Design and Implementation of the eNANOS Scheduler

Ivan Rodero and Julita Corbalan

Computer Architecture Department
Technical University of Catalonia (UPC)
Jordi Girona 1-3, Modul D6, 08034 Barcelona, Spain
{irodero,juli}@ac.upc.edu

Abstract. In his technical report we address the problem of scheduling high performance parallel applications on clusters of SMPs architectures. In particular, we focus on the cluster scheduling scenario of the eNANOS architecture. We describe the main functionalities of the eNANOS Scheduler which is responsible for performing the job scheduling based on co-allocation techniques and the coordination with processor scheduling tools on top of the Loadleveler queueing system.

Key words: Cluster Scheduling, Coordination, Loadleveler, eNANOS

1 Introduction

One of the main challenges in High Performance Computing (HPC) is the efficient scheduling of parallel applications. In the eNANOS project [1] we propose a coordinated scheduling architecture for the execution of HPC applications, from the grid level to the processor scheduling level, being complementary to the existing approaches.

The eNANOS project is based on the idea of having a good low level support for performing a good high level scheduling. Taking into account these ideas: it performs a fine grain control between the scheduling levels, and a dynamic CPU mapping to improve the system performance. It uses detailed information regarding the current scheduling details and the performance to improve future scheduling decisions and, based on this accurate information, it performs an efficient scheduling (in terms of slowdown) based on performance prediction. Since this research area is very wide, this project is targeted to HPC applications (OpenMP, MPI and MPI+OpenMP) executed on distributed architectures composed of parallel machines with shared-memory architectures (SMP and CC-NUMA). In this execution framework the applications and user-level system tools cooperate for an efficient job execution and resource usage. Efficiency is considered from the point of view of the application, matching as much as possible its requirements, and from the point of view of he system, distributing the resources in the most convenient way for optimizing the global performance. Fig. 1 shows the overall architecture of the eNANOS architecture.

In this technical report we present the eNANOS Scheduler that is responsible for the cluster scheduling in the eNANOS architecture. It is implemented as a external scheduler on top of the Loadleveler queueing system. It implements co-allocation scheduling strategies based on the coordination with processor scheduling tools (that performs dynamic CPU mapping). Our scheduling strategy will be described in a separate paper.

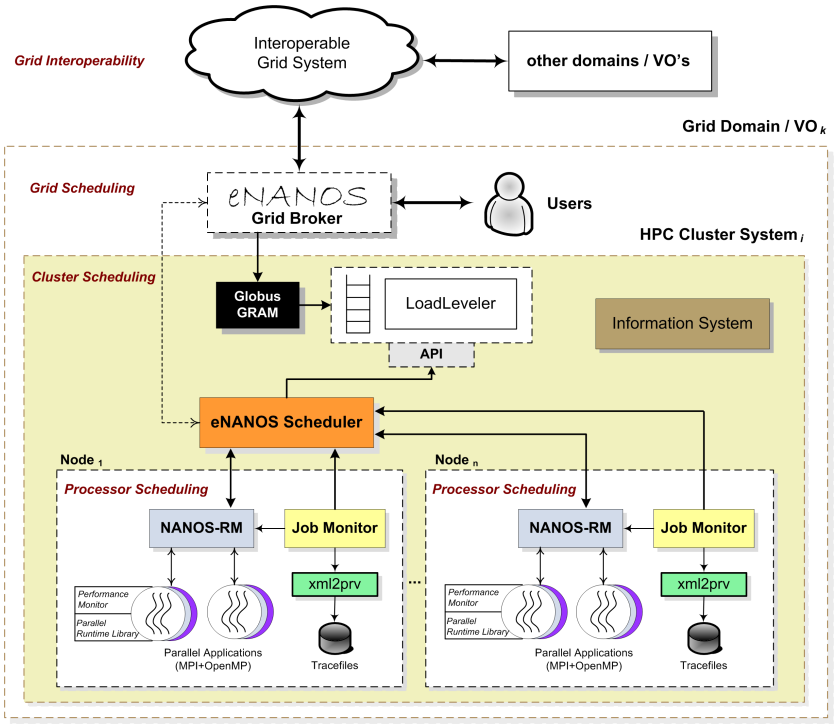


Fig. 1: eNANOS overall architecture

2 Related Work

Usually, the HPC systems are managed by queueing systems that are in charge of managing the submitted jobs and performing the resource management. Some examples of them are: Loadleveler [2], PBS (Portable Batch System) [3], LSF (Load Sharing Facility) [4] or SLURM (Simple Linux Utility for Resource Management) [5].

