UNIVERSITY OF OSLO
Department of Informatics

Improving performance of STEP/EXPRESS validation using parallel processing

Cand Scient Thesis

André Næss

2nd August 2004

Preface

This is a thesis for the Cand. Scient. degree in computer science at the Department of Informatics, University of Oslo.

I would first like to thank my supervisor associate professor *Arne Maus*, for his encouragement and support which made this work possible.

I would also like to thank my supervisor at EPM Technology, Arne Tøn, whose technical help has been invaluable.

Many thanks to EPM Technology for letting me use their office facilities, and to the staff there for much appreciated help.

Thanks to all my friends for making my time as a student a pleasant and fun time.

Thanks to SND and Kompetanseprogrammet for giving us money to buy a cluster of computers.

And finally, thanks to my parents for moral as well as financial support.

Oslo, July 2004

André Næss

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis deals with using parallelization as a performance enhancing technique in an industrial context. The problem was put forward by EPM Technology, a Norwegian company specializing in tools used to manage data stored using the STEP standard. STEP is an international standard designed to be a neutral interchange format for product data, and it is used throughout the world of engineering and manufacturing.

The STEP standard includes a data modeling language called EXPRESS. EXPRESS is used to model various domains such as electronics, mechanical engineering, construction etc. For each domain there is a set of entities as well as rules applying to these. Based on these entities it is possible to describe a product, for example a processor, a car or a house. This description is stored as a STEP model.

Our goal is to speed up the validation of models, i.e. the process of checking that the model adheres to the rules defined for the domain. The validation process can be very time consuming due to the large amount of data required to describe products and the complexity of the rules.

1.1 Overview of the thesis

In chapter 2 we define the problem and give an introduction to parallel processing and cluster computing, differentiating clusters from other parallel and distributed systems.

In chapter 3 we give an introduction to the STEP standard and the EXPRESS language used to describe the rules we wish to execute faster.

In chapter 4 we then look at the structure of STEP models and consider different approaches to breaking models down into smaller pieces that can be executed in parallel. We also describe a benchmarking methodology that we use to compare the approaches. Finally we look at the results from these benchmarks.

In chapter 5 we move on to describe how the approach selected in chapter 4 can be implemented. We describe the technology we will use, and give an outline of the architecture. We then give a more detailed description of the various components that our system requires. The main goal of this chapter is to describe a first version of the system that can be implemented fairly quickly, allowing us to study the performance of this system to pick out aspects of the implementation that need improvement.

Chapter 6 describes the environment in which we test the system, and outlines a testing methodology. We then move on to test the system, with focus on finding potentials for improvement. We also take a look at overhead in the system.

Armed with the performance results from chapter 6 we attempt to improve the system. Chapter 7 describes several improvements, showing how each affect the performance of the system. We end this chapter with the performance characteristics of the final, optimized system, and discuss some interesting aspects of these results.

Finally, chapter 8 summarizes our experiences, and gives the final conclusion of this thesis. We also discuss some things that could have been done better, and give some ideas for possible future work.

Chapter 2

Problem definition

EPM Technology is a Norwegian company which supplies products used in managing digital product data. Their tools use the EXPRESS data modeling language, which supports several international standards for product data, including ISO10303, also know as STEP. Among the products included in the *EDM Product Suite* is a tool for validating data models against one or more sets of rules, the *EDMmodelChecker*.

The process of validating a data set can take several hours, and EPM are interested in reducing this. To achieve this will attempt to use parallelization.

2.1 Express Data Manager

The Express Data Manager product suite is a set of products developed by EPM Technology to work with EXPRESS data models. The Express Data Manager is a basically a Database Management System built for working with EXPRESS data models. We will only look at one of the components, the EDM ModelChecker, as this is the part of the suite used to validate models.

As an example of a scenario we can consider automated building plan approval where the data model for a construction project is validated against rules defined by regulatory bodies like the fire department. Some example rules may look something like this:

> **Clause 2.2.7 Minimum Width No exit**, exit staircase or other exit facilities shall be narrower than the minimum width requirement as specified under Table 2.2A. The minimum clear width of an exit door opening shall be not less than 850mm
>
> **Clause 2.2.9 - Measurement of width** In the case of an exit door opening, between the edge of the door jamb or stop and the

> surface of the door when kept open at an angle of 90 degrees in the case of a single leaf door; and in the case of a double leaf door opening, between the surface of one leaf to the other when both leaves are kept open at an angle of 90degrees.

Rules like these must of course be translated into EXPRESS. There are many uses for such automated validation, and one can use them for business rules as well as engineering rules. The important point is that the models can be checked against any set of rules as long as they are written using EXPRESS.

As the size and complexity of the models grow, so does the time it takes to process them, and currently it is EPM Technology's opinion that it takes *too* long. Whenever computations take to long, there are basically three possible solutions.

1. One can increase the speed of processing, i.e. the number of operations the computer can carry out per time unit.
2. One can improve the algorithm, i.e. how the computation is performed.
3. One can increase the number of computers doing the computing, i.e. parallel processing.

The first possibility eventually reaches physical limits, and even if the speed of processors still grows year by year, it seems that the complexity of our computations follow in its heels. As for the second solution, one must presume that EPM Technology has optimized their system as far as possible, and that improvements to the algorithm can only make a minor difference.

This leaves us with the third solution, parallel processing, which is the path we wish to follow in this thesis. What we will attempt to do is to break the models down into smaller parts. The validation of these parts can then be executed in parallel, hopefully greatly speeding up the execution.

2.2 Parallel processing and cluster computing

The need for processing power seems insatiable. For every increase in processing speed, applications grow to need more. Moore's law has been holding up for decades now, but recently processor manufacturers like Intel have found it more and more difficult to increase the speed of their processors.

2.2.1 Parallel processing in general

As sequential processors reach their limit, parallel processing offers the only way to increase processing power. In it's most general sense a parallel computer is a collection of processors able to cooperatively perform a computation. This includes parallel supercomputers, distributed networks of workstations and multiprocessor workstations.

Traditionally parallel processing was applied to numerical simulations of complex systems, but we are now seeing a growing interest for parallelism in commercial application areas where processing of large amounts of data is vital. Typical examples include multimedia systems and parallel databases.

While parallel processing in some form or other has been with us since the dawn of the computing age, one approach which recently has received a lot of attention is the use of clusters.

2.2.2 Motivations for cluster computing

[Buyya99-1] provides a good list of reasons why clusters have come to be preferred over specialized parallel computers. The following is a slightly abridged version of this list:

- Workstation performance is rapidly increasing.
- Communication bandwidth between nodes is increasing.
- A cluster is easier to integrate into an existing network than a specialized parallel computer.
- Development tools for workstations are generally more mature than their counterparts in proprietary parallel systems.
- Clusters are cheaper and more available.
- It's easy to make a cluster grow. You can both easily add nodes, and upgrade existing nodes.

Today some of the most powerful computers in the world are clusters. At the time of this writing the second fastest computer in the world according to the "Top 500"[1] is a cluster of Intel Itanium computers at Lawrence Livermore National Laboratory [2]. Of the 500 systems in the list, 291 are considered clusters, and as figure 2.1 on the following page reveals the growth has been tremendous.

[1]http://www.top500.org
[2]http://www.llnl.gov/

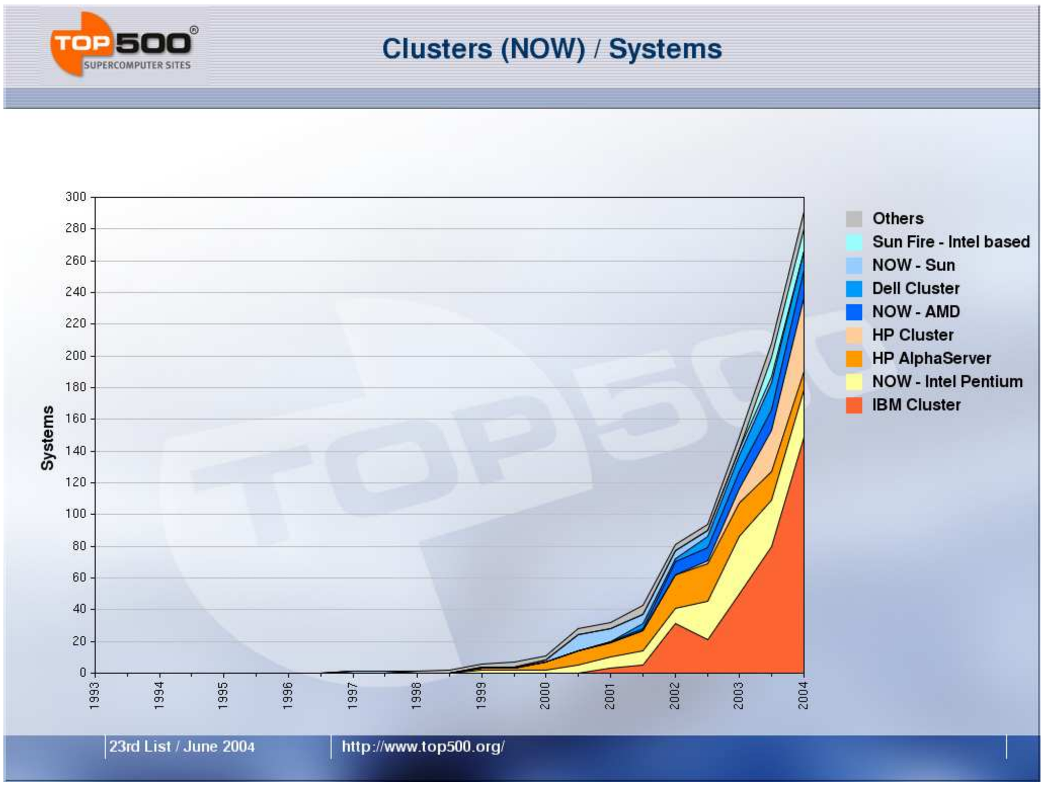

Figure 2.1: The number of entries in the top 500 list classified as clusters as of June 2004.

2.2.3 Defining clusters

But what constitutes a cluster? Gregory F. Pfister provides a straightforward definition[Pfister95]:

> A cluster is a type of parallel or distributes system that:
>
> - consists of a collection of whole computers
> - and is utilized as a single unified computing resource

But this definition doesn't make obvious the difference between clusters distributed or parallel systems. Pfister regards clusters as a "subspecies or subparadigm of distributed (or parallel) computing", and provides some insight into what differentiates clusters from parallel and distributed systems.

Clusters and parallel systems

The most important difference between a cluster and a parallel system is the fact that clusters are made from whole entities, computers. A parallel system like an SMP on the other hand is made by replicating only a part of the computer, the processor. A cluster is also more resilient to failure because of it's shared-nothing architecture, and it's generally easier to add computers to a cluster than adding processors to an SMP.

Pfister also notes that while many massively parallel multicomputers supply each processors with a memory and I/O system, there is usually less than adequate memory for each node to work as a stand-alone machine. Nor has it been common for such systems to provide access to normal operating system facilities at each node.

Clusters and distributed systems

Clusters are a bit more difficult to differentiate from distributed systems. One important difference is that a node in a distributed system has an individual identity. In many cases it's physical location is important to it's operation. For example a distributed payroll system will probably have each node store information important to the branch in which the node is located. In many cases the node can continue to function even if it's disconnected from the distributed system as a whole, there will just be some functionality missing.

In contrast to this, nodes in a cluster are anonymous. The cluster is generally viewed as a single system (hence “single unified computing resources” in the definition) , with the nodes acting merely as “cogs in the system”. It is however not uncommon to have certain nodes in the cluster perform special functions.

Chapter 3

An overview of STEP/EXPRESS

In this chapter we give a brief overview of the STEP standard and the EXPRESS language. The EXPRESS language is object oriented, but the terminology used is somewhat different from the OO norm, so we will introduce this terminology and use it throughout this thesis.

3.1 STEP

In any sort of manufacturing, data about products must be kept somehow. The representation of such product data has evolved from physical models via technical drawings to digital representation manipulated by todays sophisticated CAD and CAM tools. But different CAD/CAM tools are rarely compatible, and so a company that wishes to maximize benefits from the use of such tools must enforce a common set of tools within their organizations.

For larger companies this may not be straightforward, and as companies form joint ventures it becomes even harder. Also problematic is the fact that the product data should be usable throughout the supply chain.

STEP—the STandard for the Exchange of Product model data—is a comprehensive standard for sharing of digital product data. The following quote serves as the introductory paragraph to every part of the ISO 10303 standard:

> ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. The objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product, independent from any particular system. The nature of this description makes STEP suitable not only for neutral file exchange but also as a basis for implementing, sharing product databases, and archiving.

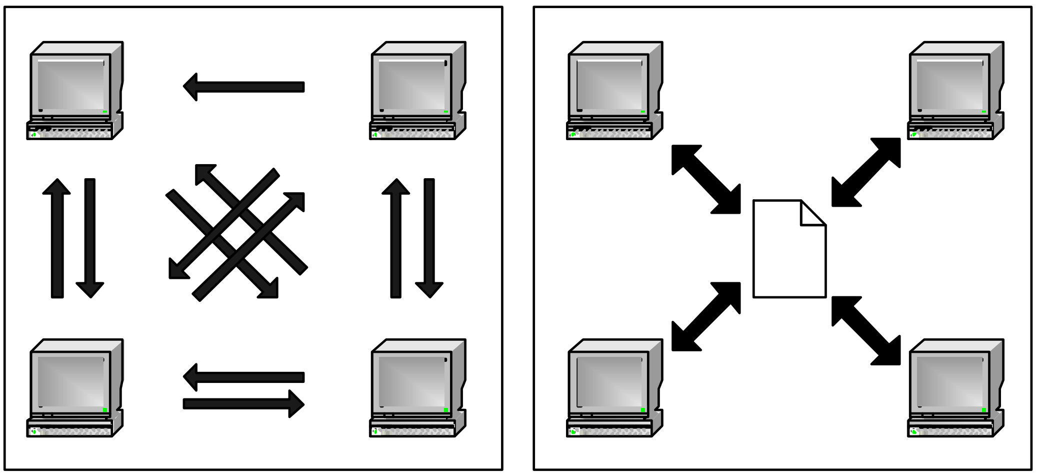

Figure 3.1: Data exchange by using system-to-system interface (left) and a neutral interchange format (right).

Digital data can be used in many different contexts throughout the enterprise, and consequently by many different computer systems. These computer systems have their own legacy formats, so the need for conversion arises. One could of course simply define interfaces between the systems, but as the number of systems grow this becomes an increasingly difficult approach as figure 3.1 illustrates.

The solution is to use a common format which acts as a data backbone between all the different systems. Each system must have an interface to this backbone, and need only convert data to and from the common format. This is precisely what the STEP architecture provides. STEP is defined through a large collection of documents, covering a large number of application areas. Some examples of STEP in action include:

- the use of STEP to support the exchange of digital mock-ups between Boeing and its engine suppliers in the process of integrating engines and their complex plumbing into an airliner, replacing expensive physical mock-ups.

- the use of STEP Draughting application protocol for the exchange of technical drawing between Japanese companies and the Ministry of Construction.

- the use of STEP in Singapore to facilitate electronic submission, processing and approval of building project documents.

STEP and the Norwegian Navy

In 2005, the Norwegian Navy receives it's first new frigate of the Nansen class from the manufacturer. This is one of five new frigates, and the total cost of the project is about 20 billion Norwegian kroner.

The Norwegian Navy chose to use STEP heavily in this project. Together with the frigates themselves comes a huge amount of data such as manuals, specifications and maintenance descriptions. Information about every part of the frigates must be available for maintenance purposes, and every part has a unique serial number used to identify it.

All this data is stored using the STEP standard, and the data will be used throughout the entire lifecycle of the frigates. Earlier such data have been maintained using a combination of paper and digital media. The digital information was then stored in proprietary formats rather than a neutral format such as STEP.

3.2 The design of STEP

STEP was designed as the successor to various exchange standards, but in addition to exchange it also added support for sharing and archiving of product data.

3.2.1 Sharing versus Exchange

Data exchange is the transfer of information from one software system to another. The data being exchanged is a snapshot of the information at the originating system. A good example of data exchange is when you receive your monthly bank statement. These are characteristics of data exchange:

- Initiated by data originator
- Transformed in a neutral format
- Content determined by discrete event in time
- Redundant copy of data created.

The challenge lies in interpretation. Information coming from system A must be understood by system B, otherwise there is no point to the exchange.

Data sharing on the other hand provides a single logical information source, to which multiple software systems have access. An example of this is internet banking where the customer accesses his accounts in real-time and works directly with the information source. Data sharing can be characterized by the following:

- Initiated by data receiver
- Data on demand
- Data access levels embedded in protocol
- Appears as a single data source
- Read (real-time) and update capabilities

Data sharing helps alleviate problems of version tracking and ownership management. When data is exchanged, any changes made to copies held by third parties must be merged back into the master copy. Data sharing is clearly an ideal to strive for.

3.3 STEP building blocks

STEP is a comprehensive standard, comprised by a large number of documents covering the gamut of engineering practices. The standard is constantly growing as new applications are embraced. To best facilitate data exchange, sharing and archiving, the standard separates data definition, data format and the data access language.

STEP has four major parts, as can be seen from figure 3.2. The description methods; the EXPRESS modeling language in it's several variants will be covered later in this chapter. We will not discuss the conformance testing component as it is not really relevant to this thesis.

3.3.1 Implementation methods

Implementation methods are "standard implementation techniques for the information structures specified by the only STEP data specification intended for implementation, application protocols." An implementation method defines a mapping between STEP data constructs and the implementation method.

Early in the standards effort four levels of implementation methods were described, from basic files to state-of-the-art multi-user knowledge database management systems. The basic file mechanism was the first to be realized, standardized as "ISO 10303-21 Clear Text Encoding of the Exchange File", also known as simply "Part 21" it specifies how the exchange file should

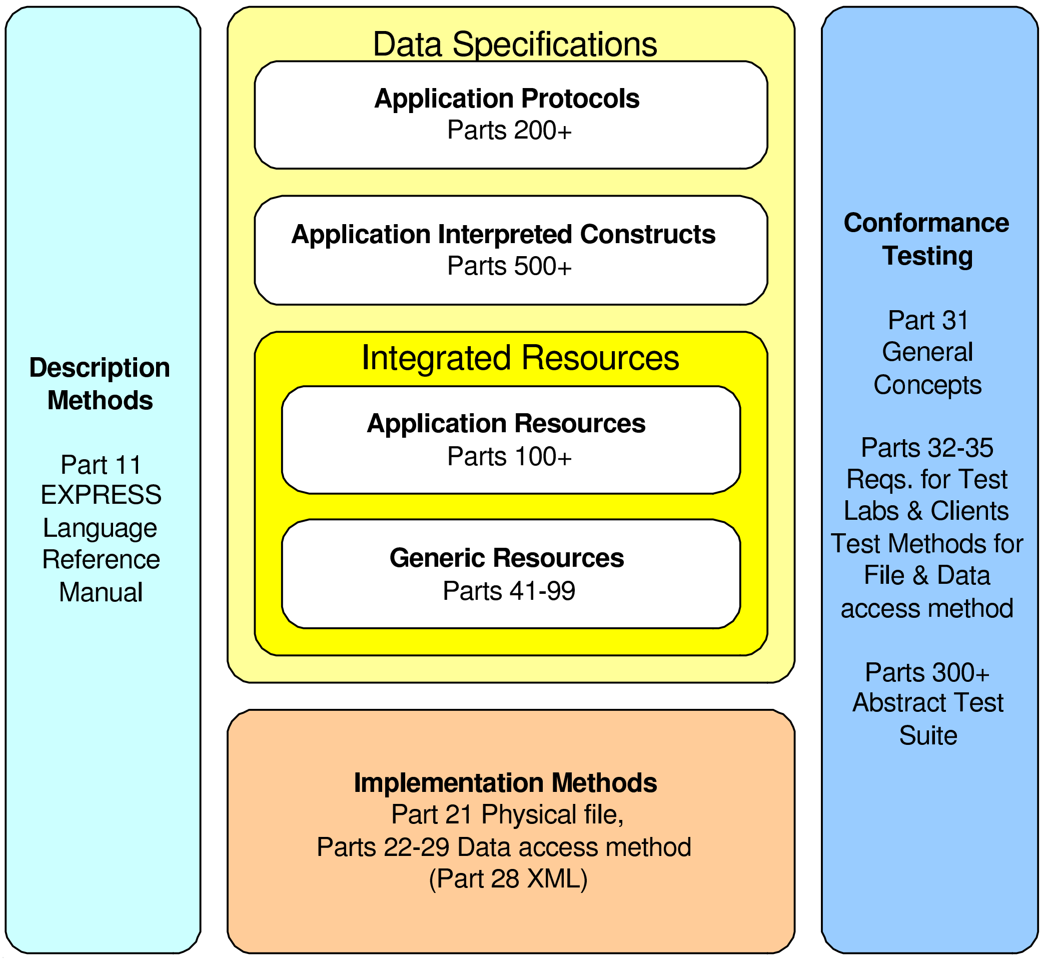

Figure 3.2: Overview of the STEP documentation structure (adapted from [Kemmerer99] page 49).

be derived from EXPRESS. However, as the name implies this offers little support for data sharing.

The desire for supporting sharing fueled the efforts leading to SDAI—Standard Data Access Interface. The intended purpose of SDAI is to provide an Application Programming Interface (API) to data described by an EXPRESS information model. SDAI is in many ways similar to interfaces to traditional database management systems such as SQL. However STEP data often has the shape of networks, and the SDAI interface reflects this, making traversal of links the predominant access method. SDAI is a family of standards, including language bindings for Java, C, C++, and IDL.

3.3.2 Data specifications

Data specification is split into four different series, but there are basically three basic types of data specification: integrated resources, application protocols, and application interpreted constructs.

Integrated resources

Integrated resources are the basic building blocks that can be used by any product description. There are two types of integrated resources, namely generic resources and application resources. Generic resources are common semantic elements, e.g. Cartesian point or date. The objective of integrated generic resources is to support the common requirements of all the different application areas.

The second type, application resources, represents concepts that are common to many application areas. Examples of such resources include drawing sheet revision, drawing revision and dimension callout. These may be used by any application that includes drawings.

Application protocols

Application protocols (APs) are the heart of STEP, and it's architecture is designed primarily to support the development of APs. APs are implementable data specifications, and include an EXPRESS information model tailored to the application area in question. APs can be implemented using any of aforementioned implementation methods. The documentation of an application protocol adheres to strict regulations.

Since application protocols are specific to an application area, it is important to define their scope precisely. This is achieved through four different components:

- The description of the functionality (AAM, Application Activity Model)
- An application-oriented reference model from a user's point of view (ARM, Application Reference Model)
- Representation of the reference model through objects from the Integrated Resources as implementation view (AIM, Application Interpreted Model)
- Implementation guidelines and conformance conditions for implementations

The AAM is developed to establish an understanding of the application tasks, processes and the information flow of the application domain. This then serves as a basis for the development of the ARM. The developers attempt to capture the information most relevant to the application area—what the AP must be able to "say". The application interpreted model (AIM) specifies a subset of the integrated resources to use with the AP.

Schemas and models

The AAM, ARM and implementation guidelines are primarily used during the development of the application protocol. The final product which is what users will work with is the AIM. To develop the AIM, the knowledge discovered through the ARM and AAM are used. The AIM is basically a schema defining entities and constraints for a particular domain. The AIM is written in EXPRESS, allowing rules and constraints to be expressed programmatically.

As an example consider "ISO 10303-210 – STEP Application Protocol for Electronic Assembly, Interconnect, and Packaging Design" informally referred to as "AP210 STEP for Electronics". Someone working on designing a complicated electronic device, such as a CPU, might do this in a CAD tool. The AP210 schema defines the necessary entities so that the CAD tool can save the design using one of the implementation methods (in practice this usually means a flat file). The CPU is thus stored as a population of the AP210 schema, and this population is usually referred to as a model.

It is these models that we are going to deal with in this thesis, and because the schemas are written in EXPRESS, we will here give a brief introduction to EXPRESS.

