Fine-Grained Parallelism in Ellie

Birger Andersen

Department of Computer Science

University of Copenhagen

Universitetsparken 1

2100 Copenhagen, Denmark

E-mail: birger@diku.dk, Fax: +45 31 39 02 21, Phone: +45 31 39 64 66

Partial Ph.D. Dissertation [1]

October 1991

1. To appear in *Journal of Object-Oriented Programming*.

Abstract

This paper briefly presents the object-oriented programming language Ellie which supports fine-grained objects and parallelism. The paper concentrates on the parallel and fine-grained programming aspects of the language. A prototype version of the language has been implemented on a parallel transputer network.[2]

The main goal of Ellie has been to achieve machine independent parallel programming, which is a major concept to reduce the cost of parallel software development. Ellie tries to meet this goal by allowing the programmer to have a large number of small processes and then having the compiler adapt the actual number of processes used to suit the actual underlying hardware.

Keywords

Object-oriented languages, Concurrency, Process synchronization, Fine-grained programming

Acknowledgements

I would like to express my thanks to Jens Clausen, Eric Jul, and Niels Christian Juul, all three of the Department of Computer Science in Copenhagen. Their comments on the Ellie language design/implementation and on this paper have been very helpful. I also thank Bjarne Hansen for the discussions we have had on the language design and implementation and for implementation of most of the Ellie kernel.

2. The transputer is a trademark of the Inmos Group of Companies.

Table of Contents

Section 1
Introduction 5

Section 2
An Ellie Overview 7

Section 3
Objects and Processes 9

Section 4
Interface and Synchronization 13

Section 5
Fine-Grained Example 17

Section 6
Related Work 21

Section 7
Final Remarks and Conclusion 23

References 25

Index 29

List of Figures

Figure 3.1: Complex Number Object 10
Figure 4.1: Semaphore Object 13
Figure 4.2: Dinning Philosophers Program 15
Figure 5.1: Fibonacci Function 17
Figure 5.2: Fine-Grained Fibonacci Objects 18

Section 1
Introduction

Ellie is a fine-grained object-oriented programming language. It has been designed to support high level fine-grained parallel programming in a flexible and efficient way on distributed memory, parallel computers.

Another goal of the language has been to achieve extraordinary language flexibility in order to be generally usable. This has been obtained by allowing definitions of new types and control structures by reusing existing ones and by having a non-strict conformity type system. Ellie is intended to be an all-purpose programming language that can meet the efficiency of languages like Fortran and C and the flexibility and expressiveness of languages like Lisp and Smalltalk.

This paper concentrates on the fine-grained parallel programming aspects of Ellie. The other aspects are only briefly mentioned. The design and compilation aspects of the language are further discussed in [Andersen 91c] and the language is presented in [Andersen 91b] and defined in details in [Andersen 91a].

What does fine-grained parallelism actually mean? Harold Stone [Stone 87] has defined a fine-grained process to be a very small number of machine instructions - even merely one instruction or operation. The amount of communication and other overhead per process can be very high. In accordance with this definition, I will define *fine-grained parallelism* in Ellie to mean that the logically finest operations in the language, such as integer multiplications and additions, are executed as parallel processes. *Medium-grained parallelism* indicates that sequential units with a small number of operations are executed as parallel processes, such as a compound expression, and *coarse-grained parallelism* concerns sequential units of large numbers of operations such as an equation solver function. The term *fine-grained objects* in Ellie refers to objects with the smallest quantity of state information such as characters and integers. The terms *medium-grained objects* and *coarse-grained objects* refer to objects with an increasing quantity of state information such as records with a number of integer fields, files of records, and whole databases.

The motivation to design a new language is best illustrated by the following observation:

> Parallel programs are often written for a particular parallel computer. This implies a new version of a program for every new computer because the number of processors and the communication speed and technique, etc., differ from existing ones. New program versions give rise to excessive software development and maintenance costs.

A potential solution to this problem is to apply the following ideas to the new language:

1) Parallelism should be expressed by many fine-grained processes which can be combined to larger less fine-grained processes at compile-time.

2) Data should be distributed in fine-grained objects which can also be combined to larger less fine-grained objects at compile-time.

Although the ideas could also be applied to other object-oriented languages such as ConcurrentSmalltalk [Yokote 86], ABCL/1 [Shibayama 87; Yonezawa 86], Beta [Kristensen 87], Emerald [Black 87; Ray 91], and Self [Ungar 87], it has been assumed to be most convenient to design a new small language as an experimental platform.

