Harvesting Large-Scale Grids for Software Resources

Asterios Katsifodimos [1], George Pallis [2], Marios D. Dikaiakos [3]

Computer Science Department, University of Cyprus, Nicosia, CY 1678, Cyprus
[1]asteriosk@cs.ucy.ac.cy, [2]gpallis@cs.ucy.ac.cy, [3]mdd@cs.ucy.ac.cy

***Abstract*— Grid infrastructures are in operation around the world, federating an impressive collection of computational resources and a wide variety of application software. In this context, it is important to establish advanced software discovery services that could help end-users locate software components suitable to their needs. In this paper, we present the design, architecture and implementation of an open-source keyword-based paradigm for the search of software resources in Grid infrastructures, called *Minersoft*. A key goal of Minersoft is to annotate automatically all the software resources with keyword-rich metadata. Using advanced Information Retrieval techniques, we locate software resources with respect to users queries. Experiments were conducted in EGEE, one of the largest Grid production services currently in operation. Results showed that Minersoft successfully crawled 12.3 million valid files (620 GB size) and sustained, in most sites, high crawling rates.**

I. Introduction

Currently, a number of large-scale Grid infrastructures are in operation around the world, federating an impressive collection of computational resources and a wide variety of application software [2], [3]. These infrastructures provide production-quality computing and storage services to thousands of users that belong to a wide range of scientific and business communities. In the context of large-scale Grids, it is important to establish advanced software discovery services that can help end-users locate software components that are suitable for their computational needs. Earlier studies have shown that the difficulty in discovering software components was one of the key inhibitors for the adoption of component technologies and software reuse [7]. Therefore, the provision of a user-friendly tool to search for software is expected to expand the base of Grid users substantially.

To motivate the importance of such a tool, let us consider a biologist who is searching for drug discovery software deployed on a Grid infrastructure. Although informative tags about installed software can be published through Grid information systems, Grid system administrators seldom follow such a practice [10]. Consequently, there are no tools or published information that would support searches of this kind. Envisioning the existence of a Grid software search engine, a biologist would submit a query to the search engine using some keywords (e.g. “docking proteins biology,” “drug discovery,” or “autodock”). In response to this query, the engine would return a list of software matching the query’s keywords, along with Grid sites where this software could be found. Thus, the user would be able to identify the sites hosting an application suitable to her needs, and would accordingly prepare and submit jobs to these sites.

Adopting a keyword-based paradigm for the search of software seems like an obvious choice, given that keyword-based search is currently the dominant paradigm for information discovery [15]. Keyword-based search traditionally relies on Information Retrieval (IR) algorithms that explore the occurrence of words in documents. However, software components usually come with few or no free-text descriptors. Consequently, the problem cannot be addressed with traditional IR approaches. Instead, we need new techniques that will discover software-related resources, extract structure and meaning from those resources, and discover implicit relationships among them. Also, we need to develop methods for effective querying and for deriving insight from query results. The provision of keyword-based search over large, distributed collections of unstructured data has been identified among the main open research challenges in data management that are expected to bring a high impact in the future [4]. Searching for software falls under this general problem, because file systems treat software resources as unstructured data and maintain very little if any metadata about installed software.

To address the software search challenge, we developed the *Minersoft* Grid harvesting system. Minersoft visits Grid sites, crawls their file systems, identifies software resources of interest (software, libraries, documentation), assigns type information to these resources, and discovers implicit associations between software and documentation files. Subsequently, it creates an inverted index of software resources that is used to support keyword-based searches. To achieve these tasks, Minersoft invokes file-system utilities and object code analyzers, implements heuristics for file-type identification and filename normalization, and performs document analysis algorithms on software documentation files and source-code comments. The major contributions of this article are the following:

- We present the design, the architecture, and implementation of the Minersoft harvester.
- We provide a study about the installed software resources in EGEE [2], one of the largest Grid production services currently in operation.
- We conduct an experimental evaluation of Minersoft, on a real, large-scale Grid testbed, exploring performance issues of the proposed scheme. In particular, we use Minersoft to harvest several sites of EGEE Grid.

The remainder of this paper is organized as follows: Section 2 presents an overview of related work and the Minersoft challenges are discussed. Section 3 presents a description of the Minersoft harvesting. Section 4 describes the architecture of Minersoft. In Section 5, we present an experimental assessment of our work. Section 6 discusses the open issues for harvesting the Grid. We conclude in Section 7.

II. Related Work

A number of alternative approaches have been proposed for addressing software-component retrieval. One of the key distinguishing traits of these approaches is the corpus upon which the search is conducted:

Searching in a software repository: The GURU system by Maarek et al [16] is one of the first efforts to establish a keyword-based paradigm for the retrieval of source code installed on standalone computers. Using standard IR techniques, GURU exploits the comments of source code and documentation files. SEC+ is a more recent keyword-based paradigm for discovering software components [13]; an ontology is used to describe the properties of software components. Maracatu [21] is another search engine for software components which makes use of folksonomy concepts. Folksonomy is a cooperative classification scheme where the users assign keywords (called tags) to software resources. A similar approach, based on file tagging, was presented in [14]; there, the authors proposed a scheme for searching and navigating huge file systems using faceted metadata, i.e., sets of key-value pairs associated with each file. The keys, which are called "facets", allow the values to be grouped into semantically meaningful ways. Several research efforts have also focused on the problem of identifying automatically the associations between source-code and software-description documents [6], [18]. Finally, another approach is to use semantic file-systems [5], [11].

