MOVE: A Distributed Framework for Materialized Ontology View Extraction[1]

Mehul Bhatt,[2] Andrew Flahive,[2] Carlo Wouters,[2] Wenny Rahayu,[2] and David Taniar[3]

Abstract. The use of ontologies lies at the very heart of the newly emerging era of semantic web. Ontologies provide a shared conceptualization of some domain that may be communicated between people and application systems. As information on the web increases significantly in size, web ontologies also tend to grow bigger, to such an extent that they become too large to be used in their entirety by any single application. Moreover, because of the size of the original ontology, the process of repeatedly iterating the millions of nodes and relationships to form an optimized sub-ontology becomes very computationally extensive. Therefore, it is imperative that parallel and distributed computing techniques be utilized to implement the extraction process. These problems have stimulated our work in the area of sub-ontology extraction where each user may extract optimized sub-ontologies from an existing base ontology. The extraction process consists of a number of independent optimization schemes that cover various aspects of the optimization process, such as ensuring consistency of the user-specified requirements for the sub-ontology, ensuring semantic completeness of the sub-ontology, etc. Sub-ontologies are valid independent ontologies, known as materialized ontologies, that are specifically extracted to meet certain needs. Our proposed and implemented framework for the extraction process, referred to as *Materialized Ontology View Extractor* (MOVE), has addressed this problem by proposing a distributed architecture for the extraction/optimization of a sub-ontology from a large-scale base ontology. We utilize *coarse-grained data-level parallelism* inherent in the problem domain. Such an architecture serves two purposes: (a) facilitates the utilization of a cluster environment typical in business organizations, which is in line with our envisaged application of the proposed system, and (b) enhances the performance of the computationally extensive extraction process when dealing with massively sized realistic ontologies. As ontologies are currently widely used, our proposed approach for distributed ontology extraction will play an important role in improving the efficiency of ontology-based information retrieval.

Key Words. Parallel and distributed systems, Coarse-grained parallelism, Semantic web, Ontologies, Sub-ontology extraction.

1. Introduction. The next generation of the internet, called the *semantic web*, provides an environment that allows more intelligent knowledge management and data mining. The main focus is the increase in *formal structures* used on the internet. The taxonomies—with added functionality, such as inferencing—for these structures are called ontologies [1], [2], and the success of the semantic web highly depends on the success of these *ontologies*. By defining the common words and concepts that are used

[2] Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, Melbourne, Victoria 3086, Australia. {M.Bhatt,A.Flahive,W.Rahayu}@latrobe.edu.au.
[3] School of Business Systems, Monash University, Clayton, Victoria 3800, Australia. David.Taniar@infotech.monash.edu.au.

to describe and represent an area of knowledge, ontologies enforce inter-operability both at a syntactic as well as a semantic level. Ontologies may be used by/for people, databases and applications that need to share domain-specific information. User groups and communities need to agree on ontologies before they can be put to good use. The reason ontologies are becoming popular is largely due to what they promise: a shared and common understanding of a domain that can be communicated between people and application systems [1]. Getting large groups to agree on an ontology means that a compromise needs to be established, and that the final ontology is the best for the community. However, for an individual alternative ontologies might be more appropriate. Often the general ontology grows very large as it attempts to model a certain problem domain accurately and completely, which poses a number of problems.

One of the major problems is that as an ontology grows bigger, user applications only require particular aspects of the ontology as they do not benefit from the plethora of semantic information that may be present in the ontology. However, using the ontology means that all the drawbacks from this extra information are encountered; complexity and redundancy rise, while efficiency falls. This brings with it a clear need to create a sub-ontology [3], [4]. For instance, if a business (application) only concerns itself with the efficiency of the workers, there is no need to access the detailed product catalog. Extracting just the part that is needed offers a smaller, more efficient, simpler solution/ontology. Ontology extraction focuses on deriving a sub-ontology from a given base ontology, based on user specified preferences. The preferences basically represent subjective information relevant to the extraction process. They characterize qualitative aspects that the target or sub-ontology should conform to. A lot of research in similar areas has been done (e.g. in [5]–[8]). Previous research by the authors pioneered in the specialized area of ontology extraction [3], [9]. It was shown that a group of requirements can be handled by addressing the appropriate sequence of single requirements. Each of these requirements can be considered an optimization for a certain aspect (e.g. optimized for some form of completeness or consistency), and individual optimization schemes were introduced to meet the requirements, and guarantee a high quality resulting sub-ontology.

The ontology *extraction process is computationally extensive*. Ontology extraction algorithms need to traverse the whole base ontology, get the appropriate nodes and relationships as requested by the user application, connect the nodes together using some established rules and re-iterate the traversal until the most optimum set of nodes and relationships is found. The complexity is only complemented by the fact that the size of the base ontology is huge. For example, the UMLS—Unified Medical Language Systems [10]—base ontology has more than 800,000 concepts (nodes) and more than 9,000,000 relationships between those concepts. Moreover, extraction is not just a one-time affair. It is required during creation time, updation and maintenance and refinement of the sub-ontology and must therefore have a reasonably quick response time. This can only be done by utilizing parallel and distributed processing techniques for ontological workload processing. Furthermore, we have to make sure that the system utilizes high-performance computing (HPC) resources easily available/affordable to the users of ontology applications. We envisage businesses utilizing our system to have access to low-cost Beowulf class clusters of inter-connected workstations. As such, a distributed memory model seems more suitable than a singular shared-memory-based parallel approach. It is worth

pointing out here that the granularity of parallelism inherent in the problem domain is very coarse. Most of the optimization schemes and/or sub-schemes do not differentiate between a whole-base ontology or a partitioned or scaled-down version of it. It is this coarseness that we are primarily interested in exploiting in *MOVE*. Also, considering the fact that most of the optimization schemes are functionally independent, procedure level parallelism too could be utilized. A more detailed discussion is presented toward the end of the paper in Section 6.

The main aim of this paper is to introduce a *coarse-grained distributed approach* to the materialized sub-ontology extraction process. Distribution not only makes the process faster, but also (and more importantly) facilitates our envisaged application of the extraction process. For now, we postpone a detailed discussion of the motivation for a distributed approach to materialized extraction to section 3.1. We think it is natural first to familiarize the reader with the idea of "*Sub-Ontology Extraction*", which itself is a new concept. Secondarily, this work could be looked upon as a general research investigating techniques to perform a coarse-grained distribution of ontological work-load processing. Currently, ontology-based applications do not make use of parallel and distributed techniques to process their ontological workload. We feel that this work will encourage other researchers in the *semantic web* community to explore the use of HPC resources for their applications.

The work presented in this paper is a culmination of our research and developmental efforts pertaining to sub-ontology extraction presented individually in various other publications. We refer interested readers to [9], [11] and [12] for a more thorough and formal account of the extraction process. In this paper we present an intuitive introduction to ontologies and sub-ontology extraction. Also, to keep the discussion simple and concise, we refrain from presenting details pertaining to all the optimization schemes. Instead, we focus on two of the rather simpler optimization schemes (OS) within the extraction process—Requirements Consistency OS (RCOS) and Semantic Completeness OS (SCOS). The rest of the paper is organized as follows: Section 2 serves as essential background for the rest of the paper. Starting with a succinct (and informal) introduction to ontologies and the semantic web, we familiarize the reader with the materialized sub-ontology extraction process. In Section 3 we utilize RCOS and SCOS as the basis for illustrating the distribution of the sequential extraction process proposed in Section 2. Section 4 provides details relevant to architecture and environment whereas Section 5 consists of performance analysis. Finally, we conclude in Section 6 with a few pointers to future work especially concerning the distribution and performance of MOVE.