A computer with hundreds or thousands of processors requires programs with many small fine-grained processes/objects to exploit the distributed memory and the processing power, whereas a computer with few processors handles a smaller number of less fine-grained processes/objects more efficiently. The reason is that a large number of fine-grained processes/objects located on a single processor causes more overhead than a smaller number of less fine-grained processes/objects. Therefore, I have introduced fine-grained parallelism and a technique called *compile-time grain-size adaption* that is used to adapt programs to the target computer.

It should be noticed that grain-size adaption in Ellie does not change the semantics of semantically correct programs. However, incorrect programs that run correctly at one grain-size level may run erroneously at another grain-size level because of synchronization fallacies. Hence, the programmer is encouraged to compile and run programs at more than one grain-size level in order to detect and correct such problems. Actually, the finest grain-size level will be the best for detecting problems.

Section 2
An Ellie Overview

Ellie is semantically and syntactically a rather simple language but it relies on some sophisticated ideas that all together constitute a very general and powerful language. This section briefly describes the most important ideas in Ellie.

- Classes, types, blocks, methods, and regular objects are all abstracted by the fundamental and single abstraction mechanism called an *Ellie object*. Ellie objects are computing entities of varying sizes, e.g., fine-grained. The nature of objects is similar to that in Smalltalk [Goldberg 89], i.e., all data items are objects on which methods may be applied in a uniform way. All abstractions are *first class*, e.g., they may be passed as parameters. An object is said to be *activated* when it is created (based on another existing object as a *prototype*) and passed an actual parameter. Hereafter the object will usually have a *process* (the object is *initializing*) and it is said to be *active*. When the process has terminated (the object is *initialized*) then the object is *passive*.
- Functional and operational methods are existing at the same time like in Emerald [Black 87; Raj 91]. Functions are without side-effects and operations may have side-effects. Hence, an Ellie object is said to have associated either a *functional process* (may be the empty process) or an *operational process*. The reasons for this distinction are improved understanding of parallel programming and the ability to define *immutable* objects where only functions can access the state, i.e., the state will be constant. The strength of immutable objects is that they can be copied in any number on a distributed memory computer.
- Interfaces that relax encapsulation can be defined in order to make locally defined attributes of objects accessible from outside by *remote addressing*. Attributes are simple objects being able to transfer and alter their values, etc., by activating value read/bind methods. This idea is similar to Self [Ungar 87]. *Label* and *constant* attributes (defined by `::` and `:`) are constantly bound to Ellie objects (a label object can not transfer its value), whereas *variable* attributes (defined by `:-`) dynamically can be bound to Ellie objects.
- Block structured programming is supported by Ellie like, e.g., Beta [Kristensen 87; Madsen 87], Simula [Birtwistle 79], and Emerald, but in Ellie blocks are object abstractions (closures) that locally may define attributes and objects. Pascal/Algol scope rules are associated with

attributes, the exception being that attributes are known inside the whole block in which they are defined. This promote recursive declarations.

- Conformity type checking is exploited to type check programs statically for efficiency reasons. The type system is analogous to that implemented in Emerald [Black 87] and Guide [Krakowiak 90]. Type checks are also available to the programmer at run-time in order to widen views of objects. Run-time type errors are only possible when widening views.
- Genericity and polymorphism [3] are available. Efficient code can often be guaranteed, since the compiler is able to compile optimally when the implementations of parameter objects are known at compile-time. Otherwise, the flexibility of conformity types is utilized instead, and the compiler generates less efficient tag-checking code.
- Delegation and multiple inheritance is supported by an Ellie delegation mechanism called *interface inclusion*. Delegation routes may dynamically be changed. Because requests may be delegated to a number of objects and the delegation mechanism takes care of name collisions, multiple inheritance abstraction is also available. The compiler decides whether to use delegation or inheritance at the code level.
- Fine-grained parallelism and synchronization is expressed explicitly by means of *bounded* and *unbounded* (remote) procedure/method calls (RPC and URPC) activating *non-future objects* and *future objects*. In the unbounded case, the result is returned as a *future result object* which is an abstract object with an undefined state. This approach is also known from ABCL/1 [Yonezawa 86]. An Ellie mechanism called *dynamic interface*, which is about as powerful as Path Expressions [Campbell 74], is used for synchronization at all levels of parallelism. A dynamic interface may delay processes trying access to local attributes. Parallelism and synchronization is the main subject for the remaining part of this paper after introducing objects and processes.