Moreover, in a cluster scenario a job scheduler is responsible for performing the job scheduling among the local resources. It can be provided by the queuing system itself (such as the EASY backfilling Loadleveler scheduler [6]), or deployed as a external component on top of a given queueing system (for example MOAB [7] or MAUI [8] on top of the PBS or SLURM queueing systems). Some of them are focused on both cluster and grid systems such as the OAR Scheduler [9]. They implement several scheduling policies and support some mechanisms such as the advanced reservations.

Our first idea was extending the MAUI scheduler (which is open source) to support our required functionalities. However, we decided implementing our own scheduler because we considered that the efforts required for extending an existing scheduler were similar to implementing a new one. Moreover, the implementation of a new scheduler allowed us to have full control of the whole scheduling process and to implement interfaces for communicating with other scheduling layers more easily.

3 The Overall Architecture

The management of the whole cluster is done by the Loadleveler queuing system with the eNANOS Scheduler as its external job scheduler. Thus, the scheduler is the responsible for executing and managing the queued jobs. As is usual in job schedulers, it works in an iterative way with a polling time interval. In each iteration the scheduler performs the following main actions:

- Searches jobs from the queuing system.
- Updates the system information.
- Evaluates the current scheduling policy.
- Executes the selected jobs in the appropriate nodes.
- Monitors the job execution.

The overall architecture of the eNANOS Scheduler is shown in Fig. 2. The most important component is the main thread which is in charge of creating the rest of working threads. Moreover, it is the responsible for the interaction with the scheduling engine and with the Loadleveler queuing system through the *LL Dispatcher*.

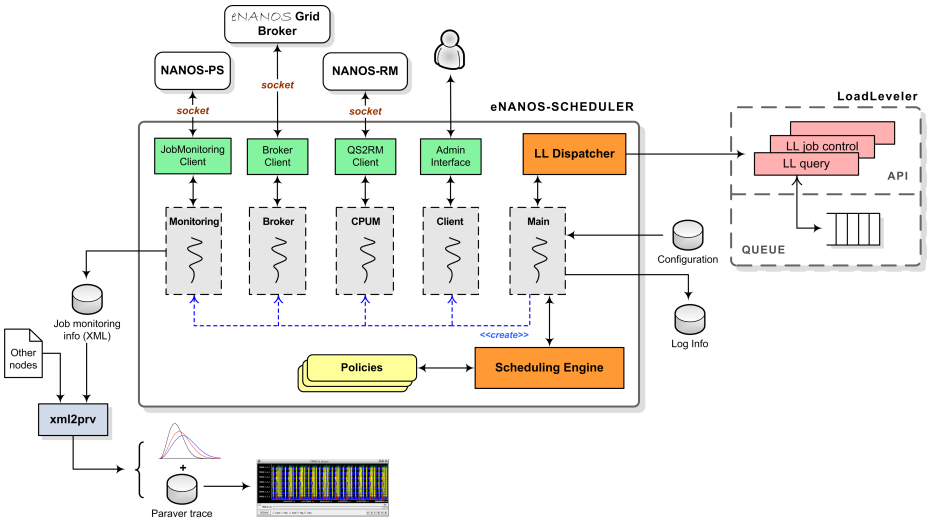


Fig. 2: eNANOS Scheduler overall architecture

Although the main thread performs the basic scheduling tasks that were listed previously, the evaluation of the scheduling policies is done externally by the *Scheduling Engine*. It implements scheduling policies based on First Come First Served (FCFS) but guided by the behavior and performance information of the running applications that is obtained in run time from the processor scheduler. The details of the scheduling strategy will be described in a separate paper.

Once a job has been scheduled, we use the *LL Dispatcher* to execute it in the system under the Loadleveler domain. For this purpose, we use the Loadleveler API that includes a very complete set of functionalities (for example *ll_get_objs* + *ll_get_data* for querying jobs or *ll_start_job* for job execution).

Internally, the scheduler has its own model for describing and managing the resources and jobs (including, for example, the job history). The client thread communicates with the final user through the *Admin Interface*. The purpose of this interface is to provide a point of access to the users to adjust scheduler parameters or to update some configuration issues in run time.