2. Background

2.1. *Ontologies and the Semantic Web.* The next generation of the internet aims to make the web resources more readily accessible to automated processes by adding meta-data annotations that describe their content. If meta-data annotations are to make resources more accessible to automated agents, it is essential that their meaning can be understood by such agents. Ontologies play a pivotal role here by providing a source of shared and precisely defined terms that can be used in such meta-data. An ontology typically consists of a hierarchical description of important concepts in a domain, along with descriptions

of the properties of each concept. The degree of formality employed in capturing these descriptions can be quite variable, ranging from natural language to logical formalisms, but increased formality and regularity clearly facilitates machine understanding [13]. Clearly, this coincides with the vision that has been originally referred to as the *semantic web* [14].

Examples of the use of ontologies include: (a) *e-commerce* sites, where ontologies can facilitate machine-based communication between two parties, (b) in *search engines*, where ontologies can help searching to go beyond keyword-based approaches, and allows results to be found that contain syntactically different, but semantically similar words and phrases, (c) in *web services* [15], where ontologies can provide semantically richer service descriptions that can be more flexibly interpreted by intelligent agents, (d) in the *medical domain* to contain the complex medical data and (e) in *multimedia information retrieval* [16].

2.2. *MOVE*: *Materialized Ontology View Extraction.*   Figure 1 shows a schematic of the sequential extraction process. The process begins with the import of ontology, which is represented using some representation standard, the requirements specification by a user (or another application) followed by the execution of the optimization algorithms that finally produce the materialized view. In the sub-sections that follow, we briefly discuss the relevance of each of the main components illustrated in Figure 1.

2.2.1. *Ontology Import Layer.*   The *import layer* (component 1) is responsible for handling various ontology representation standards that the extraction process is supposed to be compliant with. This is achieved in MOVE by transforming the external representation
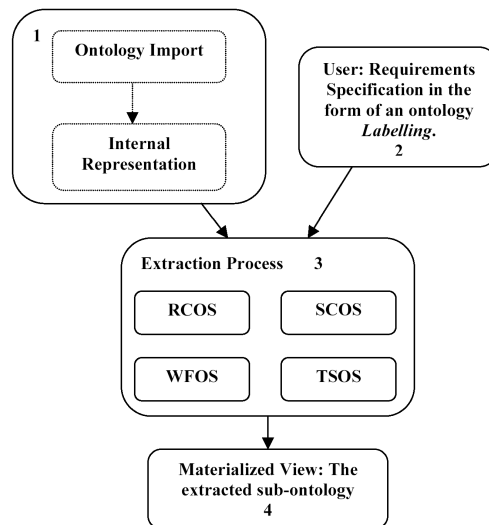
Fig. 1. The sequential extraction process.

of the ontology and its *meta-level*[4] to an internal one that is specific to our implementation. It is necessary for user applications to use our import layer to be able to utilize the extraction algorithms. The *representation layer* maintains an object-oriented view of the ontology and its meta-level. This facilitates easy extensibility as new ontology elements (new types) may easily be added in the ontology as well as its meta-level.

2.2.2. *Labeling—Requirements Specification.* "*Labeling*" (component 2) of the base ontology facilitates user manipulation of the extraction process. The labeling may also be re-applied (i.e. modification of the user-specified labeling) by the intermediate steps involved in the extraction process. This is the standard way different components of the extraction process (different extraction algorithms) may communicate with each other. Therefore, labeling is very crucial in the interaction between users and the extraction algorithms and the algorithms amongst themselves. It allows a user to provide subjective information, pertaining to what must/must not be included in the target sub-ontology, on which the extraction process is based. Moreover, an algorithm may work upon the labeling specified by the user, modify it in a certain way while preserving the semantics of the specification and pass it to another algorithm within the extraction process. Currently, every ontological element may have a labeling of *selected*, must be present in the sub-ontology, *deselected*, must be excluded from the sub-ontology, or *void*, the extraction algorithm is free to decide the respective elements inclusion/exclusion in the sub-ontology.

2.2.3. *The Extraction Process.* The extraction process (component 3) involves application of various optimization schemes that handle various issues pertaining to it. Examples include tasks such as ensuring consistency of initial requirements, maintaining semantic completeness, well-formedness and deriving a sub-ontology that is highly qualitative in a sense that it is optimum and is the best solution to the users requirements. Note that the extraction process is not limited to the optimization schemes currently being used in our framework. Also, it is possible that a particular scheme be completely left out of it. Below, we present a short discussion of each of the optimization schemes currently being used in MOVE.

- **Requirements Consistency Optimization Scheme**: As the name implies, RCOS checks for the consistency of the user-specified requirements for the target ontology in the form of the labeling. Currently, RCOS itself is a combination of four sub-schemes that check for various forms of consistency. We cover this optimization scheme in greater detail in Section 3.3.
- **Semantic Completeness Optimization Scheme**: SCOS considers the completeness of the concepts, i.e. if one concept is defined in terms of an another concept, the latter cannot be omitted from the sub-ontology without loss of semantic meaning of the former concept. Currently, SCOC consists of three sub-schemes that check for various forms of semantic completeness. SCOS is discussed in detail in Section 3.4.

---

[4] The meta-level consists of type-information pertaining to the various elements in the ontology. For example, if PERSON is a element in an ontology, the meta-level holds the type of the respective element, say a CONCEPT. So a PERSON has a meta-type of CONCEPT.

- **Well Formedness Optimization Scheme (WFOS)**: It might be possible that the user requirements (labeling) is consistent. However, there might be statements that inevitably lead to a solution that is not a valid ontology. WFOS contains the proper rules to prevent this from happening. WFOS is a combination of five sub-schemes.
- **Total Simplicity Optimization Scheme (TSOS)**: Applying TSOS to an existing solution (along with its requirements specification) will result in the smallest possible solution that is still a valid ontology. TSOS achieves this by working not only on the solution, but also its requirements specification. It may be the case that the most versatile solution is not the smallest one. In such scenarios, TSOS will not be applied. TSOS consists of three sub-schemes.

2.3. *Materialized Sub-Ontology View*.   The result of the extraction process is not just simply an extracted sub-ontology, but rather an extracted materialized ontology view (component 4). In the extraction process, no new information should be introduced (e.g. adding a new concept). However, it is possible that existing semantics are represented in a slightly different manner (i.e. a different *view* is established). Intuitively, the definition states that—starting from a base ontology—elements may be left out and/or combined, as long as the result is a valid ontology. In the process no new elements should be introduced (unless the new element is a combination of a number of original elements, i.e. the *compression* of other elements). A *materialized* ontology view is required, as the resulting sub-ontology should be an independent ontology, i.e. *should* be a valid ontology even if the base ontology is taken away.

3. Materialized Extraction Distribution Methods

3.1. *Exploiting Coarse-Grained Parallelism in MOVE*.   Often, business organizations have a cluster-like setup of inter-connected workstations as opposed to a single *shared-memory*, High-Performance Computing (HPC) facility. One reason for this is that a "*Beowulf Class Cluster*" setup is more economical in comparison with a centralized HPC facility. It is this setup that we aim to leverage upon by implementing a distributed memory architecture for the sub-ontology extraction process. Besides, the complexity of the base ontology and the multiple number of traversals that have to be performed in order to come up with the most optimum materialized view make the extraction process computationally extensive. The complexity is only complemented by the fact that the extraction process might be run frequently. Apart from a first *creation time* run of the extraction process, the extracted sub-ontology might need periodic *updates* and *maintenance* or the user might simply want to change the extraction parameters for purposes of *refinement* of the sub-ontology. Moreover, the sequential extraction process, as illustrated in Section 2.2, lends itself to easy distribution. For instance, most of the intermediate extraction algorithms do not distinguish between the complete centralized ontology and a scaled-down or partitioned version of it thereby facilitating the use of **coarse-grained data-level parallelism**. The same extraction algorithm may be applied to both sequential as well as distributed versions (albeit with some modifications) with different computing elements working on different parts of the ontology in the case of the distributed version.