Searching in the Web: In [12], authors described an approach for harvesting software components from the Web. The basic idea is to use the Web as the underlying repository, and to utilize standard search engines, such as Google, as the means of discovering appropriate software assets. In the World-Wide Web context, harvesting is a well-studied research problem. In principle, a set of seed URLs is used as the basis for the search, and those pages are fetched one by one. Each hyperlink within those pages is extracted and, if it has not been explored yet, appended to a queue of pending pages. Eventually, all pages reachable from the seed set will have been accessed, and the process can be repeated. Other researchers have crawled through Internet publicly available CVS repositories to build their own source code search engines (e.g., SPARS-J) [19].

Searching in the Grid: In the Grid context, a recent work has proposed a component search service, called GRIDLE [20]; this scheme allows users to locate the source components they need for building a Grid application. Users specify a high-level workflow plan including the requirements of each component. Then, GRIDLE presents a ranked list of components that match partially or totally user requirements.

Minersoft is different from all the above works in a number of key aspects:

- Minersoft supports searching not only for source codes but also for executables and libraries stored in binary format;
- Minersoft does not presume that file-systems maintain metadata (tags etc) to support software search; instead, the Minersoft harvester generates such metadata automatically by invoking standard file-system utilities and tools and by exploiting the hierarchical organization of file-systems;
- Minersoft introduces the concept of the Software Graph, a weighted, undirected, typed graph. The Software Graph is used to represent software resources and associations under a single data structure, amenable to further processing.
- Minersoft addresses a number of additional implementation challenges that are specific to federated infrastructures: i) Software management is a decentralized activity; different sites may follow different policies about software installation, directory naming etc. Also, software entities on a Grid site often come in a wide variety of packaging configurations and formats. Therefore, solutions that are language-specific or tailored to some specific software-component architecture are not applicable. ii) Harvesting the sites of a Grid infrastructure is a demanding task for computational, storage, and communication resources. Also, most Grid systems do not support interactive computation. Therefore, software harvesting needs to be performed in a distributed, non-interactive manner. iii) The users of a Grid infrastructure do not have direct access to local Grid sites. Therefore, a harvester has to be either part of middleware services (something that would require the intervention to the middleware) or to be submitted for execution as a normal job, through the middleware. In the Minersoft architecture and implementation we adopt the latter approach, which facilitates the deployment of the system on different Grid infrastructures.

III. Minersoft Harvesting

A software package that is installed on a Linux/Unix machine, typically comprises one or more software-related files from a variety of possible categories: *executables* (binaries or scripts), software *libraries*, *source code* written in some programming language, various *configuration files* required for the compilation and/or installation of code (e.g. makefiles), and various unstructured or semi-structured *software-description documents*, which provide human-readable information about the software, its installation, operation, and maintenance (manuals, readme files, etc). The Minersoft harvester seeks to identify software resources of interest (software, libraries, documentation), assign type information to these resources, discover implicit associations between software and documentation files, and capture this information in data structures amenable to indexing and search.

A. Software Graph Definition

To represent the software-related files found in a file system and the associations between them, we introduce the concept of the *Software Graph*, a weighted, undirected metadata-rich, typed graph $G(V, E)$. The node-set V of the graph comprises: i) nodes representing software-related files found on the file-system of a computing node (*file-vertices*) and ii) nodes representing directories of the file-system (*directory-vertices*). The edges E of the graph represent structural and semantic associations between vertices. Structural associations correspond to relationships between file-system resources (files and directories). These relationships are derived from file-system structure according to various conventions (e.g., about the location and naming of documentation files) or from configuration files that describe the structuring of software packages (RPMs, tar files, etc).

The Software Graph is "typed" because its vertices and edges are assigned to different types (classes). Each vertex v of the Software Graph $G(V, E)$ is annotated with a number of associated metadata attributes, describing its content and context:

- $name(v)$ is the normalized name of the file represented by v.
- $type(v)$ denotes the type of v (binary executable, source code, directory, etc).
- $sites(v)$ denotes the Grid sites where file v is located.
- $path(v)$ is a set of terms derived from the path-name of file v in $site(v)$'s file system.
- $zone_l(v), l = 1, \ldots, z_v$ is a set of zones assigned to vertex v. Each zone contains terms extracted either from the content of v or from the content of a file associated to v.

Each edge e of the graph has two attributes: $type(e)$ denotes the association represented by e and $w(e)$ is a real-valued weight $(0 \leq w \leq 1)$ expressing the degree of correlation between the edge's vertices. An association between two graph-nodes can be structural (e.g., directory containment, software library membership) or semantic (text similarity).

