CrossGrid Developer Manual

GridBench

Task 2.3 - Grid Benchmarking

Document Filename:	**CG-DeveloperManual**
Workpackage:	**Task 2.3 - Grid Benchmarking**
Partner(s):	**UCY**
Lead Partner:	**UCY**
Config ID:	**cg-developermanual-v1.0a**
Document classification:	**PUBLIC**

Abstract: This is the developer manual for GridBench, a tool for benchmarking Grid Resources. This documant describes the software architecture and implementation, and it complements the GridBench Installation Guide and the GridBench User Guide.

Delivery Slip

	Name	Partner	Date	Signature
From	George Tsouloupas	UCY	JAN 05	
Verified By				
Approved By				

Document Log

Version	Date	Summary of changes	Author
1.0a	Jan 05	Draft	G. Tsouloupas, M. Dikaiakos

Contents

Copyright Notice

This research is partly funded by the European Commission IST-2001-32243 Project CrossGrid

introduction

Benchmarking metrics published on the Grid can provide a basis for users to assess the "quality of service" expected of a Grid resource or a Virtual Organization providing computational services at a given cost. Grid benchmarks can be used by middleware developers to compare different middleware solutions such as job submission services, resource allocation policies, scheduling algorithms, etc. Grid-oriented benchmarks can serve as an evaluation of the fitness of a collection of distributed resources for running a specific application. As common programming models or paradigms start to emerge for programming in Grid environments, Grid benchmarks can serve as a feasibility study of running a general class of applications (or applications following a similar programming paradigm).

The heterogeneity of Grid platforms and the dynamic nature of Grid resources makes the archival and interpretation of measured metrics a complex task and raises questions about the overall applicability of benchmarking. Existing platforms are largely under continuous re-design and development, with very limited cross-platform interoperability, making the specification, submission, and management of jobs is a tedious process. Measuring and/or monitoring performance metrics at the application level of the Grid is currently a target of ongoing research work. Performance measurements are affected by a variety of factors, including the characteristics of resources allocated for a particular run, the time-dependent latency and bandwidth of shared Internet links used for communication between remote sites, the performance capacity of middleware libraries used at the application level, etc.

0.1 Abbreviations and Acronyms

1 Implementation Structure

1.1 Product Use Cases

We present 2 simple use-case scenarios for GridBench in order to illustrate the functionality visible to the end-user and the overall simplicity in using the tool to get performance metrics for Grid resources. First we describe the scenario where a user would like to get a "picture" of the currest status of a set of resources in terms of low-level performance metrics. In the second case the user has a specific application in mind and would like to select a resource onto which to execute the application. Many other use-case scenarios are possible; in fact some do not even involve an end-user. For example, metrics obtained through GridBench mechanisms can be used by a scheduler that performs resource ranking on an application basis in a way that is completely transparent to the user.

1.1.1 Use-case Scenario 1: Comparing resources

As a first use-case, we consider a user who wants to compare a set of resources in terms of 2 "basic" performance factors : CPU FLOP/s and memory bandwidth. The user would like to use "fresh" data so she opts to invoke new benchmark executions instead of fetching historical data. The user can perform the following steps:

1. Determine which metrics will tell you what you want to know about the resources. In this case, the metrics for these factors can be delivered by a set of benchmarks as summarized below:

Factor	Metric	micro-benchmark
CPU	OP/s	EPWhetstone
Memory	bandwidth	EPStream

2. Using the GridBench GUI simply drag each of the benchmarks onto each resource and submit the benchmark (Figure 1.1). When the benchmark execution finishes, the result will be automatically archived.

3. Using the GridBench GUI put together comparative charts for the resources for each benchmark (Figure 1.2).

From the results on Figure 1.2 (the charts were generated using the GridBench GUI) we observe that the three sides that were chosen for comparison vary in their measurements. At this point it is important to note that two of the resources (`ce010.fzk.de` and `gtbcg01.ifca.unican.es`) use dual-CPU worker-nodes. In terms of aggregate CPU performance they vary only sightly. In terms of memory bandwidth the performance varies greatly as shown in figure 1.2(b) (probably due to the memory technology employed at each resource). Considering a memory-intensive application where the main requirement is memory bandwidth then a user (or resource broker) can select the four worker nodes from `ce010.fzk.de` rather than the four from `gtbcg01.ifca.unican.es`.

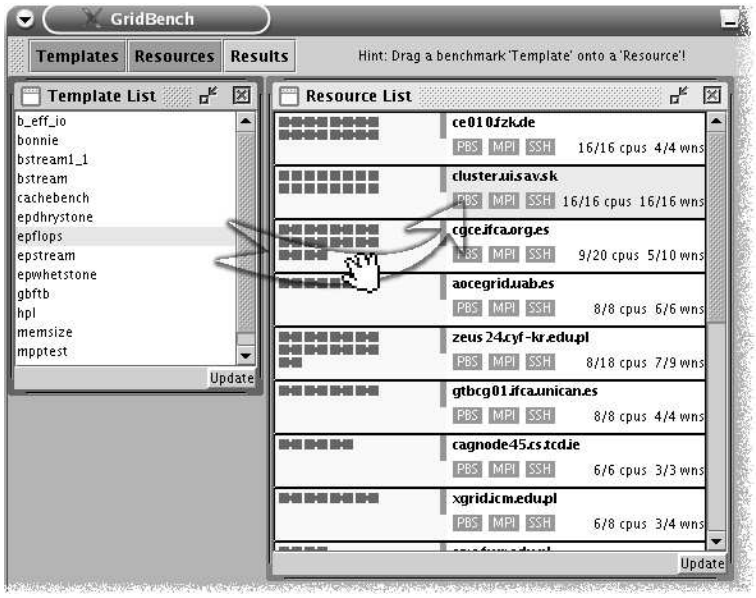


Figure 1.1: Screen-shot of the GridBench graphical user interface. The list on the left is a list of benchmarks that are integrated into GridBench. The list on the right shows the currently available resources and their status in terms of busy/free CPU's. Invoking a benchmark on a resource is as simple as dragging a benchmark from the template list to a resource in the resource list.

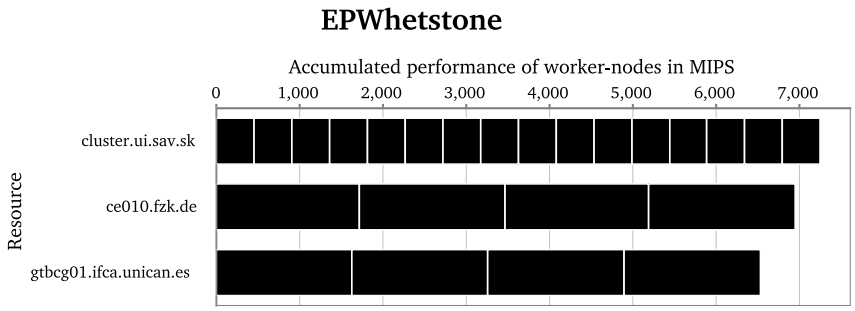
1.1.2 Use-case Scenario 2: Application Performance

As a second use-case we consider a user that wants to compare resources based on performance of a given application or kernel [1]. The user, in this case a surgeon, needs to find the best resources to run a set of simulations. The user has a given application that is used *frequently*, it is therefore justifiable to perform some trivial instrumentation/timings on the application's computational kernel (e.g. to measure iteration times or simply measure completion time on a given dataset) and make it part of the benchmarks available in GridBench.