3.4 Introduction to EXPRESS

EXPRESS is a textual *computer interpretable* language, and is used as the information modeling language by the STEP standard (it is, however, orthogonal to STEP as such). The use of EXPRESS as modeling language has several advantages, but first we should have a look at other options that were considered.

3.4.1 A brief history of EXPRESS

Some—by now at least—ancient modeling languages include ADM (Associative Data Modeling) and ER (Entity-Relationship). These influenced more recent languages like NIAM (now known as ORM), and UML. The choice of modeling language was considered to be crucial to the STEP standards effort, and at least three languages were under consideration at some point. These were IDEF0, IDEF1X and NIAM. We won't go into details about any of these.

The developers of STEP realized early on that they needed a language which would support automatic processing of models, preferably by well understood tools like a parser, as the models would surely grow large. IDEF1X failed because it was strictly graphical in nature. It was also weak on specifying constraints. NIAM is strong on constraints, and does have a textual representation but the diagrams were considered awkward and difficult to produce. However in [Kemmerer99] p. 66 the following (unattributed) quote appears: "A fair amount of rationalization and politics may also be blamed on the desire to invent something new for it's own sake." Presumably familiar words to anyone who has been involved in any sort of development...

In short, the STEP committee found the existing languages lacking in some respect, and decided to develop their own language, based on existing work by another group.

EXPRESS has provides STEP developers with several advantages: It eliminates ambiguity, it eases the process of validating models, it's user-friendly and it is possible to generate software directly from EXPRESS. However, over the course of time some flaws and weaknesses have been discovered as well. Of course, one of the ideas behind standards is that they shouldn't change too often, so major improvements to EXPRESS shouldn't be expected.

3.4.2 EXPRESS dialects

EXPRESS is an object oriented data modeling language consisting of language elements for precise, unambiguous data definition and constraint specification. It has a Pascal-like syntax, and is procedural in nature. [Kemmerer99] (p. 133) notes that EXPRESS is a language developed by and for engineers, something that has made the language conservative in some respect. Constraints are describe using functions and procedures, whereas mathematicians would prefer a declarative approach.

EXPRESS is standardized in ISO10303-11, but there are several dialects serving different purposes:

- EXPRESS (ISO10303-11) — The complete textual notation.
- EXPRESS-G (ISO10303-11) — A graphical notation similar to UML, a subset of EXPRESS.

- EXPRESS-I (ISO10303-12) — Instantiation language.
- EXPRESS-X (ISO10303-13) — Mapping and view language.
- Several proprietary dialects.

EXPRESS supports many well-known concepts from programming, like loops, conditional branching, functions, procedures (methods) and a large set of common operators to work on the built in data types. We won't go into details since we expect the reader to be familiar with these concepts. Instead we shall look at the most important elements of the EXPRESS language, the data modeling constructs.

3.4.3 EXPRESS Language elements

We will here give an overview of the various language elements usable within EXPRESS, and in most cases give examples of their use.

Schemas

Schemas were introduced as the most important document in an application protocol. Schemas are used as a grouping/partitioning mechanism for an area of interest. We can think of it as a sort of 'module'. A schema defines a scope for all it's contained declarations. The modular 'building-block' nature of schemas supports simple reuse.

```
SCHEMA electronic_assembly;
  ... declarations
END_SCHEMA;
```

One can interface between schemas, for example to allow items declared in 'foreign' schemas to be used in the 'current' schema. To interface with other schemas one can use either the `USE FROM` declaration, which allows one use `ENTITY` and `TYPE` declarations taken from the 'foreign' schema, or the `REFERENCE FROM` declaration which allows the use of `CONSTANT`, `ENTITY`, `TYPE`, `FUNCTION` and `PROCEDURE` declarations.

Types

A type declarations creates a new 'defined type' based on an 'underlying type' (built-in type). A type may thus be a simple alias for the underlying type (representation) or in some cases a restricted version. This is used to infuse the model with more semantics, and hence improve maintainability. For example, one may want the type 'label', which is represented as a string:

```
TYPE label = STRING; END_TYPE;
```

To create a new type which is a restricted version of the underlying type one can use a WHERE rule. E.g.:

```
TYPE age = INTEGER;
  WHERE SELF >= 0;
END_TYPE;
```

EXPRESS also support `SELECT` types. A `SELECT` type is basically the same as a union in C and variant records in Pascal. Finally there are enumerated types, the EXPRESS equivalent of C enums. The classical example of month names goes like this:

```
TYPE month = ENUMERATION OF
  (January,February,March, April, May, June, July,
   August, September, October, November, December);
END_TYPE;
```

Entities

An entity is the EXPRESS version of a class. An entity defines a domain of values by their common attributes and constraints. As with classes in other object oriented languages, EXPRESS supports the notion of inheritance, both single and multiple. EXPRESS also supports derived and inverse attributes, uniqueness rules, and WHERE rules. Entities can also contain procedures, which are analogous to methods in Java.

Entity attributes are analogous to attributes/properties in OOP, in addition to the well-known explicit attributes, EXPRESS supports the concept of derived attributes. Derived attributes are simply attributes that can be somehow derived from the other attributes of an entity. For example, given a person's date of birth and the current time one can derive his age. Inverse attributes are used to declare that attributes are related to other attributes (either in the same entity or in some other entity). Let's look at two examples[1]:

```
SCHEMA Geometry;
  ENTITY Circle;
      x      : REAL;
      y      : REAL;
```

[1]Taken from slides provided by EPM Technology

```
      Radius : REAL;
    DERIVE
      Area   : REAL := PI*Radius**2;
  END_ENTITY;
END_SCHEMA;

ENTITY Female
  SUBTYPE OF(Person);
    Husband : OPTIONAL Male;
END_ENTITY;

ENTITY Male
  SUBTYPE OF(Person);
  INVERSE
    wife : Female FOR Husband;
END_ENTITY;
```

Attributes can be redeclared in subtypes according to the following principles:

- The domain of an inherited attribute can only be specialized, i.e. restricted rather than expanded.
- An explicit attribute can be changed to a derived one.
- An optional attribute can be made mandatory in a subtype, but not the other way around.
- The bound specification of LIST, BAG or SET may be constrained, ARRAY bounds may not be changed.

Entity constraints

Local rules (constraints) are assertions on the domain of entity instances and applies to all instances of that entity type. There are two types of rules: the uniqueness rule, which define a uniqueness constraint on individual or combinations of attribute values for all instances of this entity *in a population* and `WHERE`-rules, which constrain the values of attributes for every entity instance. `WHERE` rules must evaluate to either a logical value[2], or indeterminate.

[2]LOGICAL is an atomic data type in EXPRESS and is basically BOOLEAN + the value unknown, i.e. it's domain is true, false and unknown

Furthermore `WHERE` rules must refer to attributes declared within the entity or any of it's supertypes.

As an example, consider a representation of Person, having an identifier Id. The Id must begin with a letter in either upper or lowercase. This can be expressed as follows:

```
ENTITY Person
    Name : STRING;
    Id   : STRING;
  WHERE Legal_id :
    (Id[1] >= 'a' AND Id[1] <= 'z') OR
    (Id[1] >= 'A' AND Id[1] <= 'Z');
END_ENTITY;
```

Note that `WHERE` rule is named, this allows one to refer to a particular `WHERE` rule.

Constants

Constant declarations are used to declare named constants, whose scope is that of the immediately enclosing function, procedure, rule or schema. You can declare several constants within a CONSTANT declaration, for example:

```
CONSTANT
    Thousand : INTEGER := 1000;
    Million  : INTEGER := Thousand**2;
END_CONSTANT;
```

Functions and procedures

Functions and procedures encapsulate code, a `FUNCTION` is a function in the mathematical sense, i.e. it is an algorithm which operates on arguments and produces a single result value of a specific type, without any side-effects. A `PROCEDURE` is an algorithm which operates on given arguments to produce the desired end state. Procedures are the EXPRESS equivalent of methods in Java, whereas functions are like static methods, except that they don't have to be declared within an entity.

Rule declarations

As we have seen, `WHERE` rules can be used both in type and entity declarations. One can also declare rules using the `RULE` declaration. Such rules apply collectively to the entire domain of an entity type, or to instances of more than one entity type.

If you are familiar with SQL, you may already be sensing a certain similarity: a `WHERE` rule in a `TYPE` declaration compares roughly to a column constraint in SQL. A `WHERE` rule within an `ENTITY` declaration mirrors the SQL concept of table constraints, and a `RULE` declaration is similar to a SQL database constraint.

The example used with entity constraints can be recast in terms of a `RULE` declaration. This rule expresses the fact that there must not exist a Person whose Id starts with anything but a letter, uppercase or lowercase:

```
RULE AllValidId FOR(Person)
  WHERE AllValid :
    SIZEOF( QUERY( p <* Person |
      (p.Id[1] >= 'a' AND p.Id[1] <= 'z') OR
      (p.Id[1] >= 'A' AND p.Id[1] <= 'Z'))) = 0;
END_ENTITY;
```

A rule declared this way is usually referred to as a global rule. Also notice the use of the query operator, this is a fairly common idiom in the world of EXPRESS. A query is executed, and the size of the returning aggregate is compared to zero or one, in other words a awkward way of saying “There exists” or “There does not exist”. EXPRESS unfortunately does not have built in functions with the meaning of the predicate calculus quantifiers.

The Query operator

EXPRESS has a query operator which allows one to formulate queries that apply a given logical expression against all the elements of a source aggregate, yielding a new aggregate with all those elements of the source aggregate for which the logical expression evaluated to true.

Furthermore, one can operate on aggregate using use well-known operators such as intersection (*), union (+), difference (-), subset (<=) and superset(>=).

3.5 A summary of terminology

The terminology of STEP/EXPRESS differs somewhat from the standard OO terminology. For simplicity and future reference we shall here summarize the terminology we will use in this thesis.

An *entity* is the STEP equivalent of a class. A *schema* is a declaration of entities, and the constraints that apply to them. A schema is much like a database schema, or in OO terms a collection of class declarations.

The constraints that apply to a given entity are called *local rules*, whereas a rule declared on it's own is called a *global rule*. Finally, a *uniqueness rule* is the EXPRESS equivalent of a candidate key in a database.

A *model* is a *population* of a given schema, i.e. a collection of *entity instances*, or *objects*. We will use the terms entity instance and object interchangeably.

For our purposes the most important part of an application protocol is the AIM. The AIM is what we refer to as the schema, and when we refer to a particular application protocol, what we mean is the AIM, i.e. schema, defined by this application protocol.

Chapter 4

Designing a parallel algorithm

Designing a parallel algorithm is a difficult task, and there are no simple recipes to follow. However, a methodical approach to the design process can help us maximize the number of options we explore. One such approach is outlined in [Foster95], and this methodology is also cited in [Buyya99-2]. What Ian Foster suggests, is to divide the design process into four distinct stages:

1. *Partitioning* The problem is decomposed into smaller task suitable for parallel execution.
2. *Communication* The communication structures and algorithms needed to coordinate task execution is defined.
3. *Agglomeration* The task and communication structures from the previous two stages are evaluated, and tradeoffs based on cost/benefit or development time are made.
4. *Mapping* Tasks are assigned to processors in a manner which attempts to balance the competing goals of maximizing processor utilization and minimizing communication costs.

A slightly different approach can be found in [Culler99]. He uses the term task to mean an arbitrary piece of work that is the smallest unit of concurrency. A process is an abstract entity that performs tasks, and a parallel program is thus composed of cooperating processes that each perform a subset of the tasks. The following is a brief outline of Culler's approach:

1. *Decomposition* of the computation into tasks.
2. *Assignment* of tasks to processes.

3. *Orchestration* of the necessary data access, communication and synchronization among processes.

4. *Mapping* or binding of processes to processors.

The assignment of tasks to processes is essentially the same as agglomeration, as the objective is to merge tasks into larger units. Hence the main difference between the two approaches is that the communication and agglomeration steps have been interchanged. However Culler uses the term partitioning to mean the first two steps taken together.

We will not rigorously follow any of these methodologies, but they can serve as a useful guide to our design process. Foster also remarks that while his list presents algorithm design as a sequential process, it is in fact a "highly parallel process, with many concerns being considered simultaneously". We should also point out that we don't expect our design and analysis to be exhaustive, and as we begin implementation we will certainly run into considerations that influence the final design.

4.1 Bottom-up versus top-down

At a very early stage in this project we discussed how to approach the problem, and the decision we made here does have an impact on our final design. There are basically two approaches, we can refer to them as bottom-up and top-down, other suitable terms are glass-box and black-box.

The important difference is that in bottom-up/glass-box approach we would look at how the EDM engine works internally, and attempt to parallelize its execution as a whole. Using a top-down or black-box approach means that we look at how the EDM ModelChecker performs a validation, and try to split this process into discrete tasks that can be performed separately, to do this we only use the interface exposed by the EDM engine.

The main benefit of a bottom-up approach is that we would parallelize the database engine itself, resulting in a much more general solution. It would allow for the parallelization of all the operations the EDM system offers rather than just the *ModelChecker*. The main drawback is that we will have to get our hands dirty with the nuts and bolts of what has so far been a simple black box. This could prove to be a major challenge, depending on how the kernel is actually implemented, for we must not forget that it was not written with parallelization in mind.

Due to the limited time available for this project we had to make an early choice, and seeing as the bottom-up approach would most likely be much more complex to implement we chose a top-down approach. This will obviously limit our freedom when designing a parallel solution, but we still hope that it will be a viable approach. With that in mind we begin the first phase.

4.2 Partitioning

The goal of this stage is to decompose the problem into fine-grained tasks to allow for maximal flexibility and freedom in designing our algorithm. Since we are working with an existing piece of software we have a sequential algorithm to base our analysis on, and this algorithm can be stated very simply:

```
for each object in model:
    for each constraint in constraints:
        validate(object, constraint)
```

This is obviously a highly idealized version, and the internals of the `validate` function may be fairly complex. But what we do know is that each iteration of the inner loop is independent. And thus we should be able to perform all the calls concurrently. In [Buyya99-2] a presentation of various well-known parallel programming paradigms can be found, and one of these is:

> *Iterative decomposition*: Some applications are based on loop execution where each iteration can be done in an independent way. This approach is implemented through a central queue of runnable tasks, and thus corresponds to the task-farming paradigm.

The task-farming paradigm is also known as master/slave or manager-/worker, and it is conceptually very easy to understand, and should also lead to a fairly simple implementation. It particularly simplifies communication, which is reduced to sending “orders” from the master to the slaves and sending work results from the slaves to the master. There is no need for communication between the slaves. The drawback is a lack of scalability, as both the load on the network and on the master increases with the number of slaves. This can be remedied by dividing the slaves into groups and have one master manage each group, with one computer acting as a sort of master for the masters.

The master/slave approach indeed seems fitting to our problem, but a typical Express model usually contains tens of thousands to millions of objects, and as we know there are nine different constraints available. This means that the number of tasks is many orders of magnitude larger than the number of processors we expect to support. Because of this we need to consider different ways of composing the tasks into bigger units. To do this we must look at the structure of a typical model, and based on this define what we will refer to as partitioning axes.

To evaluate the suitability of a decomposition Foster provides us with a four point checklist. We reproduce the checklist here for reference.

1. Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, you have to little flexibility in subsequent design stages.
2. Does your partition avoid redundant computation and storage requirements? If not, the resulting algorithm may not be scalable to deal with large problems.
3. Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
4. Does the number of tasks scale with problem size? Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks. If this is not the case, your parallel algorithm may not be able to solve larger problems when more processors are available.
5. Have you identified several alternative partitions? You can maximize flexibility in subsequent design stages by considering alternatives now. Remember to investigate both domain and functional decomposition.

Global rules and uniqueness rules

The simplified validation algorithm hides some important complications. In addition the local constraints applying to each instance, there may also be global rules. Global rules are basically pieces of code that perform some sort of computation against an entire population. The API available to us does not give us any way of partitioning the execution of these. For our purpose a global rule must be seen as a single task to be executed. The same goes for uniqueness rules.

4.2.1 Available partitioning axes

Splitting the validation into tiny tasks that validate a single instance immediately seems a little too extreme as the number of instances in a model can be in the millions. We therefore want to reduce the number of tasks, and to achieve this we need to look at other ways of decomposing the validation that merge the validation of several instances into single tasks. We will basically perform partitioning and agglomeration in a single combined step. We refer to a way of decomposing the data as an axis, and axes should be combinable.

However, having chosen a top-down approach we must work with the API exposed by EDM. This means that we cannot validate instances of different entities in a single operation. We can however validate one or more instances of a given entity, and for each instance or group of instances we can test the different rules separately. It is also possible to validate the entire model against one or more rules.

A given schema often contains many hundred entity declarations, and as we just saw the validation of a model can be split into the separate validations of individual *entity extents*. For example, consider 4.1, if a schema declares the entities Person, Car and House, and there are 6 instances of each, we can split the validation into three tasks validating 6 instances each. We will refer to this axis as the E-axis.

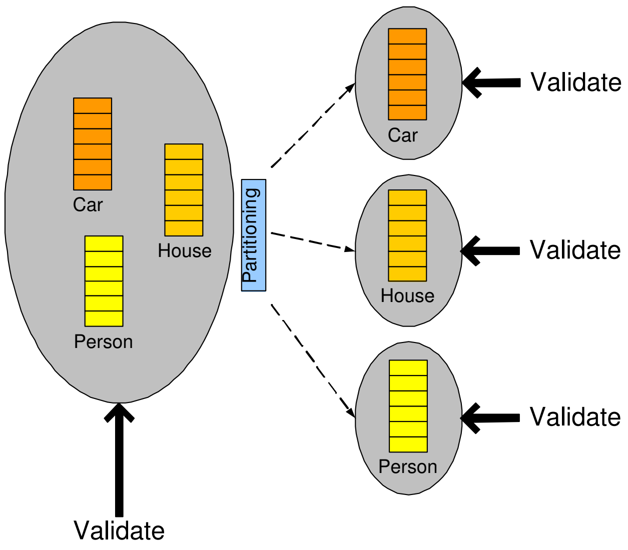

Figure 4.1: Partitioning the model into separate entity extents

Given an entity extent we can further partition it into smaller pieces. To illustrate, if we have an extent of 16 cars, we can split this into 4 operations of 4 cars each, like in figure 4.2 on the next page. Since this is a partitioning of the *population* of an entity we refer to this as the P-axis.

Finally, when validating a single instance there are eight types of constraint we need to test. It is possible to test each such constraint by itself

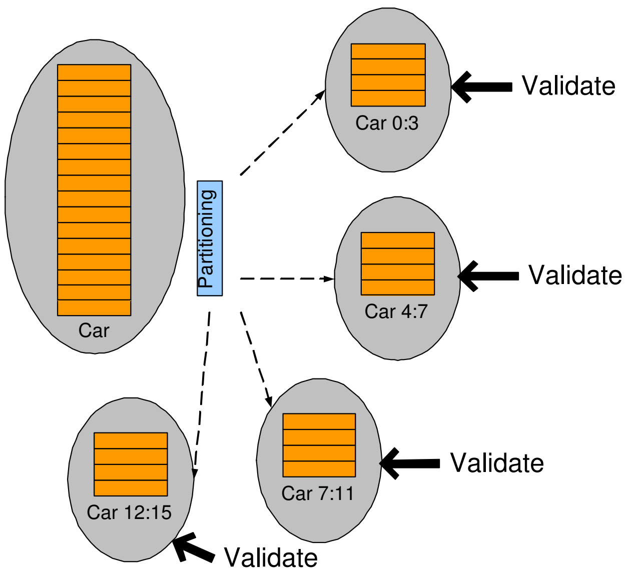

Figure 4.2: Partitioning the entity extents into subpopulations

using the available interface. We will think of these constraints as *rules*, and hence refer to this as the R-axis. We must however stress that global rules, although they are called rules, are not part of the R-axis. During a validation global rules must be validated by themselves, one by one. This note is also valid for uniqueness rules because a uniqueness rule is a constraint that applies to a collection as a whole. In fact, any uniqueness rule can be expressed as a global rule.

Based on these axes we can define some partitioning schemes. We begin by noting that a scheme based purely on either the E or R axis seems to be a bad candidate, based on Foster's fourth point. These schemes will offer a rather limited number of tasks, and — more importantly — the number of tasks is statically defined by the schema, and hence won't scale with the model size[1]. The P-axis doesn't suffer from any of these problems, but in order to apply it we are as we just saw forced to apply the E-axis first. This is because we have no other way to define subpopulations than through partitioning entity extents into smaller collections of instances.

This leaves us with just a few possible combinations. We can begin with the E axis and split this further into subpopulation using the P-axis, some-

[1]This is not entirely true, as empty entity extents will not be tested, meaning that the number of tasks for two models based on the same schema won't necessarily be the same. However, both schemes define an upper limit that does not depend on the model size, and hence they do not scale

thing we will refer to as E+P. By further partitioning this scheme using the R-axis, i.e. validating each type of constraint separately, we get the E+R+P schema. Finally, by starting with the E axis and again using the R-axis we get the E+R scheme.

To summarize, we now have three different schemes to try. Common to all these is the fact that they define the validation of individual global rules and uniqueness rules as separate tasks. This latter set of tasks is always statically defined by the schema. The other tasks are a result of the partitioning scheme being applied and the model in question. To find out which partition scheme fares the best we need some empirical data, and we now turn to this analysis.

4.2.2 Benchmark methodology

To measure the performance of the selected schemes we have written a benchmarking program. This program use the EDM C-interface to access and validate a given model. The program must be told what partitioning scheme to use, and based on the partitioning scheme the necessary tasks are created and executed.

To measure execution time we use the Unix system call `gettimeofday`. We call it once as we are about to perform a single validation task, and then a second time once the validation is done. This way we avoid measuring anything else than actual execution time. The resolution of `gettimeofday` is microseconds, but we will only use a resolution of milliseconds in our computations.

We must not forget that `gettimeofday` measure wall clock time rather than CPU time. To ensure that the numbers are valid regardless of this fact we will execute the tests on a computer running Linux with all nonessential background jobs killed to avoid interference.

The measured execution times are written to a file. We can then process this file in various ways to produce some interesting statistics:

- A list of all the task performed as well as the time and the percentage of total execution time spent on each.
- A theoretical distribution of the tasks on computer clusters of varying size, and it's impact on execution time.
- Minimal, maximal and average execution times.
- The time spent on each instance.

Ensuring measurement validity

All the values in our statistics are computed. In particular the total execution time is computed by summing up the time for each task. This has the potential of introducing more errors due to rounding. We therefore need to check if our computed values do in fact reflect reality. To do this we use conditional compiling to create a second version of the benchmark program that excludes all output code as well as all time measurements.

To see if our values represent reality we can see if the total execution time for this program is close to the execution time we have computed. To time the execution of our benchmark tool we use the unix utility `time`. `time` prints out the total execution time (wall-clock time), the actual CPU time, and the distribution between user and kernel. It also shows the percentage of the CPU that our tool was given, a value that should be as close as possible to 100% if we want to rule out interference from other programs or I/O, as this means that the program had the CPU to itself. Each validation is executed three times, and the average execution time is used.

The initial results were surprising, and revealed a bug in our code. For most models the results were good, with very little difference between computed and measured times. For the models based on the IFC2x2 schema however we found differences as large as 27%. The IFC2x2 based models generally consist of a large number of small tasks, and so we started to suspect that the problem was due to a rounding error when translating from microseconds to milliseconds. A majority of the tasks execute in less than 10ms, and so if we are truncating the numbers rather than rounding them we should on average get a 0.5ms error for all tasks!

A quick calculation showed that the difference we were seeing did in fact come close to this expected value. A look at the code revealed that the translation from microseconds to milliseconds was performed using integer division. We changed this to use floating point division, yet retaining the millisecond precision. The result was that the difference for a model called house2x2 using E+P dropped from 27.8% to 0.1%, and in the case of E+R+P from 26.3% to 1.3%. We now have an average difference between measured and computed value of 1.2% with a median of 0.5%. Based on this we are fairly confident that our computed values can be used as a foundation for further analysis.

As a an extra precaution we also measured the time it takes to performs the necessary calls to `gettimeofday`, and this shows that 100,000 calls can be performed in less than 10ms, in other words its contribution should be negligible.