B. Minersoft Algorithm

A key responsibility of the Minersoft harvester is to construct a Software Graph, starting from the contents of the file system found on a Grid site. Then, given the Software Graph, Minersoft builds an inverted index for software resources installed on the site. The Minersoft harvester implements an algorithm comprising a number of steps described below. An in-depth discussion regarding the Software Graph construction algorithm is beyond the scope of this paper.

FST construction: Initially, Minersoft scans the file-system of a Grid node and creates a *File-System Tree* (FST) data structure. The internal vertices of the tree correspond to directories of the file-system; its leaves correspond to files. Edges represent containment relationships between directories and sub-directories or files. During the scan, Minersoft ignores a *stop list* of files and directories that do not contain information of interest to software search (e.g., `/tmp, /etc`).

Classification and pruning: Names and pathnames play an important role in file classification and in the discovery of associations between files. Accordingly, Minersoft normalizes filenames and pathnames of FST vertices, by identifying and removing suffixes and prefixes. The normalized names are stored as metadata annotations inside the zone-set of FST nodes. Subsequently, Minersoft applies a combination of system utilities and heuristics to classify each FST file-vertex into a variety of possible categories that a software package comprises. Finally, Minersoft prunes all FST leaves found to be irrelevant to software search, dropping also all internal FST nodes that are left with no descendants. This step results to a pruned version of the FST that contains only software-related file-vertices and the corresponding directory-vertices.

Structural dependency mining: Subsequently, Minersoft searches for "structural" relationships between software-related files (leaves of the file-system tree). Discovered relationships are inserted as edges that connect leaves of the FST, transforming the tree into a graph. Structural relationships can be identified by: i) Rules that represent expert knowledge about file-system organization, such as naming and location conventions. For instance, a set of rules link files that contain *man-pages* to the corresponding executables. *Readme* and *html* files are linked to related software files. ii) Dynamic dependencies that exist between libraries and binary executables; Binary executable files and libraries usually depend on other libraries that need to be dynamically linked during runtime. These dependencies are mined from the headers of libraries and executables and the corresponding edges are inserted in the graph; each of these edges is assigned a weight of one, as there exists a direct association of files.

The structural dependency mining step produces the first version of the Software Graph, which captures software-related files and their structural relationships. Subsequently, Minersoft seeks to enrich file-vertex annotation with additional metadata and to add more edges into the Software Graph, in order to better express semantic relationships between software-related resources.

Keyword scraping: In this step, Minersoft performs deep content analysis for each file-vertex of the Software Graph, in order to extract its descriptive keywords. This is a resource-demanding computation that requires the transfer of all file contents from disk to memory to perform content parsing, stop-word elimination, and keyword extraction. Different keyword-scraping techniques are used for different types of files: for instance, in the case of source code, we extract keywords only from the comments inside the source, since the actual code lines would create unnecessary noise without producing descriptive features. Following the keyword extraction, Minersoft applies a feature extraction technique [16] to estimate the quantity of information of individual terms and to disregard keywords of low value. Binary executable files and libraries contain strings that are used for printing out messages to the users, debugging information, logging etc. All this information can be used in order to get useful features from these files. The extracted keywords are saved in the zones of the file-vertices of the Software Graph.

Keyword flow: Software files (executables, libraries, source code) usually contain little or no free-text descriptions. Therefore, content analysis typically discovers very few keywords inside such files. To enrich the keyword sets of software-related file-vertices, Minersoft identifies edges that connect software-documentation file-vertices with software file-vertices, and copies selected keywords from the former into the zones of the latter. As we referred above, each zone has a different degree of importance in terms of describing the content of a software file. For instance, the *content zone* of a node V is more important than its *documentation zones*.

Semantic association mining: To further improve the density of the Software Graph, Minersoft calculates the cosine similarity between the graph's file-vertices. To implement this calculation, Minersoft represents each file-vertex as a weighted term-vector derived from its associated zones. File-vertices that exhibit a high cosine-similarity value are joined through an edge that denotes the existence of a semantic relationship between them.

Inverted index construction: To support full-text search for software resources, Minersoft creates an inverted index of software-related file-vertices of the Software Graph. The inverted index has a set of terms, with each term being associated to a "posting" list of pointers to the software files containing the term. The terms are extracted from the zones of Software Graph vertices.

C. Parallelization

For the efficient implementation of the Minersoft algorithm in a Grid setting, we should take advantage of various parallelization techniques in order to:

- Distribute parts of the Minersoft computation to Grid sites, in order to take advantage of the Grid computing and storage power, to reduce the communication exchange between the Minersoft system and local Grid sites, and to sustain the scalability of Minersoft with respect to the total number of Grid sites. Minersoft tasks are wrapped as Grid jobs that are submitted to Grid sites via the Grid workload-management system.
- Avoid overloading Grid sites by applying load-balancing techniques when deploying Minersoft jobs to the Grid.
- Improve the performance of Minersoft jobs by employing multi-threading to overlap local computation with I/O.
- Adapt to the policies put in place by different Grid sites regarding the number of jobs that can be accepted by their queuing systems, the total time that each of these jobs is allowed to run on a given site, etc.