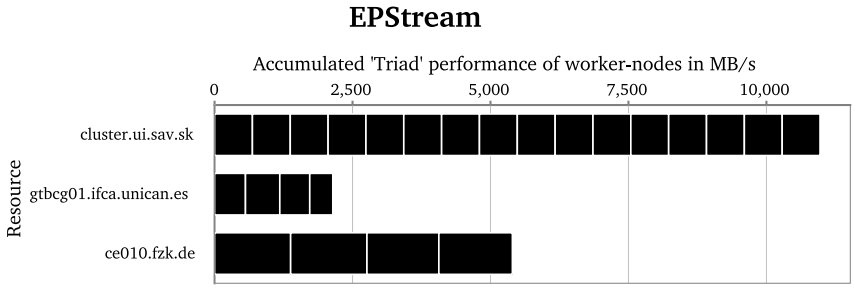
One of the primary design goals of the GridBench framework is the easy inclusion of new benchmarks/kernels. In this use-case scenario the user wishes to include a frequently used kernel; the following steps need to be taken:

1. Create a new GBDL description (model) and add it to the Archiver database;
2. Write a simple implementation of the ParameterHandler Java interface;
3. Instrument the code of the kernel to generate metrics.

[1]The kernel in question is from a medical application, developed at the University of Amsterdam, for pre-operative planning of vascular reconstruction. It involves blood-flow simulation using a Lattice Boltzmann method in arteries using 3-Dimensional data obtained from MRI scans of the patient [7].



(a) Aggregate CPU performance



(b) Aggregate memory bandwidth

Figure 1.2: Results for use-case scenario 1.

A New GBDL description

The first step in adding a new kernel is to create a new GBDL description such as the one that follows:

```
<benchmark name="bstream1_1" date="" type="mpi"
           model="true" description="B_stream 1.1 ..." >
  <parameter name="executable" type="system">bstream1.1</parameter>
  <parameter name="iterations" type="value">40</parameter>
  <parameter name="Reynolds" type="value">20</parameter>
  <parameter name="data_id" type="value">tube38x40x40</parameter>
  <parameter name="stage_file" type="system">tube38x40x40.bs</parameter>
 </benchmark>
```

This description states that:

- this is a benchmark description that is to be used as a model (*model="true"*);
- the parameters *iterations*, *Reynolds* and *data_id* are application-specific parameters required by the kernel executable;
- "bstream1.1" is the name of the *executable* and file "tube38x40x40.bs" needs to be staged;

Since the formatting of command-line arguments to the application executable is application-dependent the user needs to provide a ParameterHandler.

Writing a ParameterHandler

A benchmark-specific ParameterHandler is required for special formatting of command-line arguments (or creation of parameter files etc). In this use-case scenario the applications takes three parameters, which need to be provided in a given order on the command-line. A typical invocation would be:

```
bstream1_1 20 tube38x40x40 40
```

During translation of the GBDL to the middleware-specific job description, the class *ParameterHandler**_bstream1_1*** will be dynamically loaded:

```
public class ParameterHandler_bstream1_1 implements ParameterHandler{
  public java.util.Vector getCommandLineArguments(Benchmark benchmark){
    ParameterCollection parameters=benchmark.getParameters();
    Vector parameterVector=new Vector();

    Parameter data_id=parameters.getParameter("data_id");
    Parameter reynolds=parameters.getParameter("Reynolds");
    Parameter iterations=parameters.getParameter("iterations");

    parameterVector.add(raynolds.getValue());
    parameterVector.add(data_id.getValue());
    parameterVector.add(iterations.getValue());

    return parameterVector;
  }
  ...
}
```

The method *getCommandLineArguments()* is called and returns an ordered list of parameters correctly formatted and ready to be passed to the application executable.

Instrumenting Application codes

Instrumentation of codes is highly application-specific and usually involves trivial modification of the source code to obtain timings at a high level. In our specific use-case the application performs iterations which are controlled by a main loop. In total, about ten lines of code were added in order to time each iteration and output the following metrics onto the standard output:

```
<metric  name="iteration_times" type="vector" unit="s" step="20" period="200">
  <vector name="time">0.079617 0.079529 0.079511 0.079498 ... 0.094326</vector>
</metric>
<metric  name="completion_time" type="value" unit="s">639.633215</metric>
```

Obtaining Measurements

Once the kernel has been integrated into GridBench the user can invoke it just like any other benchmark. The same steps listed in the previous use-case apply to this case as well. One difference is that now the kernel benchmark takes considerably longer to run (tens of minutes) than the micro-benchmarks (a few seconds) in the previous use-case. In this case the user opts to use previously archived executions of the kernel benchmark because it is considerably expensive to get fresh measurements. The steps are now:

1. Retrieve archived results for this kernel;

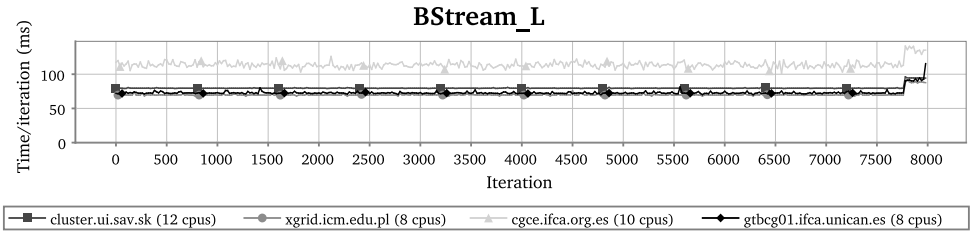


Figure 1.3: Results for use-case 2, showing iteration times of a given kernel on four resources.

2. Benchmark the resources for which there are no archived results;
3. Compare the results.

Invoking the kernel benchmark on a set of resources allows us to construct the chart shown in Figure 1.3. Based on these results a user (or resource broker) can make relatively safe decisions for resource selection, given that the criterion for a "good" selection is the performance of the given kernel.

2 Product Component Model

2.0.3 High-level Architecture

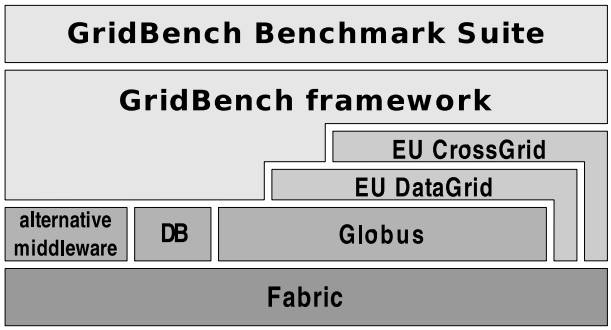


Figure 2.1: High-level view of architecture putting GridBench into perspective.

Figure 2.1 shows a high-level view of the architecture putting aiming to put GridBench into perspective. In the diagram, a component that is directly on top of another means that the component *uses* the one directly below. As shown in figure 2.1 Gridbench is based on existing Grid and other middleware. At the bottom of the diagram in figure 2.1 is the *fabric*. The *fabric* is the physical hardware, local operating systems and basic network protocols. Directly above are the basic Grid services such as *Globus*, as well as other services (for example the MySQL database, indicated by *DB*). Immediately above the basic Grid middleware are additional middleware (such as the *EU CrossGrid* and *DataGrid*) which build services either from scratch or on top of *Globus*.