Partition size

One of the challenges with the P-axis is that we must choose a partition size. By partition size we mean how many instances to validate in a single task. Based on point 3 from Foster's list the optimal size would be one that made as many tasks as possible equally sized. How to achieve this is however not a simple question, and the answer will probably rely on the analysis we are about to perform. We therefore choose to use a fixed partition size of ten instances, and postpone a further discussion of this problem till we have more information.

4.2.3 Metrics

To compare the quality of our selected partitioning schemes we need some metrics. The most important quantity is the speedup factor, i.e. how much faster a parallel version is compared to it's non-parallel counterpart. Given that a sequential execution takes T_{seq} and that the same validation takes T_{par} using parallel execution, we define the speedup factor S simply as

$$S = \frac{T_{seq}}{T_{par}} \tag{4.1}$$

Other interesting metrics are the amount of overhead involved, and the efficiency of a solution. The amount of overhead will become more important as we look at communication. Efficiency, while an important factor in a cost/benefit analysis is probably not important at this point. What we want is to figure out which partitioning gives us the most speedup, the other metrics will come into play later in the development.

We should also mention that while we will look at clusters ranging from 1 to 32 computers, the most important sizes are probably those between 4 and 16. We don't expect that many of EDM clients will want to invest in larger clusters, and so extra weight will be put on the results for these sizes.

What sort of speedup are we looking for then? While a near perfect speedup is desirable, smaller speedups may be of great interest. If for example a company has a computation that executes in 24 hours, a speedup of 3 will make this computation execute in 8 hours, meaning that they suddenly can run it overnight, which might be precisely what they require. If they need a cluster of 16 computers to achieve this speedup, then they might be satisfied, even though the solution is not particularly efficient.

Now it's time to examine how the different partitioning schemes stack up against each other.

Cml%	%	Time	Validated what?
13.6%	13.6%	21258	IFCAXIS2PLACEMENT3D-LOCAL_RULES
25.2%	11.7%	18264	IFCAXIS2PLACEMENT3D-AGGREGATE_DATA_TYPE
36.9%	11.7%	18252	IFCAXIS2PLACEMENT3D-REQUIRED_ATTRIBUTES
48.6%	11.7%	18251	IFCAXIS2PLACEMENT3D-ATTRIBUTE_DATA_TYPE
60.2%	11.7%	18247	IFCAXIS2PLACEMENT3D-AGGREGATE_UNIQUENESS
71.9%	11.7%	18239	IFCAXIS2PLACEMENT3D-ARRAY_REQUIRED_ELMS
83.5%	11.6%	18178	IFCAXIS2PLACEMENT3D-AGGREGATE_SIZE
86.6%	3.1%	4884	IFCPLACEMENTNOTSHARED-GLOBAL_RULE
89.2%	2.6%	4044	IFCPROPERTYSINGLEVALUE-AGGREGATE_TYPE

Table 4.1: The nine most time consuming tasks performed when validating the *bygga* model using the E+R scheme.

4.2.4 E+R, too simple?

E+R is not very flexible, in fact the number of tasks is completely fixed by the schema. As was explained in the section introducing the partitioning schemes, given a schema one can calculate the number of tasks as the number of entities multiplied by the eight rules in the R-axis, plus the number of global rules and uniqueness rules.

To see how much this inflexibility affects the E+R solution we turn to the data we have collected. The individual partitioning schemes are tested with a selection of different models based on varying Express schemas, but we will not reproduce all the results here. Instead we will pick a selection of data which we feel highlight important properties of each approach.

Let us begin by looking at some numbers. Table 4.1 shows a selection of the results gathered when testing the E+R scheme. The headers have the following meanings:

- *Cml%*: The cumulative percentage
- *%*: The percentage of total execution time spent on this particular task.
- *Time*: The time spent executing the task, in milliseconds.
- *Validated what?*: A description of the task, the form is `<ENTITY_NAME>-<OPERATION>`.

The *bygga* model is based on a schema called IFC2x2[2]. This schema is used to model buildings, and the *bygga* model is in fact a CAD-model of

[2]Industry Foundation Classes

a house. The schema defines 623 entities, and 3 global rules, this means that we can have a total of 5610 tasks (8 local constraints, multiplied by 623 entities, plus the 623 entities that need to be checked for uniqueness and finally the global rules, i.e: $623 * 9 + 3 = 5610$. However, because checking local constraints on empty extents is pointless we avoid the execution of this. Inspection reveals the real number of tasks to be 1090.

Looking at the figures in table 4.1 on the page before we notice that the cumulative percentage for the last task is 89.2%, in other words 9 out of 1091 tasks are responsible for almost 90% of the total execution time. This is clearly problematic, but how badly does it affect the suitability of E+R partitioning?

Based on Amdahl's Law[Foster95] we can calculate the maximum speedup. From table 4.1 on the preceding page we know that the largest sequential component takes about 21.3 seconds to execute. If we let F_{seq} represent this time as a fraction of the total execution time for the uniprocessor reference version we have $F_{seq} = 21.3/31 \approx 0.69$. Amdahl's Law then implies that the maximum possible speedup is a miserable $1/F_{seq} \approx 1.46$.

But we are also interested in knowing how the schemes behave on a cluster. To study this we simulate execution of the validation process on clusters of 1 to 32 computers. We assume a best case scenario where the longest tasks are executed first and tasks are always assigned to the next available processor. We also disregard overhead from such things as task management and communication. The numbers we acquire are thus lower bounds, but they should provide us with some insight nevertheless. The results are summarized in table 4.2.

Nodes	1	2	4	8	12
Time	157	78	39	21	21

Table 4.2: Execution times in seconds using E+R partitioning with the *bygga* model on hypothetical clusters. Upgrading from 8 to 12 or more nodes has no effect.

The execution time for a complete non-parallel validation of this particular model is about 31 seconds. Based on this we see that E+R seems to introduce quite a lot of overhead as the single node version executes in 156 seconds, about five times as slow as its non-parallel counterpart. We will come back to this in section 4.2.7. Furthermore we see that the parallel version doesn't get faster than the non-parallel version until we have 8 nodes, and finally no further improvement can be achieved using more nodes. This means that we can only achieve a maximum *theoretical* speedup of 1.4, and

Cml%	%	Time	Validated what?
0.1%	0.1%	179.0	IFCPLACEMENTNOTSHARED-GLOBAL_RULE
0.2%	0.1%	172.0	IFCAXIS2PLACEMENT3D-AGGREGATE_DATA_TYPE
0.4%	0.1%	172.0	IFCAXIS2PLACEMENT3D-ARRAY_REQUIRED_ELMS
0.5%	0.1%	172.0	IFCAXIS2PLACEMENT3D-AGGREGATE_UNIQUENESS
0.6%	0.1%	170.0	IFCAXIS2PLACEMENT3D-ATTRIBUTE_DATA_TYPE
0.7%	0.1%	167.0	IFCAXIS2PLACEMENT3D-REQUIRED_ATTRIBUTES
0.8%	0.1%	132.0	IFCAXIS2PLACEMENT3D-LOCAL_RULES
0.9%	0.1%	129.0	IFCAXIS2PLACEMENT3D-LOCAL_RULES
1.0%	0.1%	129.0	IFCAXIS2PLACEMENT3D-LOCAL_RULES

Table 4.3: The nine most time consuming tasks when validating the *bygga* model using the E+R+P scheme. With this partitioning scheme the total number of tasks is significantly larger because each E+R task is further broken down into tasks of ten instances each.

this at the price of 8 computers. We also note that this minimum execution time is in fact the execution time for the largest task (see table 4.1 on page 43), which is what we would expect based on Amdahl's Law.

The bygga model gives us the worst result for the E+R partitioning, using the results from the other models we find that the possible speedup ranges from 1.4 to 3.9 in the best case. A speedup of 3.9 isn't too bad, but overall the E+R scheme fares rather badly. This is mostly due to the fact that a few key tasks dominate execution completely, as well as its static nature and the great overhead involved. We are stuck with the same number of tasks regardless of the size of the model, and as models grow this becomes problematic. Our selected model — *bygga* — is a prime example of this, and the few large tasks that dominate do so due to their large populations.

4.2.5 E+R+P, an improvement over E+R?

E+R+P retains the partitioning from the E+R scheme but adds the P-axis which should make it a lot more flexible. In particular we expect that the poor results for the *bygga* model with E+R should be greatly improved. We will therefore again look at the numbers from the *bygga* model.

Table 4.3 shows some very interesting results. We notice that the largest task only takes 179ms to execute compared to 22 seconds using E+R. The total execution time for this model was 30807ms. According to Amdahl's law this means that we have a possible theoretical speedup of about 168, which certainly is acceptable. We also note that the 9 first tasks now only account for 1% of the total execution time. The total running time on a single computer is now 144 seconds, a little bit faster than using E+R but much slower than the non-parallel version.

Again we run a simulation of parallel execution, listing the results in table 4.4. We see that nothing is gained from parallel execution until we have 8 computers available, at which point the E+R+P partitioning is slightly faster than the E+R partitioning yielding a speedup of about 1.7.

Nodes	1	2	4	8	12	16	20	24	28	32
Time	144	72	36	18	12	9	7	6	5	5

Table 4.4: Execution times in seconds using E+R+P partitioning with the bygga model on hypothetical clusters

To summarize the results from other models the E+R+P scheme allows for a speedup in the range of 1.9 to 8, assuming a maximum cluster size of 8, which is better than with E+R. However, this speedup comes at a price, E+R+P is generally worse than or similar to E+R for clusters of 4 computers, and in several cases the difference between E+R and E+R+P on 8, 12 or 16 computers is minimal. This means that larger clusters are required to push E+R+P to it's potential, which is a strong argument against it.

4.2.6 E+P, now it gets interesting

Will removing the R axis and thus reducing granularity help? The results seem to indicate this. Again looking at the *bygga* model we find that the tasks are small, like with E+R+P. The big difference is that E+P executes in only 30 seconds on a single computer, i.e. it executes in almost the same time as the non-parallel reference version. This results in far better numbers for the simulation, in fact E+P achieves a near linear speedup, and is able to execute the entire validation in less than a second on 32 computers. It's also faster than E+R and E+R+P for all cluster sizes from 1 through 32.

But with other models the results are not so good, the worst case being a model named *house151*. This model is based on the IFC151 schema, which the reader may have guessed is an older (and outdated) version of the IFC schema. However, a closer look at the data for this model reveals that the problem is caused by a single uniqueness rule, and after a discussion with the EDM developers we have concluded that the bad results are due to a combination of a weak hashing algorithm in the EDM kernel and a badly generated identifier value in the IFC schema (this has been fixed in the IFC2x2 schema, and here the same uniqueness constraint executes in practically no time). In general we see that E+P is always faster than E+R and E+R+P for equivalent cluster sizes.

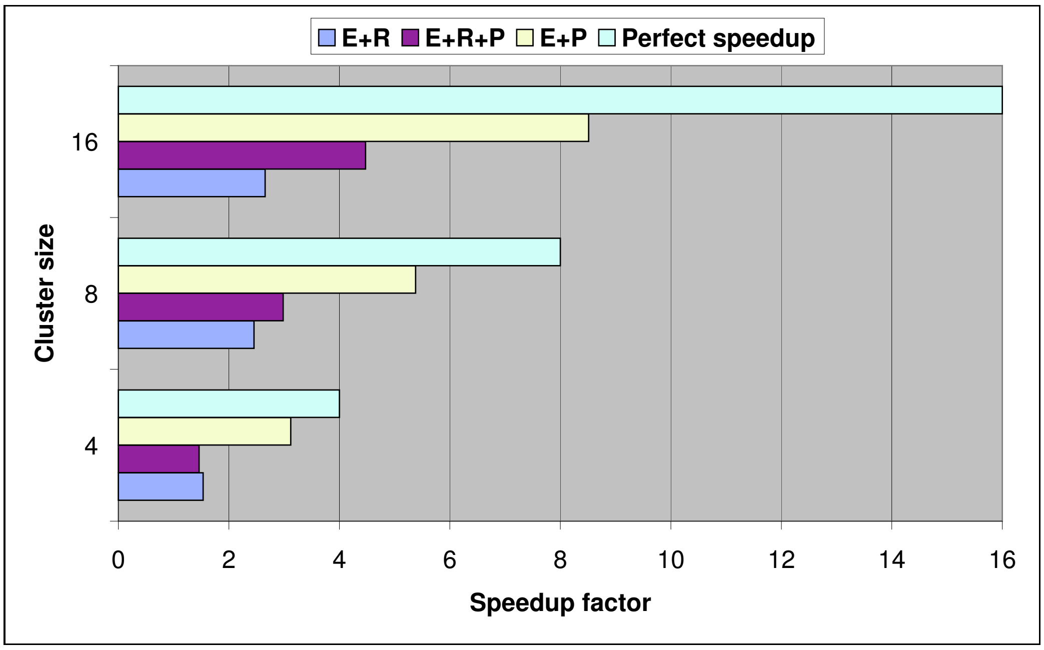

Figure 4.3: Average speedup factors for the different partitioning schemes.

4.2.7 The three schemes compared

To summarize the results we have found so far we need a comparison of the different schemes on an equal basis. To do this we compute the average speedup for each partitioning scheme across the models in our test. We do this for cluster sizes of 4, 8 and 16 computers. As a reference we will also include the perfect linear speedup. The results are shown in figure 4.2.7. From this figure it is quite clear that E+P is superior to the other two partitioning schemes.

Why does the R-axis perform so poorly?

These results are a little puzzling. E+R+P has a finer granularity than E+P, but performs badly. Our theoretical analysis shows that E+P generally runs almost as fast as the non-parallel program on a single-node cluster, whereas E+R and E+R+P adds considerable overhead in the same situation. If we let T_{scheme} denote execution time for a given validation scheme, and T_{full} denote the execution time for a normal full validation we can express the overhead O for the validation scheme as $O = T_{scheme}/T_{full}$. If we calculate the average for this value across all schemes we get an overhead ratio of 2.99 for E+R, 1.19 for E+P, and 2.89 for E+R+P. This indicates that there is a massive overhead incurred by using the R-axis.

To investigate this further we created two new benchmarking programs. One validates a given entity type using the R-axis, and one without it. In practice this means that the former executes eight calls to the validation function with a different rule as argument for each such call, whereas the latter only calls this function once, instructing it to validate all the constraints that apply to instances. We performed this test on populations of 1, 1000, 10000 and 100000 instances of a single entity. The results are summarized in table 4.5.

Instances	Without R-axis	With R-axis	Ratio
1	71	73	103%
1000	116	153	132%
10000	543	967	178%
100000	4620	13643	295%

Table 4.5: Table showing the execution times for a validation of variably sized population with and without the R-axis. The times are in milliseconds. *Ratio* is the size of *With R-axis* relative to *Without R-axis*.

In this table the problem with the R-axis is evident. As the population size grows it becomes more and more expensive to use this axis. Based on this we believe that the problem with the R-axis is that the validation function we are currently calling incurs too much overhead and so the more calls to it we have, the more time is lost. This can be due to many things, and without scrutinizing the EDM source code we cannot pinpoint the problem more precisely at this point. For now we are content to have figured out why E+R and E+R+P perform so poorly. The EDM developers will take a look at this issue, and if a simply fix exists the R-axis may be introduced again at a later stage.

4.2.8 The challenge of global rules

The speedup we're seeing with E+P is definitely acceptable, yet all is not well. The speedup varies greatly between models, and in some cases there is little to gain on using clusters larger than 4. Based on the data we have gathered, one problem stands out, and the data from a model called s214 based on the AP214[3] schema offers a very good example of this problem. This schema includes a global rule which is responsible for more than 65% of

[3]Core Data for Automotive Mechanical Design Processes

the total validation time. The validation of global rules cannot be broken up any further using any of our current schemes, and the result is that this is the largest sequential component in our algorithm. Based on *Amdahl's Law* this means that the highest possible speedup we can expect in this particular case is about 1.5.

One way to attack the problem of global rules does seem promising though. It is based on the observation that very frequently global rules are declared for entities high up in the entity hierarchy, typically with no extent (they are generally abstract entities). This means that the rule in practice is declared for an extent which is a union of all the sub-entity extents. If we could somehow push the rule down in the entity hierarchy we should be able to achieve a finer granularity. For example, if a rule is declared for abstract entity A, with sub-entities B, C and D, and each of these have 100 instances we can split the validation of this rule into three operations on 100 instances each instead of one operation on 300 instances. With some luck we should be able to implement this solution using the EDM interface only, avoiding any changes to the kernel.

A second possible approach to handling global rules comes from the fact that each global rule frequently is composed of several where-rules. We should therefore be able to split a global rule into its smaller component where-rules. This approach is somewhat limited though, for example the global rule responsible for the aforementioned problems with AP214 only consists of two where-rules, and so even if they cost exactly the same to validate, we still have a sequential component of 32.5%.

An inspection also reveals that in the case of AP214 the problematic global rule is completely dominated by one of it's where-rule components. To make matters even worse, the push-down approach previously described cannot be applied in this case because the rule is declared for an entity that has no subentities. In other words, we currently have no sure remedy for the terrible performance seen with AP214 based models.

The only possibility left is to rewrite problematic rules. The thing about STEP Application Protocols is that they are developed by modelers rather than programmers. The modelers focus on modeling — as they should — rather than implementation, and frequently the result is constructs that are extremely inefficient. This fact is frequently lamented by the EDM developers, and there are examples of simple rewrites that have reduced execution times from hours to mere seconds. We cannot simply change the Application Protocols as these are international standards and changes to them require a fairly elaborate process possibly involving several committees. Instead, the rewrites should be automatically performed by the EDM engine, for example a version of the schema optimized for validation could be stored together with the normal schema, and used for validation purposes. Implementing this automatic optimizer is however outside the scope of this project and so as an approximation we can use manually rewritten schemas in our tests. If time allows we should try to tackle the problem of global rules more generally, so that we can handle global rules that even after optimization execute too slowly, but doing so will most likely require direct changes to the EDM kernel.

4.3 Communication

Having thoroughly analyzed different partitioning schemes and arrived at what we hope is the best solution we must now consider the communication needs of our application. The first thing we must do is figure out what the communication needs for our tasks are. Based on the E+P partitioning there are basically three types of tasks, those that deal with the local constraints for an entity, those that deal with a particular global rule, and those that deal with a particular uniqueness rule. To figure out the communication needs we must look at what each node has to know in order to perform the assigned task.

In order to perform a local constraint validation a node must receive the collection of instances, and information saying which constraints apply to these instances. Because we split the data into the individual entity extents and then partition each extent into subsets we can assume that each collection of instances received by the node is homogeneous. The description of the constraints are meta-data contained in the schema underlying the model.

For global rules we need to transfer the global rule itself (i.e. the declaration of the rule from the underlying schema), as well as the necessary entity extent(s) which again is a collections of instances.

Once a validation has been executed, the results must be returned. Results will be simple structures, but their sizes may vary. The results of a local rule validation will be a (possibly empty) collection of instances[4] with the error caused by each instance recorded. The result of a global rule is simply a (possibly empty) collection of instances that violated the rule. Even if we distribute global rules across sub-entities, as discussed in section 4.2.8 on page 48, this simple scheme should work. We will just have to collect and merge the lists of violating instances from different nodes.

But our description of the communication needs so far has glossed over a major challenge. A model is a network of instances, and determining what

[4]By instances we here mean instance identifier, as that is really all we need to identify a given instance

parts of the model are needed to validate a given instance is a very complex enterprise. A single instance can be connected to thousands of instances through references, and all of these instances must be made available to the node before it can perform the validation. Furthermore, there exists functions to retrieve all instances that reference a given instance. Because these functions can be used everywhere, before a node can validate an instance it must have available to it all instances that reference the instance.

Finally there are uniqueness rules, whose validation require that the entire extent of the entity for which the uniqueness is declared are available. In the end, the validation of a single instance may require that we transfer a very large portion of the model, and the overhead involved in computing what portion to transfer can become significant.

One possibility is to make sure that each node stores all the instances it receives, essentially building a local copy of the model. This should presumably reduce the communication needs as the validation proceeds. It would however require that in addition to computing the set of nodes to transfer for a given task, the master must also keep track of which nodes have received which instances. The efficiency of this approach is very hard to analyze, and it is complex to implement, meaning that we risk wasting precious time on a possible dead-end.

A much simpler solution is to simply transfer the entire model to each node at the beginning of the validation. Once this is done the communication becomes very simple because a node just needs to know what task to perform, all the required data is readily available. Because we expect that such a simple scheme can be implemented very quickly we think it's best to simply give it a try and see how it fares.

4.4 Summary and conclusions

The first challenge in designing a parallel program is to look at the data and computations and attempt to decompose these. After an initial inquiry we found that the simplest approach is to decompose the validation process instead of parallelizing the EDM engine as the latter could potentially be a very complicated task. The limited time available to this project means that we need to favor a fast solution over a more general one.

We analyzed three different ways to decompose the validation process, and concluded that the E+P scheme is the best, and it is therefore this approach we will attempt to implement. Due to the embarrassingly parallel nature of the validation algorithm a master/slave architecture seemed natural. The master/slave architecture means that communication will be very simple.

The master must inform the slaves of their tasks, and make sure tasks are kept busy, while the slaves only need to return the results of each completed task to the master. The master/slave approach also has a built-in mapping and scheduling algorithm in that slaves simply receive new tasks as soon as they finish their current. A remaining problem in this area is that tasks are frequently of wildly varying sizes. We have not yet tried to find ways to predict the size of tasks, and so this is currently an open problem.

The models we are working with are networks, and we saw that a large amount of data may be necessary to validate a single instance. Because of this we needed to consider ways to distribute the models to the nodes in the cluster, and decided that the naive approach of transmitting the entire model to each node in the cluster should be sufficiently efficient.

Although the theoretical analysis is promising so far, we still haven't solved the problem of complex global rules. We decided to postpone a solution to this problem for the time being, noting that we should be able to find acceptable workarounds. We found that the problems with global rules is mostly due to inefficiently written code, something that could be fixed by using automatic code optimization. All in all the conclusion here is that if we try to tackle the problem of global rules at all, we should do it the algorithmic level, trying to improve the implementation of the global rules. Parallelizing a bad algorithm is simply a waste of resources. A true parallelization of global rule validation will require changes to the EDM engine, something we have already deemed to be outside of the scope of this project.

Chapter 5

Implementation

In chapter 4 we saw that a simple master/slave approach suited or problem nicely. This approach should also be fairly easy to implement and so it is what we will attempt to use. We will begin this chapter by outlining the requirements of our system. We will then look at how we will realize these requirements in practice.

Our main goal is to develop a proof-of-concept implementation, and we will thus aim for simplicity. This means that we should favor the simplest possible design, even if it may not be the most efficient. We will use an iterative design approach, where we first implement a straightforward solution. This solution can then be tested and profiled, and based on the results we can select parts of the system that may benefit from optimizations.

5.1 Choice of technology

Before we can begin actually implementing anything at all we need to consider what technologies to use. In our case we are fairly limited by the fact that the EDM library currently only offers a binding for C/C++, VB and Java, so these are the languages to choose from. VB is out of the question because we are using a Linux cluster and VB is Windows-only technology.

Task-farming has a client-server structure. While message passing can be used to program such an interaction it's not ideal because the two-way communication must be explicitly programmed with messages going back and forth. With RPC however each operation is a two-way communication channel; the client sends a request, the server acts on the request, and the result is then returned to the client.

RPC also has the advantage that writing a program in terms of procedure calls is something programmers are used to, this is after all how one write most programs. Explicit message passing on the other hand is a bit more exotic.

Both C/C++ and Java have RPC implementations available. In the case of Java it is called Remote Method Invocation (RMI), because procedures in Java are called methods. When it comes to solving the task at hand we believe both languages to be equally well-equipped. The author is most experienced with Java, so this seems like the natural choice.

5.2 Overall architecture

Our system will be written using the EDM Java binding. This is basically a library wrapping the EDM C-interface by using the Java Native Interface[1]. In practice this means that we can base our design on the benchmark program developed in the previous chapter, exchanging the C-calls with the equivalent Java calls. The rest of the implementation will deal with the infrastructure necessary to do a parallel validation.

The heart of a master/slave approach is a queue of tasks. The master distributes tasks from this queue to the slaves, who process each task and return the result. Task-farming is self-scheduling; tasks are simply distributed on demand. Once a slave completes it's current task it can receive a new. If the number of tasks is much larger than the number of processors task-farming can result in good load-balance.