3. Genericity, polymorphism, and other language concepts are considered in details in [Watt 90].

Section 3
Objects and Processes

All items in Ellie are objects. An Object may contain an (optional) process. An object may be very fine-grained, for example the representation of an integer, or more coarse-grained, for example the representation of a database. Therefore, the process of an object may also range from very fine-grained to very coarse-grained.

An Ellie object representing what is termed an *object* in Smalltalk and other languages will usually have no process or a process that merely initializes the state of the object. An Ellie object representing what is termed a *method* in Smalltalk and a *procedure* in other languages will usually have a process changing the state of its surrounding object (the *origin object*) but no state itself.

An object with a *bounded process* returns its output parameter when the process has terminated (RPC). An object with an *unbounded process* returns its output parameter in an undefined state when activated (URPC). The state of the output parameter will then be defined when the process has terminated. The object representing the undefined state (the *future result object*) delays any requests to access the state but the reference to it may be used immediately.

Objects may be defined to be either *functions* or *operations*. In the first case, the processes are not allowed to have any side-effects; in the latter case, the processes are allowed this possibility. This is compile-time determinable since side-effects are defined in a conservative way. The difference between functions and operations facilitate compile-time reduction of the number of functional processes.

Now let us consider a part of an implementation of complex numbers in Ellie (see Figure 3.1) which uses an existing implementation of real numbers. The name `complex` is constantly bound (indicated by colon) to the object defined by the succeeding object defined by the object constructor `[ ... ]`. The implementation is straightforward but it should be noticed, that the expressions are defined in a sugared syntax form with operator preceding levels of a more clumsy punctuated expression syntax. Expressions are evaluated from right to left with respect to operator preceding levels and parenthesis. The methods operating on complex numbers are defined to be future functions which means that they are without side-effects and activations will produce fine-grained parallelism. A formal input parameter is defined right to `<:` and a formal output parameter right to `:>`.

A complex number object is created by an activation with a record [4] conforming to the formal input parameter as actual input parameter:

```
complex [ r : 2.25; i : 0.5 ] [];
```

4. The record must be an activated object. A record defined by the definition `[ ... ]` may be activated by providing a simple actual input parameter `[]` (the simplest object in Ellie).

```
complex : [ function <: c : parameter :> self;
   export *, +, =, ...;  $ Interface definition
   parameter  :  [  $ Input parameter type definition
                    r, i : real
                 ];
   $ Definition of methods that operate on complex numbers:
   *  :  [ future function <: p : complex :> o :- complex;
            rp, ip :- real;
            rp <- c.r * p.r - c.i * p.i;
            $ Un-sugared form of the expression above:
            $ rp.<- (c.r.* p.r.-> []).- c.i.* p.i.-> [];
            ip <- c.r * p.i + c.i * p.r;
            o <- complex [ r : rp; i : ip ] []
         ];
   +  :  [ future function <: p : complex :> o :- complex;
            o <- complex [ r : c.r + p.r; i : c.i + p.i ] []
         ];
   =  :  [ future function <: p : complex :> o :- Boolean
            o <- (c.r = p.r) & (c.i = p.i)
         ];
   ...
];
```

Figure 3.1: Complex Number Object

This expression returns the created complex number object, i.e., a reference to it. Notice, that the process of creating the object is a bounded functional one and that the complex number object locally defines functions only. The object is then *immutable* and the reference may be substituted by a copy of the object itself at the implementation level without semantic problems.

Let us now, as an example, try to multiply a complex number variable by a complex number constant:

```
cplx :- complex;                                          (a)
cplx <- complex [ r : 2.25; i : 0.5 ] [];                 (b)
cplx <- cplx.* complex [ r : 1.5; i : 1.0 ] [];           (c)
```

In (a) we define a name `cplx` variably bound to a complex number object. This name may dynamically be bound to any object conforming to the complex number object. In (b) we activate (by RPC) a complex number object and bind it to the name `cplx`. In (c) we then activate (by URPC) a complex number multiplication object, which is defined by the object constantly bound to the punctuated name `cplx.*`, by activating it (by URPC) with an actual input parameter being a new activated complex number object. The process of creating (and initializing) the multiplication object is an unbounded functional one. This means that the name `cplx` temporarily will be bound to an undefined state object. This is the future result object which is to some extent similar to a future object in ABCL/1 [Shibayama 87; Yonezawa 86] and Cantor [Athas 87]. Let us continue the example with an succeeding addition:

```
cplx <- cplx.+ complex [ r : 2.75; i : -1.5 ] [];         (d)
```