More details on the parallelization of the Minersoft algorithm and its deployment on the EGEE Grid are given in the following section.

IV. Minersoft Architecture

Creating a search engine for software that can cope with the scale of emerging Grid infrastructures presents several challenges. Fast crawling technology is required to gather the Grid software resources and keep them up to date. Storage space must be used efficiently to store indices and the files themselves. The indexing system must process hundreds of gigabytes of data efficiently.

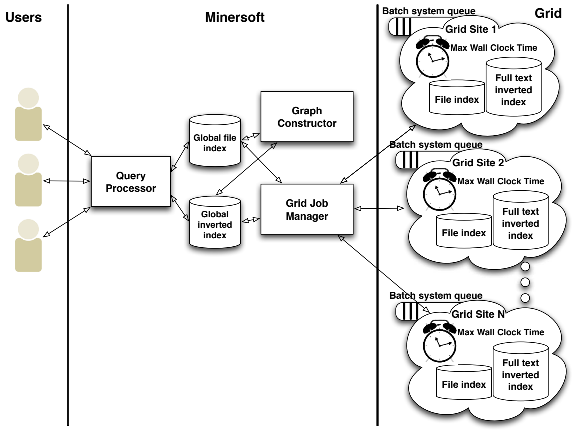

Fig. 1. Minersoft architecture.

In this section, we will provide a description of how the whole system works as depicted in Figure 1. The crawling and indexing is done by several distributed multi-threaded crawler and indexer Grid jobs, which run in parallel on different Grid sites for improved performance and efficiency. The crawler and indexer jobs process a specific number of files, called *splits*. A key component of the Minersoft architecture is the *Grid job manager*, which has the overall supervision for crawler and indexer jobs. The *graph constructor* module is responsible for constructing the Software Graph and identifying the software components. The *query processor* module is responsible for providing quality search results efficiently.

A. Crawling phase

Crawling the Grid is a challenging task that needs to address various performance, reliability and site-policy issues since it involves interaction with hundreds of Grid sites, which are beyond the control of the system. Minersoft undertakes the crawling of Grid sites in a distributed manner. The *Grid job manager* sends a number of multi-threaded crawler jobs to each Grid site. A challenge for crawler jobs is to harvest all the software resources residing within Grid sites, without exceeding the time constraints imposed by site policies: jobs which run longer than the allowed time are terminated by the sites' batch systems. The maximum wall clock time for a Grid site usually ranges between 2 and 72 hours.

Considering that a Grid site contains a large volume of files, we decompose the file system of each Grid site into a number of *splits*, where the size of each split is chosen so that the crawling can be distributed evenly and efficiently within the time constraints of the underlying site. The splits are assigned by the *Grid job manager* to crawler jobs on a continuous basis: As a site finishes with its assigned splits, it is receives more splits for processing. If a site becomes laggard, the crawler job is canceled and rescheduled to run when the site's workload is reduced. Furthermore, if the batch system queue of a Grid site is full and does not accept new jobs, the *Grid job manager* stops submitting crawler jobs to that site until the batch system becomes ready to accept more. Minersoft crawlers undertake the task of classifying software files into categories, as described earlier (e.g., binaries, libraries, documentations). The files found to be irrelevant are dropped from the FST data structure. The results of a crawler are stored at the Storage Element of each Grid site. In particular, we keep in a *metadata store file* the file-id, name, type, path, size and structural dependencies of the identified software resources. Then, the *Grid job manager* fetches the resulted *metadata store files* from all Grid sites and merges them into a *file index*. The *file index* comprises information about each software resource and is stored in Minersoft's dedicated infrastructure. From this index we can easily construct the first version of the Software Graph, which captures software-related files and their structural relationships.

B. Indexing phase

During the indexing phase, the *file index* is used by the *Grid job manager* in order to create multi-threaded indexer jobs, and to dispatch them for execution to Grid sites. The task of the indexers is to read and parse local files of interest and create a full-text inverted index. Since most sites do not allow jobs running more than 48 hours, several indexer jobs should be submitted and executed simultaneously on each site, to improve the file-processing throughput and to reduce the overall indexing time per site. Each indexer job is responsible for a specific number of files in a Grid site.

Similarly to the crawling process, the list of files of each Grid site is decomposed into a number of splits where the size of the split is chosen to ensure that indexing can be distributed evenly and efficiently within the time constraints of the system. In this context, the *Grid job manager* assigns the splits to a number of indexer jobs, taking into account the current status of the Grid site. As a site finishes indexing the assigned splits, it receives the next ones. When the indexing has been completed, each Grid site has a full-text inverted index.