In figures 2.1 and 2.2 GridBench makes up the top two layers, but before we look at more detail we need to recall that GridBench is a system for benchmarking grids, and to accomplish this it has to accomplish two main objectives:

1. Generate metrics that characterize the performance capacity of resources belonging to a Virtual Organization and *spanning across multiple Grid nodes*, in terms of computational power, file-transfer speed, inter-process communication bandwidth, application-kernel performance, scalability etc.

2. Provide a *tool* for researchers that wish to investigate various aspects of Grid performance, using well-understood kernels that are representative of more complex applications deployed on the Grid. Having access to a corpus of such kernels and being able to easily specify and dispatch parameterized runs of these kernels on Grids, facilitates the characterization of factors that affect application and infrastructure performance, the quantitative comparison of different middleware solutions, algorithms for scheduling, resource allocation, etc.

To address the two main objectives mentioned, Gridbench has two constituents: the GridBench Benchmarking Framework and the GridBench Benchmark Suite. The GridBench Benchmark

suite is a collection of micro-benchmarks , micro-kernel benchmarks and application benchmarks; its purpose is to generate the metrics that will characterize resources and virtual organizations. The GridBench Benchmarking Framework provides facilities for defining and running benchmarks as well as storing, retrieving and analyzing the results of the GridBench benchmark suite. Keeping this in mind, GridBench appears in figure 2.1 as well as in figure 2.2 as two layers. In both cases the *GridBench Benchmark Suite* appears as a different layer from the *GridBench Framework* (also referred to as the *GridBench toolkit*).

Figure 2.2 shows a high-level view of the architecture by including more detail as to which *services* or *API*'s are used. It is important to clarify that in figure 2.2 internal entities do not necessarily use entities are directly below them. They simply indicate that services/API's in the same enclosed shaded area belong to the same middleware/system. The hatched area (i.e. the top two layers are the GridBench Framework and Suite). From *Globus* the main services used are **(i)** the *Gatekeeper* for (GRAM) job submission, **(ii)** the *MDS* for resource discovery and **(iii)** *GridFTP* for file transfer. *MySQL* is used by the *Archiver* and *MPICH-G2* (MPI) is used by most benchmarks in the benchmark suite. From the *EU Datagrid* and *EU Crossgrid*, *R-GMA*, *JIMS* and *SANTA-G* are used for infrastructure monitoring, the *Resource Broker* (RB) for job submission and the *Replica Catalog* (RC) for I/O data management.

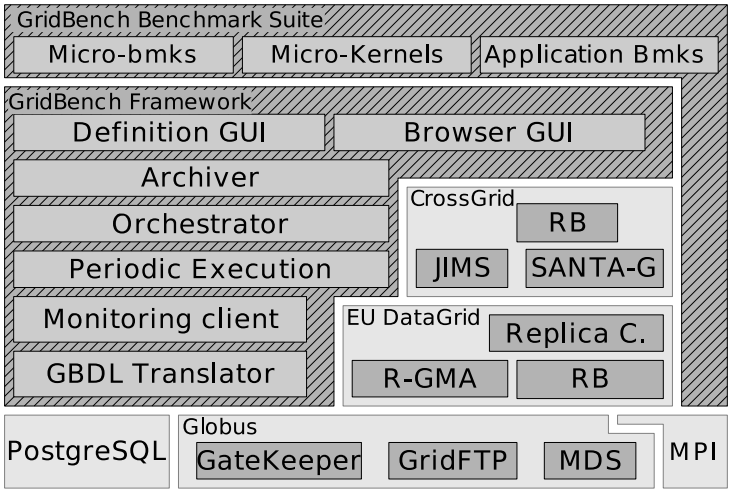


Figure 2.2: High-level overview of the design of GridBench services and components building upon an existing services and middleware. The design allows for easy replacement of the underlying middleware

One of the design goals is to be as independent of specific middleware as possible. The design is open enough to allow easy replacement of the underlying middleware by the use of *Middleware plugins*. To this end we describe the design for *two* underlying middleware: Globus and the EU Datagrid middleware. From the first stages of implementation, the user will be able to use the Globus MDS for information retrieval, and either the EU DataGrid Resource Broker or the Globus GRAM for job execution.

Figure 2.3 outlines the software architecture of GridBench. The main components of this architecture are:

- the GridBench Suite (i.e. The benchmark executables, e.g. Linpack);

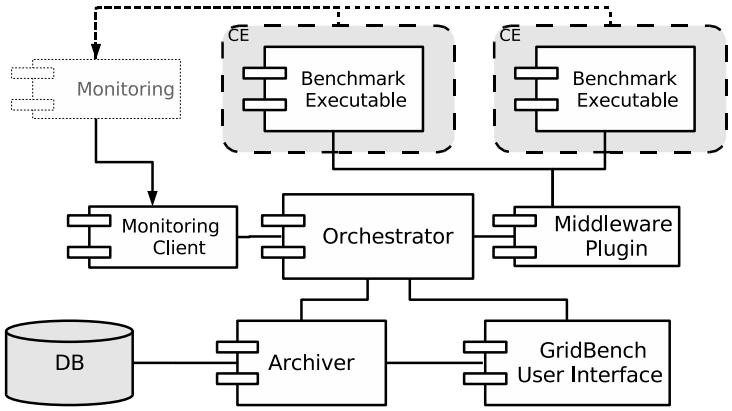


Figure 2.3: The GridBench architecture overview. An outline of system's major components and their interaction.

- the GridBench UI (GUI for defining benchmarks and browsing/analyzing benchmark results).
- the Orchestrator Webservice(manages benchmark execution and collects results);
- the Archiver Webservice (interface to the benchmark database);
- the Middleware plugin (converts XML descriptions of benchmarks to a job description language, performs benchmark execution);
- the Monitoring Client (collects monitoring information);

The diagram in figure 2.3 shows the main components of the GridBench system and (at a very high level) indicates which components interact with each other. This is indicated by a line connecting the two interacting components. Starting from the bottom of the diagram the *Browser GUI* interacts with the *Archiver*. This is in fact a one-way interaction where the *Browser GUI* retrieves the benchmark results from the *Archiver web service*. The *Definition GUI* interacts mainly with the *Orchestrator* but needs to retrieve *models*[1] from the *Archiver*. The *Archiver* maintains a repository of benchmark results and model definitions. The *MySQL Archiver* (described later) is one implementation of an *Archiver* which needs to interact with a MySQL RDBMS. The *GBDL Translator* translates a benchmark definition expressed in the GridBench Definition Language to a middleware-specific job description language. There are currently two implementations of the GBDL Translator interface: the RSLTranslator and the JDLTranslator for generating job descriptions for Globus and the EU-CrossGrid respectively. The *Orchestrator*'s functionality is central to the system. It takes benchmark definitions from the *Definition GUI* (or any other source) and manages their execution. It monitors the job status for each benchmark job and on completion retrieves and archives the resulting metrics. The *Orchestrator* also collects monitoring data (as specified in each GBDL) by using different

[1] A *model* definition is a template benchmark definition with some default parameters, monitoring etc. A model is used for the creation of a new benchmark definition.