In some task-farming systems, slaves can generate new tasks while processing the current task. This complicates matters slightly as the slaves must be able to add tasks to the queue. Also, if the queue is empty, a slave must be able to determine if any of the other slaves are working, as this may lead to new tasks becoming available. This creates a dependency between the slaves. We can avoid these complications altogether because our tasks won't create new tasks. The complete list of tasks can be generated before execution begins.

Figure 5.1 on the facing page illustrates the overall architecture of the system. Each node will be running an instance of the Express Data Manager, and we access the database via this instance using the EDM Java interface. The master is written against this interface, and exposes the necessary RMI calls to the slaves, which are implemented in the same way as the master by using the EDM Java interface.

The flow of our system is described using an UML like notation in figure 5.2 on page 56. First, the master receives a model to validate, this will

[1]http://java.sun.com/j2se/1.4.2/docs/guide/jni/spec/jniTOC.html

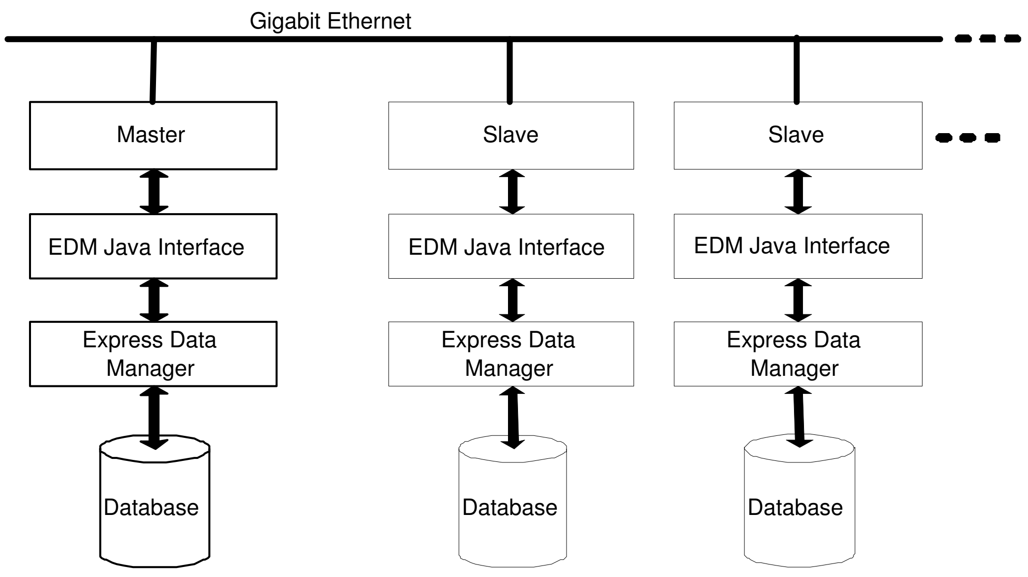

Figure 5.1: System architecture overview

typically be in the form of a simple text file. The model must be made available for download through the FileServer. The model must then be imported into the database, once this is done we can analyze the model and generate tasks based on the partitioning scheme we have chosen. The list of tasks is thus generated before the validation starts. Once the queue of tasks is set up the slaves are ordered to start the validation. They will then connect to the FileServer and download the model file. Once the file has been downloaded it must be imported. The slaves then start requesting tasks from the master. During the validation, results will be returned from the slaves containing information about any errors found. These must be stored at the master and used to generate a final error report as the validation completes.

5.2.1 Distributing models

Slaves receive tasks as messages telling them what to do, but as was discussed at the end of chapter 4 the slave must also have available to it the data on which it should operate. As we saw, knowing what instances will be needed to validate a given instance is very difficult due to the way instances can arbitrarily refer to other instances in Express. Because of this, we found that the simplest approach would be to distribute the entire model to all the slaves at the beginning of the validation.

This naive solution seems rather wasteful, and certainly has the potential

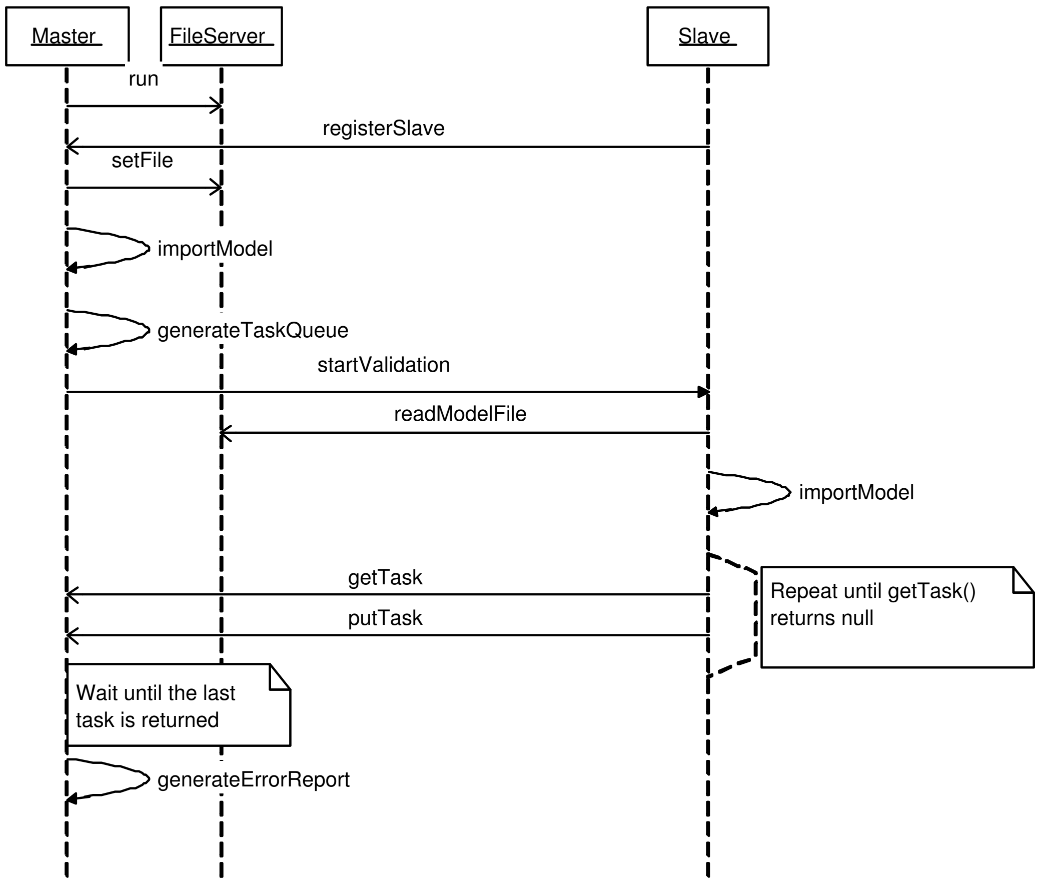

Figure 5.2: The general sequence of a validation using our task farming approach. We have included only one slave, but each slave will perform the same operations in the same order.

of becoming the largest sequential component in our solution. One would expect that only a small part of the model changes between validations, and so it would make sense to transfer only the changes. But this won't work because currently models are moved from the application used to build the model—e.g. a CAD tool—and to the EDM database by exporting it to a file and then consequently importing it from this file. While this is certainly inefficient it is how the system is used today, and it is what we have to relate to.

5.2.2 Distributing schemas

In order for the master and slave to be able to import the models they must know the schema used by the model. Because schemas are static entities defined as international standard we make the assumption that the necessary schemas will be available at the nodes when a model is submitted for validation.

Hence our solution will not feature any schema management. A full-blown solution will most likely offer this, although the schemas should be managed by the cluster manager rather than the users submitting models for validation. We should also note that in most cases a company works with data from very few schemas. Companies involved in construction for example will mostly use the IFC schema only.

We should also point out that it is possible to validate a model against a specified rule schema. In this case it is necessary to distribute the rule schema together with the model. We will not support rule schemas in our implementation. Support for rule schemas should however be an integral part of a complete solution because validation against rule schemas is a fairly common operation.

5.2.3 Distributing tasks

Task farming is fairly straightforward, we use a central queue to hold all available tasks and as slaves request new tasks they are removed from the queue and sent out for processing. Once processed, the result is returned and stored. When slaves request a task but find the queue empty they know that the current validation is complete, and can idle until new orders arrive.

A slight variation is possible here; when there are no tasks left in the queue, a task can be taken from the pool of tasks being currently processed. If a slow slave has received a large task, this may be beneficial because a faster slave may be available for processing almost at the same instant, and can finish the task much faster. This could also make the system more responsive to failed slaves. To finish the validation we would have to send a terminate signal to all the slaves once all tasks have been completed as in this approach slaves may be working on tasks that have already been completed by other slaves.

This latter approach has some advantages, but as we shall see, practical issues make it difficult to simply kill a slave once it's in operation. Also, our cluster is a homogeneous one, so for all practical purposes our slaves can be considered equally fast. Because we use a fairly large number of tasks it is also not likely to make any major impact as it will only make it possible to finish the last few tasks of the validation slightly faster. The rest of the validation will not benefit from such an approach.

5.3 Error handling

In a distributed application there are more sources of errors than in a normal application. The network is a source of many possible errors as nodes can lose their connection at any point during an execution. Concurrency is also a source of many challenges. Writing concurrent programs is much harder than writing sequential ones, and bugs can be very subtle and hard to track down due to the indeterministic nature of concurrent programs. These are issues we must deal with.

Ideally our solution should be able to run even if we unplug one or more of the slaves. If a slave disconnects for some reason we should be able to discover this and take the appropriate action. We will however not be able to handle cases where the master crashes or loses it's network connection. It is certainly possible to write master/slave solutions of a more egalitarian nature, where any of the machines in the cluster can act as master. Such solutions can tolerate the loss of the currently acting master by somehow choosing a new master which then must be set up with the necessary knowledge about the state of the current computation. Such extreme fault-tolerance however adds a lot of complexity, and we wish to keep things simple.

5.3.1 Disconnected slaves

It is important to know the state of the slaves. When a slave becomes unavailable (either as the result of a network problem, or a software crash) we must be able to detect this as soon as possible. There are basically two approaches here, the master can poll the slaves at regular intervals, or the slaves can send heartbeats to the master to assert their liveness.

We will use the heartbeat approach. Each slave sends a heartbeat once every second, this can be implemented as a simple RMI call. When the call arrives we record the timestamp of the heartbeat and maintain a mapping between each slave and it's most recent timestamp. The master should have a routine which executes every three seconds (for example) and scans this mapping to see if any timestamps are older than three seconds in which case we assume the slave to be dead.

A polling approach is somewhat problematic. If a slave disconnects then the attempt to poll it might hang until a timeout occurs. Java makes no guarantees about the length of such timeouts by default, but it is possible to define this timeout manually. It also requires us to make callbacks to the slaves, thus adding methods to the slave interface. Because we want to keep communication as simple as possible we prefer the heartbeat approach as it goes from the slave to the master.

But this addresses only network errors or complete failures where the slave simply crashes. What about other errors that may occur at the slave?

5.3.2 Handling other failures

There are numerous things that can go wrong at the slaves. But as a start they can be grouped into fatal and non-fatal errors. The former are errors that makes it impossible for the slave to continue executing. If for example some unrecoverable error occurs while the slave is copying the model from the master it will not be able to perform any subsequent validation operations at all. The same is the case if errors occur during the import step. The simplest way of handling these errors is probably to close down the slave. The master will then detect this as outlined in the previous section.

This is a crude form of error handling, but for our proof-of-concept it should suffice. In general we will be more interested in detecting and analyzing errors once they occur rather than have the system handle them. After all we want to measure performance in an error-free environment as this should be the normal case.

5.3.3 Errors and task management

When a slave is disconnected this may just be temporarily, although this is very unlikely in our cluster solution. If it finished a validation during it's disconnect it will send the result as soon as it gets back up. But if we assume that the master has discovered that the slave is down it will have put the task on which the slave was working back into the queue. This will result in the master receiving a result for a task which is still in the queue. Clearly the master must be able to detect this and remove the task from the queue again. This also means that rescheduling tasks by inserting them at the back of the queue can be advantageous because it should increase the chances that the connection to the disconnected slave can be fixed before the task is rescheduled. But even if we do so, we can also find ourselves in a situation where some other slave has computed and returned the particular task while the problematic slave was disconnected. In this case we must simply discard the result from the disconnected slave should it reconnect.

Errors can occur during the execution of a task, meaning that the slave in question is incapable of fulfilling the task. Because we are running in an

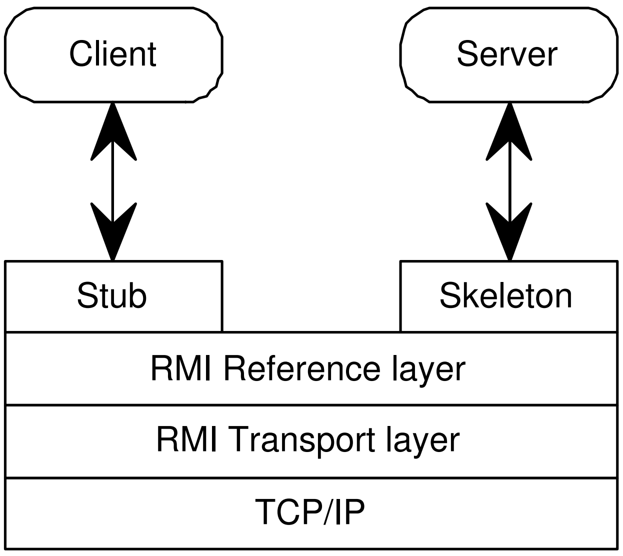

Figure 5.3: The layers of the RMI architecture

homogeneous environment this probably means that no other slaves will be able to execute the task. But if we keep resending failed tasks and none of the slaves are able to execute them we will end up in a livelock. We can of course maintain information about which slaves have tried to execute a task and failed, and based on this stop trying once every slave has tried to execute the task. But again, because we are mostly interested in detecting and fixing errors as soon they occur we will just stop the slave and report the error when it occurs.

5.4 Writing applications using RMI

This section is a very brief description of the usage of RMI, and is mostly based on [Andrews00]. As a reference the architecture of RMI is depicted in figure 5.3.

In it's simplest form an application developed using RMI has three components: an interface defining headers for remote methods, an implementation of this interface, i.e. the server, and one or more clients that call the remote methods.

One usually starts by defining the interface, it must extend the `Remote` interface defined in the `java.rmi` package. The server must then be written as an extension of the `UnicastRemoteObject`, implementing the methods in the interface. It must also contain code which instantiates the server and registers this under an appropriate name in the registry service (to be covered shortly). Finally a client class can be written. To use the methods exposed by the server the client must start by getting a reference to the server from the registry. The rest of the client code can then make normal method calls to this server object as if it was a local object.

When the client performs a remote call it must be handled differently than a local call. For this interaction to happen transparently a program called *rmic* is used to generate the server *skeleton* and the server *stub*. The *stub* sits at the client and translates a remote call into a message (a process called *marshalling*). This message is then sent to the server *skeleton* which *unmarshals* the call and generates a local method invocation to the actual server implementation. Once complete, the result is then *marshalled* and sent back to the client. The *stub* and *skeleton* thus hides all the details of the network communication.

Because clients and servers typically are on different hosts (though they don't have to be.) they must be able to refer to each other somehow. For this purpose URL-like names are used, having the form `rmi://hostname:port/service`. The `hostname` is the Internet domain name for the host on which the server is running, or an IP address. `port` is the port on which the registry service is running, by default 1099. Finally, `service` is the name under which the server registered, so this name must be unique.

The registry service is a daemon running in the background on the server host. The registry maintains a mapping between service names and the code implementing the services. The server registers a new mapping by calling `Naming.bind()`, the client fetches a reference to the server by calling `Naming.lookup()`.

According the RMI Specification [SunRMI] section 3.2, the server "may or may not" execute an RMI call in a separate thread. This means that remote method invocations on the same remote object may execute concurrently. As a result a remote object implementation must take care to synchronize access to shared data structured, making the implementation thread-safe.

5.5 Modules

To implement our solution we should define modules that have clear areas of responsibility. The master must deal with two important entities, slaves and tasks. To manage the slaves and keep track of which slaves are currently connected we should write a separate SlaveManager. To handle the queue of tasks we should write a separate TaskManager.

5.5.1 The SlaveManager

The SlaveManager is responsible for keeping track of the status of the slaves. For this purpose it must maintain a list of slaves currently expected to be alive, which we shall refer to as the live set. It must also maintain a list of all the slaves that have registered with the master. This list should be a mapping from the slave to a timestamp indicating when the last heartbeat was received.

The SlaveManager must allow slaves to register and unregister with the master. It must also expose methods for requesting information about what slaves are alive, and methods for getting references to these slaves for the purpose of callbacks.

Finally the SlaveManager must have a routine which scans the slave-to-timestamp mapping looking for slaves that have not sent a heartbeat within some defined time interval, and remove these from the live set on the assumption that they are unreachable.

If a slave which has been removed from the live set sends a heartbeat this event must either be detected by the routine which scans the list of timestamps, or it can be handled by the function used to handle the heartbeat. The former approach means that the routine scanning the timestamp table must both add and remove slaves to the list of live slaves. The advantage of this is that the heartbeat handler is very simple because all it needs to do is to update the timestamp. It is also a cleaner design, in that the scanner has the sole responsibility of keeping track of which slaves are live and which are not. The drawback is that a slave will not be known to be live until the scanner discovers this, and depending on the frequency by which the scanner runs this may take several seconds.

The other approach is to see if the slave is in the live set when a heartbeat arrives. This will make the slave known as live as soon as it's heartbeat arrives, at a very slight extra cost to the heartbeat handler as this must check whether the slave exists in the live set for every request. It also separates the maintenance of the live set between the scanner and the heartbeat handler. Furthermore, we don't expect the disconnection and subsequent reconnection of a slave to be a very frequent event. Because of this we opt for the cleanest design and let the scanner handle all maintenance of the live set.

To make the scanner run at timed interval we can make us of `Timer.scheduleAtFixedRate` This method requires an instance of the `TimerTask` class as it's argument, as well as an interval in milliseconds. To run the scanner we pass an anonymous instance of the `TimerTask` which simply calls the scanner routine in the SlaveManager. Because the class is anonymous and within the scope of the SlaveManager, it has direct access to the SlaveManager methods, allowing it to call the scanner routine.

Finally the SlaveManager must be able to notify other modules about the death of a slave. In particular it must be able to notify the TaskManager about this event, as the TaskManager must have this information to reschedule the task (if any) which the disconnected slave was working on.

5.5.2 The TaskManager

The task manager module is responsible for maintaining the queue of available tasks. It must offer methods to request new tasks as well as returning finished tasks. It must also keep track of which slaves are doing what, and take care of rescheduling tasks should a slave fail to execute it.

The TaskManager must also offer a method that is used to build the internal list of tasks when a new validation is about to begin. This method will be called from the master once the current model has been imported into the database and is ready for analysis. The TaskManager thus implements the partition scheme that we have chosen.

As results are returned from slaves they must be stored until the validation is complete for the purpose of producing a final error report. The TaskManager is responsible for storing the results, and should offer a method to return these results. It is not natural for the TaskManager to produce the final error report, so it simply holds the raw data needed to generate this report.

5.5.3 The Master

Having modularized the master means that the slaves will need to know about the modules to use their functionality. Furthermore, the modules must be written as if they are remote services. Clearly this is not desirable. The slaves should only need to know about one single service, not the modules it is made up of. To achieve this we follow the Mediator design pattern [Gamma et al 95], making the master act as a mediator between the slaves and the TaskManager and SlaveManager. The interface exposed by the master is the only thing the slaves need to know about. Whenever a remote call is made to the master it must forward the call to the appropriate module. This approach is visualized in figure 5.4 on the next page.

This means that the code implementing the Master interface will be extremely simple. But the master must also provide the user interface. For this purpose we should have a class RunMaster, which is used to actually run a master. It starts the different modules, connects to the EDM database, creates the server used to serve the model files, creates a binding in

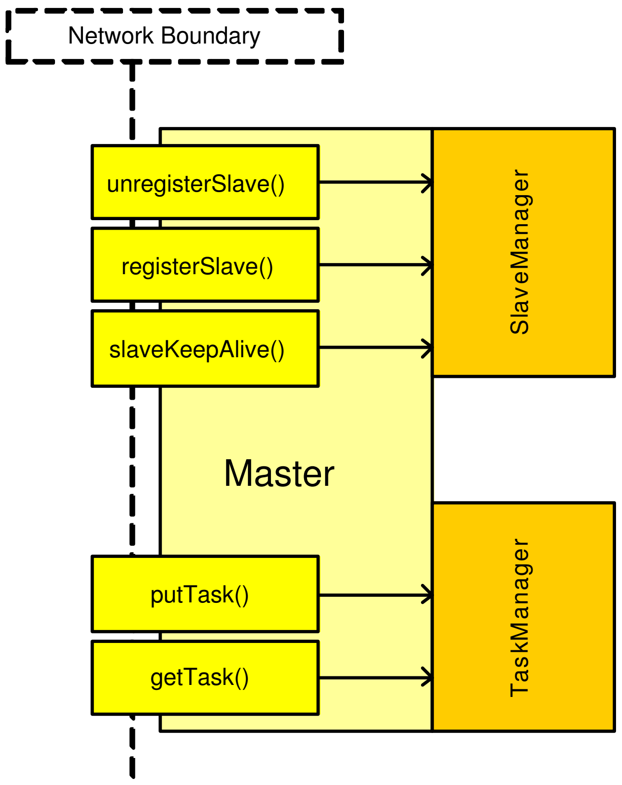

Figure 5.4: Using the Mediator design pattern to simplify the remote interface

the rmiregistry so that the slaves can access the master, and controls the overall validation process. The source code listing for this class can be found in Appendix A.1.

For our purposes testing is obviously very important. The testing will require many executions, and this should be as automated as possible. The master should offer both a batch mode and an interactive mode. A simple approach to the batch mode is for the master to read a predefined configuration file containing a list of models to validate. For data collection we wish to validate models multiple times, and we wish to do this in random order. But we must also make sure that the order is reproducible. We will therefore use a simple utility to generate the list of models which guarantees that the same input will always yield the same output.

The interactive mode simply waits for the user to input the name of a model to validate. This can be used to investigate properties of selected models. In a real implementation the models will probably arrive over the network or be selected through some sort of GUI.

When a model has been selected the master must import the file containing the model data. Once this is completed it must call the TaskManager to make it build the list of tasks. Then it can instruct the slaves to start the validation process.

5.5.4 The FileServer

The first stage in the validation process will be to transfer the model to all the slaves. To achieve this we should have a tiny file server module running at the master node. This module is responsible for accepting connections from slaves and serving the current file to the slave.

When the validation of a model is about to start the model file is made available for download from the file server. As the slaves are ordered to start validation by the startValidation() call they connect to the file server and download the file. This happens sequentially, i.e. each slave finishes downloading a file before the next slave can connect and do the same.

5.5.5 The Slave

The slave is a lot simpler than the master. Basically it must retrieve a reference to the master and register with the master. When it registers it can send a reference to itself as a parameter, allowing the master to make callbacks to the slave without manually going through the registry. It must also connect to the database, and set up the heartbeat timer. We have included a source code listing for the Slave in Appendix A.2.

While we prefer to have the communication be one-way, the slaves must somehow know when they are to start a validation cycle. While they could certainly poll the master at regular intervals, this is not very efficient. Instead the slave should offer a remote method startValidation() which the master can use to inform the slave that a new model is ready for validation. Once the master calls this method the slaves connect to the master, reading the model data file and writing it to a temporary file locally. This file must then be imported. Once imported the slaves can start requesting and processing tasks.

When called, startValidation() will block the master until it returns. Because of this the method must create a new thread which will deal with the actual validation process, this way the startValidation() call can return immediately. We will thus need a ValidatorThread class which contains the actual validation code.

5.5.6 The ValidatorThread

Each slave runs the validation code in a separate thread. Once started, this thread begins by reading the model from the master. It then executes the model import, and enters a loop alternating getTask() and putTask() requests.