Since a large percentage of duplicate software resources exists in Grid sites, Minersoft uses a *duplicate reduction policy* to preprocess the *file index* and identify the exact duplicate files. Its ultimate goal is to further improve the performance of indexing. Specifically, a file may belong to more than one Grid sites. Files with the same name, path and size are considered to be duplicates. According to our policy, a duplicate file is assigned to the Grid site which has the minimum number of assigned files that should be indexed. The key idea behind this policy is to distribute the duplicate software resources in Grid sites so as to prevent their overloading. In this context, for each Grid site, the following steps take place:

1) The *file index* is sorted in ascending order with respect to the count of sites that a file exists.
2) The files which do not have duplicates are directly assigned to the corresponding Grid site.
3) If a file belongs to more than one Grid sites, the file is assigned to the site with the minimum number of assigned files.

Finally, the *Grid job manager* fetches all the resulted local inverted-indexes and merges them into a global full-text inverted index which is stored in the Minersoft repository.

C. Harvester Implementation and Deployment

The implementation of the *Grid job manager* relies upon the Ganga system [8], which is used to create and submit jobs as well as to resubmit them in case of failure. We adopted Ganga in order to have full control of the jobs and their respective arguments and input files. The *Grid job manager* (through Ganga scripts) monitors the status of jobs after their submission and keeps a list of sites and their failure rate. If there are sites with a very high failure rate, the *Grid job manager* eventually puts them in a black list and stops submitting jobs to them.

The crawler is written in Python. The Python code scripts are put in a tar file and copied on a storage element before job submission starts. The tar file is being downloaded and untarred to the target site before the crawler execution starts. By doing that, the size of the jobs' input sandbox is reduced, thus job submission is accelerated because the Workload Management System has to deal with much less files per job. The indexer is written in Java and Bash and uses an open-source high performance, full-text index and search library (Apache Lucene [1]). In order to execute the indexer jobs, we follow the same code-deployment scenario as with crawlers.

Before the job submission starts, the *Grid job manager* has to distribute the crawling/indexing workload. This is done by creating splits for each site that Minersoft has to crawl. The input file for each split is uploaded on a storage element and registered to an LCG File Catalog (LFC). Every Job has its own ID (given as an argument during submission). A job's ID is the split number that the job will have to process. The split input is then downloaded from a storage element and used to start the processing of files. The split input is a text file containing the list of files that have to be crawled or indexed. After execution, the jobs upload their outputs on storage elements and register the output files to an LFC. The logical file names and the directories containing them in the LFC are properly named so that they implicitly state the split number and the site that they came from or going to.

D. Constructing Software Graph

The main task of *graph constructor* is to construct the Software Graph described earlier. The Software Graphs are built locally in each Grid site. For each site, this module performs structural dependency mining, keyword scrapping, keyword flow and semantic association mining tasks (described in previous section). These tasks result in enriching the *full-text inverted indexes* of Grid sites.

E. Searching Software Resources

The goal of searching is to provide quality search results efficiently. The *query processor* module receives the results

Grid Site	**# of Files**	**Size (MB)**	**# of CPUs**
HG-03-AUTH	2.659.554	153.560	118
RO-08-UVT	284.426	24.527	28
MK-01-UKIM-II	182.876	3.671	16
AEGIS01-PHY-SCL	150.874	5.162	689
HG-05-FORTH	1.434.525	94.039	232
BG01-IPP	3.632.212	108.814	18
CY-03-INTERCOLLEGE	91.783	2.648	10
HG-02-IASA	3.366.365	195.193	118
CY-01-KIMON	550.762	31.891	78
Total	**12.353.377**	**619.505**	**1.307**

TABLE I
TESTBED.

and ranks them with respect to the users' queries. To this end, a ranking algorithm is used to improve the accuracy and relevance of the replies, especially when keyword-based searching produces very large numbers of "relevant" software packages. Minersoft uses the Lucene relevance ranking [1]. In particular, Lucene provides a scoring algorithm that includes additional data to find best matches to users queries. The default scoring algorithm is fairly complex and considers such factors as the frequency of a particular query term with individual software files and the frequency of the term in the total population of software files. Finally, the users interact with the *query-processor* module in order to receive the results.

V. EXPERIMENTAL EVALUATION

Testbed: Our experiments were conducted on EGEE, one of the largest Grid production services currently in operation. Table I presents the Grid sites that have been crawled and indexed by Minersoft.

In this paper, we elaborate on the performance evaluation of the crawling and indexing tasks of Minersoft. Our objective is to show that Minersoft works sufficiently on a real, large-scale Grid testbed. Regarding the information retrieval perspective, we validated Minersoft by harvesting a site of the EGEE infrastructure (CY-01-KIMON). For the experiments, we used a collection of queries, expressed as a single keyword search or multiple-keyword compound search. Results showed that the Minersoft algorithm achieves both high search efficiency (the response time is less than 1 sec) and accuracy (high recall and precision values), and outperforms existing state-of-the-art methods. We do not present any results regarding the evaluation of the Software Graph since it is out of the scope of this work.