3.2. *Ontological Notation.*   Below we present some notation consistent with [9], which is useful to define a common vocabulary pertaining to the ontological workload. Although we refrain from using this notation in the definitions that follow, it is useful for illustrating the pseudo-code associated with the respective optimization schemes.

1. $\delta_B(b)$: Denotes a binary relationship between concepts.
2. $\delta_C(\Pi_1(b))$: First concept associated with $\delta_B(b)$.
3. $\delta_C(\Pi_2(b))$: Second concept associated with $\delta_B(b)$.
4. $\delta_{attr}(t)$: Denotes an attribute-concept relationship.
5. $\delta_C(\Pi_1(t))$: A concept with associated attribute, the concept in a $\delta_{attr}(t)$.
6. $\delta_{\mathcal{A}}(\Pi_2(t))$: An attribute with an associated concept, attribute in $\delta_{attr}(t)$.
7. $\delta_B(b_i)$, $i \in [0, N]$: Many binary relationships linked to make a path.[5]
8. $\delta_B(b)$: Denotes a binary relationship between concepts.

3.3. *Requirements Consistency Optimization Scheme.*   In this section we focus on the *Requirements Consistency Optimization Scheme* (RCOS), which is one of the simpler phases within the entire extraction process, and illustrate our distribution scheme pertaining to the ontological workload associated with it. RCOS ensures that the requirements as expressed by the user (or any other optimization scheme) are consistent, i.e. there are no contradictory statements in the *labeling*, as set up by the user. By ensuring requirements consistency, we eliminate the possibility that no sub-ontology (based on user preferences) is derivable in the first place. RCOS is currently the very first rule to be applied during the extraction process. Moreover, it is also one of the rather simpler rules within the overall extraction process; both from a conceptual as well as an implementation viewpoint. RCOS is a suite of four optimization schemes, which without any implicit ordering or execution priority, we denote as RCOS1–RCOS4.

Before we proceed with illustrating the distribution scheme, it is necessary that each of RCOS1–RCOS4 be defined, albeit informally for the purposes of this paper. A formal introduction to RCOS (and the entire extraction process) along with a practical walk through with intuitive examples can be found in [3].

- **RCOS1**: The RCOS1 rule (see Figure 2) stipulates that *if a binary relationship between concepts is* selected *by the user to be present in the target ontology*, *the two concepts that the relationship associates cannot be* disqualified/deselected *from the target ontology*.
- **RCOS2**: The RCOS2 rule (see Figure 3) is similar to RCOS1 with the difference that instead of a binary relationship over the set of concepts, it is applied on a special relationship, called an attribute mapping, that exists between concepts and their attributes. This rule enforces the condition that *if an attribute mapping has a* selected *labeling*, *the associated attribute as well as the concept it is mapped onto must be* "selected" *to be present in the target ontology*.
- **RCOS3**: This rule (see Figure 4) imposes a requirement on a more specific characteristic of an attribute mapping. It stipulates that *if an attribute mapping has a* deselected *labeling*, *its associated attribute must also be disqualified from the target ontology*.

---

[5] We postpone the discussion of the notion of a path in this context until the illustration of RCOS4.

```
Input : A collection of relationships, [δ_B(b)], with a selected
labeling.
Output: A collection of invalid relationships, [δ_B(b)], as per
RCOS1 criteria.

BEGIN
Result = ∅
Loop through each relationship in [δ_B(b)]
{
  If δ_C(Π_1(b)) == deselected
  {
    add current δ_B(b) to Result
  }
  If δ_C(Π_2(b)) == deselected
  {
    add current δ_B(b) to Result
  }
}
Return Result.
END
```

Fig. 2. RCOS1 pseudo-code.

Basically, no contradicting preferences are allowed between an attribute mapping and the associated attribute. RCOS3 together with RCOS2 imposes this condition.

- **RCOS4**: RCOS4 (see Figure 5) is relatively more complex than each of RCOS1–RCOS3. We utilize the notion of a ***Path***, again informally, for illustrating RCOS4. Paths are very important in the specification of ontology views. They provide seemingly new relationships (new information) that are semantically correct, albeit only implicitly present in the ontology definition. A path is defined as the chain of relationships that connect concepts, where the end concept of one relationship is the start concept of

```
Input : A collection of mappings, [δ_attr(t)], with a selected
labeling.
Output: A collection of invalid mappings, [δ_attr(t)], as per
RCOS2 criteria.

BEGIN
Result = ∅
Loop through each mapping in [δ_B(b)]
{
  If δ_C(Π_1(t)) == deselected
  {
    add current δ_attr(t) to Result
  }
  If δ_A(Π_2(t)) == deselected
  {
    add current δ_attr(t) to Result
  }
}
Return Result.
END
```

Fig. 3. RCOS2 pseudo-code.

```
Input : A collection of mappings, [δ_attr(t)], with a deselected
labeling.
Output: A collection of invalid mappings, [δ_attr(t)], as per RCOS3
criteria.

BEGIN
Result = ∅
Loop through each mapping in [δ_B(b)]
{
  If  δ_A(Π_2(t)) == deselected
    add current δ_attr(t) to Result
}
Return Result.
END
```

Fig. 4. RCOS3 pseudo-code.

the following relationship in the chain. Note that the same relationship can appear only once in the entire path. From an RCOS4 viewpoint, the emphasis of a path lies on the first and last concept it visits, as these are the two that are connected by the path. However, alternative formulations of the "*path concept*" have been utilized by imposing certain qualification criteria on the concepts that are a part of the connection and/or the relationships that form the chain for other optimization schemes. Based on the above formulation of a path, we can now introduce the fourth requirements consistency rule. *If an attribute is selected, but the concept it "belongs" (mapped) to is deselected, RCOS4 stipulates that there must be a path from the attribute to another*

```
Input : A collection of attribute mappings, [δ_attr(t)], with
a deselected δ_C(Π_1(t)) and a selected δ_A(Π_2(t)).
Output: A collection, [δ_B(b_i)], of valid and invalid paths
found.

BEGIN
valid_path list   = ∅
invalid_path list = ∅
Loop through each mapping in [δ_attr(t)]
{
  δ_B(b_current) = findSolution(δ_A(Π_2(t)))
  if δ_B(b_current) is not valid
  {
    Mark δ_A(Π_2(t)) as invalid
    add δ_B(b_current) to invalid_path list
  }
  else
  {
    Mark δ_A(Π_2(t)) as valid
    add δ_B(b_current) to valid_path list
  }
}
Return valid and invalid path list
END
```
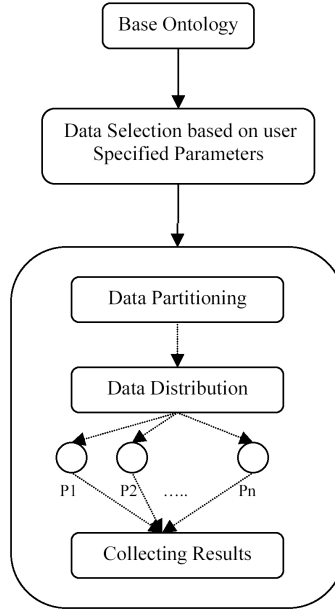
Fig. 5. RCOS4 pseudo-code.