It is impossible in Ellie, opposed to ABCL/1, directly to determine whether a name is bound to a future result object or an object with a defined state but the language guarantees that an undefined state is never accessed before it has been defined. Therefore, the activation of the complex addition object will be delayed until the process of the multiplication object has terminated and thus has defined the state. But since the process of the addition object is also an unbounded one, the name `cplx` will temporarily be bound to a new future result object. Hence, the process evaluating the expression (d) will not be delayed at all. Only processes attempting to access names in the interface of a future result object will be delayed.

If it is critical for some process whether a certain object is in fact a future result object or an object with defined state, then an inspection object with an unbounded process may be defined in several ways in order to resolve this.

Usually in most object-oriented languages, numbers and characters, etc., are value attributes whereas objects are reference attributes. As opposed to this, reals, integers, and characters, etc., are all objects, i.e., reference attributes, in Ellie. They are implemented in the same manner as complex numbers, i.e., methods to process them are unbounded functions and the objects are immutable. The abstract types are defined by *abstract object definitions* (objects with all expressions left out) available to the programmer. The expressions to be evaluated in order to, for example, add two integers, are only implemented in a lower level code but the objects are not some kind of simple objects like in most other languages. In order to meet the higher efficiency of using value attributes, the compiler can decide (in case of immutable objects) to represent a reference by a tagged value or just a value in case of code specialization.

Section 4
Interface and Synchronization

The *object interface* defines which of the locally defined names of an object are to be accessible from outside the object by *remote addressing* when the process of the object has terminated. Until then, the names are inaccessible, i.e., processes trying to access them will be delayed.

An Ellie object may define either an *export interface* or an *accept interface* in order to make one or more locally defined names accessible by processes of other objects having references to the object. The export interface has already been mentioned. The accept interface is an extension to the export interface and will now be further elaborated.

The accept interface allows the process of the object and processes of local objects to dynamically change the interface and for periods to encapsulate names defined to be a part of the interface. By encapsulating names temporarily, process synchronization and state protection are possible. This is a very important quality in a fine-grained world where a large number of processes may all try to modify the state of one object. The *dynamic interface* synchronization mechanism is similar in powerfulness to Path Expressions [Campbell 74].

```
semaphore : [ function <: val :- integer :> self;
   accept wait, signal;

   wait :: [ operation;
      val <- val - 1;
      option [
         $ Activating the Ellie selective statement
         if : val = 0;    $ Value to be tested
         then : [ include [ signal :: any ]]
      ]
   ];

   signal :: [ operation;
      val <- val + 1;
      option [
         $ Activating the Ellie selective statement
         if : val = 1;    $ Value to be tested
         then : [ include [ wait, signal :: any ]]
      ]
   ]
];
```

Figure 4.1: Semaphore Object

The example in Figure 4.1 shows an Ellie implementation of a semaphore object. Objects being activations of this object will always allow activations of the object constantly bound to the name `signal`, whereas it will only allow activations of the object bound to the name `wait` if the integer bound to the name `val` (the

semaphore value) is one or greater. Whenever the semaphore value is decreased to zero, the name `wait` is encapsulated temporarily until the semaphore value is increased to one. This is done by the activation of `include` [5] with an actual parameter defining which names that shall be included in the accept interface from now on. By this manner, activations of the semaphore wait object will then from time to time be suspended awaiting an activation of the semaphore signal object.

Mutual exclusion is guaranteed in Figure 4.1 since the accept interface encapsulates all names as long as a process activated through the interface has not terminated. This may at first seem equivalent to the monitor concept [Hoare 74; Brinch Hansen 75] implemented in, e.g., Emerald [Hutchinson 87a, 87b; Ray 91] but Ellie allows as an exception to the monitor concept unbounded processes to execute freely inside an object with accept interface since it is only a restricted interface. When a bounded process activated through the interface has terminated, the interface will no longer encapsulate the names last included in the interface by the last activation of `include`. The interface reacts to the parameter return.