4 Integration with the Processor Scheduler

As we discussed in [1] and it is also shown in Fig. 1, in each computational node a NANOS-RM is in charge of its applications management and the eNANOS scheduler centralizes the information provided by each NANOS-RM. To perform this interaction, the CPUM thread (named CPUM from *CPU Manager*) makes polling calls to the NANOS-RM of each node in short interval times to update the availability and status of the resources. This data is obtained through a well defined API between the levels that we have implemented to both facilitate the scheduling decisions (for instance specific allocations) and getting detailed information in run time (for instance the real performance reached by a job or the allowed multiprogramming level of a node). The idea is sharing the required information for improving the whole system performance without penalizing the applications performance independently. This API is shown below in Listing 1.1.

```
1  int  ConnectQS()
2  void DisconnectQS()
3  int  NewInfo(void)
4  int  CpusNode(void)
5  int  NodeLoad(void)
6  int  MaxJobs(void)
```

Listing 1.1: API between processor and cluster schedulers

ConnectQS connects to the NANOS-RM. It returns 0 if the connection is stablished successfully and -1 if there is any error. *DisconnectQS* disconnects the job scheduler from the NANOS-RM. *NewInfo* blocks the calling thread until the job scheduler provides new information. *CpusNode* returns the node number of CPUs, *NodeLoad* the load of the node, and *MaxJobs* the maximum number of jobs that are allowed to run on the node simultaneously.

Although the design and implementation of the eNANOS scheduler is oriented for using the low level support, we made several modifications to the NANOS-RM to perform the integration. On one hand, we have implemented the scheduling by dividing

the process into different phases that have dependences between them. On the other hand, we have performed several changes and improvements of the NANOS-RM. In particular, we have extended its functionality set, we have updated its communication mechanisms and interfaces(from shared memory mechanisms to sockets), the protocol to interchange data, and the kind of data that it provides.

In the first version of the NANOS-RM the communication between the components was based on shared memory mechanisms because it was designed for single node systems. Since our objective requires the scheduling in multiple nodes systems with heterogeneous configurations, we have updated the communication interfaces using sockets. In order to provide the NANOS-RM data in a simple and robust way, we have implemented a protocol divided into two phases. After having a successful connection, the first one *(NewInfo)* blocks the calling thread until the NANOS-RM has new information to provide. The second one *(CpusNode, NodeLoad, MaxJobs)* returns the data from buffers. The new data that the NANOS-RM provides includes useful information for connections (such as hostname or communication port) or information concerning the full resource node (such as CPUs, resource manager index or running applications). Some of this new information is useful to estimate if a node is able to execute a new application without degrading the system performance (newApplAllowed). Finally, we also have modified the NANOS-RM to provide the eNANOS scheduler some control interfaces such as forcing the multiprogramming level (number of concurrent processes running simultaneously in a node).

5 Coordination with the Grid Level

The Broker thread purpose is to provide detailed information concerning the local jobs and the scheduler status to the eNANOS Grid Broker. Although we have implemented a mechanism to export job progress and performance information to the grid broker [10], we are still using a direct communication way between the local job scheduler and grid broker. Moreover, the broker thread is prepared to receive direct commands from the grid broker such as those related to the quality of service required by the job. However, the grid scheduling layer of the eNANOS architecture is not in the target of this technical report.

6 Monitoring Functionality

The monitoring functionality is in charge of providing the required information to analyze the system behavior and the statistical information for evaluating the workloads execution in a simple way. Thus, the Job Monitoring thread collects fine-grain information concerning the running jobs directly from the NANOS-RMs using their *NANOS-PS* interface (details concerning the processes, threads and CPUs). With this information and the scheduler internal job events, the monitoring thread generates a set of XML files that include the full job life cycle details. We have selected the XML language because it is extensible, standard, and easy to manage. The main attributes of the monitoring files are listed below:

- Name of the workload that contains the job.
- Job identifier at grid level (JobID assigned by the eNANOS Broker in case that the job provides from the grid layer).
- Job identifier at local level (the StepID assigned by Loadleveler).
- Job topology (number of MPI processes and OpenMP threads).
- A set of local events (concerning the scheduler and the queuing system: *ll_submission*, *ll_start*, *ll_finish*, etc.).
- A set of processes events (*pids*, *threads*, *CPUs*, etc.).