Fig. 6. Classic task-farm distribution model for RCOS.

concept that is not deselected. Moreover, the path can only contain relationships with a label other than "deselected". The **findSolution**$(\delta_{\mathcal{A}}(\Pi_2(t)))$ procedure used in the pseudo-code in Figure 5 attempts to find a path (as defined above) from $\delta_{\mathcal{A}}(\Pi_2(t))$ to a concept that is not deselected.

3.3.1. *RCOS Distribution.*   Figure 6 illustrates the schematic of the classic task-farm model that is being utilized for the distribution of the RCOS. Although RCOS1–RCOS3 essentially use the same model as depicted in Figure 2, we use a slightly modified/extended version for RCOS4. As is evident from the pseudo-code for RCOS1–RCOS3, their respective implementations are indeed very simple. However, the complexity actually lies in the sheer size of the workload that needs to be processed for ensuring consistency. One can also draw the obvious parallels that would exist in a distributed implementation of RCOS1–RCOS3. Hence, we refrain from demonstrating the distribution schemes for each of RCOS1–RCOS3 separately. Instead, we illustrate the general distribution scheme used by us (based on Figure 7) for RCOS1–RCOS3.

Let O denote the Ontology and **filter(type, label)** be an operation defined over it such that it returns a collection of elements, $\vec{e}_{type}$, of the type given by "**type**" such that the label of each of those elements matches "**label**". Similarly, let **partition**$(\vec{e}_{type})$ be a generic operation that partitions the $\vec{e}_{type}$ according to some data *partitioning scheme*. If N computing elements are available (numbered 1 to N), the result of **partition**$(\vec{e}_{type})$ will be N distinct $\vec{e}^{i}_{type}$ where i represents the number of the computing element to which the respective partition has been assigned. Finally, let **tribute and gather** be *asynchronous* data distribution and result collection primitives. Also, **barrier** can be

```
1. Initialise O;
2. ē_type = filter(type, label);
3. [ē^i_type]  = partition(ē_type);
4. for(i = 1 To N)
   {
       distribute(ē^i_type, i);
   }
5. set a execution barrier
6. for(i = 1 To N)
   {
       gather(ē^i_type, i);
   }
7. merge results.
```

Fig. 7. RCOS1–RCOS3 distribution.

thought of as a synchronization primitive that puts the calling program in a wait mode until the satisfaction of some condition. In this case the condition is the completion of all the worker processes where the workload has been sent for processing.

Note that depending on which rule is being implemented, the parameters to the filter($\cdots$) operation change accordingly. However, the rest of the algorithm is exactly the same for each of RCOS1–RCOS3. Currently, we adopt an equal workload distribution scheme. So the partitioning scheme divides the entire ontological workload into N distinct partitions, when N computing elements are available for use. As mentioned before RCOS4 utilizes an extended model of the one depicted in Figure 7. For RCOS4 (see Figure 8), the main (distributing process) and the worker processes follow a bi-directional

```
1. Initialise O; terminate = false; k = 0;
2. ē_type = filter(type, label);
3. [ē^i_type]  = partition(ē_type);
4. for(i = 1 To N)
   {
       distribute(ē^i_type, i);
   }
5. while(terminate != true)
   {
       src = readMessage;
       if(readMessage == REQUEST_UPDATE)
       {
           ē^i_type = update(O);
           distribute(ē^i_type, src);
       }
       else if(readMessage == COLLECT_RESULT)
       {
           gather(ē^i_type, src);
           ++k;
       }
       if(k == workerCount)
           terminate = true;
   }
```

Fig. 8. RCOS4 Distribution

communication protocol consisting of *update requests* (sent by the worker processes) and *ontology updates* sent back by the main process. Note that although every worker processor for RCOS4 has its own data partition to work on, it does not have access to the whole base ontology. The *findSolution*($\cdots$) algorithm (see RCOS4 pseudo-code in Figure 5) works by traversing the ontology, trying to find a concept that is not deselected. This traversal may span parts of the ontology that are not *locally* present with the worker processor. The worker processors keep track of such a situation and ask for an *update* from the main processor. The updateRequest-updateReceive protocol continues as long as the worker processes do not reach a dead end or a best path is found, which is then returned as the locally found solution. Notice that RCOS4 does not involve any merging of results like RCOS1–RCOS3. Every solution returned by any of the worker processes will be complete in itself.

3.4. *Semantic Completeness Optimization Scheme.*   The idea of semantic completeness of an ontology can be interpreted in a number of ways. However, for the purposes of sub-ontology extraction, it amounts to the inclusion of the *defining elements* for the elements selected by the user by way of requirements specification. A defining element is a concept, relationship or attribute that is essential to the semantics of another element of the ontology. For example, a concept selected to be present in the sub-ontology would be semantically incomplete if its super-concept (the defining element in this case) is deselected at the same time. This could be further generalized into a situation where a set of elements are connected by an IS-A relationship unto any arbitrary depth. The scenario can only get more complex in the presence of more complex relationships such as multiple-inheritance, aggregation, etc. The Semantic Completeness Optimization Scheme (SCOS) exists to guard against such inconsistencies. As mentioned previously in Section 2.2.2, the requirements specification takes the form of a labeling of the ontological elements.

SCOS consists of a collection of three optimization schemes, which we (as before) refer to as SCOS1–SCOS3. Before illustrating the distribution scheme, it is necessary that each of SCOS1–SCOS3 be defined, albeit informally for the purposes of this paper. A formal introduction to SCOS (and the entire extraction process) along with a practical walk-through with intuitive examples can be found in [3]. Note that because of space restrictions, we do not discuss SCOS3 in this paper. Again, we refer interested readers to [3] for a formal introduction. Figure 9 shows the pseudo-code for SCOS1. SCOS2 is similar to SCOS1 except that the data elements that it works on consist of binary aggregation relationships instead of binary inheritance ones as is the case with SCOS1. SCOS1–SCOS3 are as follows:

- **SCOS1**: *If a concept is selected, all its super-concepts, and the inheritance relationships between the concepts and its super-concepts, must be selected.*
- **SCOS2**: *If a concept is selected, all the aggregate part-of concepts of this concept, together with the aggregation relationship, must also be selected.*
- **SCOS3**: *If a concept is selected, then all of the attributes it possesses, with a minimum cardinality other than zero, and their attribute mappings should be selected.*

3.4.1. *Problem Representation.*   SCOS1 and SCOS2 are conceptually similar with the difference that the former deals with a collection of inheritance relationships while

```
Input : A collection of binary relationships, [δ_B(b)].
Output: Boolean result indicating semantic
        completeness/incompleteness.

BEGIN SCOS1
Result = false;
Loop through each relationship in [δ_B(b)]
{
  btemp = current δ_B(b)
  bsub = findSub([δ_B(b)], getSubconcept(btemp))
  while(bsub != NULL)
  {
    btemp = bsub;
    bsub = findSub([δ_B(b)], getSubconcept(btemp))
  }

  csub = getSubconcept(btemp)
  csup = getSuperconcept(btemp)
  if(getLabel(csub == selected))
  {
    if(getLabel(btemp) == deselected
         || getLabel(csup) == deselected)
      result = false;
    else
    {
      modifyLabel(btemp, selected)
      modifyLabel(csup, selected)
    }
  }
}
Return Result.
END
```

Fig. 9. SCOS1 pseudo-code.

the latter deals with a collection of aggregation relationships. This collection can be conceptualized as a forest of sparsely connected undirected graphs with the concepts representing the vertices and the relationships representing the edges of the graph. Such a conceptualization (and representation) using a graph-theoretic approach is optimal (and convenient) for purposes of distribution of SCOS. For example, consider checking the semantic for the completeness of a (potentially huge) set of concepts related/connected by binary inheritance relationship, i.e. SCOS1. If the set is to be partitioned and distributed to different processors so that SCOS1 may be run on each of the partitions in parallel, it is obviously desirable to allocate one connected component to each of the processors. It is necessary to utilize standard graph-theoretic approaches to perform such a partitioning of data.

