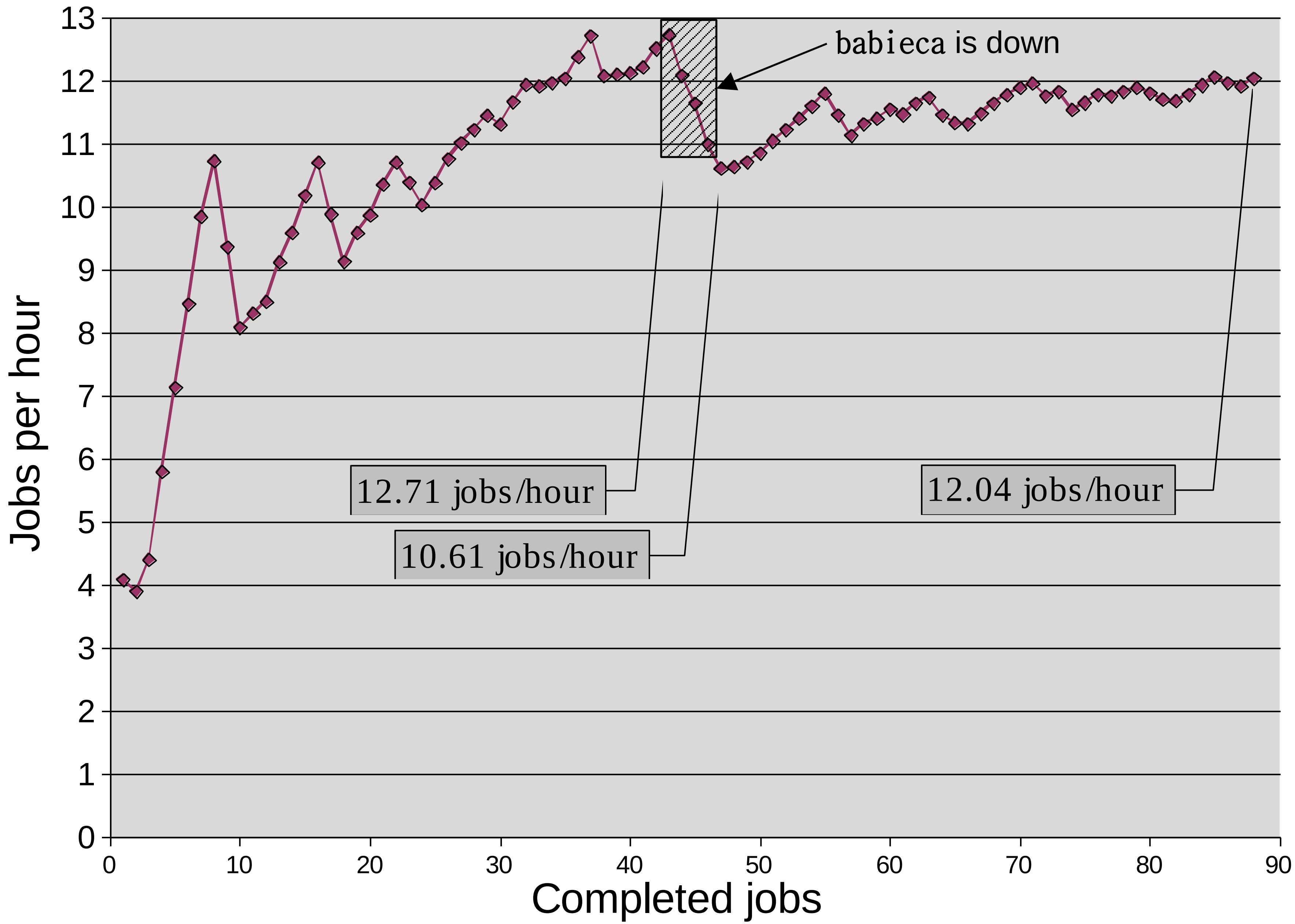
Experiment:

- Protein structure prediction algorithm applied to families of orthologous proteins
- Analysis of 88 sequences of the Triose Phosphate Isomerase enzyme present in different organisms