In the design of Ellie, the monitor concept has been considered in order to guarantee mutual exclusion. Protected regions as in, e.g., ABCL/1 [Shibayama 87; Yonezawa 86] has also been considered. It was, however, concluded that these concepts might be too restrictive in a fine-grained world. On the other hand, the Ellie dynamic interface concept will definitely cause programming fallacies if unbounded processes are activated without attention, so the programmer has to be careful when writing fine-grained synchronizing processes. This is a tradeoff between safety and freedom to generate parallelism.

The semaphore object can be used in a deadlock and starvation free Ellie implementation of Dijkstras solution to the Dinning Philosophers problem (see Figure 4.2).

An interface definition also allows requests to a name to be delegated to the same name defined locally within another object and accessible from outside that object. The special Ellie delegation mechanism is called *interface inclusion*. The inclusion definition specifies which remote names, i.e., names in the interface of another object, to make accessible for users of itself. It does not specify whether to make the names accessible by delegation or inheritance which is considered the same at the language level. The compiler decides which code to generate. Consider, as an example, the following interface definition of an object *X*:

```
export local_1, (local_2), (local_3).(remote_1, remote_2);
```

This interface makes a locally defined name `local_1` accessible by users of *X*. It also makes accessible all names which are accessible from the object bound to the locally defined name `local_2` and the two names `remote_1` and `remote_2` accessible from the object bound to the locally defined name `local_3`. The names

5. The name `include` is bound to a locally defined built-in object that has the ability to change the accept interface of a surrounding object.

```
program :: [

   fork :: vector
      [ length : 5; of : [ elem : semaphore 1 ] []] [];
      $ A vector of 5 semaphores initialized
      $ to the value one

   room : semaphore 4;
      $ A semaphore initialized to the value 4

   philosopher :: [ future operation <: no : integer;
      loop [  $ An endless loop
         $ Think
         room.wait [];
         (fork#no).wait [];
         $ The fork vector is indexed by no
         (fork#((no + 1) % 5)).wait [];
         $ Eat
         (fork#no).signal [];
         (fork#((no + 1) % 5)).signal [];
         room.signal []
      ]
   ];

   philosopher 0;
   philosopher 1;
   philosopher 2;
   philosopher 3;
   philosopher 4
];
```

Figure 4.2: Dinning Philosophers Program

`local_2` and `local_3` will be inaccessible by users of *X* (encapsulated).

The delegation routes may dynamically be changed. This might in case of the object *X* be done by rebinding the name `local_2` to another object. If we think in terms of inheritance, this will cause the inherited names (and the objects bound to them) to be substituted with another set of identical names (and new objects bound to these names) inherited from another object. The name `self` is implicitly defined by any object to refer to the object itself. This diverges from most class/inheritance based languages like, e.g., Smalltalk [Goldberg 89]. Accordingly, actions must be taken explicitly in order to start the name search anywhere else than in the object itself.

If a name is included in an interface more than once, only the first version will be accessible. This is a simple solution to the name collision problem. By explicitly mentioning each remote name, the correct version can always be selected by the programmer. Dynamic selection between versions can be implemented by writing objects to make the decisions.

Section 5
Fine-Grained Example

As a last fine-grained example, we will consider the Ellie object in Figure 5.1 which when activated makes its process compute the *n*th Fibonacci number in parallel by recursively activating the object itself hereby generating a number of unbounded functional processes.

```
Fibonacci : [ future function <: N : integer
   :> fib :- integer;
   option [
      if : N > 1;
      then : [
         fib <- Fibonacci (N - 1) + Fibonacci (N - 2)
      ];
      else : [
         fib <- N
      ]
   ]
];
```

Figure 5.1: Fibonacci Function

When activated, e.g., by evaluating the expression `Fibonacci 5`, the definition in Figure 5.1 will spawn a binary tree of 15 Fibonacci objects with unbounded processes. The object takes an integer *n* as actual input parameter and returns the *n*th Fibonacci number as output parameter (future result object). The name `option` refers to a control structure object that activates its input parameter and hereby binds the constant `if` to the value of `N > 1`. Depending on the value of `if`, the uninitialized object bound to either the constant `then` or the constant `else` will be activated. The operator names >, −, and + are bound to fine-grained objects with fine-grained processes that are also activated.