3.4.2. *Problems with Graph-Based Representation.* A major problem with representing the data partitioning problem (for SCOS) in a graph-theoretic manner is that our underlying ontology representation is not graph theoretic. During ontology import, we construct an object-oriented representation of the ontology as well as its meta-level. No structural information regarding the connectivity of the ontological elements is present. Since an ideal ontology would be massive in size and complex in structure, it would

be optimal from a performance view-point that the graph-based representation be constructed at the time of the initial ontology import. Our object-oriented design represents a trade-off decision we took given the fact that other optimization schemes (such as RCOS) do not benefit from a graph-based representation. Moreover, the ontological workload for SCOS currently only deals with binary inheritance and aggregation relationships between concepts. So a graph-based representation encompassing all different types of ontological elements would not be particularly useful.

3.4.3. *Proposed Solution*.   Prior to data partitioning for SCOS, a graph-based representation for elements specific to SCOS (binary inheritance and aggregation relationships) is constructed. We use the standard adjacency structure representation (comprised of adjacency lists) to construct the ontoGraph. This involves additional work in the form of pre-processing of the SCOS workload to extract the concept set (i.e. the vertices of the graph). This is necessary to construct the adjacency structure representation. Once the graph-based representation is complete, we use a technique similar to a depth first search algorithm on the graph to get the set of connected components (called partitions hereafter) so as to schedule each of those for distribution to worker processors. Below, we discuss the three main steps, namely ontology pre-processing, ontoGraph construction and partition formation, involved in ontology pre-processing.

As before, let O denote the Ontology and let *filter*(*type*, *label*) be an operation defined over it such that it returns a collection of elements, $\vec{e}_{type}$ of the type given by "*type*" and that the label of each of those elements matches "*label*". Let V denote the *set* of (indexed) vertices, each representing an ontological element and let *findIndexOf*(*element*, V) be a primitive that returns the index of the "*element*" that is present in the set given by V. Let G denote the desired graph-based representation of the ontology. As a note, all elements have been referenced in the pseudo-code for partition formation using their integral id's. We refrain from explaining some other self-explanatory primitives used in the pseudo-code.

- **Ontology pre-processing**: The pre-processing phase (shown in Figure 10) basically involves constructing the vertex *set* to be used by the ontoGraph construction module. The input to this phase is the whole ontology. Processing begins by extracting the *list* of binary (inheritance and aggregation for SCOS1 and SCOS2, respectively) relationships from the ontology and inserting the elements related by each of the relationships in the list to a *set-based container* thereby avoiding duplicates. Moreover, the unique vertices in the vertex set are keyed from 0 to $N - 1$, where N is the cardinality of the vertex set.
- **OntoGraph construction**: As mentioned before, we represent the ontoGraph using the standard adjacency structure representation. The adjacency structure consists of a *vector* of *lists* of graph nodes. Each node in turn consists of other information such as an integral id of the ontology element it represents, a pointer to the element it represents, etc. This ancillary information is necessary during the next phase, namely partition formation. Semantically, the nodes in the *list* in the ith position of the vector represent those elements that are directly connected to the ith vertex/element in the ordered set of vertices. The pseudo-code for this stage is shown in Figure 11.

```
Input : The Ontology (O).
Output: An ordered collection of unique vertices (V).

BEGIN preProcess
1. ResultSet = ∅;
2. type = inheritance | aggregation; //Depending on
3. label = any;                      //SCOS1 or SCOS2
4. [δ_B(b)] = filter(type, label);
5. counter = 1;
6. Loop through each relationship in [δ_B(b)]
   {
      subConcept = δ_C(Π_1(b));
      superConcept = δ_C(Π_2(b));
      if(not(subConcept ∈ ResultSet))
      {
         set index for subConcept to counter;
         counter++;
         ResultSet = ResultSet ⋃ subConcept;
      }
      if(not(superConcept ∈ ResultSet))
      {
         set index for superConcept to counter;
         counter++;
         ResultSet = ResultSet ⋃ superConcept;
      }
   }
7. Return Result.
END
```

Fig. 10. Pre-processing phase.

```
Input : The ResultSet from the Pre-Processing Stage (V)
and the Ontology (O).
Output: The OntoGraph (G).

BEGIN buildOntoGraph
1. ontoGraph G = ∅ //is a vector of lists.
2. type = inheritance | aggregation; //Depending on
3. label = any;                      //SCOS1 or SCOS2
4. [δ_B(b)] = filter(type, label);
5. counter = 1;
6. initializeOntoGraph(|V|);
7. Loop through each relationship in [δ_B(b)]
   {
      subConcept = δ_C(Π_1(b));
      superConcept = δ_C(Π_2(b));
      indexOfSub = findIndexOf(subConcept, V);
      indexOfSup = findIndexOf(supConcept, V);
      tempNode = initNode(subConcept, indexOfSub);
      G[indexOfSub].push_back(tempNode);
      tempNode = initNode(supConcept, indexOfSup);
      G[indexOfSup].push_back(tempNode);
   }
8. Return G.
END
```

Fig. 11. OntoGraph construction.

```
Input : The OntoGraph (G)
Output: A List of Partitions (P).

BEGIN PartitionGraph
1. Result = ∅;
2. if(ontoGraph G == ∅)
     return ∅;
3. int numberOfVertices = |G| ;
4. int visitedFlags[numberOfVertices];
5. for(i = 0; i < numOfVertices; ++i)
     visitedFlags[i] = 0;
6. for(i = 0; i < numOfVertices; ++i)
   {
     if(visitedFlags[i] == 0) //if not visited
     {
        partition = retrievePartition(i, G);
        Result.push_back(partition);
     }
   }
7. Return Result.
END
```

Fig. 12. Partition formation—I.

- **Partition formation**: Partition formation in our case is equivalent to finding the different connected components in the ontoGraph. We currently use a technique similar to a depth first traversal (of the ontoGraph), as shown in Figures 12 and 13, to achieve this. The input to this phase is the ontoGraph whereas the result consists of a list of partitions. Looking at the pseudo-code, one might get the impression that all of the partitions need to be formed before any of them are assigned to worker processors. This is because we have (for brevity) separated the illustration of partition formation (above) and distribution of the same (see Section 3.4.4) to a worker processor. However, in actuality, there is no reason to wait for the next partition to be generated before

```
Input : The OntoGraph (G) and a Vertex Number/ID to start
Traversal (i).
Output: A Connected Component Starting From Vertex i.

BEGIN retrievePartition(i, G)
1. Result = ∅;
2. nodeList lst = G[i];
3. visitesNodes[i] = 1;
4. Loop through each node in lst
{
   tempNode = gextNextNode(lst);
   int connectedTo = tempNode.connectedTo;
   if(visitedNodes[connectedTo] == 0)
       retrievePartition(connectedTo, G);
}
5. Result.push_back(V[i]);
END
```

Fig. 13. Partition formation—II.

the current one is scheduled for distribution to a free processor as the partition sets are all going to be disjoint. We make use of asynchronous distribution primitives to assign the most recently generated partition to a free processor without waiting for the next one to be formed. This is advantageous as working out the semantic completeness (for the assigned partition) and formation of the next partition can proceed in parallel. The asynchronous nature of the primitives only adds to this optimality.