Because validation errors are stored in the EDM database using a special schema, the ValidatorThread is also responsible for querying this database after every task has been executed. Should it find that any errors or warnings have been created it must collect these and store them in some data structure. We will simply use a ValidationError class to hold these errors, and putTask() should return an array of ValidationError objects, which will be empty if no errors or warnings were generated.

5.5.7 Crosscutting issues

The most important purpose of our system is to give us test results that can tell us something about the performance of our approach. But to get these results we will need to time different parts of the code, and we will also need to generate miscellaneous logs. These are so called crosscutting issues because they affect all parts of the system. To time the different steps in a validation, timer code must wrap each step of the cycle. The timer and logging code must be replicated wherever we need to time things, and the this extra code will also clutter up the system code and make it harder to read.

A more efficient way of dealing with such crosscutting issues is through the use of Aspect Oriented Programming (AOP) [Kiczales et al 97]. AOP adds another level of abstraction on top of OOP and for example allows us to declare that calls methods matching some pattern should all be wrapped by code which starts and stops a timer. This is only a very simple example of what can be done with AOP, but it is what we will be using it for.

AOP is usually implemented by using a *weaver*. A *weaver* takes the description of crosscutting code and weaves it into the original source files. If we for example have two functions, get() and put(), and we wish to time them both, we declare this in a file. The weaver will then modify both methods to include the timing code.

For Java there are several implementations available. The most popular seems to be AspectJ[2], and Aspectwerkz[3]. Of these two, AspectJ is the most

[2]http://aspectj.org

[3]http://aspectwerkz.codehaus.org/

mature, so we will use it. AspectJ is a very small extension to the Java language.

5.6 Synchronization and state analysis

Concurrent programming is hard. Making sure that the different processes don't interfere with each other is necessary to ensure the correctness of a concurrent system. In our case there are two different issues to consider. First, there is the relationship between the slaves and the master. These will communicate using RMI, and when remote calls are made we must ensure that the callee is in a suitable state to deal with the request. Secondly, both the master and the slave makes use of threads. These must be carefully designed so as to not interfere with each other. One of the nice features of Java is that it offers simple constructs for working with threads.

We first begin by looking at the communication between the master and the slave. By enumerating the states that the slave and the master can be in, and relating these to the different calls that can be made, we can analyze which combinations are acceptable and which are not, allowing us to decide how to deal with erroneous combinations.

5.6.1 The master states

As the master begins it's validation cycle it will first import the model and build a list of task. This ensures that once the master instructs the slaves to begin validation it is in a correct state to fulfill the requests that will come from the slaves. When the slaves begin validation they will start by importing the model. Once this is done they start calling getTask() and putTask(), always in a strict alternating sequence. The master will not exit the validation stage until the last task has been executed and returned. Once this happens, the master will go back to the ready stage.

Table 5.1 on the following page show the states the master can be in and relates these states to the possible calls it can receive. Combinations that are acceptable have the value OK. The state are as follows:

Ready In the Ready state the master is ready to accept a new model for validation. In our solution this either means that the user is expected to supply a model, or that the batch system is about to select one.

Import Once a model has been selected it must be imported into the database. This state also includes the processing needed to generate the list of tasks based on the newly imported model.

	State			
Call	**Ready**	**Import**	**Validate**	**Errors**
registerSlave	OK	OK	OK	OK
unregisterSlave	OK	OK	OK	OK
slaveKeepAlive	OK	OK	OK	OK
getTask	—	—	OK	—
putTask	—	—	OK	—

Table 5.1: The states in which the master can be and the calls it can receive. The master will always change between the states from left to right, cycling back to **Ready** when finishing **Errors**.

Validate As the list of tasks has been built the master is ready to proceed with the validation. This implies ordering all the slaves to start validating, and wait until the last task is finished.

Errors With the validation completed the only remaining task is to generate an error report detailing any errors found with the model. Once this state completes the master returns to the Ready state.

As is clear from table 5.1 we don't have to worry about the `registerSlave()`, `unregisterSlave()` and `slaveKeepAlive()` calls. The first two are used to register and unregister a slave with the SlaveManager, and these should be allowed to happen at any time without it causing problems.

The `getTask()` and `putTask()` calls on the other hand only make sense in the **Validation** state. This is the only state in which the task queue contains any tasks. But handling erroneous calls to `getTask()` or `putTask()` shouldn't be difficult. A call to `putTask()` in any other state should simply be ignored. A call to `getTask()` can simply return null.

5.6.2 The slave states

The slave only accepts a single call, `startValidation()`. This makes the call/state table very simple. The states are much the same as for the master, except for the new Read state which means that the slave is downloading the model file from the master.

As we can see from table 5.2 on the facing page there are three problematic cases here, and they are a lot more difficult to handle than the ones in the master. If the slave is in the Read state it is reading data from the FileServer and writing this to a local temporary file. Should a new call to startValidation() arrive at this point, the slave will enter the Read state,

	State			
Call	**Ready**	**Read**	**Import**	**Validate**
startValidation	OK	—	—	—

Table 5.2: The states in which the slave can be and the calls it can receive.

and thus two threads will be writing to the same file. This file is also being read in the Import state, so if the slave is in the Import state and receives a startValidation() call this will certainly cause problems. Finally, if the slave is in the Validate state a call to startValidation() will cause the slave to run two validation threads. This will cause the validation to break down completely as the model data which the first thread operates on will be overwritten as soon as the newly created thread reaches the Import state.

5.6.3 Using barrier synchronization to ensure acceptable state transitions

A simple way of preventing all these problems is to use barrier synchronization [Tanenbaum01] [Andrews00]. The master and all the slaves must all come back to the ready state before they can start a new validation. Finishing the validation of a single model is the barrier they all must reach.

By using barrier synchronization we can guarantee that as the master reaches the Validate state, all the slaves will be in the Ready state. The master can then safely call `startValidation()`, which will make the slaves enter the Validation state. Until all slaves have finished the validation both the master and the slaves are known to be in the Validate state. After the slaves have finished, they will go back to the Ready state. The master will go through the Errors state, safe from any unwanted requests from the slaves. As the Errors state ends, the master goes back to the Ready state and a new cycle can begin.

Barrier synchronization may however introduce inefficiencies. Consider the situation in which some slaves are in the middle of reading or importing a model, while others have already started validating. If the validating slaves are able to process all the tasks before the reading or importing slaves are able to finish reading or importing there is no point in continuing this reading or importing. What we want is a way to instruct these slaves to simply stop what they are doing and go back to the Ready state once the validation has completed. The master should thus be able to inform the slaves of this event, and the slaves must be written so as to allow them to abort the current read or import.

But this is somewhat tricky. In Java threads can be interrupted, causing them to raise an `InterruptException` and terminate. However, according to [Lea02] nothing forces a thread to terminate once interrupted. This allows the thread to do necessary clean-up, but also makes the interrupter responsible for checking that the thread has actually been terminated before moving on. In addition to this minor complication, the import process is performed by a Java method wrapping a native call to a C routine. Abruptly aborting this C routine may cause database corruption because it is in the process of altering the database. Had the import process been a Java routine we could have added the necessary clean-up code to allow for such sudden termination.

Due to these complications it is not a good idea to abort the thread. Instead we simply have to wait for all the slaves to finish, i.e. reaching the barrier. We also believe that the likelihood of this event occurring is minimal. For it to happen the validation must be very short compared to the read and import stage, in which case there is little or nothing to gain from a parallelization anyway. It can also happen if one slave gets stuck importing for a very long time, but because we are using a homogeneous cluster and all slaves have the same operating environment this is also an unlikely event.

Finally we should note that while the read stage is implemented in Java and can be gracefully aborted, the probability of a slave being in this state as the other slaves finish the validation is very small. We therefore choose to ignore it and stick with our simple strategy of simply waiting for all slaves to finish before continuing with the next validation. Should this prove to be a problem we may have to return to this point at a later stage.

5.6.4 Local synchronization requirements

The SlaveManager and the TaskManager both maintain internal data structures to keep track of the state of slaves and tasks. These data structures must be protected from concurrent access. Here Java provides a very easy to use mechanism. By making the methods that may access the data structures concurrently `synchronized` we can ensure that they will be thread-safe. The slave doesn't maintain any such data structures, so here there is no danger of concurrent access.

5.7 Summary

In this chapter we have tried to cover as many aspects of our design as possible. Our approach is iterative, the design described here is the first version, as we turn to the test phase we will gather performance data that will help us in making informed choices about which parts of the system will need improvements[4].

For the implementation we chose Java, using RMI for communication. We discussed the overall structure of the system, and how to properly modularize it. Testing the performance and correctness of our system is important, and we have decided to use Aspect Oriented Programming for a simple way of adding and removing timing and logging of the system as the need arises during testing.

Designing a distributed system is complicated by the need for more error handling due to the asynchronous nature of when and how errors can occur. We discussed various strategies for how to deal with errors, but decided to go with very simple error handling as we are writing a prototype. We also looked at how to manage synchronization and ensuring that the different parts of the system don't end up in an erroneous state.

[4]Assuming of course that the system has potential of being a viable solution at all!

Chapter 6

Performance analysis

In this chapter we shall look at how we tested our parallel solution. The main purpose is to find weaknesses in our solution that can be improved. We begin by giving a description of the test environment. We then describe how we will perform our tests, and discuss the quality of our test results. We then turn to our results, and study several important aspects, such as performance, overhead and scalability.

6.1 Test environment

For the purpose of this project we received a grant from "Komp-programmet", a program initiated to strengthen the cooperation between the University of Oslo and the local industry. The grant was used to buy a cluster of 8 computers. The cluster was set up on site at EPM technology, and was under our control for the duration of the project. No other people had access to the cluster, it was solely dedicated to our purpose.

6.1.1 Hardware configuration

The cluster consists of 8 identical Dell computers. They are all equipped with Pentium 4 2.4GHz processors and 1GB of main memory. The cluster is connected through an 8-port gigabit switch, and hence has a dedicated gigabit network available. One of the computers was designated as the master and is the only computer with an extra network interface card allowing this computer to access the rest of the local area network. The other 7 computers are designated as slaves. The master was named Snow White, and the slaves were named Dopey, Grumpy, Doc, Happy, Bashful, Sneezy and Sleepy. Hopefully they will not display the same variances in mood as their namesakes.

6.1.2 Software setup

All the computers run a very minimal installation of RedHat Linux 9.0[1]. We have tried to avoid installing anything but the absolutely necessary software to ensure that no computational resources are wasted. All the computers were configured using RedHat's kickstart installation system[2] which allowed us to easily do identical installations on all the computers via the network.

To manage the cluster we use Cluster Command and Control (C3)[3], a simple set of command line tools developed by the Computer Science and Math Division at Oak Ridge National Laboratory. Most important is a program which lets us distribute files to all the computers in the cluster, and a program which lets us execute the same command on all the computers in the cluster.

Since our initial study the EDM library has been upgraded to version 4.7 and we chose to use this version for our system, mainly because the Java interface had seen some extensive improvements since version 4.5 which was what we previously used. The EDM library has been compiled using *gcc* without any optimization settings.

As for Java we are using version 1.4.2 of the Java SDK which at the time of this writing is the most current version.

6.2 Testing methodology

For testing purposes we have gathered a collection of various models available to us. This has been a bit problematic because Express models typically contain data considered to be industrial secrets, and thus nobody wants to give them away. But with some effort we have been able to find a collection of models which should highlight the most interesting aspects of our solution.

There are several variables that may have an impact on performance of the cluster. First of all the number of slaves can vary, and testing with different cluster sizes should tell us something about the scalability of the solution. We also expect to see performance vary greatly between models, and from our initial survey we know that this is probably mostly influenced by what schema the model is based on, so we have models based on different schemas.

[1]http://www.redhhat.com

[2]http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/custom-guide/ch-kickstart2.html

[3]http://www.csm.ornl.gov/torc/C3

We should also have a selection of models based on the same schema that vary in size as this will tell us something about how size affects performance. Finally the validation can be executed using different task granularities by varying the population partition size, and we are interested in knowing if there is a single partition size which works well across the board, or if it depends on the models.

6.2.1 Measuring execution times

To measure the execution times we simply insert timers into the code that should give us the numbers we need. We want to know how long it takes to transfer a model from the master to a slave, how long it takes to import a model, and how long it takes to validate the entire model. Because we want to be able to closely scrutinize examples of bad performance the slaves will also time each operation they perform. The timing data is returned to the master together with the standard return information for each completed task, and written to a log file there. This simplifies data collection and will allow us to find bottlenecks in the execution as well as keep track of which slaves did what and in what order.

All these measurements are done in Java, using `System.currentTimeMillis`. As with our initial studies this gives us a resolution of milliseconds[4] which should be adequate given that the sort of operations we are working with generally takes minutes or more to execute.

6.2.2 Test setup

It is important that we are able to compare the two solutions on an equal basis. We have already covered the hardware and software setup which will be the same for all tests. We must also ensure that the two programs do the same thing. The two programs will therefore validate the same models, the same number of times, in the same order. To achieve this they both read the list of models from the same configuration file.

To further improve the quality of our measurements, each model will appear 10 times in this list allowing us to check the variance and to compute averages. In a real world situation, models will typically arrive in random order, and so to reflect this the list of models will be permuted so that the models are validated in random order.

[4]While `currentTimeMillis` has a resolution of 1ms on Linux, it's resolution under Windows 2000 is only 10ms, despite the name.

6.2.3 What to measure

Our implementation works by first importing the model on the master, which then builds a list of tasks to be performed. Once this is done it informs the slaves to start the validation process. The slaves then connect to the master to read the model file and import the model. As they finish importing they start the actual validation by requesting tasks from the master.

It is important to understand that when measuring the execution time of our distributed solution we exclude the time it takes to import the model at the master. The time it takes to validate a model in our distributed environment is defined to be the time from the master is done importing the model to the final task has been completed and returned to the master. This means that transmitting the model to the slaves as well as importing the model at each slave *is* part of the validation time.

The rationale behind this is the way models are generally used. As we have previously discussed a user typically works with the model using dedicated software like a CAD tool. When he or she deems it necessary to perform a validation the model is exported from the CAD tool to a file. To validate it using the Express Data Manager it must first be imported. This latter import stage is thus necessary regardless of whether a sequential or parallel validation is used. In the parallel case this import stage is the same as importing the model at the master. Once the model is imported, validation begins, and to the user there should be no difference between a parallel and a sequential validation (except hopefully the time it takes).

6.2.4 Data quality

Before we can look at the test results we should look at the quality of our measurements. If they vary greatly we may need to reevaluate our testing methodology or possibly alter the test programs.

One must expect some variance due to factors out of our control, such as the effect of other programs running. Paging and cache effects are also to be expected. The execution times for the parallel system is further influenced by the performance of the network and the task distribution, so a greater variation in the parallel results is to be expected.

Sources of variance

One possible source of error is the many daemons that the Linux operating system depends on. While we have made a very minimal install of the operating system, several daemons are essential to it's operation and can cause problems for us. Because our tests take several hours to execute we must expect that some of the daemons will start running during the tests and will interfere with them. On the positive side it should however be noted that most of the daemons require very little resources and generally only execute for a very short time.

Models that are quick to validate will be most affected by these daemons because the relative effect of the daemon running will be bigger. One of the worst examples of this is the distributed validation of the house model where the values generally show little variation, hovering around the median of 950ms, but where one of executions took 2956ms to execute.

A second major source of variance is the effect of memory. EDM manages it's own memory, and uses a lot of caching in an attempt to speed up execution. In chapter 7 we will discuss the effect of caching and the memory management in more depth, as we look at some very interesting effects this has.

Measured variance

To characterize the spread of our results we use the semi-interquartile range. This is computed as one half the difference between the 75th and the 25th percentile of the measured results. It is fairly resistant to extreme values which inspection shows that our data contain some rare cases of. Because we have several models, we need to normalize the computed ranges if we are to compare the values across models. The easiest way to do this is to express the semi-interquartile range as a percentage of the average execution time. The resulting values are shown in table 6.1 on the next page and visualized in figure 6.1 on the facing page. The relative semi-interquartile range has the header % Variance.

6.3 Findings

6.3.1 Overall speedup

The first thing we want to look at is simply how well the distributed solution performs. We measure execution times using partition sizes of 1, 10, 100 and 1000. The results are summarized in table 6.2 on page 78. We have also extracted the best result for each model and these are shown in figure 6.2 on page 78.

As we were expecting based on the theoretical analysis, the results span a wide range. Two models (*house* and *bygga*) actually have a speedup of less than 1, and thus perform worse in our distributed environment than on a

		% Variance	
Schema	**Model**	**Parallel**	**Sequential**
AP214	**as214**	1.4%	1.9%
	io214	4.4%	0.7%
IFC2X2	**bygga**	3.8%	2.1%
	condo	5.3%	1.5%
	ginza	7.4%	0.4%
	house	4.6%	3.1%
AP203	**conrod**	3.8%	2.2%
	part21	1.4%	0.4%
	s203	5.6%	0.2%
AP210	**cpu**	0.8%	0.1%

Table 6.1: Normalized semi-interquartile range of execution times

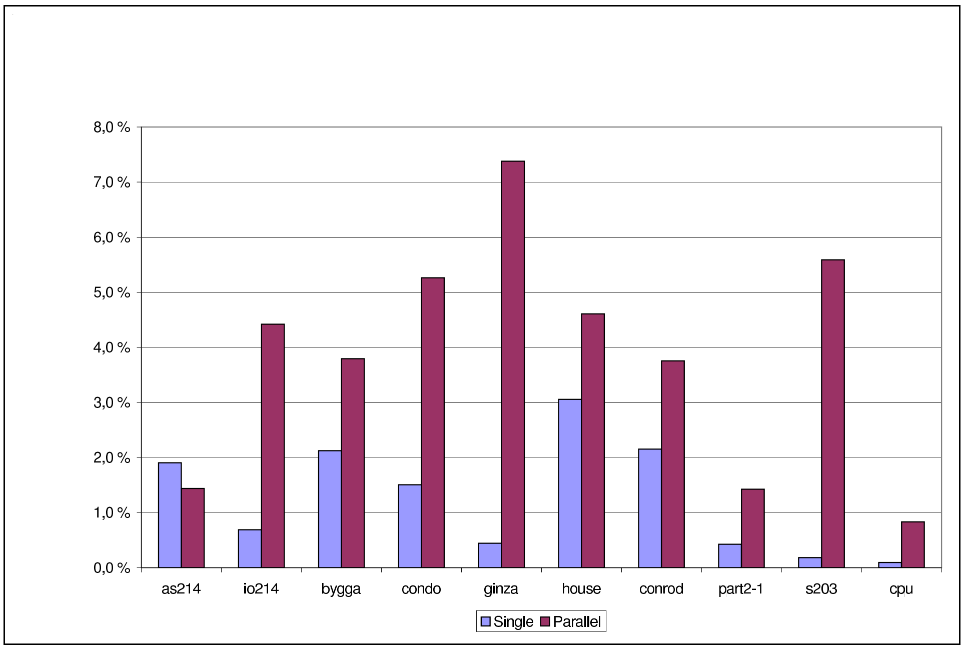

Figure 6.1: Normalized semi-interquartile range of execution times

		Partition size			
Schema	**Model**	**1**	**10**	**100**	**1000**
AP214	**as214**	1.60	1.57	1.59	1.59
	io214	2.39	3.43	4.07	4.08
IFC2X2	**bygga**	0.18	0.83	0.98	0.78
	condo	—	1.45	4.61	4.77
	ginza	—	0.80	1.73	1.72
	house	0.04	0.50	0.67	0.54
AP203	**conrod**	4.27	4.65	3.99	2.18
	part21	0.86	3.90	5.55	4.72
	s203	0.92	3.99	5.16	3.09
AP210	**cpu**	3.43	6.48	6.64	3.67

Table 6.2: Measured speedup for a selection of models using a 8 node cluster with 7 slaves and 1 master and partition sizes ranging from 1 to 1000.

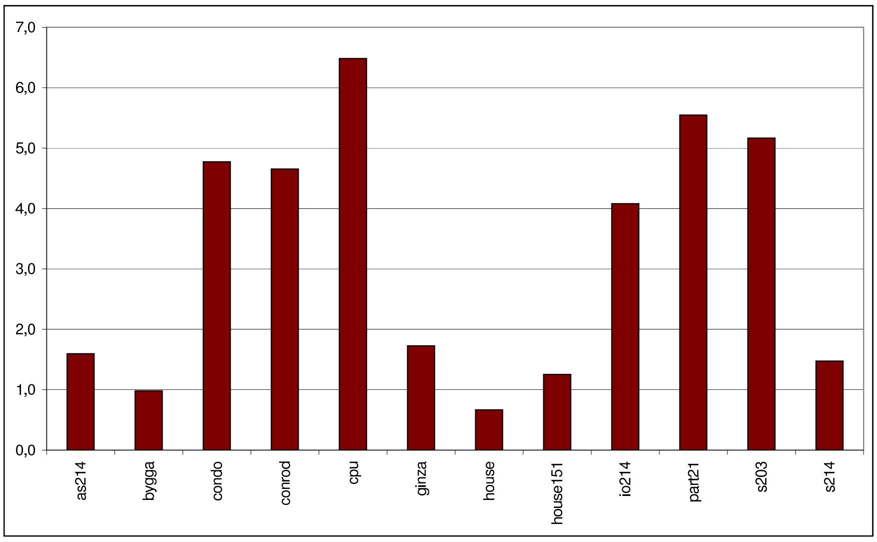

Figure 6.2: Best initial speedups achieved for the various models in our test set.

single computer. On the other end of the spectrum we find the *cpu* model, which has a maximum speedup of 6.64, which is very close to the perfect speedup given that we have 7 working slaves. We should also remark that no data is available for *ginza* and *condo* using a partition size of 1 because the Java VM ran out of memory due to the extreme amount of tasks this generated.

The two worst performing models—*bygga* and *house*—are both instances of the IFC2x2 schema. So is the *ginza* and *condo* model. But these four models display very different performance. From the initial study we suspected that the schema would influence performance to a great degree. But with the four models in question there is one more variable to consider, and this is size. The two problematic models, *bygga* and *house* are 2.3MB and 3.1MB respectively. *ginza* is 57MB, and *condo* is the largest at 122MB. This is very interesting as performance seems to improve with model size.

For a user the absolute speedup is obviously also very important. For the two smallest model there is very little to gain in terms of wall-clock time, they execute in a few seconds anyway. But with *condo* the situation is different, this model takes about 36 minutes to validate on a single computer. With our distributed solution it validates in roughly 7 minutes. It is this type of models that we want to tackle with our solution, and thus the results so far seems very promising regardless of the terrible results for *bygga* and *house*.

The problem with IFC based models seems to be that they are computationally lightweight. This means that the import stage at the slaves tends to dominate the execution, and thus performance suffers. This is unfortunate as the IFC schema is one of the most used schemas in the industry.

The best performing model (*cpu*) is not particularly large in terms of data, but it is computationally intensive. It takes 40 minutes to validate on a single computer, and only 6 minutes to validate using the cluster. The advantage with computationally intensive models is that overhead is reduced, most importantly because the import stage becomes very small compared to the validation stage.

6.3.2 The impact of partition size

From our results it is clear that no particular partition size works best overall. There is one model that perform best with a partition size of 10, six models come out best with a partition size of 100 and three models show best performance with a partition size of 1000. This is no surprise, the models vary greatly in their number of instances, and for the largest models a partition size of 10 means that the task count grows extremely large, adding much overhead due to task management. But this is not the sole cause. The *cpu* model provides a good case in point, this is actually our third largest model, yet it comes out best with a partition size of 100. And the performance drop going from 100 to 1000 is quite large, from a speedup of 6.64 to a speedup of 3.67.

The problem here seems to be that certain entities have very complex rules declared on them, and thus take very long to validate. When these are grouped into chunks of 1000 instances the resulting tasks become larger in terms of computation. Because of this the granularity of the tasks goes down, and the load balance becomes less optimal. To verify this hypothesis we can use the logs generated from our test runs and sum the execution times of all the tasks for each slave. We expect to find that with a partition size of 100, the slaves are doing pretty much the same amount of work, whereas with a partition size of 1000 certain slaves are much more heavily loaded. Inspection shows this to be correct, using a partition size of 100 the work done by the slaves range from 315 seconds to 346 seconds. With a partition size of 1000 we find that one slave only works for 215 seconds, while a different slave has to work for 641 seconds. The validation can't be completed until this latter slave is done, and thus most of the other slaves lie idle as the last slave struggles to finish it's final complex task. Inspection reveals that the last task to be finished takes 541 seconds to validate in our test case. The results are shown in figure 6.3 on the next page.