In Figure 5.2 we examine the first level of the process/object structure generated, i.e., before the recursive activations, when the fibonacci object is activated by the simple expression `Fibonacci 3`. Each box contains in the top the expression activating the process represented by the box. In the bottom of the boxes the returned result, if applicable, is shown. The tilde (~) means a future result object, whereas the mark (') is used to distinguish different versions of objects with the same state. The double lined boxes represent unbounded processes, whereas the single lined boxes represent bounded processes. Process no. 5 will not be activated until the future result object ~<true> has been evaluated to the concrete object <true>. Processes generated by process no. 4, 6, 7, 8, 9, and 10 have been left out for convenience.

An important idea of Ellie is compile-time combination of fine-grained processes to larger processes and fine-grained objects to larger objects when the target

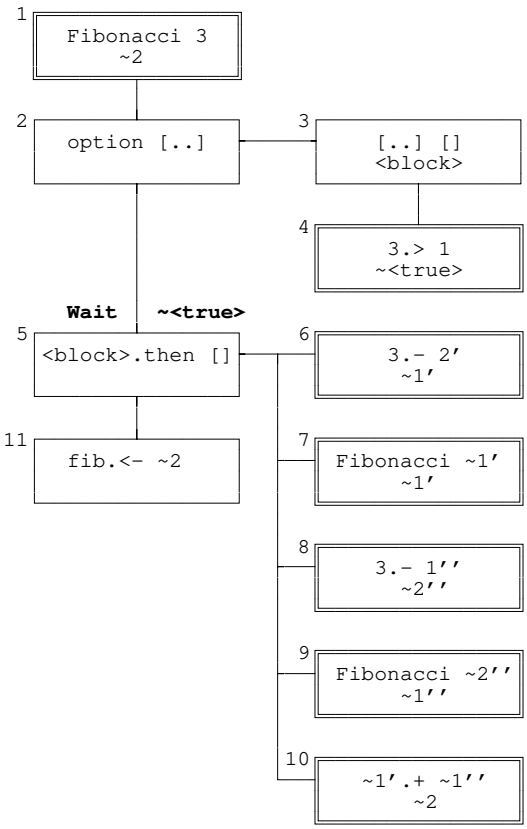

Figure 5.2: Fine-Grained Fibonacci Objects

computer for a program compilation does not efficiently support the fine-grained level of programming supported by Ellie. In the case of the Fibonacci object, we observe:

1) The object has an unbounded functional process. The semantics of Ellie allows the process to be reduced to a bounded functional process thereby removing much of the fine-grained parallelism from the Fibonacci object. It is also legal to have two versions, one with an unbounded process and the other with a bounded process in order to maintain more parallelism.

2) The actual parameter object for the activation of `option`, the object bound to the name `then`, and the object bound to the name `else` are implicitly defined to have bounded operational processes. Bounded processes may immediately be included in their activating processes. This is done by code inclusion and stack inclusion (enrolling). It is also legal to have both included and not included versions.

3) The names `Fibonacci`, `option`, `then`, and `else` are constantly bound to objects know at compile-time, i.e., manifest concrete object types. Hence, it is possible to actually include code and stack at the place of activation as suggested above.

4) If the fibonacci object is initially activated by evaluation of the expression `Fibonacci 3` (or with another parameter of the integer object type) then it is known at compile-time that the names `N`, `fib`, and `if` will be bound to integer objects (not just objects conforming to an integer object). Supposing the compiler knows what an integer object actually is (a immutable object with a state of one machine word), the references to these objects can be replaced with integer values (machine words) not needed to be tagged since the code can just be specialized.

By now it seems clear that an activation of the Fibonacci object can be replaced by sequential code that activates no objects at all, but we must not forget the recursion making it inefficient just to include code and stack, i.e., unroll the calls (could be infinite enrollment). A decision must be taken to stop at a certain level on which it is necessary to create two new Fibonacci objects. That decision (and other grain-size adaptation decisions) can be taken by the compiler based on architectural information on the target computer. This is further discussed in [Andersen 91c].

It must be noticed, that the fine-grained example above easily can be compile-time grain-size adapted. In real parallel applications written in Ellie there are a lot of further problems to be aware of. In the worst case, only very little can be done. It must also be noticed, that some run-time grain-size adaption combined with intelligent load balancing for parallel computers is indispensable in order to achieve a real high performance. Further works will be carried out on these subjects in the future.

Section 6
Related Work

The idea of compile-time grain-size adaption/reduction seems to be new. Hence, related work is difficult to find here. Parallelizing sequential code is the usual subject to study, not the opposite, although there have been a few interesting approaches to more general grain-size determination for efficient execution [Kruatrachue 88; McCreary 89] that my additional work relates to.