3.4.4. *SCOS Distribution.*   For implementing the RCOS we utilized a modified version of the classic task-farm model. SCOS essentially utilizes a similar distribution model with the exception that there is continued two-way interaction between the master and worker processors. Moreover, unlike the RCOS distribution scheme, the worker processes do not need to post *updates* (requests for missing data) as they have all of the data elements that are needed to perform the most recently assigned task (i.e. any of SCOS1–SCOS3). Also, data partitioning by the master and processing by the workers happens concurrently as the master dynamically creates partitions and assigns them to worker processes in a round-robin manner. Note that to maintain clarity, the pseudo-code presented (here and in Section 3.4.3) does not reflect this fact. We use three asynchronous data distribution/result collection primitives namely *gatherModifiedLabelingsFrom*($\cdots$), *recvPartition*($\cdots$) and *sendModifiedLabelings*( ). *recvPartition*($\cdots$) is used by the worker processes to receive the ontological workload that needs to be processed. Likewise, *gatherModifiedLabelingsFrom*(...) is used by the main processor to gather results from the worker processors, which they send using the *sendModifiedLabelings*( ) primitive. As explained in Section 2.2.2, this result takes the form of the modified labeling set. Note that it may be possible for the main processor to receive a "semantic incompleteness" message and still get a modified labeling set. This is because following the rules for SCOS1–SCOS3, the worker processors attempt to make the extracted view as semantically complete as possible even if a 100% completeness is not possible.

Master processor execution (see Figure 14) consists of performing the necessary pre-processing of the ontology as explained in Section 3.4.3. To re-iterate, it involves ontology initialization, extraction of the unique vertex and edge set, building the ontoGraph representation and performing the ontoGraph partitioning coupled with asynchronous distribution to the worker processors. Once the distribution is achieved, the only thing that remains to be done for the master is collection and application of results to the solution set or the extracted view. As mentioned previously, irrespective of the results (i.e. semantic completeness/incompleteness), the master always applies the labeling modifications worked out and serialized back by the worker processes. Depending on the number of partitions scheduled to the worker processors, the master can figure out when results for each of them have been received and it is appropriate for the workers and itself to exit.

Worker execution (see Figure 15) involves checking for an "*execution command*" from the main processor. Possible commands are to receive the workload pertaining to SCOS1–SCOS3 or actually execute SCOS1–SCOS3. Execution of any of the optimization schemes is followed by the sending back of results (modified labeling) and an indication of whether or not semantic completeness is possible for the most recently received partition. Note that the worker only terminates upon receiving an "*SCOS_EXIT*" message from the main processor.

```
BEGIN Master
1. terminate = false;
2. resultsCounterSCOS1 = 0;
3. O = intialiseOntology();
4. V = preProcess(O);
5. G = buildOntoGraph(V);
6. P = partitionGraph(G);
7. partitionCount = |P| ;
8. while(terminate == false)
   {
      msg = getMessage();
      src = getSender();
      if(msg == SCOS1_SEMANTIC_COMPLETENESS)
        resultsCounterSCOS1++;
      else if(msg == SCOS1_SEMANTIC_INCOMPLETENESS)
        resultsCounterSCOS1++;

      modifiedLabelings = gatherModifiedLabelingsFrom(src);
      applyModifications(modifiedLabelings);

      if(resultsCounterSCOS1 == partitionCount)
        terminate = true;
   }
9. broadcastExitMessage(ALL_WORKERS);
END
```

Fig. 14. Master processors.

```
BEGIN Worker
1. terminate = false;
2. while(terminate == false)
   {
      msg = getMessage();
      if(msg == SCOS_EXIT)
         teminate = true;
      else if(msg == SCOS1_PARTITION)
      {
         [δ_B(b)] = recvPartition(SCOS1_PARTITION)
         O = initLocalOntology([δ_B(b)]);
         Result = SCOS1_Single(O);
      }
      else if(msg == SCOS2_PARTITION)
      {
         [δ_attr(t)] = recvPartition(SCOS2_PARTITION)
         O = initLocalOntology([δ_attr(t)]);
         Result = SCOS2_Single(O);
      }

      sendMessage(Result);
      sendModifiedLabelings();
   }
END
```

Fig. 15. Worker processors.

4. Architecture and Environment

4.1. *Plugin Architecture for the Extraction Algorithms.* Our implementation consists of two main components, one pertaining to ontology representation and materialized view extraction, the other pertaining to management of the distributed extraction process. We adopt a plugin-based architecture to facilitate seamless integration of both the components into other ontological applications. It is possible for a user application to incorporate our functionality in the form of a plugin. Moreover, each phase within the extraction process, consisting of various optimization schemes, has been implemented independent of other phases. Therefore, applications importing our functionality are not compelled to use the entire extraction process. Depending upon intended use, any of the optimization schemes may be used in isolation. However, note that there might exist a functional dependency between some of the optimization schemes, for example, Requirements Consistency must be checked before checking for Semantic Completeness.

The ontological workload distribution pertaining to all the optimization schemes turns out to be pretty standard. For instance, almost all optimization schemes involve the same data partitioning algorithms, distribution primitives, etc. Only the rules that qualify data selection (for example, for partitioning purposes) involve minor modifications. Therefore, the same plugin-based design has also been adopted for the distributed component, keeping in mind the following two perceived advantages offered by the said approach:

1. It facilitates our own (ongoing) development involving distribution of ontology-related workload, materialized extraction being just one application within the scope of this paper.
2. The distribution component encapsulates the classic task-farm distribution model, data partitioning and distribution primitives as well as facilities for merging back results. So our distributed approach may be replicated by users in other innovative application domains.

4.2. *Implementation Environment.* All implementation has been done using C++ on an Alphaserver SC supercomputer running Tru64 Unix 5.1. It has also been ported to a Linux Cluster environment with minimal modifications. Our distribution management component does not directly tackle issues pertaining to the cluster architecture, processor initialization, etc. Instead, the *Messaging Passing Standard* [17], [18], which encapsulates such architecture-specific details and provides high-level message-passing primitives suitable for distributed systems, has been utilized. As such, porting to other environments should not involve anything more than a recompilation on the target platform. Also, note that although homogeneous computing elements are being utilized currently, this is not a requirement for the implementation. Any distributed architecture is acceptable as long as it supports the MPI message passing standard. It is up to the standard to handle the underlying architectural details pertaining to the cluster setup.

5. Performance Evaluation. As has been emphasized time and again in this paper, distribution not only makes the computationally extensive extraction process faster, but also facilitates our envisaged application for it. The performance results strongly support

our use of the coarse-grained data-level parallelism inherent in MOVE. The next two sections provide an evaluation of the distribution methods described in this paper. The first is an optimization evaluation that estimates the number of computations required to complete the task. The second is a performance evaluation of one of the optimization schemes.

5.1. *Optimization Evaluation.*   Before any test runs were carried out, an optimization evaluation was conducted to determine the possible load on the processors. It can be expected that given the right processing techniques and a large enough processing load, the distribution of the load will improve the efficiency of the system. This evaluation estimates the optimum load and the optimum number of processors required for the system to be at its most efficient.

The RCOS was analyzed to determine the number of computations that would need to be processed in order for one iteration to be completed. The number of iterations vary according to the size of the ontology. From these iterations the number of operations are calculated. Next, the distribution methods were analyzed to determine the number of operations required to employ a worker process. These operations were used to calculate the overhead of employing multiple processors. The main costs were associated with the messages that had to be created and sent between the master and the worker processors.