Monitoring Clients. The *Periodic Execution* component is used by the *Orchestrator* for periodic execution of jobs as specified in the GBDL.

It is important that we give a short introduction to the GridBench Definition Language at this point because it is central to the operation of the GridBench system, and it will help understand the rest of the system.

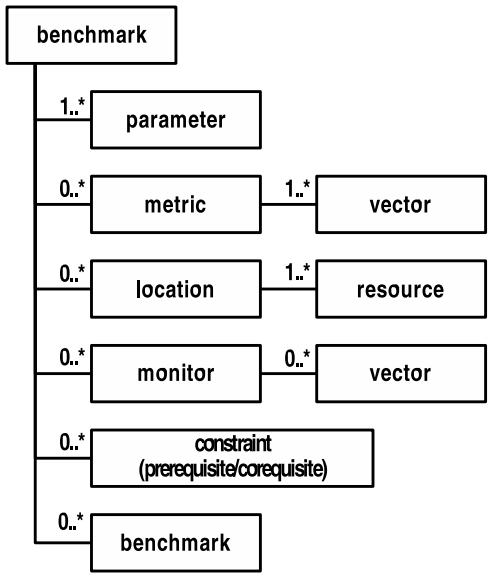


Figure 2.4: A schematic overview of GBDL. Shown in boxes are the main parts of a GBDL document.

Figure 2.4 provides a high-level schematic view of the GridBench Definition Language which is described in detail in chapter 3.2.4. The benchmark definition includes all necessary information needed to run a benchmark. It includes a set of *parameters* which specify details for the benchmark execution (such as the path to the executable and benchmark-specific parameters). It also contains a *location* which specifies the resources on which it should run. A *benchmark* can be hierarchical in nature, meaning that it can be made up of other *benchmarks*. A work-flow can be established by the use of *constraints*. *Constraint* elements can be of type *prerequisite* or *corequisite* and denote dependencies between components. For example, a benchmark involving three component benchmarks, `comp 1`, `comp 2` and `comp 3`, where `comp 2` needs to start after `comp 1` has finished, and `comp 3` must run concurrently with `comp 2` but after `comp 2` starts, would be specified like this:

```
<benchmark name="flow">
  <benchmark id="comp 1"/>
  <benchmark id="comp 2">
    <constraint type="prerequisite" id="comp 1">
  <benchmark id="comp 3"/>
    <constraint type="corequisite" id="comp 2">
</benchmark>}
```

3 Detailed Implementation Model

3.1 Archiver Webservice

3.1.1 Archiver Description

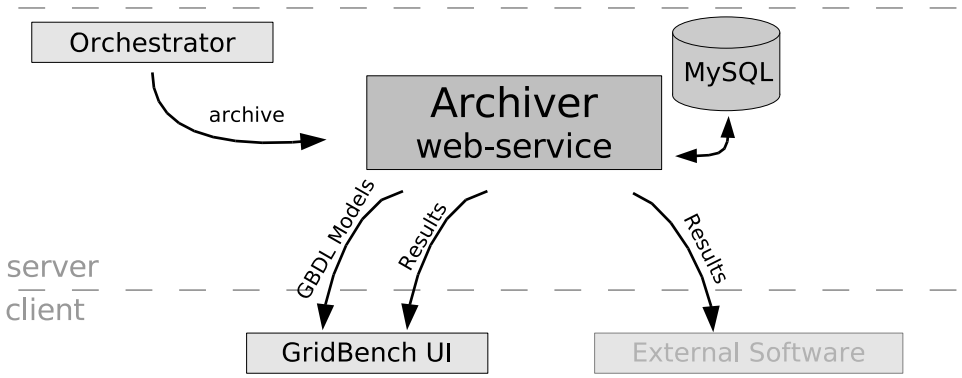


Figure 3.1: Diagram describing the Archiver functionality

The *Archiver* allows the storage and retrieval of results generated by executions of the GridBench Suite Benchmarks through the Gridbench Framework.

The *Archiver* was introduced in order to serve the following purposes:

- To manage a potentially large number of results depending on the size of the Grid under study, the number of benchmarks and the frequency of their execution.
- To provide a central repository for the results allowing access to measurements for users or Grid services.
- To hold a set of *model* definitions serving as customizable benchmark definitions.

The *Archiver* is an *interface* implemented as a web service. The *Archiver* interface may have several implementations depending on the back-end in use. There are already implementations for using the *Apache Xindice* native XML database as a back-end and the newer *MySQLArchiver* implementation using the MySQL relational database as a back-end.

3.1.2 Interface

String getBenchmarks()

Get a list of all benchmark description ID's archived in the database. The list is returned as a newline-separated string of numeric ID's. The benchmark description itself can be retrieved using the *getBenchmark(int ID)* method.

int archiveResult(String gbdl)
Store this GBDL description (XML) and return the assigned unique ID.

String getBenchmark(int ID)
Retrieve the benchmark description (GBDL) with the given unique ID.

String getUniqueBenchmarkTypes()
Return a newline-separated list of unique benchmark names. This will return all benchmark names for which there is a model or at least one archived result.

String getUniqueBenchmarkTypes(String ceName)
Return a newline-separated list of unique benchmark names that have been executed on a given Computing Element. This will return all benchmark names for which there is at least one archived result.

String getUniqueResourceNames()
Return a newline-separated list of unique resource (Computing Element) names. This will return the name of all resources for which there is at least one archived result.

String getDoclist(String benchName,String ceName)
Return the list of benchmark description ID's for the *benchName* benchmark executions on resource *ceName*. The list is newline-separated.

String getModelIds()
Return a list of benchmark descriptions that have the "model" flag set to true (i.e. benchmark descriptions that serve as templates).

3.1.3 The MySQL Archiver

A transition to a relational database and away from a native XML database was decided since the XML database proved overly difficult to query and sometimes extremely slow. Moving to a more structured database allows for more powerful expression of queries with much less effort.

The MySQL Archiver performs the following main tasks:

- Archive a GBDL XML document by generating the correct queries and inserting the data in the appropriate tables while maintaining the structure of the original document.
- Retrieve a requested GBDL document by querying the dataset and reconstructing the original GBDL document.
- Implement the remaining requests specified by the Archiver Interface.

The following is a description of the main tables used for storing the benchmark definition and results. Additional but less important tables are required (for normalization and other issues) but are omitted for simplicity.

```
TABLE benchmarks (
    id integer ,
    name character varying(50) ,
    composite boolean ,
    parent_id integer,
    type_id integer ,
    exec_date timestamp with time zone,
    description character varying(1024),
    model boolean DEFAULT false
);

TABLE metrics (
    id integer,
    name character varying(50),
    value character varying(50),
    valueref integer,
    b_id integer,
    type_id integer
);

TABLE monitors (
    query text,
    b_id integer,
    id integer
);

TABLE resources (
    id integer,
    name character varying(100) ,
    resource_type_id integer ,
    b_id integer,
    cpucount integer,
    wncount integer
);
```

```
TABLE metric_data (
    id integer,
    data text,
    metric_id integer,
    name character varying
);

TABLE monitor_data (
    id integer,
    data text,
    m_id integer,
    name character varying(50)
);

TABLE parameters (
    id integer ,
    name character varying(50) ,
    value character varying(50) ,
    b_id integer ,
    data_type_id integer ,
    parameter_type_id integer
);
```

3.2 Orchestrator Webservice

3.2.1 Orchestrator Description

When a new benchmark description (in the form of GBDL) is delivered to the *Orchestrator* web service for execution the GBDL is translated to the Job Description Language required by the underlying middleware (as indicated by the *mechanism* parameter). All specified monitoring data collection is initiated and the job is submitted. When the job finishes, it's output (the metrics) are incorporated into the benchmark, as well as all the collected monitoring data. The final GBDL is then archived using the Archiver service.