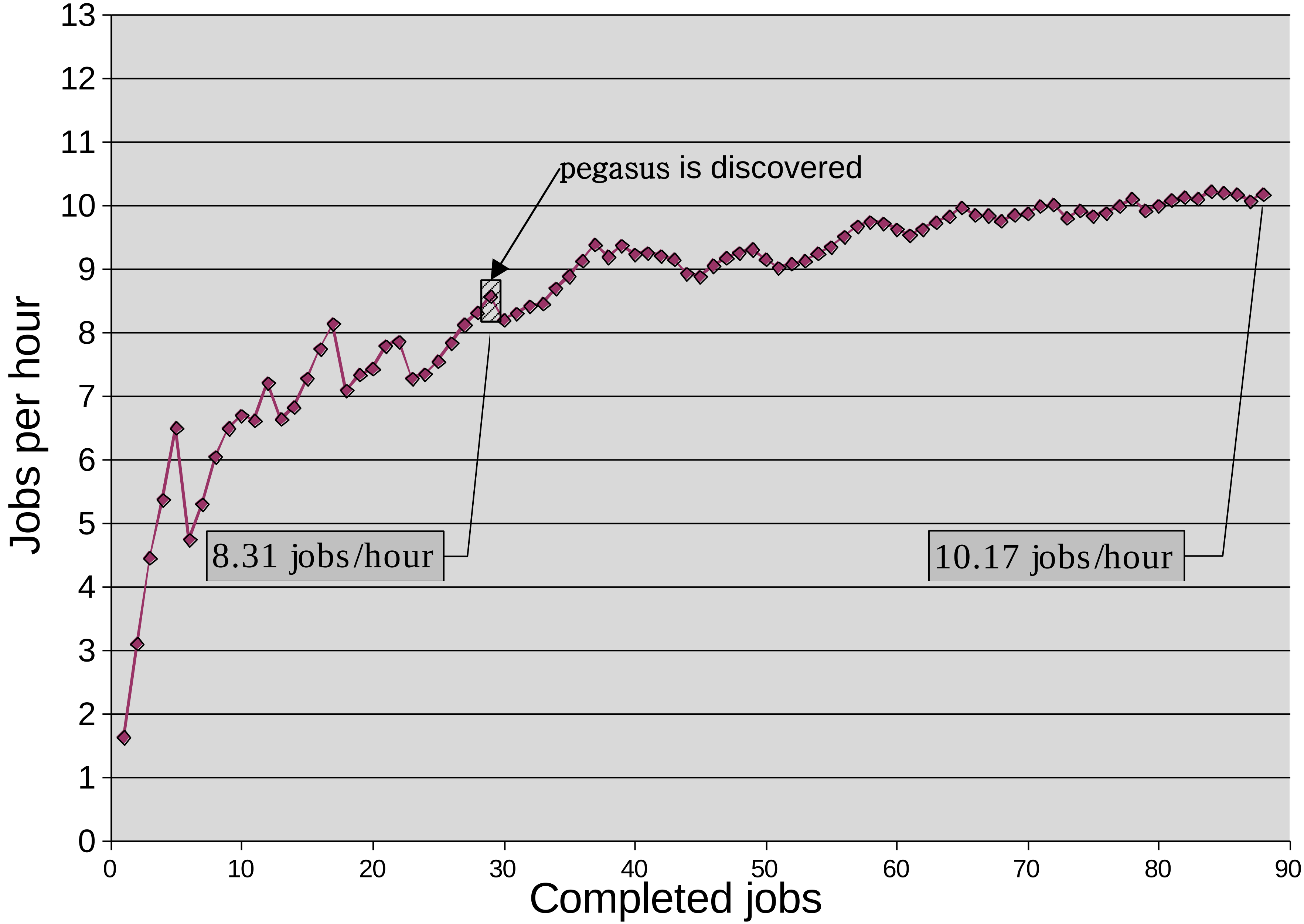
Results: Maximal Throughput



Results: Fault Tolerance



Results: Performance Improvement



Conclusions and Future Work

We have tested the **Grid*W*ay** tool in our research testbed with a **high-throughput** application.

We have seen the benefits of **adaptive scheduling** and **adaptive execution** to provide both **fault tolerance** and **performance improvement**.

This promising application shows the potentiality of the Grid to the study of large numbers of protein structures, and suggest the possible application of this methods to the whole set of proteins in a **complete microbial genome**.