Examined metrics: To assess the crawling and indexing in Minersoft, we investigate the performance of crawler and indexer jobs; recall that each job is responsible for a number of files (called splits) that exist on a Grid site. In this context, we use the following metrics:

- Run time: the average time that a crawler/indexer job spends on a Grid site, including processing and I/O; this metric measures the average elapsed time that Minersoft needs to process (crawl or index) a split.
- CPU time: the average CPU time in seconds spent by a crawler/indexer job while processing a split on a Grid site.
- File rate: the number of files that Minersoft crawls/indexes per second on a Grid site.
- Size rate: the size of files in bytes that Minersoft crawls/indexes per second on a Grid site.

In our experiments, each crawler and indexer job was configured to run with five threads. We also ran experiments with different numbers of threads (from 1, 5, 9 to 13) and concluded that 5 threads per crawler/indexer job provide a good trade-off between crawling/indexing performance and Grid site workload. Smaller or larger numbers of threads per crawler/indexer job usually result to significantly higher run times, due to poor CPU utilization or I/O contention, respectively.

Crawling Evaluation: Figure 2 depicts the *per-job average run-time* and *per-job average CPU-time* for crawling the Grid sites. The per-job CPU time takes into account the total time that all the job's threads spend in the CPU. The run-time values are significantly larger than the CPU times due to the system calls and Input/Output that each crawler performs while processing its file split. I/O is much more expensive in the case of sites with shared file systems. Another observation is that the run-time and CPU-time of crawler jobs vary significantly across different Grid sites. This imbalance is due to several factors, including the hardware heterogeneity of the infrastructure, the dynamic workload conditions of shared sites, and the dependence of the crawler processing on site-dependent aspects. For example, the crawler performs expensive "deep" processing of binary and library files to deduce their type and extract dependencies. This is not required for text files. Consequently, the percentage of binaries/libraries found in each site determines to some extent the corresponding crawling computation. Overall, the crawling of a site can take a few hours; for instance, the total crawling time for CY-01-KIMON is about 4 hours. Table III depicts the throughput achieved by the Minersoft crawler on different sites, expressed in terms of the number of files and the number of bytes processed.

The files found by the crawlers to be irrelevant to software search are pruned from subsequent processing. Figure 3 presents the percentage of files that have been dropped. We observe that a large percentage of content in Grid sites (50%-90%) includes software resources. These files are categorized with respect to their type (Table IV). From Table IV, we can see that most software-related files in the Grid infrastructure are documentation files (man-pages, readme files, html files) and sources[1](e.g. Java, C++). Finally, each crawler stores within the storage element of its site a *metadata store file* capturing the file-id, name, type, path, size and structural dependencies of the identified software resources. The size of each *metadata store file* is presented in Table II.

Indexing Evaluation: Figure 2 depicts the *per-job average run-time* and the *per-job CPU time* for indexing Grid sites. As expected, we observe that indexing is more computationally-

[1] Sources are files written in any programming language. Executable scripts (e.g. python, perl, bash) are also considered as sources.

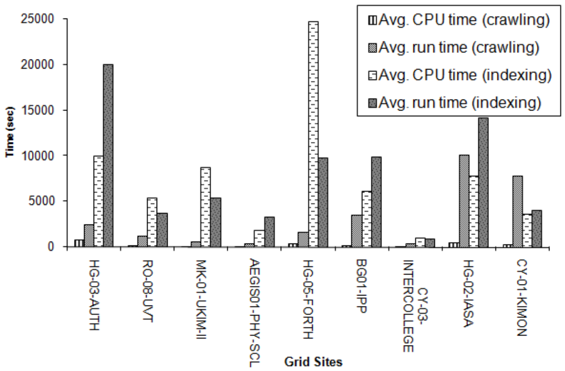

Fig. 2. Average times for jobs.

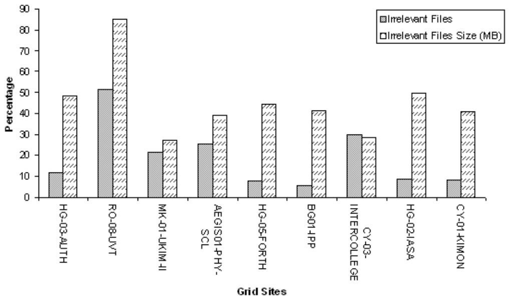

Fig. 3. Percentage of irrelevant files.

Grid Sites	Crawling Statistics		Indexing Statistics	
	# of splits	Metadata store file size (MB)	# of splits (including duplicates)	Inverted Index size (MB)
HG-03-AUTH	9	41	6 (8)	1261,33
RO-08-UVT	1	3,1	1 (1)	156,58
MK-01-UKIM-II	1	2,45	1 (1)	121,75
AEGIS01-PHY-SCL	1	2,2	1 (1)	163,55
HG-05-FORTH	5	20,5	5 (5)	586,05
BG01-IPP	13	38	6 (6)	850,9
CY-03-INTERCOLLEGE	1	1,5	1 (1)	75,06
HG-02-IASA	12	51	6 (11)	1511,75
CY-01-KIMON	2	7,3	2 (2)	128,9

TABLE II
CRAWLING & INDEXING STATISTICS.