The diagram in figure 3.2 describes the Orchestrator functionality in a series of steps. The steps are given below:

The *Orchestrator* receives a benchmark description in the GridBench Description Language (XML) (as indicated by step **1**). This will originate from the GridBench User Interface or from

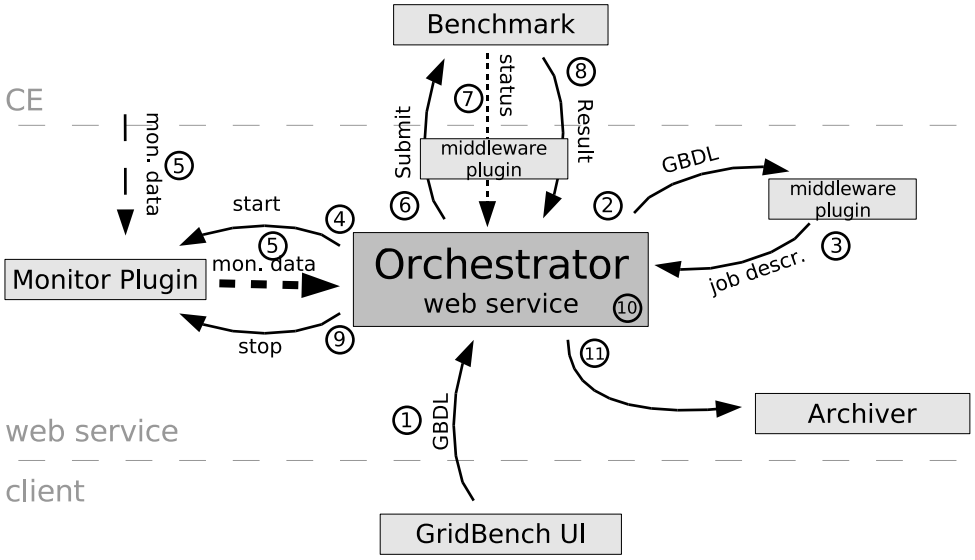


Figure 3.2: Diagram describing the Orchestrator functionality

an automated system performing automated / periodic executions;

The GBDL is passed to the *Middleware plugin* which generates a middleware-specific job description in the syntax and format required by the underlying middleware (step **3**);

The middleware-specific job description is then returned to the *Orchestrator*;

The *Orchestrator* determines all monitoring that need to be performed by looking at the *monitor* tag(s) of the GBDL. Using the *type* and *query* attributes of the *monitor* the correct monitoring plugin is used and monitoring data collection is started (step **4**);

The *Orchestrator* collects streamed monitoring data via the *Monitoring plugin* (step **5**);

In step **6** the benchmark job is then submitted using the *Middleware plugin*;

The benchmark job's status is monitored either by an "in-process wait" (e.g. until the *globus-run* command returns) or by polling the *Middleware plugin* (e.g. using *edg-job-get-status*), as indicated by **7**;

In step**8** the benchmark job finishes and the result (i.e. the standard output containing the *metrics*) is returned to the *Orchestrator* by the *Middleware Plugin*.

The *Monitoring Plugin* is then signaled to stop collecting monitoring data (step **9**) and the collected data is returned to the *Orchestrator*.

The results of the benchmark in the form of *metric* elements, as well as the monitoring data, are incorporated into the original GBDL (step **10**). If the *resources* specified in the *location* tag were not specified explicitly (i.e. resources were allocated by the system) then the *name* and *queue* attributes of the *resource* are correctly inserted.

Finally, in step **11** the resulting GBDL is passed to the Archiver, concluding the *Orchestrator*'s role as it relates to this specific benchmark.

3.2.2 The Orchestrator Interface

String submitGBDL(String gbdl, byte[] proxy, String mechanism)
This method accepts the benchmark definition in GBDL (as described in section 3.2.4) ans is responsible for its execution via the given *mechanism*. The currently allowable mechanism constants are:

> **rsl** or ucy.gridbench.jobmanagement.Orchestrator.USERSL for using Globus directly.
>
> **jdl** or ucy.gridbench.jobmanagement.Orchestrator.USEJDL for using the CrossGrid Resource Broker, using the *Job Description Language*.

The *proxy* must be a valid proxy certificate that will be used for executing the benchmark on the target resource(s).

The method returns a job identifier.

String getJobList()
Return a list of recent jobs submitted though the webservice in tabular form. The fields (delimited by the TAB character) are the following:

job id, *status*, *standard output*, *standard error*

3.2.3 Middleware plugins

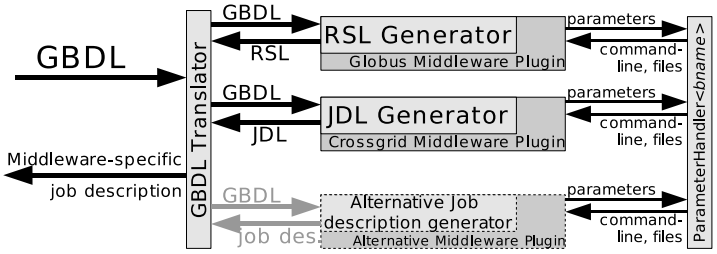


Figure 3.3: Diagram describing the Translator functionality.

The benchmark definitions are expressed in the GridBench Definition Language. In order for the benchmarks to run using a given Grid middleware, they must be expressed in a middleware-specific job definition, for example the *Resource Specification Language* (RSL) in the case of Globus or the *Job Description Language* (JDL) in the case of the EU-Datagrid Resource Brokers. The *Middleware plugin* has the task of performing this translation.

Additionally, the execution of a benchmark (as with any other executable) usually requires either a specially formatted command-line or a specially formatted parameter file. To accomplish this, a Java class must be defined for every benchmark that needs special formatting of parameters, or requires the generation of parameter file(s).

Figure 3.3 outlines this functionality. A request for a GBDL translation given a specific job description language is forwarded to the related middleware plugin, which uses the GBDL data (mainly the *parameter* and *location* tags) to create the job description. The parameters are

then passed to a benchmark-specific parameter handler. For example, for the "Cachebench" benchmark the Java class ParameterHandler_cachebench will be dynamically loaded and the information contained in the GBDL *parameter* tags is used to construct the correct command line.