Each of the job events includes its time stamp, the event identification, a human readable description, and the required information concerning the event for further analysis. The set of events allows us to follow the jobs life cycle with precision of seconds (scale of our time stamps). Since each job are completely identified, a set of these XML files describes the whole workload. Currently, the job monitoring files include the information concerning the two lower layers of our architecture (local and processes events). However, the XML schema includes an element for grid events but for the moment it is always empty. The full XML schema is shown in Fig. 3.

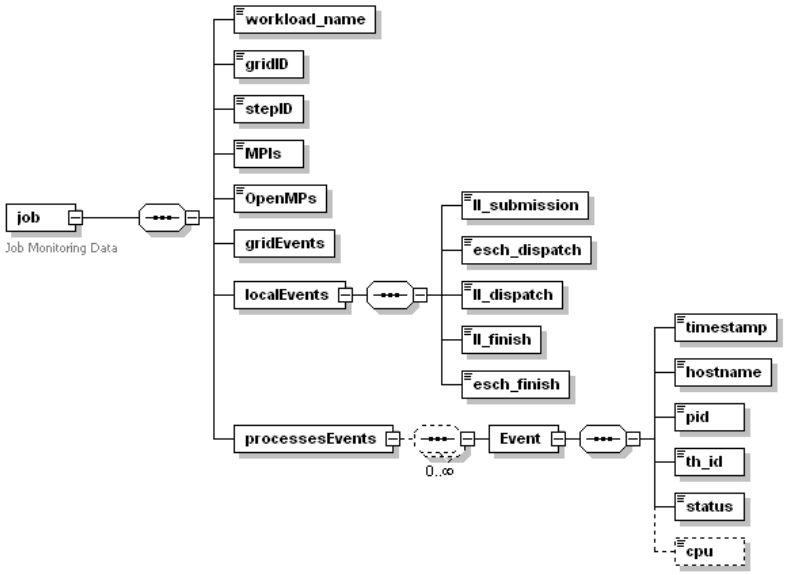


Fig. 3: Job Monitoring XML schema

In order to visualize and analyze the workloads we have implemented an external tool (called *"xml2prv"*). Given a set of job monitoring XML files describing a workload, it generates a text file with some statistics and a trace file that can be visualized and analyzed with the Paraver visualization and analysis suite [11]. Firstly, xml2prv combines the information of the input XML files in one file that contains all the workload

data. Using the tasks and events details of the new file it generates the paraver trace. It is worth noting that these traces include data from job that can be executed in multiple nodes simultaneously (co-allocated). We use two kinds of records (that corresponds to each paraver trace row) for generating the traces: event records and state records. The event records correspond directly to the local events, and the state records define the job status during a particular time interval. We generate these last records using the time stamps and the type of the events by pairs. Furthermore, when the trace is being generated *xml2prv* calculates some statistics that will be given in a text file (named, for example *"tracename.stats"*). The statistic data includes the workload number of jobs, and the following metrics for each job and in average: execution time, response time, waiting time, and slowdown.