6.3.3 Scalability

Unfortunately we don't have a very large cluster to test our solution on, and this prevents us from testing scalability properly. However we decided to test the solution with 6, 3 and 1 slave. Using only 1 slave should also tell us something about the overhead incurred by our solution. We won't go into details here, the results showed that for most models using 6 models was about twice as fast as using 3. For two models however this was not the case, but these two models both suffer from the global rule problem we saw in our initial survey. They are dominated by a single global rule, and the slave executing this single rule is always the slowest slave. Because of this it makes no difference if we use 3, 6 or 100 slaves for these models. They are both limited to a speedup of less than 2, so anything beyond 2 slaves makes no difference.

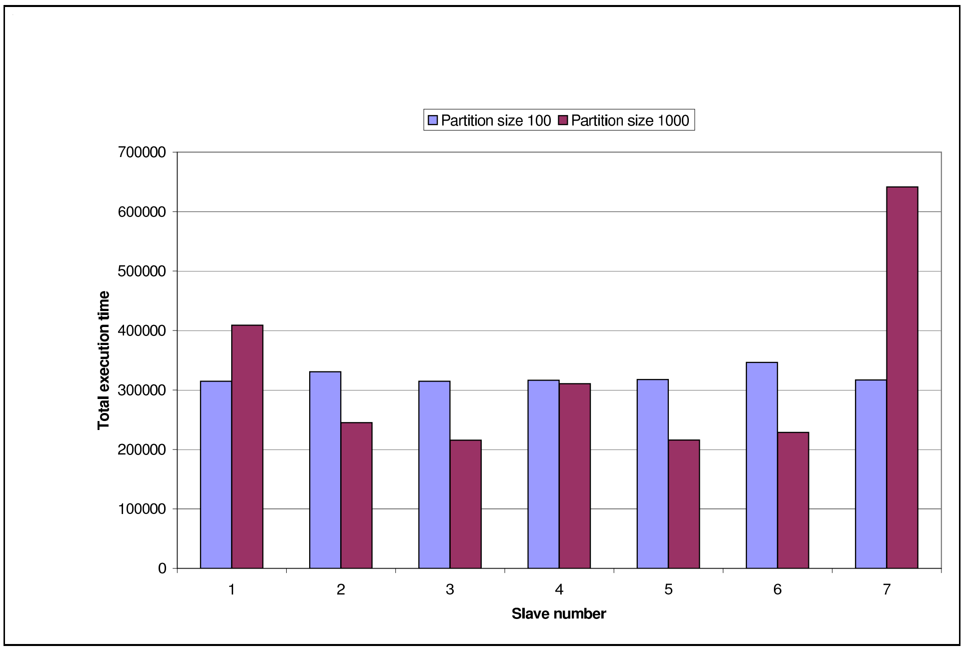

Figure 6.3: Load distribution using partition sizes of 100 and 1000 on the *cpu* model. The problem with a size of 1000 is evident for slave 7, whose total execution time is much larger than any others.

6.4 Overhead

A distributed solution must necessarily always carry some overhead due to the extra management necessary. We will here discuss the possible sources of overhead in our implementation. We have briefly touched this subject in previous sections but a more complete discussion is desirable.

There are several possible sources of overhead in our system. We are using Java with native calls to C, and one can expect some overhead due to this. To perform a distributed validation several operations which are not necessary on a single computer must be done. First, a model must be analyzed, and tasks generated, these must be inserted into a queue. Next the model must be transferred to all the slaves and imported there. Once this is complete each slave must fetch and return tasks, something which is done through RMI calls. As these calls are made the master must keep track of which tasks are in what states. Once all tasks have been completed the master must summarize any errors found. All of these operations must considered as overhead as they are not necessary in a regular single computer validation.

6.4.1 Measuring overhead

A simple approach to measuring overhead is to simply study the efficiency of our solution. [Foster95] defines relative efficiency as

$$E_{relative} = \frac{T_1}{PT_P} \tag{6.1}$$

where T_1 is the execution time on one processor and T_P is the execution time on P processors. An efficiency of 1 clearly means there is no overhead, and the lower the efficiency the more overhead. From this it is clear that our solution has one major source of inefficiency—the use of a master which does nothing but manage the slaves. In a cluster of 8 computers this means that $\frac{1}{8}$ of the computing resources are wasted[5]. This puts an upper limit on the relative efficiency of our solution at 0.875, or 87.5%[6]. A second source of inefficiency is load imbalance, something which we have already touched upon several times.

Figure 6.4 on the facing page shows the relative efficiency of the models in our test set. The results here shouldn't be surprising as relative efficiency

[5]Of course, some of the computing resources are actually used for managing the slaves, so this is a simplification

[6]Disregarding the possibility of superlinear speedup

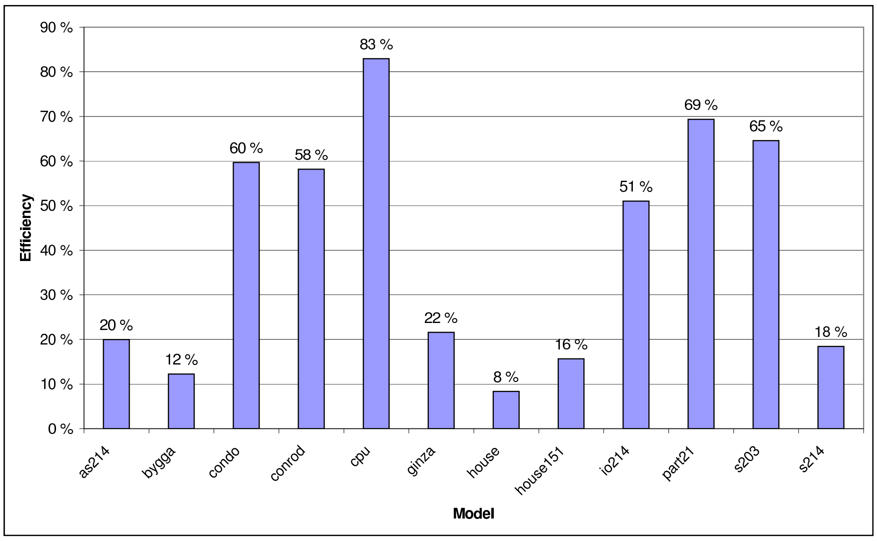

Figure 6.4: Relative efficiency in percent for the models in our test

is closely related to the speedup which we have already studied. In fact, the two can be defined in terms of each other.

But while relative efficiency gives us a birds-eye view of the overhead, it doesn't say anything about where the extra time is actually spent. To study this we need to analyze the different sources of overhead and their contribution to the total.

6.4.2 JNI overhead

There are really only three methods implemented with JNI that we call frequently, these are the methods responsible for the different validation operations. These calls are fairly simple in that they merely wrap the C functions which implements the validation. The native functions do not perform any callbacks into Java nor do they access any Java objects except for the data passed as arguments.

The overhead from JNI has been studied in [Kurzyniec, Sunderan 01], and the findings here indicated that JNI incurs very little overhead if no callbacks are made, and there are no uses of Java objects within the native code. This is the case for our solution, hence we don't expect the use of JNI to have a noticeable effect on the performance of our solution, and therefore choose to disregard it.

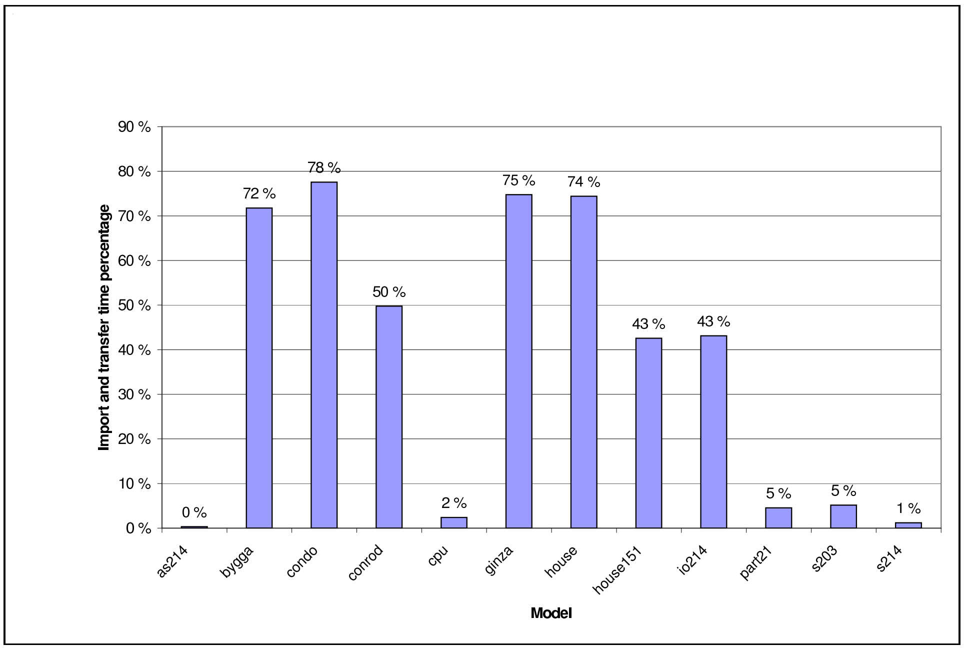

Figure 6.5: Percentage of the total validation time spent transferring and importing the model.

6.4.3 Transfer and import overhead

To get an idea of what impact the import stage has on the efficiency of our solution we can compare the time spent performing the import to the total validation time. By doing this we can express the import time as a percentage of the total validation time. This gives us the results displayed in figure 6.5.

The overhead from the transfer and import of a model is very dominant for many of the models, so this is certainly a target for optimization. We will have more to say on this when we turn to making improvements in the next chapter.

6.4.4 Load balance

Load imbalance is a source of inefficiency because it means some computers are doing much more work than others, which in practice means that some computers are idle as they wait for the harder working computers to finish.

Thanks to the self-scheduling nature of the master/slave approach we know that slaves are working at their max as long as there are tasks available. As soon as a slave finishes a task, it fetches the next and starts executing it.

When we looked at how the partitioning affected the performance of the

Schema	**Model**	**Difference**	**Relative**
AP214	**as214**	29241	74%
	io214	314	24%
IFC2X2	**bygga**	215	2%
	condo	78	0%
	ginza	390	0%
	house	326	5%
AP203	**conrod**	19	1%
	part21	999	7%
	s203	737	13%
AP210	**cpu**	23705	7%

Table 6.3: Difference in finishing time for the first and last slave, expressed in absolute terms (milliseconds), and as a percentage of the total execution time.

cpu model we analyzed load balance by computing the total execution time for each slave. But a simpler approach is also possible; we can timestamp when a slave starts and stops a validation. These two timestamps can then be used to calculate the time spent in the particular slave. An even simpler approach is to timestamp only when a slave is done validating. What really matters after all is when the slaves finishes, if one slave has to work twice as much as the others, the others lie idle as they wait for the slave to finish. If we choose the first slave to finish as our reference value we can calculate the finishing times of the other slaves relative to this. We also express the difference in execution time as a percentage of the total execution time as this makes it easier to compare the various models.

We've already seen that *as214* performs poorly, and based on the load balance results it is clear that this is mostly due poor load-balance. The other AP214 based model (*io214*) also seems to have rather poor load balance, but here the absolute difference is only 314ms. The problem here is not a large global rule as with *as214* but rather that the validation of *io214* is very fast so that the relative effect is more pronounced. Looking at the two largest models in our test set, *condo* and *ginza*, we see that the load balance here is extremely good.

The way our solution is implemented, load balance is only influenced by the partitioning scheme chosen. In cases where complex global rules dominate execution time there is little we can do to improve the load balance. In section 6.3.2 on page 79 we saw how much partition size influenced the performance of the *cpu* model. A statically defined partition size is problematic because it doesn't adapt to the model.

6.5 Reference data collection

To obtain our reference data we use a small C-program. Our first version of this program was simply a program that called the API function `validateModel` to validate the entire model. This call was then timed using the same method as we did with our benchmark program in chapter 4.

As we were executing our tests we noticed something strange with *condo*, the largest model in our test set. When validating the *condo* model using only a single slave we found that it was 37% faster than using a single computer and the `validatemodel` program. This is clearly not possible as the parallel version should carry a lot of overhead, and so we decided to investigate further.

We began by reviewing our code. The first thing we did was to reassure ourselves that the two different executions actually perform the same validation with the same data yielding the same result. The first is simple to verify; the options sent to the validation API calls determine what is to be validated, and using a debugger we could easily check that the same options were used both in the distributed and single computer solution. We can also be sure that the models are equal in both cases, because once again the same C function is used to import the models (while we call a Java method in the parallel version, it merely passes a filename to the C-function which performs the actual import.). But the distributed solution partitions the data into chunks, so a bug in the partitioning code could result in parts of the data not being validated properly. We have used the same partitioning algorithm as we did in the initial study, and nothing so far had indicated a bug in this code. Further scrutiny of the logs reassured us about the correctness of the partitioning. We also found that the validation results were the same in each case.

At this point we were baffled. We decided to try the benchmarking tool from our initial study. Because this program was the basis of our parallel program it uses the same partitioning of the data as the distributed solution, it then executes all the tasks generated and measures the time for these. We will refer to this program as `validatemodel-split`. Using `validatemodel-split` we validated the *condo* model, and the result was surprising indeed. `validatemodel-split` validates the model twice as fast as `validatemodel` which we had used as our reference so far.

The resolution

The results for `validatemodel-split` prompted us to profile the code. We recompiled both `validatemodel` and `validatemodel-split` to include profile information and validated *condo* with both. The results were clear, in `validatemodel` two functions responsible for internal memory management account for a major percentage of execution time, this is not the case for `validatemodel-split`. These findings were given to the lead engineer at EPM, who after some studying could verify that this was indeed the problem. What happens is that during a complete validation using no partitioning the EDM memory manager keeps track of far too much data, it doesn't discard data quickly enough. This leads to certain internal data structures growing very large, and thus slow. With a partitioned execution this is not a problem because data are discarded between each function call used to implement the partitioning. This is clearly a weakness in the EDM engine, and will be fixed.

But this is problematic from our point of view. For does it invalidate all our results? Certainly we can claim that our current solution makes it possible to validate the condo model faster than is currently possible with the EDM software package. But once this weakness is fixed, our distributed solution will most likely show much weaker performance for the condo model. Luckily it looks like the other models are not affected by this problem, it is the sheer size of the condo model which is the problem.

Because we feel that the `validatemodel-split` program best represents reality we will use this as our reference program henceforth. We do not wish to introduce the new EDM code into our system as that could potentially introduce other problems. The version we currently have is a stable release that has been thoroughly tested.

6.6 Summary

In this chapter we have described how we set up our cluster and how we went about testing our implementation. We then looked at the measurements and analyzed some aspects of how our implementation performs. We found that certain models were difficult to speed up. Two models clearly suffered under the global rule problem that we predicted in our theoretical analysis.

We also looked at overhead, and found that the sequential import stage is slowing things down. We also saw that the use of a static partition size is problematic. These are issues that need to be addressed.

Chapter 7

Optimizing the system

The first version of the system made some simplifying assumptions that made writing the system easier. As the first testing phase came to an end it was clear that there was room for some improvements. In this chapter we will describe these improvements and how to implement them, as well as study what performance gains we are able to achieve.

7.1 Improving the import stage

The first version of our system has one sequential stage which should be possible to parallelize further. The import stage works by first importing the model on the master, then distributing the model to the slaves who finally import the model themselves. The transfer time is generally orders of magnitude smaller than the import time, so in practice the slaves are importing in parallel. But while the master is performing it's import step the slaves are completely idle, if we could begin by transferring the model to the slaves and then start the import process in parallel on both the master and all the slaves we should expect to see improved performance. In particular we should see improved performance for the important class of models based on the IFC schemas. These tend to have a relatively large import stage and be computationally simple.

The two ways of distributing the models are shown using a timeline in figure 7.1 on the next page. There are two sets of timelines, one for each approach. Each set of timelines has the master timeline on the far left, this is of course the only timeline without a transfer stage, as the master already has the model data available. Note that we don't begin to import the model at the master until it has been transferred to all the slaves. This is a pragmatic choice, if we were to both transfer and import at the same time we would have to use a separate thread to import the model at the master, and more threads means more complexity. We also don't expect to see any performance gains, in fact it is possible that such parallelism would slow down the transfer stage so much that nothing is gained.

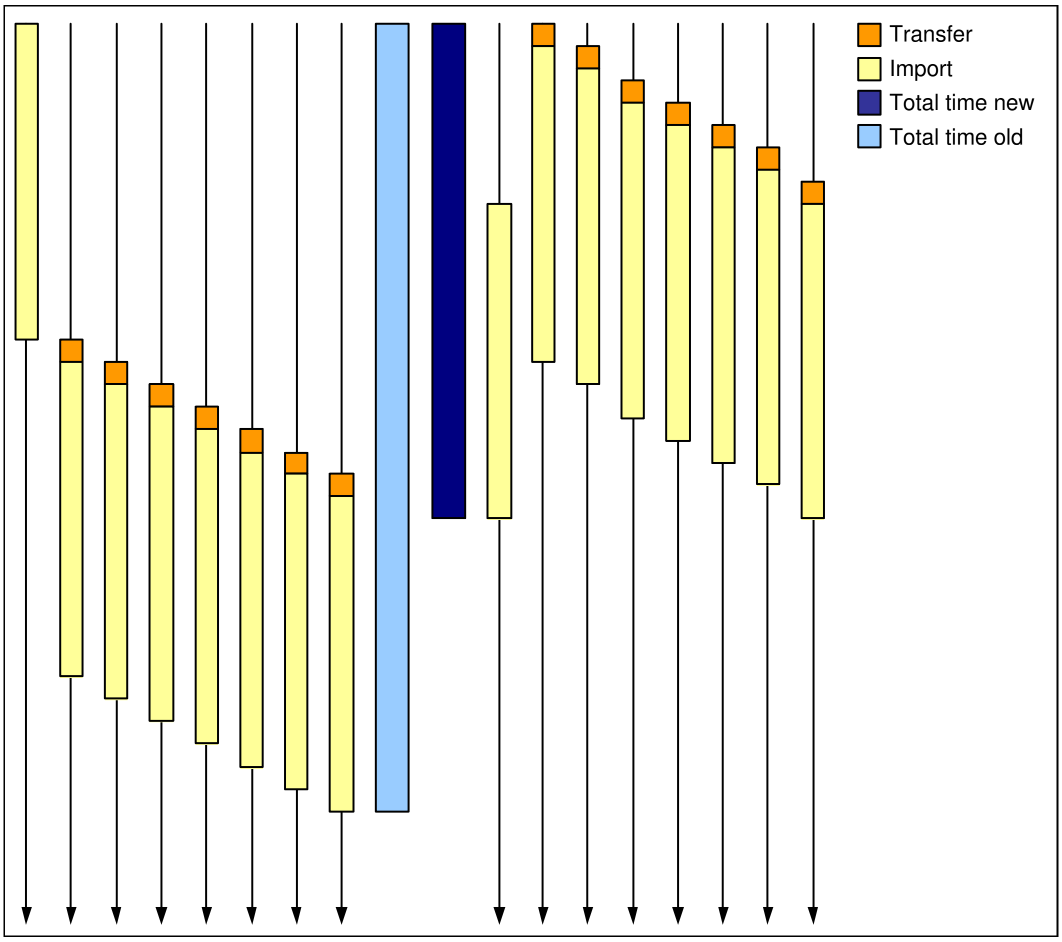

Figure 7.1: A timeline showing the two approaches to model distribution. On the left is the old approach, on the right is the new approach. The difference in length between the two blue bars in the middle show the performance increase possible with the new approach due to the increased parallelism.

The reason why this wasn't done initially has to do with simplicity. The advantage with our current solution is that once the master is done importing and setting up the list of tasks it is in a state which ensures that the subsequent validation can begin. It then instructs the slaves to begin their validation process which also is a sequential process beginning with the transfer and import of the model, so when the slaves start validating they are also guaranteed to be in the correct state.

If we are to perform the import in parallel, complexity increases because the number of possible states increases. When a slave is done importing the model we can no longer be sure that the master is ready to start handing out tasks because the master may not have finished it's import step, and we thus have to implement wait states to handle this case. Once the master has completed it's import stage we can't guarantee that all the slaves have finished theirs, and thus they might not be in a state that allows the validation to start. This case also has to be catered for.

We should also mention that during the implementation of the first version of the system an attempt at using this improved solution was made, but it failed. The author has limited experience with this sort of programming, and because of time constraints the improvement was discarded. As our testing phased ended the author was more experienced with concurrent programming and Java threads. A new attempt was made at implementing the improved import stage, and this time it succeeded.

7.2 Improving the partitioning scheme

During the theoretical study a fixed partition size was used. Because the theoretical study did not consider the effects of task management and communication overhead the partition size had little impact. As our tests revealed however the partition size can have a major effect on performance. The problem is that no particular partition size is optimal, meaning that the desired partition size must be specified up front. Because it's impossible to predict what size will work best this is now a very good solution.

A better approach would be to calculate the partitioning based on information from the model. This would make the partitioning scheme adapt to the model and remove the need for a statically defined partition size. Because the tasks vary so much in complexity it is a good idea to have a fairly large number of tasks as this should make load balancing easier. If we have a small number of heavy tasks then some slaves are bound to do much more work than others.

When the population of a given entity is subdivided into equally large chunks, each chunk take about the same time to validate. This means that if we have 8 slaves, and we subdivide a population of 800 instances that normally take 8 seconds to validate into chunks of 100 instances each, we can expect each such chunk to take about 1 seconds to validate.

Based on this observation we suggest to divide each population using the following formula:

$$\begin{aligned}
chunkSize &= \frac{numInstances}{2 * numSlaves} \text{ if } numInstances \geq 2 * numSlaves \\
chunkSize &= \frac{numInstances}{numSlaves} \text{ if } numInstances \geq numSlaves \\
chunkSize &= numInstances \text{ if } numInstances < numSlaves
\end{aligned}$$

We use a factor of two because this makes the partitioning somewhat more finely grained and ensures that for large populations there are at least two tasks available per slave for each population. For populations that have less than $2 * numSlaves$ instances we see if they have at least $numSlaves$ instances, if they do we simply don't multiply by two in the divisor. If they have less than $numSlaves$ instances we generate one task to validate all the instances. Once the $chunkSize$ has been determined we proceed to generate tasks based on this value.

The division used to calculate $chunkSize$ is assumed to be integer division. In other words the number of tasks generated (assuming $numInstances$ is larger than $2 * numSlaves$) will thus be

$$numTasks = 2 * numSlaves + (numInstances \bmod numSlaves) \tag{7.1}$$

7.3 Performance of the improved system

The two different aspects of the system that we changed might have different effects. The parallelization of the import stage should increase performance, especially for models with much data but simple computations. The new partitioning isn't expected to improve performance very much. It will however make the system more practical in use, and prevent performance problems due to badly chosen partition sizes from arising. The system will be able to efficiently adjust the partitioning to the model.

The new partitioning solution is compared against the initial solution using a partition size of 100 as this is the partition size which works best in most cases.

Because the two improvements are different in effect we first implemented them separately to test each by itself. Thee results are shown together in 7.2 on the following page. The figure shows the performance of the new solution relative to the old. A value less than 1 indicates a performance decrease, whereas a value above 1 indicates an improvement.