The interface is as follows:

```
public interface MiddlewarePlugin {
    public String getPluginId();
    public String getJobDescription(String gbdl,File dir);
    public String getJobDescriptionLanguage();
    public String execute(String gbdl,byte[] credential,boolean async);
    public String executeNative(String resource,String description,byte[] credential);
    public String getResult();
    public String getJobStatus();
    public Job getJob();
    public javax.swing.JPanel getVisualisationPanel();
    public void visualize(String gbdl);
}
```

3.2.4 The GridBench Definition Language

The GridBench Definition Language was introduced to the system for several reasons:

- To allow easy definition of benchmarks, including work-flow benchmarks;
- To introduce a middleware-independent definition of benchmarks;
- To serve as a container for associating a definition to the resulting metrics as well as the collected monitoring data.

The complete GBDL DTD can be found in Appendix **??**.

GBDL Documentation

Benchmarks

```
<!ELEMENT benchmark (location,(metric|monitor|parameter|benchmark)*)>
<!ATTLIST benchmark
    id ID #REQUIRED
    date CDATA #REQUIRED
    name CDATA #REQUIRED
    model CDATA #REQUIRED
    description CDATA #REQUIRED
    type (mpi|simple) #REQUIRED >
```

In a GBDL document the top element in the *benchmark*. Each *benchmark* has a set of attributes:

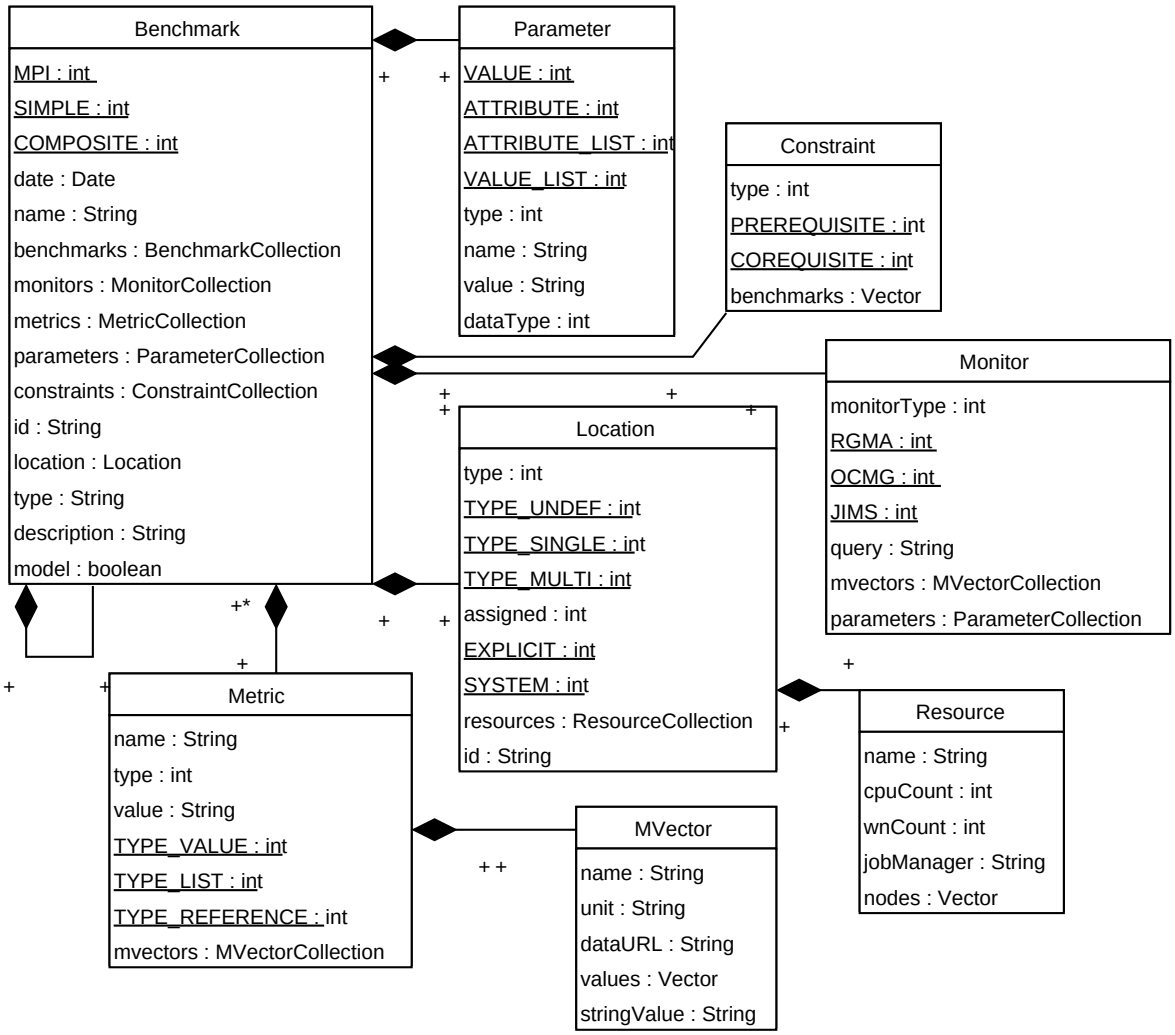


Figure 3.4: UML diagram of the GridBench Definition Language (GBDL) showing the internal structure of a benchmark definition and resulting metrics.

attribute **id**: is a unique id for this GBDL document. When a benchmark contains other benchmarks they should be assigned their own unique id.

attribute **date**: is the date and time this benchmark is submitted. The format is "YYYYMMDDhhmmss".

attribute **name**: is the name of the benchmark (e.g. epstream, mpptest)

attribute **model**: is a boolean indicating that this benchmark definition is to be flagged as a *model*. A *model* can be used as a basis for defining a new benchark run. A *model* does not contain a *location* definition or any *metrics*.

attribute **description**: is a free-text description of the benchmark

attribute **type**: can be one of:

mpi: indicates the benchmark is a MPI application

simple: indicates the benchmark is not a MPI application

A benchmark consists of a *location* specification, a set of *parameters*, a set of *monitors* and possibly a set of *benchmarks*.

Parameters

```
<!ELEMENT parameter (#PCDATA)>
<!ATTLIST parameter
    name CDATA #REQUIRED
    dataType CDATA #IMPLIED
    type (value|list|attribute|attributelist) #REQUIRED >
```

The *parameter* element is used for setting the benchmark parameters

Location

```
<!ELEMENT location (resource)*>
<!ATTLIST location
    assigned (explicit|system) #IMPLIED >
```

The *location* element defines where the *benchmark* will run. It contains a set of *resource* elements. Each *location* has a set of attributes:

attribute **assigned**: can be one of:

explicit: indicates that the location was explicitly defined.

system: indicates the the location was automatically assigned by the system (e.g. a scheduler).

Metrics

```
<!ELEMENT metric (#PCDATA |vector)*>
<!ATTLIST metric
    name CDATA #REQUIRED
    unit CDATA #REQUIRED
```

The *metric* tags contain the results of the benchmark. A *metric* can either contain (i) one or more values or, (ii) a set of *vector* elements. Each *location* has a set of attributes:

attribute **name** is the name of the metric.

attribute **unit** is the unit of the measurement.