Cantor [Athas 87] is to my knowledge the only truly fine-grained object-oriented language existing besides Ellie. Cantor has been designed at Caltech in the mid-eighties and its purpose is lower level parallel machine programming. Therefore, the flexibility is somewhat higher than Ellie and so is the amount of code to write. The language relies as opposed to Ellie on classes and dynamic typing. Inheritance is unknown to Cantor. Cantor has only simple types and data of these types are not objects. Parallelism is generated by URPCs and parameters may be supplied later on by explicit communications. Hence, parallel multiplication like in the Ellie expression `a * b + c * d` requires more work to be done by the programmer.

ABCL/1 [Shibayama 87; Yonezawa 86] is a medium-grained language. It has been designed at Tokyo Institute of Technology in the early eighties. Parallelism is generated by message passing. *Now* type messages are equivalent to RPC, whereas *future* type messages are equivalent to URPC. *Past* type messages are equivalent to URPC with no result returned. Processes synchronize by means of atomic sections. ABCL/1 has simple types (atoms) and lists. Unlike in Ellie, it is difficult to evaluate an expression like `a * b + c * d` in parallel. Although it would be possible by writing a lot more code, this indicates that ABCL/1 is not a fine-grained language.

MELD [Kaiser 89] is in many ways an interesting language which, like Ellie, supports multiple levels of parallelism including the fine-grained level. It has been designed at Columbia University in the late eighties. Fine-grained parallelism is generated by writing unordered statements. Sequential statements may also be written. Apparently, parallelism in the multiplication expression is established by writing two unordered multiplication statements and a third addition statement to be executed after the two unordered. This means that temporary variables are needed. Methods can be invoked synchronously (RPC) and asynchronously (URPC). Methods executing in parallel synchronize by means of atomic blocks (a whole method may be an atomic block).

Section 7
Final Remarks and Conclusion

An Ellie compiler generating fine-grained virtual code has been implemented and is running on MS-DOS based Personal Computers and Unix based VAX systems, Sun 3 and Sun 4, and HP 9000 workstations. The compiler is written in C (about 6000 lines). All semantic checks have been implemented in the compiler. This includes conformity type checks. The object fragmented fine-grained virtual code (a byte code) is only slightly optimized. More work has to be done in order to realize the generation of more efficient and optimally optimized fine-grained code.

A kernel including an interpreter has also been implemented. The kernel runs on the same computers as the compiler and is also written in C. Run-time conformity type checks, object comparisons, floating point numbers, and vectors/arrays have currently not been implemented. Some optimizations remain also to be done, including the implementation of a memory garbage collector.

The kernel has also been moved to a parallel network consisting of 16 T800 transputers by adding a message router and a naive load balance mechanism to the kernel [Hansen 91]. A more powerful load balance mechanism to be developed will later on decide when to move objects in order to optimize the object distribution dynamically. Although this parallel implementation already has shown interesting results in terms of speedup measurements, not much can be concluded on this at the moment. The speedup factor 11 has been observed for a `fibonacci 12` activation (see Figure 5.1) but this is without any compile-time adaption. A grain-size adaption compiler are to be implemented in order to generate practical results in adapting Ellie programs to different parallel computers. Future research will show which adaption methods are the most attractive and interesting results can be expected later from this ongoing research project.

I conclude that Ellie in several aspects seems to be a promising language for programming future fine-grained computers and for programming computers already existing. Ellie seems to be the language known today that most easy can express fine-grained parallelism at a high abstraction level. Future work must show if the ideas in Ellie also will succeed in practice when more complicated and meaningful Ellie applications are available.

References

[Agha 86] Gul Agha. An overview of Actor langauges. *ACM SIGPLAN Notices*, 21(10):13-22, October 1986.

[Agha 87] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 49-74, The MIT Press, 1987

[Andersen 91a] Birger Andersen. *Ellie Language Definition Report*. PhD thesis (partial), DIKU Report no. 91/3, Department of Computer Science, University of Copenhagen, Denmark, June 1991 (2nd ed.). Also *ACM SIGPLAN Notices*, 25(11):45-64, November 1990 (1st ed.).

[Andersen 91b] Birger Andersen. Ellie - A General, Fine-Grained, First Class Object Based Language. Research Paper, Department of Computer Science, University of Copenhagen, Denmark, October 1991. To appear in *Journal of Object-Oriented Programming*.