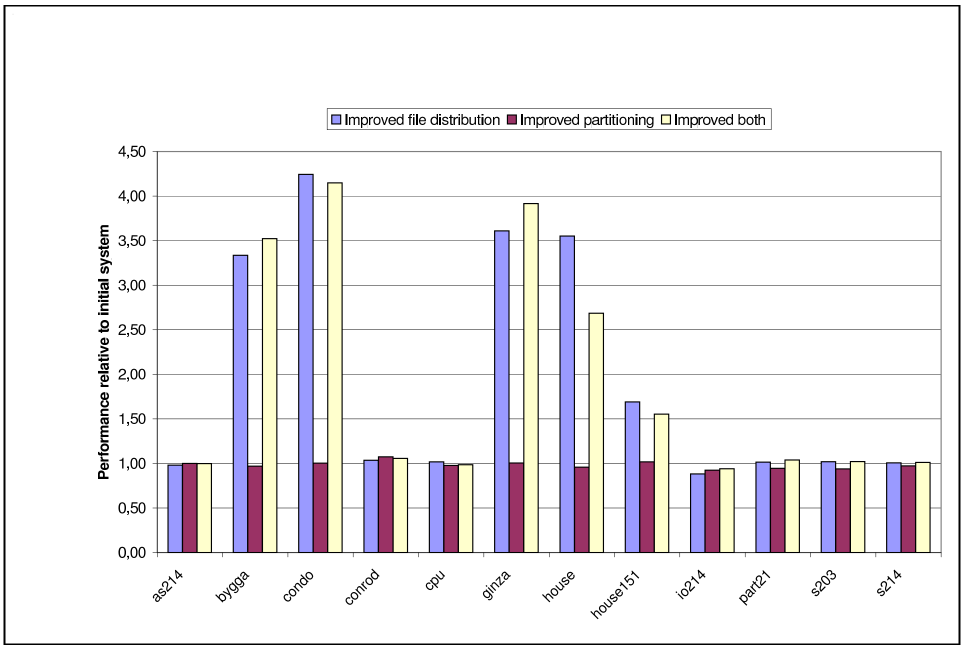

Figure 7.2: Performance of the improved system relative to the initial version.

The new import stage

As we can see the difference is quite amazing for models based on the IFC2x2 schema. We have previously seen that models based on IFC2x2 generally take a long time to import compared to their computational requirements, so it comes as no surprise that these model benefit enormously from this new approach.

The new partitioning scheme

The new partitioning scheme has a very small impact on performance compared to the best results in our previous tests. Most models take a very slight performance hit, but this is certainly acceptable considering the benefits of this new approach.

7.3.1 A new performance profile

With the new import stage in place we have removed the biggest source of overhead, but this alters the performance profile of the system. We therefore rerun our tests to see if there are any bottlenecks left.

The results show that the most obvious bottleneck now is the transfer stage, for large models like *ginza* and *condo* this stage account for up to 30% of the total execution time. That is enough to warrant an attempt at optimization.

Current transfer stage implementation

The way the models are currently transferred is simple, and not rather inefficient. As the master starts up a simple thread is created to act as a file server (see section 5.5.4 in chapter 5). When a new model is selected for validation the file server is informed of this fact, and prepares the model file for download. Slaves then connect to this file server and download the model.

The file server is implemented in a way that prevents more than a single slave to connect at the time, the distribution of the file to all the slaves thus happens sequentially. One possible improvement here would be to create one file server thread to serve each of the slaves. These threads could then execute concurrently. But this approach still has a source of inefficiency in that the entire model file must be read from file and written to a socket one time for each slave. Due to the fact that model files can be very large, there is no use in caching the model file.

7.4 Improving the file transfer

7.4.1 A better server implementation

What we want is to read through the entire file only once. We still have to write to each of the slaves sockets. But if we for every block read from the model file write the block to all the slave sockets before reading a new block, each block would only be read once.

To achieve this we will be to wait for all the slaves to connect, and then transfer the model file to all the slaves simultaneously. This would result in the file only being read once. For this to work the server must be rewritten so that it knows how many slaves that will connect, and doesn't begin transfer until all slaves have connected. A disadvantage of this approach is that it is less fault tolerant as it will break if any of the slaves lose their connection. The old solution simply served files on request, completely oblivious to what sort of environment it was in.

Preliminary tests of this new approach showed that the transfer time was cut in half for the largest models. But this still meant that it was a dominating step. To make the transfer even faster we would have to reduce the amount of data to transfer. This can be achieved through compression.

7.4.2 Model compression with GZip

We did some simple tests using the gzip utility that comes with Linux to see if anything could be gained from using compression. If the cost of compression was larger than the improvement in transfer time, little would be gained. It was evident from these tests that performance could be improved, but only if we used the least possible compression. The compression time increases dramatically as we increase the level of compression, but the size of the compressed data file is only marginally reduced.

Having found that gzip compression should improve performance we tried to implement it using the `java.util.zip` package. One of the advantages of the Java IO libraries is that you can interchange different stream implementations depending on your needs. In this case we could take the current streams used to transfer the model file to the slaves and replace it with a stream using GZip compression.

But this failed miserably, performance was drastically reduced instead of increased. It seems that the Java implementation of gzip is a lot slower than the implementation used by the gzip utility. We therefore decided to use the gzip utility from within Java, which means that we must compress the STEP file and write it to a temporary file which is then transferred to the slaves. The Java way is certainly a lot more elegant.

7.4.3 Performance of the improved file transfer

The new transfer stage is a great improvement over the old, all models benefit, although the improvement is very dependent on the size of the model. The larger the model the greater the improvement. Results are summarized in figure 7.3 on the next page.

7.5 Utilizing the master

As we previously noted one source of inefficiency is the fact that the master performs no validation. If the master is working very hard to keep track of the slaves and manage tasks then there is little to gain from making the master do any validation. In fact, it may even slow down the validation as the master may become so busy with validation that the task management doesn't get enough CPU cycles to handle returned task and distribute new tasks efficiently. This will mean more idle time at the slaves, and consequently decreased performance.

Making the master execute validations as well would mean increasing processing from 7 to 8 nodes. This means a potential speed increase of $\frac{1}{7}$,

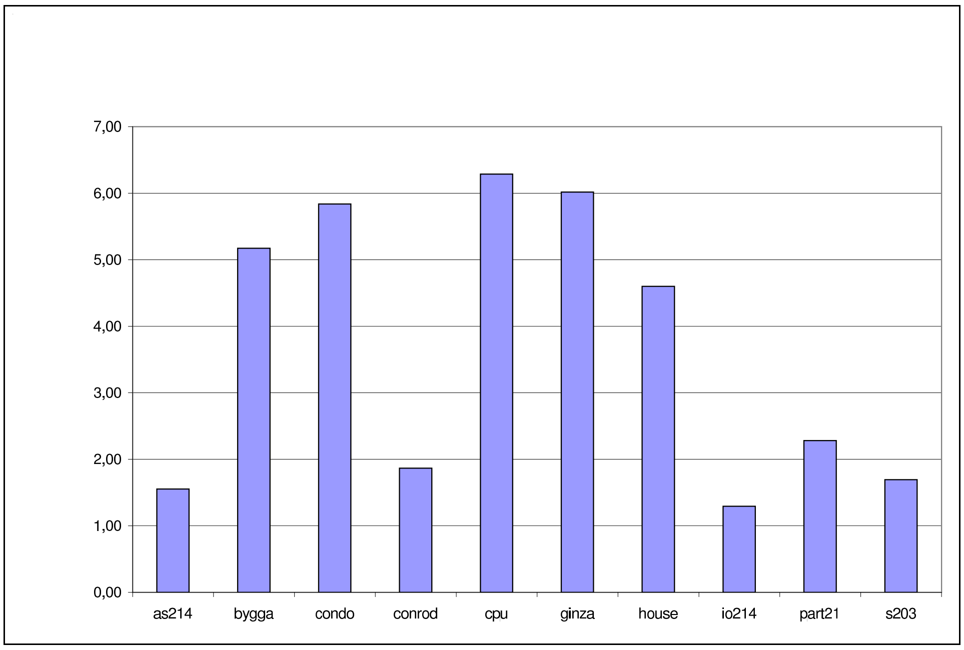

Figure 7.3: Speedup of the new transfer stage relative to the old

or about 0.14. Implementation shouldn't be difficult because the validation process is implemented in a separate thread, a thread which can be executed at the master as if it was a regular slave.

To compare solution with and without a validating master we express the performance increase by dividing the time taken using no master on the time taken using a master. The results are shown in figure 7.4 on the following page and listed in 7.1 on the next page. We see a general increase in performance, and for two models the increase is even larger than the potential increase of 14%, but this is due to measurement errors and the fact that we are working with average values. In fact, if we subtract half the standard deviation for io214 from the difference between using the master and not using the master, we find that the increase is 13%.

The effect of using the master for validation is obviously greatly affected by cluster size. The bigger a cluster the less can be gained from making the master validate, and the higher the risk that doing so will slow things down. Based on our results we can say that for clusters of 8 or less nodes it is certainly worth it, and we will use the master for validation when doing our final performance tests.

Schema	Model	Increase
AP214	**as214**	0%
	io214	19%
IFC2X2	**bygga**	16%
	condo	14%
	ginza	11%
	house	10%
AP203	**conrod**	8%
	part21	13%
	s203	9%
AP210	**cpu**	13%

Table 7.1: Performance increase when utilizing the master for validation.

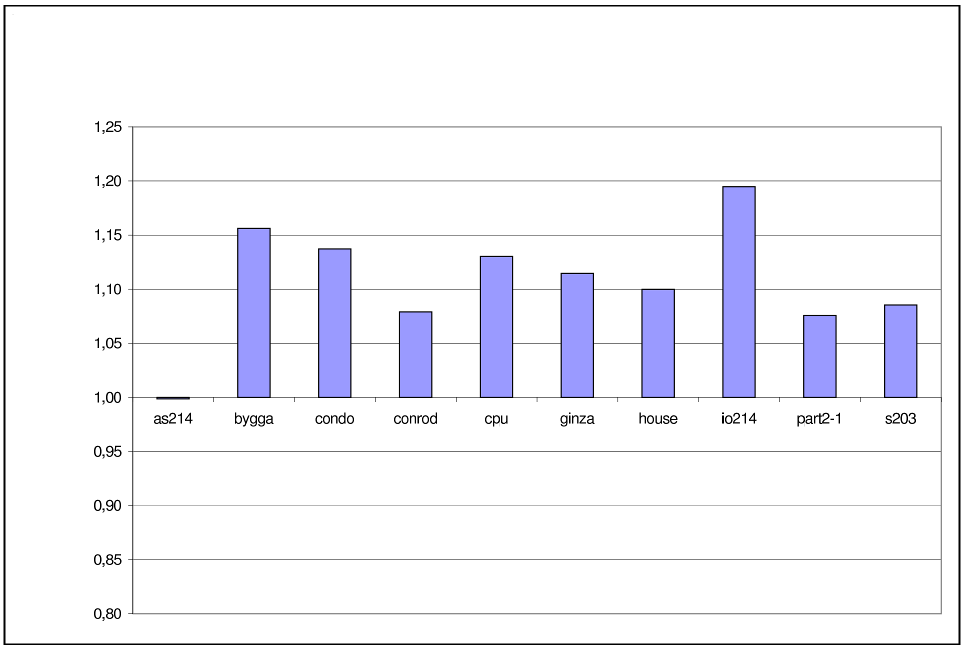

Figure 7.4: Performance increase when utilizing the master for validation.

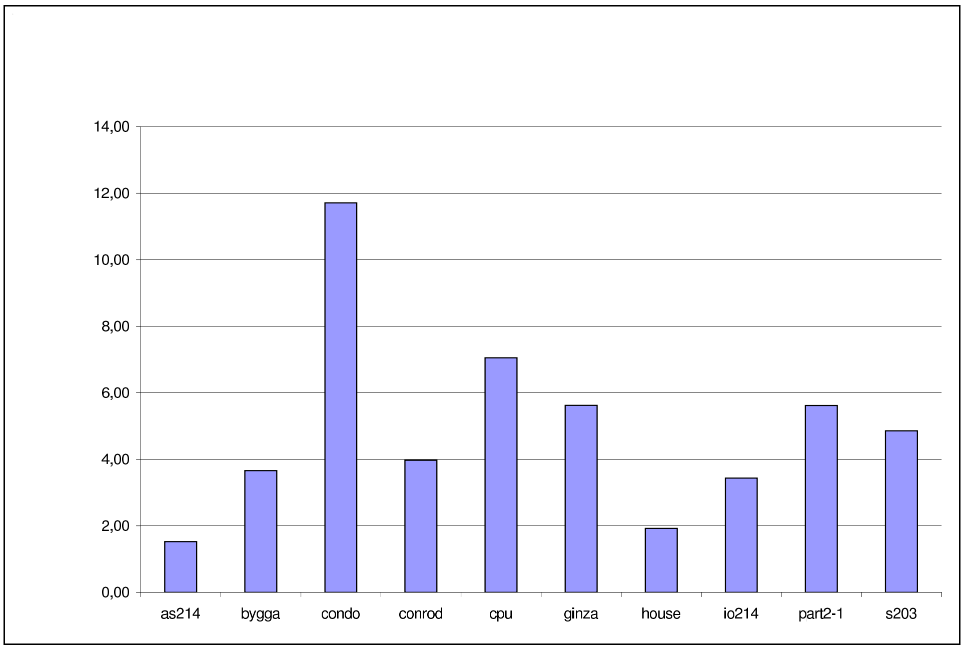

Figure 7.5: Final speedup results.

7.6 Final performance figures

With the optimizations in place we are now content with our system. It is time to look at how well this system performs. We begin by looking at the speedup for the models in our test set. The results are shown in figure 7.5 and listed in table 7.2 on the next page.

One immediately striking feature with these figures is that we have superlinear speedups. The biggest model in terms data, *conrod*, is speeded up by a factor of almost twelve.

7.6.1 Why superlinear speedup?

The speedup seen with *condo*, must have an explanation. We know that it's is not attributable to measurement variance, and the effect is too big to be caused by cache effects[1]. The most likely explanation is memory. The IFC2x2 models all tend to be data-heavy but computationally lightweight. The result of this is that a lot of memory is needed during validation. In chapter 6 we saw that a weakness in the EDM memory management code caused terrible

[1]By cache we here mean the processor cache, as we will soon see, EDM operates with it's own cache, which makes matters a lot more interesting

Schema	**Model**	**Speedup**
AP214	**as214**	1.53
	io214	3.66
IFC2X2	**bygga**	3.66
	condo	11.71
	ginza	5.62
	house	1.92
AP203	**conrod**	3.97
	part21	5.62
	s203	4.86
AP210	**cpu**	7.05

Table 7.2: Final speedup results.

performance for the sequential validation of this model, and we changed to a better reference program. Had we stuck with the old reference program, the speedup for *condo* would have been more than 30! But our current reference program does not suffer from this problem, something we are able to verify through profiling.

Returning to the EDM Memory Manager

The true explanation must lie in the way the EDM kernel works. Consider for example the validation of several instances of some entity. During the validation of these objects, a lot of temporary memory structures will be created. The intention for using these structures is to speed up the validation. For example, derived attributes really only need to be computed once, after which they can be kept in memory and referred whenever needed. Computing a derived attribute can be expensive because the data needed to compute it may reside on disk. Also, because the EDM manages it's own memory it has it's own cache. If during validation of object A we visit object B, object B can be loaded from disk. When we later need to validate object B or some other object referencing object B it will be in memory and we save ourselves the extra read.

When a validation is executed in parallel, each computer participating in the validation will validate much fewer objects. This means that the supporting in-memory data structures will be smaller and there is a reduced chance that the total memory needed by EDM outgrows the physical memory available. In addition, the chance that an object must be evicted from the EDM cache and subsequently reloaded is much smaller. The net effect is that less memory will be used, less paging will occur, and CPU utilization thus increases.

As an example, consider the validation of 16,000 instances of some entity, say `Person`. Let us say that each `Person` object has a derived attribute age, and that the age is stored as an integer requiring 4 bytes. During the sequential validation of these objects, the derived age must be computed and stored in memory consuming a total of $16,000 * 4 = 64KB$ of memory. During the parallel validation on the other hand, each node will ideally only receive 2 tasks of 1000 objects each[2]. This means that at each slave there will only be a need for $2,000 * 4 = 8KB$ of memory.

If we also say that each Person object requires 64 bytes, the sequential version will have loaded $16,000 * 64 = 1MB$ from disk and into the EDM cache. Each slave on the other hand will only have loaded $2,000 * 64 = 128KB$. If we pretend that this is executed on rather ancient hardware with only 1MB of memory available, the sequential version will have to start evicting objects from the cache once it starts validating something else. If this something else are objects that reference `Person` objects, the sequential version might have to reload the `Person` objects from disk every time such a reference must be resolved, whereas the parallel version still has plenty of room in the cache.

But it's not just the speed

One other interesting effect of this is that certain models may not be possible to completely validate at all on a single computer[3]. The operation may simply require to much memory. If the computer runs out of virtual memory, the EDM Virtual Machine cannot continue to function. By using a cluster, there is a much smaller chance that this happens, as the total virtual memory will be much bigger. Because the nodes in a cluster will most likely use very different amounts of memory during a validation one can't say that a cluster of N homogeneous nodes has a total memory size of $N * VirtualMemorySize$, but it should at least be an improvement.

7.6.2 Scalability revisited

In chapter 6 we looked at scalability by varying the node count. With the master now also executing tasks we can test with node counts of 8, 4, 2 and 1.

[2]Assuming that there are 8 nodes in a cluster and that we use the partitioning algorithm previously outlined

[3]Unless one manually validate different aspects of the model, much like doing a partitioning along the R-axis, only by hand.

		Node count			
Schema	**Model**	1	2	4	8
AP214	**as214**	70 %	55 %	37 %	19 %
	io214	71 %	53 %	50 %	43 %
IFC2X2	**bygga**	68 %	57 %	53 %	46 %
	condo	61 %	93 %	129 %	146 %
	ginza	30 %	63 %	70 %	70 %
	house	62 %	35 %	34 %	24 %
AP203	**conrod**	71 %	55 %	54 %	50 %
	part2-1	67 %	75 %	74 %	70 %
	s203	66 %	71 %	73 %	61 %
AP210	**cpu**	58 %	66 %	75 %	88 %

Table 7.3: Relative efficiency using 1, 2, 4 and 8 nodes.

We executed these tests in the normal fashion using the same batch sequence of models as always. To see how the solution scales, it is best to look at the relative efficiency for each node count. The results are listed in 7.3.

It should be noted that the single node execution was in no way optimized. Using only one node means that the only node is the master. But the master still compresses and prepares the model data for distribution, so some time is lost there.

For the three largest models, *ginza*, *condo* and *cpu* we see that the relative efficiency increases with the node count. This is probably mostly due to the memory effect we covered in the previous section. For most other models, efficiency decreases as we increase the number of nodes. This is what one would expect as the overhead due to data transfer becomes higher and the master becomes more loaded.

7.6.3 Task size distribution

It has so far been a somewhat unstated assumption that when the population of a given entity is split into subpopulations which are validated separately as tasks, each subpopulation will take about the same amount of time to validate. This assumption is based on the fact that the same operations needs to be performed on every instance in the population. But simply assuming isn't good enough, so we decided to collect some data to study the task size distribution.

To do this we use a collection of models that are created by taking the *part2-1* model and multiplying it's contents. We have versions of the model which are multiples of 2, 4, 8, 16 and 32. Quick inspection found us three entities that were quite numerous in this model: `CARTESIAN_POINT`, `DIRECTION` and `VERTEX_POINT`. We chose these because they are not trivial to validate and there are many instances of them in the models.

To study the variance of execution time we collected the execution times for each of the tasks and simply computed the range for each entity type. We then normalized this range by dividing it on the average task size, giving us a percentage value. The results of this are plotted in 7.6.

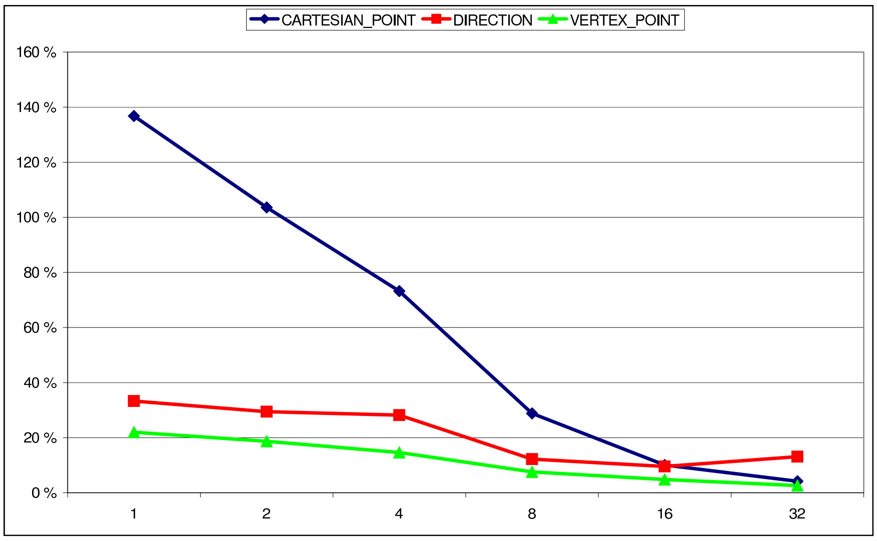

Figure 7.6: Task size variance for three entities in the part2-1 model multiplied 2, 4, 8, 16 and 32 times in size.

Our assumption is clearly invalidated by these results, variance is rather large, in particular for `CARTESIAN_POINT`. But the variance decreases as the model (and hence subpopulations) grow. This seems to suggest that as models grow larger, tasks of the same entity will be of more and more comparable size.

However, the variance between tasks of different entities is much larger than the variance between tasks of the same entity. For example, each `CARTESIAN_POINT` task on average takes three times as much time to validate as each `VERTEX_POINT` task. But because we generate so many tasks compared to the number of nodes, and because our cluster is homogeneous, this sort of variance shouldn't cause any problems.

Chapter 8

Conclusion and ideas for future work

8.1 Summary

Using clusters of cheap commodity hardware to speed up the validation of STEP models may lead to increased productivity for companies that work with large and complex models. Frequent validation is also an important factor in ensuring data quality and consistency.

We have implemented a system which is based on the EDM product suite and allows for the parallelization of data validation as performed by the EDM ModelChecker component. Our approach required only the very minor addition of a single function to the EDM kernel. Apart from this the system runs on top of a standard edition of EDM and uses only the interface exposed by EDM to perform it's operations.

Our implementation is a proof-of-concept rather than a production-ready system, error handling in particular is very simplified, and the user interface is only the most basic needed to benchmark the system.

The solution is based on a master/slave approach, and works by splitting up the validation of a model into a number of separate, independent tasks. The tasks are then distributed using the task-farming principle, which means that it is self-scheduling. The system is designed so that it should be possible to insert as a piece of middleware between a client program and the EDM suite. This means that the system will allow users to validate models without any special considerations, the parallelization happens behind the scenes.

For implementation we chose to use Java with RMI for communication allowing us to program at a comfortably high level of abstraction and benefit from the use of Java threads. The system was developed by first creating a straightforward implementation, and then tuning certain aspects of the implementation based on feedback from performance testing and profiling.

8.2 Conclusion

Results have been both positive and negative. We have seen a range of results from complete failure to superlinear speedup. But overall the results are quite good, and the three largest models in our test set are also the three best performing models.

The single biggest remaining issue is global rules. Many schemas contain at least one complicated global rule whose computational cost grows exponentially, meaning that the larger the model the more dominating this rule becomes.

The excellent performance seen with large IFC-based models is interesting not just because it shows good performance even for large, data-heavy models. IFC is the schema used by companies involved in building and construction, and there is a lot of activity in this area. Specialized systems to deal with IFC data only (frequently referred to as IFC Model Servers) are starting to appear, and such systems could definitely gain a lot from using our approach. The most interesting result here is that very large models may be almost impossible to validate on a single computer because of memory limits. A cluster solution can remedy this because the total memory available becomes larger.

It is also important to remember that most companies tend to work with only one or a very few types of product data. If a company mainly works with a schema that is not marred by the global rule problem, they could certainly benefit from a system based on our approach.

Finally we should note that the global rule problem frequently is due to poorly written rules. In most cases it is possible to rewrite a rule to execute much faster, retaining semantic equivalence. A model server specialized for a particular domain could be created that used our parallelization method but used an optimized schema.