Vectors (metric/monitor vectors)

```
<!ELEMENT vector (#PCDATA)>
<!ATTLIST vector
    name CDATA #REQUIRED
    unit CDATA #IMPLIED
    type (value|reference) #REQUIRED
    unit CDATA #IMPLIED>
```

A set of *vector* elements make up a *metric*. The following attributes make *vector* element:

attribute **type** can be one of:

name is an identifier of the measurement that makes up the *metric*.

value: indicates that the metric value(s) are explicitly included inside this tag.

reference: indicates that the measurements are given in the referenced file or URL.

unit is the unit of the measurement(s).

The values are given as a list of measurements delimited by a space. Two special *vectors*, that carry the *name* **hostname** and *timestamp*, contain the Worker Node names on which the benchmark was executed and the timestamp of each of the measurements.

Constraints

```
<!ELEMENT constraint (#PCDATA)>
<!ATTLIST constraint
    id IDREF #IMPLIED
    type (prerequisite|corequisite) #REQUIRED >
```

The *constraint* element aims to define a work-flow when a *benchmark* is of type *composite*.

attribute **id** is a unique identifier of a *benchark* of which the execution is either a *prerequisite* or a *corequisite*. It must match the **id** specified by any other *benchmark*.

attribute **type** indicates a *prerequisite* or a *corequisite*. When a *prerequisite* is specified then the containing *benchmark* must execute after the *benchmark* specified by the **id**. When a *corequisite* is specified then the containing *benchmark* must execute *concurrently* with the *benchmark* specified by the **id**.

Resources

```
<!ELEMENT resource EMPTY >
<!ATTLIST resource
    cpucount CDATA #REQUIRED
    wncount CDATA #REQUIRED
    jobmanager CDATA #IMPLIED
    name CDATA #IMPLIED >
```

As part of the *location* element, the *resource* element specifies a single resource (CE) to be used for the execution of the containing *benchmark*. A *resource* is a specified by the following attributes:

attribute **name** is the hostname of the *resource* (Computing Element).

attribute **cpucount** is the number of CPUs specified.

attribute **wncount** is the number of worker nodes onto which the *cpucount* will be allocated.

attribute **jobmanager** is the Globus job-manager to be use (in effect specifying the *job queue* to be used on the specified resource.

If the **name** is not provided or it is blank, it signifies that the underlying middleware is expected to automatically match any resource for the specified *cpucount* and *wncount*.

Monitors

```
<!ELEMENT monitor (vector*)>
<!ATTLIST monitor
    type (jims|rgma) #REQUIRED
    query CDATA #REQUIRED>
```

The *monitor* element specifies the type of monitoring to be performed during the execution of the benchmark. The following attributes provide the details of the monitoring to be performed:

attribute **type** can be one of:

jims which indicates that JIMS will be used for collection of monitoring information.

rgma which indicates that R-GMA will be used for collection of monitoring information.

attribute **query** is the middleware-specific query specifying the system parameters to be monitored .

After execution the monitoring data will be incorporated into the *monitor* element in the form of *vectors* (specified above).

4 Benchmarks

4.1 Micro-benchmarks

4.1.1 Metrics

A critical step in our methodology is the selection of a concise set of metrics for the low-level characterization of the Grid's computational resources. It is a reasonable assumption to make that the resource's performance depends mainly on the performance of its CPU's, the performance of its memory and caches, and the performance of its interconnects. Of course there is a wealth of other factors affecting machine performance ranging from I/O performance to Operating System robustness to fitness for running a specific application. We chose to limit the set of metrics to a concise size, but kept the design open for easy inclusion of more metrics as deemed necessary. In terms of specific metrics we have chosen (i) *Operations Per Second* for CPU performance (integer/floating-point), (ii) *Available Memory* and *Bytes per second* for writing and reading to and from main memory/cache, and (iii) *Latency* and *Bandwidth* for evaluating the machine's interconnects. These metrics are easily understood and well-established for evaluating their respective performance factor.

Table 4.1: Metrics and Benchmarks.

Factor	Metric	Delivered By
CPU	Operations per second (mixture of floating point and integer arithmetic)	EPWhetstone
CPU	Floating-Point operations per second	EPFlops
CPU	Integer operations per second	EPDhrystone
memory	sustainable memory bandwidth in MB/s (copy,add,multiply,triad)	EPStream
memory	Available physical memory in MB	EPMemsize
cache	memory bandwidth using different memory sizes in MB/s	CacheBench
Interconnect	latency, bandwidth and bisection bandwidth	MPPTest
I/O	Effective I/O bandwidth	b_eff_io

4.1.2 Micro-benchmarks

In order to deliver the required metrics, eight benchmarks are employed:
(i)*EPWhetstone*, (ii)*EPFlops*, (iii)*EPDhrystone*, (iv)*EPStream*, (v)*CacheBench*, (vi) *EPMemsize*, (vii)*MPPTest* and (viii) *b_eff_io*.

During the execution of each of the benchmarks listed above, it is imperative that the only process imposing substantial load on the CPU is the benchmark process, especially since the

Table 4.2: Metrics returned by the "EPFlops" benchmark and the operation counts in each reported metric.

Metric name	FADD	FSUB	FMUL	FDIV	Total
mflops-1	21 (40.4%)	12 (23.1%)	14 (26.9%)	5 (9.6%)	52
mflops-2	58 (38.2%)	14 (9.2%)	66 (43.4%)	14 (9.2%)	152
mflops-3	62 (42.9%)	5 (3.4%)	74 (50.7%)	5 (3.4%)	146
mflops-4	39 (42.9%)	2 (2.2%)	50 (54.9%)	0 (0.0%)	91

results are calculated using wall-clock time. Another thing to note is that the "EP" prefix of some benchmark names (namely EPWhetstone, EPFlops, EPDhrystone and EPStream) denotes the "embarrassingly parallel" nature of its execution, which means that each process runs on a CPU independently without any communication during the computation. The accumulated result from all the processes is then reported as the performance of the whole resource. In many cases it is useful to have results from benchmarks executed as both 1) one process per CPU and 2) one process per SMP node (see the description of *EPStream* for an example).

EPWhestone is a simple adaptation of the traditional Whetstone CPU benchmark [2] so that it runs simultaneously on a set of CPU's using MPI. It is implemented in C, and uses MPI for collecting the final measurements from each process (communication time is excluded from measurements). Each process performs a mixture of operations, such as integer arithmetic, floating point arithmetic, function calls, trigonometric and other functions. The benchmark gets the current time using *gettimeofday()*, runs for a few seconds, calculates the wall-clock time difference and reports the rate at which these operations were performed on average. The typical execution time is less than 10 seconds.

EPFlops is a floating-point CPU benchmark adapted from the "flops" benchmark [1]. It is modified so that it runs simultaneously on a set of CPU's using MPI. It measures the performance of a CPU's floating-point operations in different "mixes" of floating-point operations. The benchmark employs a set of 8 modules, where each module is made up of a different mix of operations. Different combinations of the 8 modules yield a set of four metrics ("ratings") with different ratios of each of the four floating-point operations. The benchmark tries to maximize register usage in order to be as independent as possible from the performance of the memory subsystem. It is implemented in C. Table 4.1.2 gives a summary of the distribution of floating-point operations in the four metrics delivered by the "EPFlops" benchmark. For example, the *mflops-2* metric, which is also reported in Figure **??**, does 152 operations per loop. Out of the 152 operations, 58 (38.2%) are additions, 14 (9.2%) are subtractions, 66 (43.4%) are multiplications, and 14 (9.2%) are divisions.