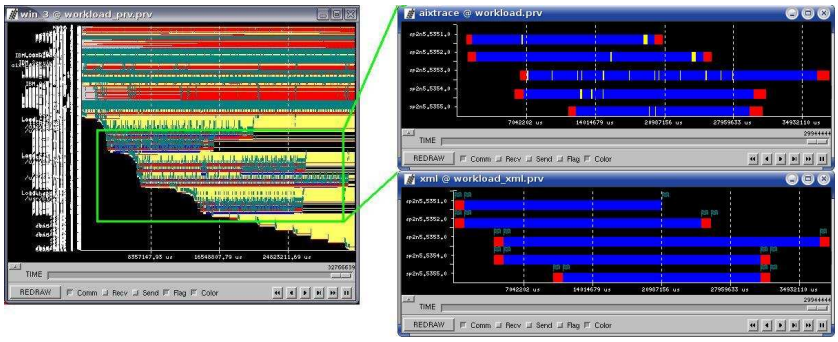


Fig. 4: Job execution traces: AIXtrace vs. our monitoring functionality

As well as the quantitative data included in the statistical output file, we are very interested on studying the system behavior to understand the reasons for obtaining the numeric results. To achieve this goal, our first approach for job executions on individual nodes was obtaining a complete trace of the job execution using the IBM AIXtrace functionality [12]. AIXtrace returns very exhaustive data in a binary trace that can be converted to Paraver format using the *"AIXtrace2prv"* tool. The main problem of AIXtrace is the huge size of the traces that generates. In Fig. 4 we show the appearance of some traces. In these traces each row is a job, the blue color represents running status regions and the red ones represent idle regions. On the left there is a trace obtained through AIXtrace and converted to the paraver format. On the right there is a portion of the same trace but the upper one is the cleared AIXtrace and the lowest one is the trace obtained directly with our monitoring system for the same job execution. We can appreciate that both traces look very similar except a few differences that come from the timestamps granularity and the trace scales.

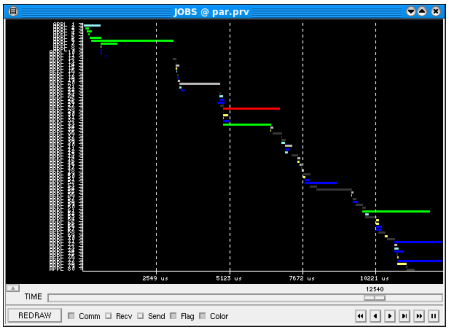
One of the main reasons for obtaining the traces with our monitoring system is that we can obtain quite exhaustive traces with a reduced size. In Table 1 we present a comparison of the different trace formats size for workloads of different durations. The first one (binary trace) is the binary trace obtained with AIXtrace, the second one

is the same binary trace but filtering the unnecessary data and converted to paraver format, and the last one is the trace obtained with our monitoring system and *xml2prv*. Moreover, since we want to monitorize different scheduling levels (from processors to the grid) it is very difficult to combine different traces obtained with AIXtrace from different nodes. We also have taken into account that XML is standard, very extended, and easy to read, to manage, and to convert to other formats.

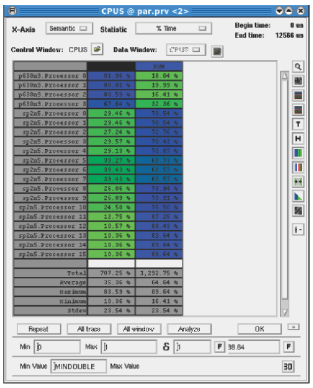
execution time	binary trace	binary filtered	paraver (from XML)
10 s	12 MB	8 MB	1,5 KB
1 min	150 MB	132 MB	40 KB
10 min	670 MB	640 MB	300 KB

Table1: Comparison of differnt trace format sizes

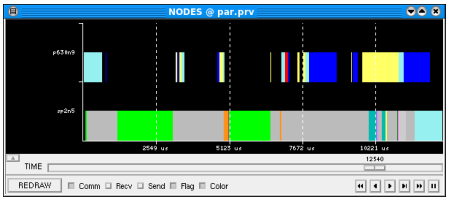
Using our full monitoring functionalities we can analyze full workloads being executed in different nodes simultaneously. In Fig. 5 is shown the workload execution as an example. Fig. 5a shows a trace with the different jobs of the workload. The rows represent the different jobs that are printed in different colors. With this view of the