Grid Sites	File Rate (files/sec) Files/Run time	Size Rate (MB/sec) MB/Run time
HG-03-AUTH	117,82	6,80
RO-08-UVT	231,82	19,99
MK-01-UKIM-II	275,24	5,52
AEGIS01-PHY-SCL	339,66	11,62
HG-05-FORTH	172,65	9,43
BG01-IPP	79,15	2,06
CY-03-INTERCOLLEGE	225,43	6,50
HG-02-IASA	27,74	1,60
CY-01-KIMON	34,93	2,02

TABLE III
CRAWLING RATES.

Grid Site	Binaries	Sources	Libraries	Docs	Irrelevant
HG-03-AUTH	32.276	958.811	96.698	1.252.504	319.265
RO-08-UVT	8.134	56.784	4.199	68.332	146.977
MK-01-UKIM-II	15.083	62.856	8.010	57.806	39.121
AEGIS01-PHY-SCL	6.064	31.734	7.669	66.814	38.593
HG-05-FORTH	28.351	495.982	65.507	729.980	114.705
BG01-IPP	28.684	738.354	83.332	2.580.035	201.807
CY-03-INTERCOLLEGE	26.971	8.925	3.644	24.945	27.298
HG-02-IASA	50.096	1.313.373	121.979	1.597.527	283.390
CY-01-KIMON	28.690	166.285	22.571	287.304	45.912
Total	**224.349**	**3.833.104**	**413.609**	**6.665.247**	**1.217.068**

TABLE IV
FILES CATEGORIES.

Grid Sites	File Rate (files/sec) Files/Run time	Size Rate (MB/sec) MB/Run time
HG-03-AUTH	38,65	1,75
RO-08-UVT	24,03	0,82
MK-01-UKIM-II	14,94	0,31
AEGIS01-PHY-SCL	59,04	1,98
HG-05-FORTH	10,75	0,40
BG01-IPP	92,25	1,93
CY-03-INTERCOLLEGE	37,09	1,82
HG-02-IASA	65,07	2,92
CY-01-KIMON	66,50	3,36

TABLE V
INDEXING RATES.

intensive than crawling, since we need to conduct "deep" parsing inside the content of all files.

Removing the duplicate files via the *duplicate reduction policy* leads to reducing either the number of splits or the number of files within splits since we found that 33% of files belong to more than one Grid sites. For instance, HG-02-IASA has 6 splits instead of 11 splits (including duplicate files). Consequently, the total indexing time is significantly reduced. Table II presents the number of splits and the size of inverted file indexes in each Grid site. Note that less number of splits are required for indexing than crawling since the irrelevant files have been deleted. Finally, Table V depicts the throughput of the indexer expressed in terms of the number of files and the number of bytes processed per second in each Grid site. The performance of indexing is affected by the hardware (disk seek, CPU/memory performance), file types, and the workload of each site.

To sum up, our experimentations concluded to the following empirical observations:

- Minersoft successfully crawled 12.3 million valid files (620 GB size) and sustained, in most sites, high crawling and indexing rates.
- A large percentage of duplicate files exists in Grid sites. Specifically, 33% of files belongs to more than one Grid sites.
- The crawling and indexing is significantly affected by the hardware (local disk, shared file system), file types and the current workload of Grid sites.
- It is important to establish advanced software discovery services in the Grid since, in most cases, more than 50% of files that exist in the workernodes file systems of Grid sites are software files.

VI. OPEN ISSUES

We are currently exploring a number of open issues that require further analysis:

- Determining the size of splits: As we referred above, the files of each Grid site are split into a number of splits where the size of the split is chosen to ensure that the crawling and indexing can be distributed evenly and efficiently within the time constraints of the underlying site. Considering that in a Grid infrastructure the execution time and workload cannot be determined in advance using historical data, the size of splits should be adapted to the working environment. To meet this challenge, the *Grid job manager* should have a global view of the system's workload so as to dynamically rearrange the size of splits as well as schedule them to Grid sites.

- Determining the number of threads per crawler/indexer jobs: From our experiments it is obvious that the number of threads per crawler/indexer job affects significantly the performance of Minersoft. To improve the efficiency of crawling and indexing, Minersoft should enhance self-adaptive mechanisms in order to assign a sufficient number of threads to Grid resources (CPUs in Grid sites). Grid monitoring systems provide information about resource utilization (CPU utilization, memory utilization, disk utilization, etc.) and network connectivity in Grid sites. Thus, if the current status of a Grid site has changed, the *Grid job manager* should modify the number of threads per crawler/indexer jobs in this site.
- Detecting duplicate software files: Files that are exact duplicates of each other can be identified by either heuristic techniques or checksumming techniques. In order to prevent duplicate files, crawler jobs need to periodically communicate to coordinate with each other. However, this communication may result in overhead. Can we minimize this communication overhead while maintaining the effectiveness of the crawler job? Authors in [9] dealt with this problem in the context of the Web. Another issue is the identification of near-duplicate files. If we could successfully identify these files, we could improve the performance of indexing, since a percentage of files will be deleted. Manber [17] has developed algorithms for near-duplicate detection to reduce storage in large-scale file systems.
- Determining politeness: Minersoft jobs should not obstruct the normal operation of Grid sites. Minersoft should adhere to strict rate-limiting policies when accessing poorly provisioned (in terms of workload) Grid sites. To address this issue, Minersoft should implement a flexible policy that would avoid running multiple crawler jobs to overloaded Grid sites.