EPDhrystone is an integer operations benchmark, adapted from the C version of the "dhrystone" benchmark [8]. It is modified so that it runs simultaneously on a set of CPU's using MPI. Dhrystone is based on a workload from an extensive set of applications, but does not target numerical computations. It focuses on "systems programming" applications which perform

mainly integer operations. As before, the benchmark has been adapted to run concurrently on a set of CPU's using MPI. The benchmark returns the accumulated result from all the processes in "dhrystones" per second.

EPMemsize is a platform independent benchmark that aims to measure memory capacity. It is written in C and it runs simultaneously on a set of CPU's using MPI. It first determines the maximum amount of memory that can be allocated. It then proceeds to determine the maximum amount of memory that can be allocated *in physical memory*. The size of physical memory available is important to memory-intensive applications that profit from allocating as much memory as possible while avoiding the use of slow swap memory. Detecting the physical memory in the machine in a platform-idependent way may not depend on any system-specific system call to get the memory size. More importantly, the value that is returned by a "get_free_memory()" system call is usually not the real amount of physical memory that can be allocated by an application; the system kernel, services as well as other processes also take up memory, filesystem caches etc. The benchmark operates by accessing memory until a substantial delay occurs (determined by a configurable delay threshold). The process is performed repeadedly and the maximum amount of memory allocated without incurring swapping is returned.

EPStream is a simple adaptation of the C implementation of the well-known STREAM memory benchmark [4] so that it runs simultaneously on a set of CPU's using MPI. The STREAM benchmark measures the sustainable local memory bandwidth (MB/s). It is a simple synthetic benchmark program and in addition to providing memory bandwidth it also gives an idea of the corresponding computation rate for simple vector kernels. The STREAM benchmark measures bandwidth while performing four operations: *copy*, *scale*, *sum* and *triad*. Table 4.3 outlines each operation. In the case of SMP machines, such as clusters of dual-CPU or quad-CPU machines, this benchmark can provide useful information when run in either of two modes: **1)** One process per SMP node (e.g. 1 process on a dual node) and **2)** One process per CPU (e.g. 4 processes on a quad node). This information can be crucial since the memory bandwidth available may be shared between more than one CPU's.[1]

Table 4.3: The STREAM benchmark Operations.

Name	Operation
copy	a[i]=b[i]
scale	a[i]=q*b[i]
sum	a[i]=b[i]+c[i]
triad	a[i]=b[i]+q*c[i]

The typical execution time of EPStream is around 10 seconds.

CacheBench is a benchmark aiming at evaluating the performance of the local memory hierarchy of a machine [5]. The benchmark is implemented in C and performs a set of operations

[1] An example of this is that Dual Intel "Xeon" nodes typically have a single memory controller per SMP machine, while AMD "Opteron" SMP machines typically have a separate controller for each CPU and can thus achieve a highter memory bandwidth.

– *read*, *write*, *read/modify/write*, *memset()* and *memcopy()* – varying the underlying array size thus exposing the performance of the (potentially multi-level) cache. For example, a knee can be observed at the different cache sizes when the results are plotted on a graph (see Figure **??**). An instance of CacheBench is invoked on each CPU of the resource under study and results are reported independently for each CPU. The operations at each size run for a configurable amount of time (default is 2 seconds) and the average bandwidth (MB/s) is reported. Table 4.4 outlines each operation. While this benchmark produces a similar metric to the STREAM benchmark, it runs for a longer time since it takes measurements at different memory sizes (execution times are typically in the order of 5 minutes). The time it takes to finish depends strictly on the input parameters. It also focuses on the performance of memory *caches* providing insight to the different levels of cache available to the CPU's. Since this benchmark runs considerably longer than the EPStream benchmark, it would make sense to invoke it when need arises (i.e. the user is explicitly interested in cache performance, and the sustained memory bandwidth produced by EPStream is not adequate).

Table 4.4: The CacheBench Operations.

Name	Operation
read	register=m[i]
write	m[i]=register++
read/write	m[i]=m[i]++
memset()	(system call)
memcopy()	(system call)

MPPTest is a benchmark that tests MPI communication speeds by various ways and provides a variety of options for a detailed performance analysis [3]. MPPtest is platform and MPI-implementation independent and can therefore be used with any MPI implementation. MPPtest aims to make reproducible measurements of MPI performance and results are claimed by the MPPTest creators to be reproducible since the reported measurements are the minimum of several runs. For the purpose of resource characterization it is desirable to have a focused set of measurements and to this end, only three types of measurement are performed: (i) Latency, (ii) point-to-point bandwidth and (iii) bisection bandwidth. "Bisection bandwidth" refers to an *all-to-all* measurement of bandwidth in contrast to the *point-to-point* measurement where only two processes communicate at any time. The typical execution time is in the order of minutes (depending on the measurement detail) and results are calculated using wall-clock time.

The b_eff_io benchmark is included in order to evaluate the shared I/O performance of (shared) storage at a resource (site). This benchmark is used "to achieve a characteristic average number for the I/O bandwidth achievable with parallel MPI-I/O applications" [6]. *B_eff_io* produces a metric given in Megabytes per second, which represents the average obtained by performing several storage access patterns. Access patterns include: (i)Multiple processes read/write data scattered in a file; (ii) Multiple processes read/write adjacent data; (iii) Multiple processes read/write data in separate files; and (iv) each of the multiple processes accesses data in a different segment of a segmented file (a detailed description of the access patterns can be found in [6]). Given that shared disk I/O is usually performed over the network, the results obtained by

this benchmark may be correlated with the results obtained by the MPPTest benchmark. The benchmark is implemented in C.

4.1.3 Application benchmarks

The main kernels for the blood-flow simulation application (WP1.1) and the pollution simulation (WP1.4) were instrumented and made available as kernels.

BStream kernel benchmark

This kernel was instrumented to measure iteration times, communication time and completion time. (Please see use-case 2 in the first section)

VERTLQ kernel benchmark

This kernel was instrumented to measure completion time.

Bibliography

[1] Al Aburto. flops.c version 2.0. ftp://ftp.nosc.mil/pub/aburto, 1992.

[2] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[3] William Gropp and Ewing L. Lusk. Reproducible measurements of MPI performance characteristics. In *PVM/MPI*, pages 11–18, 1999.

[4] John D. McCalpin. *Sustainable Memory Bandwidth in Current High Performance Computers*. Advanced Systems Division Silicon Graphics, Inc., October 1995.

[5] Phillip J. Mucci and Kevin London. The cachebench report, 1998.

[6] Rolf Rabenseifner, Alice E. Koniges, Jean-Pierre Prost, and Richard Hedges. The parallel effective i/o bandwidth benchmark: b_eff_io. Message Passing Interface Developer's and User's Conference (MPIDC), March 2000.

[7] P.M.A. Sloot, A. Tirado-Ramos, A.G. Hoekstra, and M. Bubak. An interactive grid environment for non-invasive vascular reconstruction. In *2nd International Workshop on Biomedical Computations on the Grid (BioGrid'04), in conjunction with Fourth IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2004)*, Chicago, Illinois, USA, April 2004. IEEE. CD-ROM IEEE Catalog # 04EX836C.

[8] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.