Based on these calculations we were able to determine when it would be necessary to employ another worker processor and when to stop employing a worker processor. The calculations have been scaled down for each of the different sized ontologies so that they can be viewed on the one graph. This graph is shown in Figure 16.

The graph (in Figure 16) shows the effectiveness of using multiple processors for different sized ontologies. The different sized ontologies correlates to different sized workloads. It shows how each level of workload is handled by the distribution. The graph shows that for a very small workload the cost of employing worker processors actually increases the overall number of operations.

It is not until a workload equivalent to that of a 10,000 concept ontology is supplied, that any improvement in increasing the number of worker processors can be noticed.
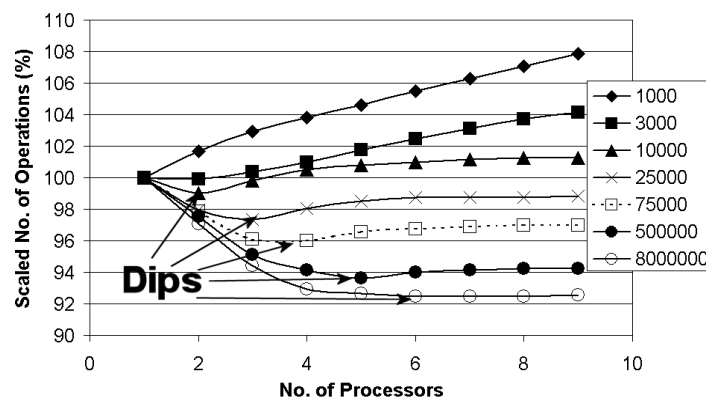

Fig. 16. RCOS evaluation.

Table 1. Optimum number of processors.

Size of processors	Optimum no. of processors
Less than 10,000	1
10,000–25,000	2
25,000–75,000	3
75,000–500,000	4
500,000–8,000,000	5
8,000,000 plus	6

This is the point when the number of operations being completed in parallel overcomes the number of operations of the overhead required to perform the parallel task. Up until this size, the cost of employing a second processor is not worth the effort.

The point of the most efficient use of processors to complete the given workload is highlighted on the graph by the "Dips". After the dip, the number of operations being completed in parallel is overwhelmed by the overhead of employing more worker processors. This is because the division of work required becomes smaller and the overhead increases. For large workloads the cost of employing more worker processors becomes negligible as the number of operations still being completed in parallel far outweigh the overhead.

Based on these "Dips", the number of processors required to complete a certain size task can be estimated. For example, if we required an ontology with a size of 100,000 concepts to be processed, how many processors should we employ to be assured of an efficient outcome? Judging by the "Dips" in the graph, we would make sure MOVE uses four processors.

Table 1 is a summary of the "Dips" indicating the optimal number of processors to use if given a certain size of ontology.

For ontologies that are small in size, i.e. less than 10,000 concepts, only one processor should be used as the workload is small. For ontologies that are between 10,000 and 25,000 concepts in size, two processors should be employed if the most efficient outcome is to be obtained. The number of processors increases along with the size of the ontology. For very large ontologies, like those over 8 million, increasing the number of processors makes little difference to the overall efficiency. It is here that the evaluation requires more exact measurements to be of any assistance in determining the most efficient number of processors to employ.

5.2. *Performance Analysis.* We restrict the performance analysis in this section to one of the optimization schemes within MOVE—SCOS. We believe the results (in terms of actual speedup gained) for other optimization schemes are similar and as such would be redundant.

Figure 17 presents the combined performance results for SCOS. It consists of results obtained over five test cases, each consisting of different sized ontologies. The time taken to process an ontology workload of these sizes are vastly improved when more than one processors is employed. For instance, in the case of the 5000 concept ontology (the top line on the graph), there is a saving of well over 15 minutes. In this case, using more than five processors results in the processing time being 15 times quicker.
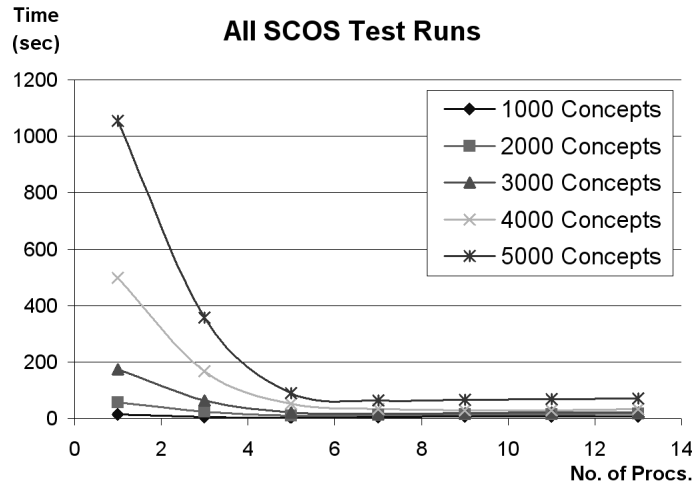
Fig. 17. All SCOS test runs.

A greater impact of multiple processors for smaller ontologies is seen in this graph compared with the number of operations estimated for RCOS in the previous section. This is because SCOS is computationally more extensive to complete per iteration and the iterations increase almost exponentially with the number of concepts. Thus employing multiple processors for smaller ontologies does not just depend on the number of concepts but also on the complexity of the optimization scheme(s) used.

Figure 18 shows the SCOS running time for relatively small (a) and relatively large (b) ontologies. These two graphs contains the same data as Figure 17 but displaying results separately to compare the processing of large ontologies with that of processing small ones.

Figure 18(a) shows a rapid increase in speedup over one processor when multiple processors are employed. However, it also shows that for a large number of processors, the time taken to process the workload gradually starts to increase beyond five processors. This is a similar phenomenon to that experienced with the RCOS evaluation mentioned in the previous section. These results show that the cost of splitting up a small amount of work into a large number of processors does not improve the overall speed up. This is largely due to the extra messaging required between the many processors. Instead, better results can be achieved by using just a single processor or only a few processors to complete the workload for small sized ontologies.

Figure 18(b) shows the SCOS running time for the larger ontologies. This graph clearly shows that a larger number of processors significantly improves the time taken to process the larger ontologies. The extra overhead costs become negligible when dealing with a larger number of processors. This is because more work is required to be done by each processor for the same amount of messaging. However, once the number of processors rises above seven, the cost of splitting and messaging takes over thereby reducing the efficiency.

It is important to note from each of these graphs (Figure 18) that the efficiency improves vastly when multiple processors are employed. If these processing costs were
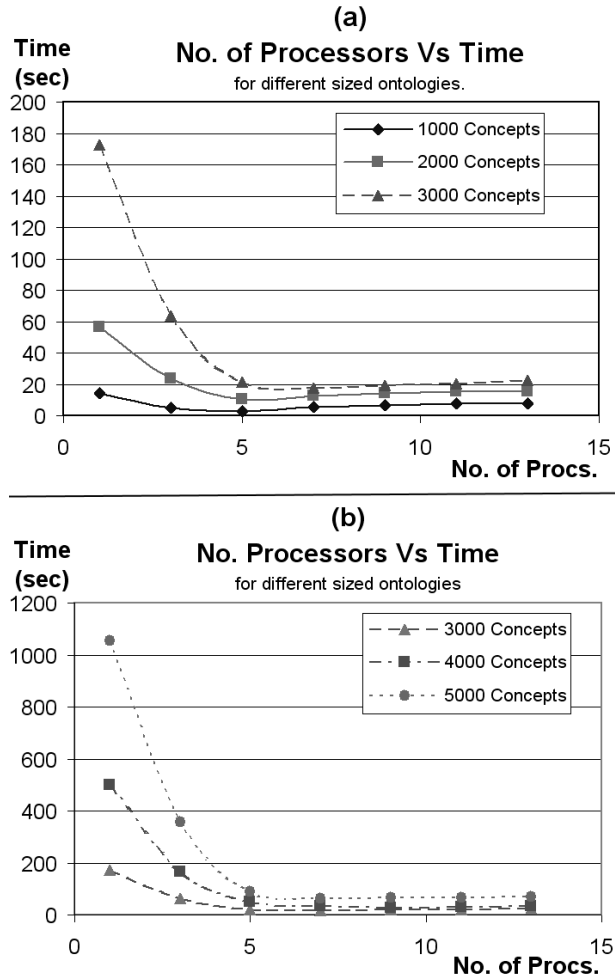
(a)

Time (sec)

No. of Processors Vs Time
for different sized ontologies.