8.3 Concluding remarks

It would have been very interesting to test the solution on a really large model from the industry, but this proved to be hard. On one occasion an engineer at EPM Technology received a very complicated model that a client was having problems with. But with the model came the orders to delete it as soon as the problem was fixed. No one else was to have access to it.

The code is not very clean, and the main class which is used to execute the master is quite messy with user interaction code mixed with the normal flow of validations. In addition the main master class and the main slave class, although quite similar in overall structure, has been written in two different ways. But after all, the system was developed with the intention of being a prototype. A proper implementation will have to use a cleaner design, do a lot more error handling, and offer a much better interface.

One of the major lessons learned during development was that debugging this sort of application is very hard. Threads make debugging harder because of the nondeterminism introduced by threads. So is the case with RMI. The use of Java native calls also makes debugging harder because simply using a normal Java debugger isn't enough, you also have to use a debugger that can interface with the underlying C code. Because our system uses threads, RMI and native calls, debugging was quite painful. This, coupled with the lack of experience, lead to a considerable slowdown in development. We should probably have considered this in more detail before we started working on the implementation. By spending more time on setting up an environment that would ease debugging, we might have saved ourselves quite a few headaches.

The use of AOP to add logging and timing wasn't a very good idea. This is partially due to the less than optimal structure of the code. We had to supplement the AOP code by adding logging and timing at certain points in the actual code because it wasn't possible to insert the necessary code using AOP. This lead to a mix of the two approaches which made making changes harder because timing and logging had to be maintained at two conceptually different levels.

We also should have considered what we wanted to time and how more thoroughly during the design phase. Our approach to this became rather ad-hoc, and the timing data was written in several different formats. This meant that we had to use different approaches to massage the timing data into usable formats, something which probably slowed down testing a bit.

8.4 Ideas for future work

8.4.1 A general parallelization

When working with STEP product data there are other operations such as the merging and mapping of models, and the execution of complex queries, that tend to take quite some time to perform. If the EDM suite could be entirely parallelized it would be possible to execute mergings, mappings and queries in addition to validations.

But implementing this can prove to be a major undertaking. While the method of parallelization achieved in this thesis was fairly simple and straightforward to implement, it still took quite some time to do.

8.4.2 Rule schema validation

Currently there is no support for rule schema validation. A rule schema contains the same type of declarations that a normal schema does, and it is always based on an underlying schema. A rule schema is basically a way to extend an existing schema with more rules. Rules can only be added to entities and types already declared in the underlying schema.

Implementing rule schema support shouldn't be very hard. Because a rule schema extends an existing schema, the set of entities will be the same. The only thing that is needed is for the system to call on the underlying EDM to validate against a given rule schema. Because additional global rules can be declared in a rule schema these must also be extracted from the rule schema and added to the task list, but this can be done in exactly the same manner as in the current solution.

Rule schemas are becoming more and more important. In the introduction to STEP we had an example of automated building approval. The rules that govern such an approval would probably be implemented as a rule schema based on the IFC schema.

8.4.3 More than one validation at the time

The current system can only validate one model at a time. Work is underway to create a multi-threaded version of EDM that can support multiple users through a standard client/server approach.

One possible approach here is to let the client act as the master, keeping copies of the necessary meta-data available at the client. The client can then generate the tasks and send them as separate operations to the server. The server must be responsible for

8.4.4 Spare cycle harvesting

Most companies involved with STEP data already have a network of fairly powerful workstations, as this sort of work requires quite a bit of processing power. The current system is tuned towards homogeneous clusters dedicated to run validations. If the system could be improved to better support regular networks of heterogenous computers, existing resources could be utilized more efficiently.

We already discussed one change to the system that would facilitate such usage in section 5.2.3 on page 57, where a slight variation on task distribution was considered. In addition, the software which acts as the slave must be nicer in terms of CPU usage. The current version tries to get as much resources as possible, something that would surely annoy anyone using one of the workstations.

The performance requirements of such a solution can probably be lowered. If you have 15 computers in your network, and adding a little bit of software allows you to speed up validation with a factor of 4, then that's probably fine because you got it almost for free. But if you go out and buy a cluster that you will dedicate to speeding up validation you presumably expect a much more efficient solution.

The biggest problem may be the speed of the network. The current solution performs well because of the gigabit interconnect. Todays local area networks normally offer no more than 100Mb/s. Data-heavy models such as the IFC models will probably suffer quite a bit in such an environment.

A second problem is memory. Because the EDM kernel requires a lot of memory to work well, the workstations that are executing a slave client may become very unresponsive even if the slave client is nice in terms of CPU, because memory runs it. It is possible to limit the memory usage of the EDM virtual machine, but doing so may result some models not being possible to validate at all.

8.4.5 A turnkey solution

Being a prototype, the current solution is not ready for the marketplace. It must be integrated better with the rest of the EDM software so that installation is painless. In addition, a special version of EDM that allows for straightforward setup of an entire cluster would be nice.

In particular this would require that more monitoring and managing software is included. There are already existing systems that offer this sort of functionality, one could either integrate these into the EDM suite, or implement something similar.

8.4.6 Tackling global rules

Global rules is the Achilles Heel of the current implementation. For entities the system works well, they are split into equally sized tasks and distributed. If the global rule problem could be handled by parallelizing it from inside the

EDM kernel, the current simple method used to parallelize the validation of entities could still be used. This would hopefully be simpler than parallelizing the entire EDM kernel. But it is also quite possible that doing so would require almost the same as parallelizing the kernel.

As we have also seen, global rules can frequently be rewritten. It would be interesting to see if it was possible to automatically optimize the execution of global rules, much like queries in SQL are optimized. If proved to be possible it could also benefit the execution of queries against models as these rely on the same EXPRESS constructs as global rules.

Appendix A

Sourcecode for the Master and Slave

A.1 RunMaster.java

```java
import java.io.*;
import java.rmi.*;
import java.util.*;
import java.net.*;
import edm.edmi.*;
import edm.sdai.*;
import edm.util.*;
import iso10303.sdai.*;
import edm.access.*;
import edm.extensions.*;

public class RunMaster {

    // Flag indicating whether a validation is in progress.
    private boolean running = false;

    // Flag indicating whether the master and slaves are running on
    // the same machine
    private boolean local = false;

    // Flag indicating whether or not to run a validation thread on
    // the master node
    private boolean runValidation;

    private boolean automode;
    private String automodeCfg; // Automode configuration file

    // Login information for the EDM Database
    private String dbPath;
    private String dbName;
    private String dbPass;

    private MasterImpl master;
    private SdaiModel model = null;

    private EDMDatabase database;

    public RunMaster(boolean local) {
        this.local = local;
        this.dbPath = System.getProperty("edm.dbpath", "/home/databases/");
        this.dbName = System.getProperty("edm.dbname", "cluster");
        this.dbPass = System.getProperty("edm.dbpass", "123");
        this.automodeCfg = System.getProperty("edm.automodecfg",
                                              "/home/edm/automode.cfg");
        this.automode = System.getProperty
            ("edm.automode", "false").equalsIgnoreCase("true");
        this.runValidation = System.getProperty
            ("master.validate", "false").equalsIgnoreCase("true");

        TaskManager taskManager = new TaskManager(this);
        ModelLoader modelLoader = new ModelLoader();
        SlaveManager slaveManager = new SlaveManager(this, taskManager);
        slaveManager.init();

        // This will serve the step files to the slaves.
        ServerThread stepServer = new ServerThread("ServerThread", 4000);
        stepServer.start();

        try {
            master = new MasterImpl(slaveManager, taskManager);
            Naming.rebind ("Master", master);
            System.out.println ("Master running....");
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(MalformedURLException e) {
            System.out.println("Invalid URL in call to rebind()");
            e.printStackTrace();
            System.exit(1);
        }

        openDatabase();

        LineNumberReader ui = new LineNumberReader
            (new InputStreamReader(System.in));
        while(true) {
            System.out.println("### Hit enter when slaves have been " +
                                "connected");
            String modelFile = "";
            try {
                modelFile = ui.readLine();
            } catch(IOException e) {
                e.printStackTrace();
            }

            Set slaves = slaveManager.getSlaves();
            int numSlaves = slaves.size();

            // Assuming a cluster size of 8, if the master also runs a
            // validation thread we need to act as if there are 8
            // slaves. But the ServerThread will only see 7 clients as
            // there is no need to download the file at the master.
            stepServer.setClients(numSlaves);

            if(this.runValidation)
                numSlaves++;
            if(numSlaves == 0) {
                System.out.println("No slaves connected, please connect " +
                                   "some slaves and try again");
                continue;
            }

            if(this.automode) {
                String[] modelList;
                try {
                    modelList = getModelList();
                } catch (IOException e) {
                    System.out.println("Couldn't find " +
                                       "/home/edm/automode.cfg, please " +
                                       "make sure it exists and try again");
                    continue;
                }
                for(int k = 0; k < modelList.length; k++) {
                    modelFile = modelList[k];
                    System.out.println("\nSELECTED MODEL: " + modelFile);
                    String modelName = "";
                    if(modelFile.indexOf('.') == -1) {
                        modelName = modelFile;
                        modelFile = modelFile + ".stp";
                    } else {
                        modelName =
                            modelFile.substring(0, modelFile.indexOf('.'));
                    }
                    if(!(new File(modelFile)).exists()) {
                        System.out.println("WARNING: Couldn't open " +
                                           "model file");
                        continue;
                    }

                    compressFile(modelFile);
                    stepServer.setFile(modelFile);
                    distributeFile(slaves);

                    try {
                        model = loadModel(modelFile, modelName);
                    } catch(SdaiException e) {
                        e.printStackTrace();
                        // If the model couldn't be loaded we stop
                        // processing by breaking out of the loop.
                        break;
                    }

                    taskManager.buildTaskList(model,
                                              this.database, numSlaves);

                    System.out.println("" + taskManager.getTaskCount() +
                                       "␣tasks");
                    System.out.println("Validating...");

                    runValidation(slaves, model, master, modelName);

                    List validationErrors =
                        taskManager.getValidationErrors();
                    if(!validationErrors.isEmpty())
                        summarizeErrors(validationErrors);
                    System.out.println("COMPLETED␣VALIDATION␣CYCLE");
                }
                // Exit once the automated mode is complete.
                break;
            } else {
                String modelName = "";
                if(modelFile.indexOf('.') == -1) {
                    modelName = modelFile;
                    modelFile = modelFile + ".stp";
                } else {
                    modelName =
                        modelFile.substring(0, modelFile.indexOf('.'));
                }
                System.out.println(modelFile);
                if(!(new File(modelFile)).exists()) {
                    System.out.println("Couldn't␣open␣model␣file");
                    continue;
                }

                compressFile(modelFile);
                stepServer.setFile(modelFile);
                distributeFile(slaves);

                try {
                    long start_load = System.currentTimeMillis();
                    model = loadModel(modelFile, modelName);
                    if(model == null) {

                    }
                    long stop_load = System.currentTimeMillis();
                    System.out.println("STEP import took " +
                                       (stop_load - start_load) + "ms (" +
                                       ((stop_load - start_load) / 1000) +
                                       "s)");
                } catch(SdaiException e) {
                    e.printStackTrace();
                    continue; // Request new model file
                }

                taskManager.buildTaskList(model, this.database, numSlaves);
                System.out.println("" + taskManager.getTaskCount() +
                                   " tasks");
                System.out.println("Validating...");

                // The task manager is ready, the slaves can start
                // executing.
                runValidation(slaves, model, master, modelName);

                List validationErrors = taskManager.getValidationErrors();
                if(!validationErrors.isEmpty())
                    summarizeErrors(validationErrors);
                System.out.println("COMPLETED VALIDATION CYCLE");
            }
        }
    }

    private void runValidation(Set slaves, SdaiModel model,
                               Master master, String modelName) {
        try {
            for(Iterator i = slaves.iterator(); i.hasNext(); ) {
                Slave current = (Slave) i.next();
                current.startValidation(modelName);
                this.running = true;
            }

            if(this.runValidation) {
                MasterValidatorThread validator =
                    new MasterValidatorThread("Validator", model, master);
                validator.start();
            }

            /* At this point we want to wait for the validation
             * process to finish. This must be signalled by the task
             * manager which cooperates with the slave manager. When
             * the TaskManager sees that the validation is complete it
             * calls the method validationCompleted in this class,
             * resulting in a notify being called on this Object.
             */
            synchronized(this) {
                try {
                    wait();
                } catch(InterruptedException e) {
                    e.printStackTrace();
                }
            }
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    private String[] getModelList() throws IOException {
        List configuredModels;
        BufferedReader in =
            new BufferedReader(new FileReader("/home/edm/automode.cfg"));
        String line;
        configuredModels = new LinkedList();
        while ((line = in.readLine()) != null) {
            configuredModels.add(line);
        }
        in.close();
        String[] finalList = new String[configuredModels.size()];
        int c = 0;
        for(Iterator i = configuredModels.listIterator(); i.hasNext();) {
            finalList[c] = (String) i.next();
            c++;
        }
        return finalList;
    }

    private void distributeFile(Set slaves) {
        for(Iterator i = slaves.iterator(); i.hasNext(); ) {
            Slave current = (Slave) i.next();
            try {
                current.loadFile();
            } catch(RemoteException e) {
                e.printStackTrace();
                System.exit(1);
            }
            this.running = true;
        }
    }

    public void allSlavesDead() {
        if(this.running) {
            validationCompleted(false);
        }
    }

    public void compressFile(String modelFile) {
        try {
            Runtime r = Runtime.getRuntime();
            String[] cmdarr1 = {"mv", "-f", modelFile, "tmpfile.stp"};
            Process p = r.exec(cmdarr1);
            p.waitFor();
            String[] cmdarr2 = {"zip", "-1", "tmpfile.zip", "tmpfile.stp"};
            p = r.exec(cmdarr2);
            p.waitFor();
            String[] cmdarr3 = {"mv", "tmpfile.stp", modelFile};
            p = r.exec(cmdarr3);
            p.waitFor();
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(-1);
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.exit(-1);
        }
    }

    public boolean isLocal() { return local; }

    public synchronized void validationCompleted(boolean success) {
        if(success) {
            System.out.println("VALIDATION_COMPLETE");
        } else {
            System.out.println("VALIDATION_FAILED!!");
        }
        notify();
    }

    public void summarizeErrors(List errors) {
        // Note that required_attribute errors have a list of missing
        // required attributes, and that the way EDM counts this is as
        // the number of missing attributes. With the method used here
        // however we simply count how many instances have missing
        // required attributes, disregarding how many attributes are
        // actually missing.
        Map errorSummary = new HashMap();
        for(Iterator iter = errors.iterator(); iter.hasNext();) {
            ValidationError cur = (ValidationError) iter.next();
            if(errorSummary.containsKey(cur.name)) {
                int cnt = ((Integer)
                           errorSummary.get(cur.name)).intValue();
                cnt += 1;
                errorSummary.put(cur.name, new Integer(cnt));
            } else {
                errorSummary.put(cur.name, new Integer(1));
            }
        }
        for(Iterator iter = errorSummary.keySet().iterator();
            iter.hasNext();) {
            String name = (String) iter.next();
            int count = ((Integer) errorSummary.get(name)).intValue();
            System.out.println(name + ":␣" + count);
        }
    }

    private void openDatabase() {
        CUserAccessController controller =
            CUserAccessController.getAccessController();
        this.database = controller.getDb();
        try {
            this.database.create(this.dbPath, this.dbName, this.dbPass);
        } catch(SdaiException e) {
            // Triggered if database exists, which is fine. What
            // we want however is a method like dbExists(). The
            // SdaiException is absolutely useless.
        }

        try {
            this.database.open(this.dbPath, this.dbName, this.dbPass, true);
        } catch (SdaiException e) {
            e.printStackTrace();
        }
    }

    private SdaiModel loadModel(String modelFile, String modelName)
        throws SdaiException
    {
        // Start importing the model
        SdaiRepository dataRep =
            this.database.getRepository("DataRepository");
        dataRep.open(SdaiModel.READ_WRITE);
        CReadStepFile reader = new CReadStepFile(modelFile,
                                                 null,
                                                              "DataRepository",
                                                              null,
                                                              modelName,
                                                              null,
                                                              null,
                                                              0);
        reader.setOptions(CReadStepFile.DELETING_EXISTING_MODEL);
        try {
            reader.read();
        } catch(SdaiException e) {
            if(e.getMessage().indexOf("Error/warning during STEP File " +
                                      "read operation") != -1) {
                System.out.println("Warning: Error/warning during STEP " +
                                   "File read operation");
            } else {
                throw e;
            }
        }
        return dataRep.openModel(modelName, SdaiModel.READ_ONLY);
    }

    public static void main(String[] args)
    {
        try {
            if(args.length > 0 && args[0].equals("-l")) {
                RunMaster runner = new RunMaster(true);
            } else {
```

A.2 SlaveImpl.java

```
import java.rmi.*;
import java.rmi.registry.*;
import java.io.*;
import java.net.InetAddress;
import java.rmi.server.*;
import java.net.*;
import java.util.*;
import java.net.UnknownHostException;
import java.text.*;
import java.text.Format.*;
import edm.edmi.*;
import edm.sdai.*;
import edm.util.*;
import iso10303.sdai.*;
import edm.access.*;
import edm.extensions.*;

public class SlaveImpl
    extends UnicastRemoteObject
    implements Slave
{

    public String masterHost;
    private Master master;
    public int id;
    private String ip;
    private boolean isLocal;
    private String dbPath;
    private String dbName;
    private String dbPass;

    public boolean importDone = false;
    public boolean importRunning = false;
    public SdaiModel model;

    private SdaiSession session;
    private SdaiRepository dataRep;
    private EDMDatabase database;

    private static final String registerFailedMsg =
        "Could not register with the master. Please try with a different " +
        "slave name";
    private static final String badUrlMsg =
        "The supplied master hostname seems to be invalid";
    private static final String notBoundMsg =
        "Failed to bind the master, make sure the master is running. " +
        "This error can also occur if the rmiregistry isn't running.";
```

```java
    private static final String noRegistryMsg =
        "Failed to contact the rmi registry , is rmiregistry running?";

    public void ping() throws RemoteException {
    }

    public SlaveImpl(String masterHost, boolean isLocal)
        throws RemoteException
    {
        super();
        this.isLocal = isLocal;
        this.masterHost = masterHost;
        this.dbPath = System.getProperty("edm.dbpath", "/home/databases");
        this.dbName = System.getProperty("edm.dbname", "cluster");
        this.dbPass = System.getProperty("edm.dbpass", "123");

        if(this.isLocal) {
            this.dbPath = this.dbPath + this.id;
        }

        try {
            this.ip = InetAddress.getLocalHost().getHostAddress();
        } catch(UnknownHostException e) {
            e.printStackTrace(); System.exit(1);
        }

        openDatabase();

        try {
            master = (Master) Naming.lookup
                ("rmi://" + masterHost + "/Master");
            // registerSlave returns a negative value if a slave
            // already has registered with the same ip as this
            // slave. Unless we are using the local version.
            id = master.registerSlave(ip, this);
            System.out.println("Received id " + id);
            if(id < 0) {
                System.out.println(registerFailedMsg);
                System.exit(1);
            }
            // If we are using the local version (testing), we must
            // make sure that the rmiregistry name is
            // unique. Otherwise the slave will simply replace other
            // slaves with the same name.
            String slaveName = isLocal ? "Slave" + id : "Slave";
            System.out.println ("Slave running as " + slaveName);
            // As soon as the slave is up and running we start sending
            // heartbeats to the master.
            initHeartBeats();
        } catch(NotBoundException e) {
            System.out.println(notBoundMsg);
            e.printStackTrace();
            System.exit(1);
        } catch(RemoteException e) {
            System.out.println(notBoundMsg);
            System.exit(1);
        } catch(MalformedURLException e) {
            System.out.println(badUrlMsg);
            System.exit(1);
        }
    }

    public void setLocal(boolean isLocal) { this.isLocal = isLocal; }
    public boolean isLocal() { return this.isLocal; }
    public Master getMaster() { return this.master; }
    public int getId() { return this.id; }

    public boolean startValidation(String modelName) throws RemoteException
    {
        System.out.println("Beginning validation of " + modelName);
        try {
            ValidatorThread validator = new ValidatorThread("Validator",
                                                            this,
                                                            this.master);
            validator.start();
        } catch(Exception e) {
            e.printStackTrace();
        }
        return true;
    }

    public void initHeartBeats()
    {
        Timer heartbeat = new Timer();
        heartbeat.scheduleAtFixedRate
            (new TimerTask() {
                public void run() {
                    keepAlive();
                }
            }, 0, 1000);
    }

    private void keepAlive() {
        try {
            this.master.slaveKeepAlive(this.id);
        } catch(RemoteException re) {
            System.out.println("KeepAlive request failed, exiting.");
            System.exit(1);
        }
    }

    private void openDatabase() {
        CUserAccessController controller =
            CUserAccessController.getAccessController();
        this.database = controller.getDb();
        try {
            this.database.create(this.dbPath, this.dbName, this.dbPass);
        } catch(SdaiException e) {
            // Triggered if database exists, which is fine. What we
            // want however is a method like dbExists(). But the API
            // doesn't contain anything like this.
        }

        try {
            this.database.open(this.dbPath, this.dbName, this.dbPass, true);
        } catch (SdaiException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public boolean loadFile() {
        if(this.importRunning) {
            System.out.println("I'm currently importing...");
            int c = 0;
            while(this.importRunning) {
                try {
                    Thread.sleep(100);
                    c++;
                } catch(InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Waited " + c + " times for import thread");
        }
        try {
            ImportThread importer = new ImportThread("ImportThread", this);
            importer.start();
            System.out.println("Importer started...");
        } catch(Exception e) {
            e.printStackTrace();
            return false;
        }
        return true;
    }

    public SdaiModel loadModel(String modelFile)
```

```java
        throws SdaiException
    {
        SdaiRepository dataRep =
            this.database.getRepository("DataRepository");
        String modelName = "tempmodel";
        CReadStepFile reader = new CReadStepFile(modelFile,
                                                 null,
                                                 "DataRepository",
                                                 null,
                                                 modelName,
                                                 null,
                                                 null,
                                                 0);
        reader.setOptions(CReadStepFile.DELETING_EXISTING_MODEL);
        try {
            reader.read();
        } catch(SdaiException e) {
            // This error is OK as it doesn't have to be fatal, and we
            // really can't detect if it is until we try to validate
            // the model.
            if(e.getMessage().indexOf("Error/warning during STEP File " +
                                      "read operation") != -1) {
                System.out.println("Warning: Error/warning during STEP " +
                                   "File read operation");
            } else {
                e.printStackTrace();
                System.exit(1);
            }
        }
        this.model = dataRep.openModel(modelName, SdaiModel.READ_ONLY);
        return this.model;
    }
```

Bibliography

[Foster95] Ian Foster, “Designing and Building Parallel Programs”

[Buyya99-1] Rajkumar Buyya ed. “High Performance Cluster Computing Volume 1: Architectures and Systems”

[Buyya99-2] Rajkumar Buyya ed. “High Performance Cluster Computing Volume 2: Programming and applications”

[Culler99] David E. Culler & Jaswinder Pal Singh, “Parallel Computer Architecture, a hardware/software approach”

[Andrews00] Gregory R. Andrews, “Foundations of Multithreaded, Parallel, and Distributed Programming”

[Lea02] Doug Lea, “Concurrent Programming in Java - Design Principles and Practices”

[Tanenbaum01] Andrew S. Tanenbaum, “Modern Operating Systems”

[Kiczales et al 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, “Aspect Oriented Programming” in *proceedings of the European Conference on Object-Oriented Programming (ECOOP).*

[Gamma et al 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design patterns: Elements of reusable object-oriented software.”

[SunRMI] Sun Microsystens, “The Java Remote Method Invocation Specification” (http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html)

[Kemmerer99] Sharon J. Kemmerer (Editor), STEP—The grand experience (NIST SP939)

[Kurzyniec, Sunderan 01] Dawid Kurzyniec and Vaidy Sunderan "Efficient cooperation between Java and native codes – JNI performance benchmark" in *In The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPDA)*, Las Vegas, Nevada, USA, June 25-28 2001.

[Pfister95] Gregory F. Pfister "In Search of Clusters — the coming battle in lowly parallel computing"