[Andersen 91c] Birger Andersen. *Grain-Size Adaption in the Fine-Grained Object-Oriented Language Ellie*. PhD thesis (partial), DIKU Report no. 91/5, Department of Computer Science, University of Copenhagen, Denmark, 1991. In preparation.

[Athas 87] William C. Athas. *Fine Grain Concurrent Computations*. PhD thesis, Technical Report 5242, Department of Computer Science, California Institute of Technology, Pasadena, California, May 1987.

[Birtwistle 79] Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden, 1979.

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65-76, January 1987.

[Brinch Hansen 75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199-207, June 1975.

[Campbell 74] R.H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions in operating systems. In *Operating Systems Proceedings*, April 1974. Lecture Notes in Computer Science, (16):89-102, Springer-Verlag, 1974.

[Goldberg 89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Publishing Company, 1989.

[Hansen 91] Bjarne Hansen. *Object Activation in a Transputer Implementation of the Ellie Language*. MSc thesis, DIKU Report no. 91/6, Department of Computer Science, University of Copenhagen, Denmark, March 1991.

[Hoare 74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549-557, October 1974.

[Hutchinson 87a] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, Technical Report 87-01-01, Department of Computer Science, University of Washington, Seattle, Washington, January 1987.

[Hutchinson 87b] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. *The Emerald Programming Language*. DIKU Report no. 87/22, Department of Computer Science, University of Copenhagen, Denmark, October 1987. Also Technical Report 87-10-07, Department of Computer Science, University of Washington.

[Kaiser 89] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush, and Shyhtsun Felix Wu. MELDing multiple granularities of parallelism. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 147-166, Cambridge University Press, July 1989.

[Krakowiak 90] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, and X. Rousset de Pina. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, 3(3):11-22, October 1990.

[Kristensen 87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The Beta programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7-48, The MIT Press, 1987.

[Kruatrachue 88] Boontee Kruatrachue and Ted Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, (1):23-32, January 1988.

[Madsen 87] Ole Lehrmann Madsen. Block-structure and object-oriented Languages. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 113-128, The MIT Press, 1987.

[McCreary 89] C. McCreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32(9):1073-1078, September 1989.

[Shibayama 87] Etsuya Shibayama and Akinori Yonezawa. Distributed computing in ABCL/1. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 91-128, The MIT Press, 1987.

[Stone 87] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley Publishing Company, Massachusetts, 1987.

[Ungar 87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1987. ACM SIGPLAN Notices, 22(12):227-242, December 1987.

[Yokote 86] Yasuhiko Yokote and Mario Tokoro. The design and implementation of ConcurrentSmalltalk. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1986. ACM SIGPLAN Notices, 21(11):331-340, November 1986.

[Yonezawa 86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1986. ACM SIGPLAN Notices, 21(11):258-268, November 1986.

Index

Abstract object 2, 8, 11, 25
Abstraction 7, 8, 23
Accept interface 13, 14
Actors 25
Actual input parameter 9, 10, 17
Actual parameter 7, 14, 18
Beta 6, 7, 26
Classes 7, 21
Coarse-grained 5, 9
Conformity 5, 8, 23
Delegation 8, 14, 15
Distributed memory 5-7
Dynamic interface 8, 13, 14
Emerald 6-8, 14, 25, 26
Encapsulation 7
Export interface 13
Fibonacci function 17
Fine-grained 1, 2, 5-9, 13, 14, 17, 18, 17-19, 21, 23, 25
Functions 7, 9-11
Future objects 8
Future result object 8-11, 17
Grain-size adaption 6, 19, 21, 23, 25
Inclusion 8, 14, 18
Inheritance 8, 14, 15, 21
Input parameter 10, 9, 10, 17
Interface inclusion 8, 14
Keywords 2
Medium-grained 5, 21
Mutual exclusion 14
Object activation 26
Object constructor 9
Object interface 13
Operations 5, 7, 9
Origin object 9
Output parameter 9, 17
Path Expressions 8, 13, 25
Polymorphism 8
Remote name 15
RPC 8-10, 21
Scope rules 7
Semaphore 13-15, 14
Side-effects 7, 9
Simple objects 7, 11
Synchronization 2, 6, 8, 13, 25
Type checking 8
Type checks 8, 23
URPC 8-10, 21