No. of Procs.

(b)

Time (sec)

No. Processors Vs Time
for different sized ontologies

No. of Procs.

Fig. 18. SCOS test runs for (a) small and (b) large ontologies.

added to that of RCOS and other optional optimization schemes, the time taken to extract a materialized ontology view on a single processor machine would take so long that it would make the whole idea computationally infeasible. This is because the main aim of extracting materialized views of ontologies is to make the ontologies faster and easier to use on ordinary single-user machines.

The main concern with these results is that extracting a materialized ontology view from large ontologies may be out of reach for the average computer user, as they would not usually have access to a parallel processing machine to complete such an optimization task. These results do show that using a coarse-grained distribution scheme is effective for processing small and large ontologies alike. Both data and procedural level parallelism inherent in the problem domain may be cited as the main reasons for the drastic improvements in performance.

6. Conclusion and Future Work. In this paper we have demonstrated a novel approach for sub-ontology extraction from a large-scale base ontology. We have developed and implemented a distributed framework for the execution of the extraction process. The Requirement Consistency Optimization Scheme (RCOS) was divided into four sub-schemes (RCOS1–RCOS4) to distinguish the optimization scheme rules. The Semantic Completeness Optimization Scheme (SCOS) currently consists of three sub-schemes SCOS1–SCOS3, which handle various issues pertaining to semantic completeness. As mentioned previously, we have defined semantic completeness to be the inclusion/exclusion of certain *defining elements* for the ontological elements selected/deselected by the user or other application. Obviously, this notion of semantic completeness could be expanded and new rules could be specified by other researchers in the ontology domain. The distributed architecture that we have proposed and implemented is general enough to be used with other ontology-based applications. The plugin-based design of the optimization schemes as well as the distribution primitives facilitates seamless integration with other ontology applications.

The use of dynamic process management capability needs to be implemented into the system. Until the development of SCOS [11], the need for such capability within the framework did not arise as RCOS merely consists of partitioning the data to be processed based on the number of processors available. However, with the development of newer (and more complex) optimization schemes, for instance, SCOS, the number of partitions generated is a property of the structure/connectivity of the ontoGraph. As such, a capability to allocate an optimal number of processors based on the number of data partitions will be necessary for optimized execution.

Currently, we use a very coarse-grained distribution scheme. It is possible to utilize more sophisticated distribution schemes given the fact that there are no constraints to the order of execution of functionally independent optimization schemes. Also, more fine-grainedness could be introduced by parallelizing individual sub-schemes within a particular optimization scheme. However, these enhancements would only be justified if optimal performance of the extraction process is an absolute necessity. As previously mentioned, our research and the resulting distributed architecture is strongly driven by our envisaged application of the extraction process in a distributed business environment.

We have implemented and tested our distribution algorithms in an AlphaServer and Linux cluster environment, utilizing the Message Passing Interface (MPI) that provides message passing primitives suitable for distributed systems. The evaluation shows how the performance of the extraction process can be improved in a multi-processor distributed environment by employing an appropriate number of worker processors. The result of this work will be beneficial for other web applications that utilize large ontologies such as the bioinformatics area, data warehousing and medical sciences. By using our extraction process and distributed algorithms, an application can extract the necessary concepts and relationships from the base ontology and work independently on them; since the extracted sub-ontology is complete, semantically correct and independent of the base ontology.

References

[1]   Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum.-Comput. Stud.*, 43(5–6):907–928, 1995.

[2] Nicola Guarino, Massimiliano Carrara, and Pierdaniele Giaretta. An ontology of meta-level categories. In *KR*, pages 270–280, 1994.

[3] Carlo Wouters, Tharam S. Dillon, J. Wenny Rahayu, and Elizabeth Chang. A practical walkthrough of the ontology derivation rules. In *DEXA '02: Proceedings of the 13th International Conference on Database and Expert Systems Applications*, pages 259–268, London, 2002. Springer-Verlag, Berlin.

[4] Peter Spyns, Robert Meersman, and Mustafa Jarrar. Data modelling versus ontology engineering. *SIGMOD Rec.*, 31(4):12–17, 2002.

[5] Nicola Guarino and Christopher Welty. Evaluating ontological decisions with ontoclean. *Commun. ACM*, 45(2):61–65, 2002.

[6] Michel Klein, Dieter Fensel, Altanas Kiryakov, and Damyan Ognyanov. Ontology versioning and change detection on the web. In *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management*, pages 197–212, Spain, 2002. Volume 2473 of LNAI. Springer-Verlag, Berlin.

[7] Deborah L. McGuinness, Richard Fikes, J. Rice, and S. Wilder. An environment for merging and testing large ontologies. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning*, pages 483–493, San Francisco, CA, 2000. Morgan Kaufmann, San Mateo, CA.

[8] Natalya Fridman Noy and Michel C. A. Klein. Ontology evolution: not the same as schema evolution. *Knowl. Inform. Syst.*, 6(4):428–440, 2004.

[9] Carlo Wouters, Tharam S. Dillon, Wenny Rahayu, and Elizabeth Chang. Large scale ontology visualisation using ontology extraction. *Int. J. Web Grid Serv.*, 1(1), 2005.

[10] Unified medical language system, 2003. http://www.nlm.nih.gov/research/umls/.

[11] Mehul Bhatt, Andrew Flahive, Carlo Wouters, Wenny Rahayu, David Taniar, and Tharam Dillon. A distributed approach to sub-ontology extraction. In *Proceedings of the Eighteenth International Conference on Advanced Information Networking and Applications (AINA '04)*, pages 636–641, Fukuoka, Japan, March 2004.

[12] Mehul Bhatt, Carlo Wouters, Andrew Flahive, J. Wenny Rahayu, and David Taniar. Semantic completeness in sub-ontology extraction using distributed methods. In *ICCSA* (3), pages 508–517, 2004.

[13] Ian Horrocks. Daml+oil: a reasonable ontology language. In *Proceedings of EDBT-02*, pages 2–13, 2002. Volume 2287 of LNCS. Springer-Verlag, Berlin.

[14] Tim Berners-Lee. *Weaving the Web*. Harper, San Francisco, CA, 1999.

[15] D. Wollersheim and J. Wenny Rahayu. Ontology based query expansion framework for use in medical information systems. *Int. J. Web Inform. Syst.*, 1(2):101–115, 2005.

[16] Sonja Zillner and Werner Winiwarter. Integration of ontological knowledge within the authoring and retrieval of multimedia meta objects. *Int. J. Web Grid Serv.*, 1(3), 2005.

[17] Anthony Skjellum, William Gropp, and Ewing Lusk. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, second edition. MIT Press, Cambridge, MA, 1999.

[18] Ewing Lusk, Rajeev Thakur, and William Gropp. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.