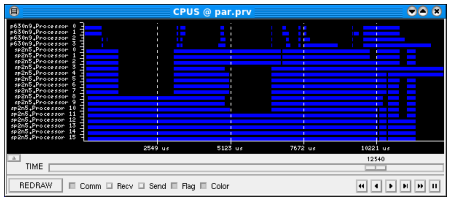
(a) Job view of the workload execution



(b) Analyzer 2D of CPU usage



(c) Nodes utilization view of the workload



(d) CPU view of the workload execution

Fig. 5: Example of a workload analysis with Paraver

trace we can analyze the job scheduling performed during the workload execution. In this case the workload is composed by 80 jobs and the scheduling policy is able to allocate several jobs at the same time but following a FCFS policy. All the time there are jobs running but when long jobs are running the waiting times are bigger, probably because these long jobs or the following ones require several CPUs. In Fig. 5c is shown the nodes utilization view of the workload execution. Each row represents a node and each color depends on the number of used CPUs during the regions. We can see that the second node (*sp2n5*) almost all the time is used while the first one (*p630n9*) is not used during some regions. Since Fig. 5d shows the CPU view, each row represents a different CPU. When a job forces the others to wait for a long time, during these time intervals there are empty regions in the trace because some CPUs are idle. In Fig. 5b we can find the numerical analysis of the CPU usage obtained with the *"Analyzer2D"* functionality of Paraver. This functionality returns the usage percentage of each CPU individually (second column), and other information such as the accumulated, average, maximum, minimum, and the standard deviation values. The colors encode the values: in this example light green is the minimum and dark blue is the maximum. We can appreciate that the first CPUs (first node) are used much less that the second node CPUs. As well as applying the 2D analysis to whole system, it is also possible to apply it to a particular node or CPU selection. Moreover, the workload can be analyzed with the processes or the threads views, or with other Analyzer2D semantic configurations.

Since the traces include information from different nodes and jobs, it is important to identify each entity in detail. We consider the identifier (ID) from the grid to the thread level; therefore, the ID of a thread (which is the smallest involved entity) includes: grid ID (e.g. *"1@1128247604981"*), Loadleveler StepID (e.g. *"p630n9.3002.0"*), process ID, and thread ID. An example of a complete thread ID can be the following string: *"1@1128247604981.p630n9.3002.0.3.1"*.

7 Implementation Issues

Since the scheduling system is designed for multiple nodes, the eNANOS Scheduler threads use socket connections to interact with the external system components. Moreover, it allows us to extend the system for a multi-cluster scenario that can be interconnected though LAN and WAN connections. For the socket connections implementation we use two interfaces: one for the server side and another one for the client side (named, for example: *QS2RM_server* and *QS2RM_client* respectively). The eNANOS Scheduler configuration is specified through a configuration file that includes, for example, the logging detail level, output log file, the initial multiprogramming level, or the current scheduling policy. Moreover, the scheduler provides logging information concerning the jobs and system status, performed actions, and some general details (for example the number of queued jobs, the reason of an error, or the allocation for a running job). The implementation of the eNANOS Scheduler has been done mainly in C++ but also in C as the majority of the components in our local scheduling scenario.

8 Conclusions and Future Work

In this technical report we have presented the eNANOS Scheduler and its main functionalities. Our main lines of future work are focused on evaluating the scheduling strategy that the eNANOS Scheduler implements as well as the coordination with the grid scheduling layer.

Acknowedgements

This paper has been supported by the Spanish Ministry of Science and Education under contract TIN200760625C0201.

References

1. I. Rodero, F. Guim, J. Corbalan, and J. Labarta, “enanos: Coordinated scheduling in grid environments,” *Parallel Computing (ParCo)*, pp. 81–88, 2005.
2. S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. Skovira, *Workload Management with LoadLeveler*, November 2001, iBM Redbooks.
3. (1998) Portable batch system (pbs). [Online]. Available: http://www.nas.nasa.gov/Software/PBS
4. (2008) Load sharing facility (lsf). [Online]. Available: http://www.platform.com/Products/platform-lsf
5. A. Yoo, M. Jette, and M. Grondona, “Slurm: Simple linux utility for resource management,” *Job Scheduling Strategies for Parallel Processing*, pp. 44–60, 2003, lNCS 2862.
6. J. Skovira, W. Chan, and H. Zhon, “The easy - loadleveler api project,” *Job Scheduling Strategies for Parallel Processing*, pp. 41–47, 1996, lNCS 1162.
7. (2008) Moab cluster suite. [Online]. Available: http://www.clusterresources.com/pages/products/moab-cluster-suite.php
8. (2008) Maui cluster scheduler. [Online]. Available: http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php
9. (2008) Oar scheduler website. [Online]. Available: http://oar.imag.fr/
10. I. Rodero, F. Guim, J. Corbalan, and J. Labarta, “Design and implementation of a general-purpose api of progress and performance indicators,” *Parallel Computing (ParCo)*, pp. 501–508, 2007.
11. V. Pillet, J. Labarta, T. Cortes, and S. Girona, “PARAVER: A tool to visualise and analyze parallel code,” *Proceedings of WoTUG-18: Transputer and occam Developments*, vol. 44, pp. 17–31, 1995.
12. (2008) Ibm aix trace facility website. [Online]. Available: http://publib.boulder.ibm.com/infocenter/pseries/index.jsp