VII. Conclusions - Future Work

In this paper, we present Minersoft - a Grid harvester which enables keyword-based searches for software installed on Grid computing infrastructures. The results of Minersoft harvesting are encoded in a weighted, undirected, typed graph, called the Software Graph. The Software Graph is used to annotate automatically the software resources with keyword-rich metadata. Then, each Grid site indexes its Software Graph. Using a real testbed, we present the performance issues of crawling and indexing. Minersoft successfully crawled 12.3 million valid files (620 GB size) and sustained, in most sites, high crawling. Except of Grids, Minersoft can also be used as keyword-based paradigm for any large-scale distributed computing platform, such as Clouds, as well as, for stand-alone computers. In future work we intend to enhance the Map-Reduce paradigm in the Minersoft architecture.

Acknowledgement: This work was supported in part by the European Commission under the Seventh Framework Programme through the SEARCHiN project (Marie Curie Action, contract number FP6-042467) and the Enabling Grids for E-sciencE project (contract number INFSO-RI-222667).

References

[1] Apache Lucene. http://lucene.apache.org (last accessed February 2009).

[2] Enabling Grids for E-SciencE project. http://www.eu-egee.org/ (last accessed February 2009).

[3] teragrid. http://www.teragrid.org/ (last accessed February 2009).

[4] R. Agrawal and et al. The claremont report on database research. *SIGMOD Rec.*, 37(3):9–19, 2008.

[5] A. Ames, C. Maltzahn, N. Bobb, E. L. Miller, S. A. Brandt, A. Neeman, A. Hiatt, and D. Tuteja. Richer file system metadata using links and attributes. In *MSST '05*, pages 49–60, Washington, DC, USA, 2005. IEEE Computer Society.

[6] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[7] L. Bass, P. Clements, R. Kazman, and M. Klein. Evaluating the software architecture competence of organizations. In *WICSA '08*, pages 249–252, 2008.

[8] F. Brochu, U. Egede, J. Elmsheuser, and K. H. et al. Ganga: a tool for computational-task management and easy access to Grid resources. *Computer Physics Communications (submitted)*, 2009. http://ganga.web.cern.ch/ganga/documents/index.php.

[9] J. Cho and H. Garcia-Molina. Parallel crawlers. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 124–135, New York, NY, USA, 2002. ACM.

[10] M. D. Dikaiakos, R. Sakellariou, and Y. Ioannidis. *Information Services for Large-scale Grids: A Case for a Grid Search Engine*, chapter Engineering the Grid: status and perspectives, pages 571–585. American Scientific Publishers, 2006.

[11] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. James W. O'Toole. Semantic file systems. In *SOSP '91*, pages 16–25, New York, NY, USA, 1991. ACM.

[12] O. Hummel and C. Atkinson. Extreme harvesting: Test driven discovery and reuse of software components. In *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IRI - 2004, Las Vegas Hilton, Las Vegas, NV, USA*, pages 66–72, 2004.

[13] S. Khemakhem, K. Drira, and M. Jmaiel. Sec+: an enhanced search engine for component-based software development. *SIGSOFT Softw. Eng. Notes*, 32(4):4, 2007.

[14] J. Koren, A. Leung, Y. Zhang, C. Maltzahn, S. Ames, and E. Miller. Searching and navigating petabyte-scale file systems based on facets. In *PDSW '07*, pages 21–25, 2007.

[15] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD 2008*, pages 903–914, New York, NY, USA, 2008. ACM.

[16] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng.*, 17(8):800–813, 1991.

[17] U. Manber. Finding similar files in a large file system. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 2–2, Berkeley, CA, USA, 1994. USENIX Association.

[18] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE 2003*, pages 125–135, May 2003.

[19] M. Matsushita. Ranking significance of software components based on use relations. *IEEE Trans. Softw. Eng.*, 31(3):213–225, 2005.

[20] F. Silvestri, D. Puppin, D. Laforenza, and S. Orlando. Toward a search architecture for software components. *Concurrency and Computation: Practice and Experience*, 18(10):1317–1331, 2006.

[21] T. A. Vanderlei, a. Frederico A. Dur A. C. Martins, V. C. Garcia, E. S. Almeida, and S. R. de L. Meira. A cooperative classification mechanism for search and retrieval software components. In *SAC '07*, pages 866–871, New York, NY, USA, 2007. ACM.