A Parallel Framework for Computational Science

Fernando Rubio and Ismael Rodríguez*

Departamento de Sistemas Informáticos y Programación
Universidad Complutense, 28040 – Madrid, Spain
{fernando,isrodrig}@sip.ucm.es

Abstract. Parallel languages based on skeletons allow the programmer to abstract from implementation details, reducing the development time of the parallelizations of large applications. Unfortunately, these languages use to restrict the set of parallel patterns that can be used. The parallel functional language Eden extends the lazy functional language Haskell with expressions to define and instantiate process systems. These extensions also make possible to easily define skeletons as higher-order functions. By doing so, skeletons can be both defined and used in the same language, using a high level of abstraction. Due to these facts, the advantages of skeleton-based languages are kept in Eden, while we do not inherit the restrictions they have, as the set of skeletons can grow as needed. Moreover, in our approach the sequential code of the programs can be written in any language supporting a COM interface.

Keywords: Parallel computing, skeletons.

1 Introduction

Due to the size of the applications in computational science, it is particularly important to be able to take advantage of parallel architectures to reduce the computation time. Unfortunately, conventional parallel programming languages use to require too much programming effort to correctly implement the parallel versions of the applications. Moreover, usually the parallelization of these programs heavily depend on the underlying architecture. Thus, porting a program to a different machine is not a trivial task at all.

Fortunately, during the last years several parallel languages (see e.g. [8,1,7] based on skeletons have been developed. A skeleton [2] is a *parallel problem solving scheme* applicable to certain families of problems. For example, the divide&conquer family can be abstracted in a single skeleton. The specific functions to be performed in the nodes of the process topology are abstracted as parameters. Thus, for instance, to parallelize a mergesort it is enough to specify it in terms of the divide&conquer method, while the actual parallel implementation will be delegated to the underlying skeleton.

The main advantages of using skeletons are two. Firstly, the parallelization effort is reduced, as predefined skeletons can be used; secondly, the probabilities of errors are reduced a lot, as the programmer does not need to handle all the gory details of the

* Work partially supported by the Spanish CICYT projects AMEVA and MASTER, and by the Spanish-British Acción Integrada HB1999-0102.

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2658, pp. 1002–1011, 2003.

parallelization. However, skeleton-based languages use to restrict the set of skeletons available, so that all the programs must fit somehow a structure representable with those available skeletons. Fortunately, the language we present in this paper allows the user both to use predefined skeletons, and to define new ones. Let us remark that being able to extend the set of skeletons is a very important issue, as programmers can add new skeletons specific of their particular working areas.

In this paper we show how the Eden language can be used to define skeletons dealing with any process topology. These skeletons will not only be simple *schemes* that the programmer can *follow:* They will be actual programs parameterized by the code that need to be executed in each processor. Moreover, the programmer will be able to slightly modify the skeletons in case he want to include any characteristic particular to his working area. Thus, for each area of computational science, it could be possible to adjust the skeletons to the corresponding peculiarities.

Let us remark that, as Eden is a functional language, it is needed some knowledge of the functional paradigm to understand how to *define* new topologies. However, it is not necessary this knowledge to *use* the topologies, and it is even possible to *modify* them with only some knowledge.

The rest of the paper is structured as follows. In the next section we introduce the basic features of our language, while in Section 3 we present how we can use our language to develop generic topologies. In Section 4 we show the actual speedups obtained with one application running in a Beowulf architecture. Finally, in Section 5 we present our conclusions.

2 The Eden Language

Eden [9,6] extends the (lazy evaluation) functional language Haskell [11] by adding syntactic constructs to explicitly define processes. A new expression of the form `process x -> e` is added to define a *process abstraction* having variable x as input and expression e as output. Process abstractions can be compared to functions, the main difference being that the former, when instantiated, are executed in parallel.

Process abstractions are not actual processes. In order to really create a process, a *process instantiation* is required. This is achieved by using the predefined infix operator #. Given a process abstraction and an input parameter, it creates a new process, and it returns the output of the process. Each time an expression `e1 # e2` is evaluated, the instantiating process will be responsible for evaluating and sending e2, while a new process is created to evaluate the application `(e1 e2)`.

Once a process is running, only fully evaluated data objects are communicated. The only exceptions are lists, which are transmitted in a *stream-like* fashion, i.e. element by element. Each list element is first evaluated to full normal form and then transmitted. Concurrent threads trying to access not yet available input are temporarily suspended. This is the only way in which Eden processes synchronize. Notice that process creation is explicit, but process communication (and synchronization) is completely implicit.

In addition to the previous constructions, a process may also generate a new *dynamic channel* and send a message containing its name to another process. The receiving process may then either use the received channel name to return some information

to the sender process (*receive and use*), or pass the channel name further on to another process (*receive and pass*). Both possibilities exclude each other, to guarantee that two processes cannot send values through the same channel.

Eden introduces a new expression `new` (*ch_name*, *chan*) *e* which declares a new channel name *ch_name* as reference to the new input channel *chan*. The scope of both is the body expression *e*. The name should be sent to another process to establish the communication. A process receiving a channel name *ch_name*, and wanting to reply through it, uses an expression *ch_name* `!*` e_1 `par` e_2 . Before e_2 is evaluated, a new concurrent thread for the evaluation of e_1 is generated, whose result is transmitted via the dynamic channel. The result of the overall expression is e_2, while the communication through the dynamic channel is a side effect.

Let us remark that it is trivial to extend the previous constructions to provide functions that deal with list of dynamic channels. For instance, the following function creates a list of n dynamic channels, where `[]` denotes an empty list, while `x:xs` is a list with `x` as head, and with `xs` as tail:

```
generateChannels 0 = []
generateChannels n = new (cn,c) ((cn,c) : generateChannels (n-1))
```

while the next function sends a list of values through their dynamic channels, and returns the evaluation of its second argument:

```
sendValues [] e           = e
sendValues ((v,ch):more) = ch !* v par sendValues more e
```

In most situations —in particular in all the topologies presented in this paper— by using only process abstractions and instantiations it is possible to create the same topologies that could be created by using dynamic channels. However, dynamic channels can be used to *optimize* the efficiency of the implementations. In this sense, this feature can be seen as an optimization using a low-level construct provided by the language rather than as a radically new concept.

Let us remark that, in contrast to most parallel functional languages, Eden includes high-level constructions both for developing *reactive* applications and for dynamically establishing direct connections between any pair of processes. This allows handling *low-level* parallel features that cannot be used in conventional functional languages. Thus, Eden provides an intermediate point between very high-level parallel functional languages (whose performance use to be poor), and classical parallel languages (which do not allow using high-level constructions). We do not claim that Eden can obtain optimal speedups, but it can obtain quite *acceptable* speedups with small programming effort.

Eden's compiler[1] has been developed by extending the most efficient Haskell compiler (GHC [4,10]). An important feature of Eden's compiler is that it reuses GHC's capabilities to interact with other programming languages. Initially, GHC only allowed to include C code, but currently it also provides a COM interface. Thus, the sequential

[1] The compiler can be freely downloaded from `http://www.mathematik.uni-marburg.de/inf/eden`

parts of our programs could be written in any language supporting COM interfaces. So, Eden can be used as a coordination language, while the computation language can be, for instance, C.

In order to easily port the compiler to different architectures,[2] Eden's runtime system has been designed to work on top of a message passing library. In the current compiler, the user can choose between PVM [3] and MPI [13].

3 Defining Topologies

In this section we present how processors topologies can be expressed in Eden. For the sake of clarity, we start presenting the most simple skeleton (`map`), then we introduce an example of how to define a simple topology of processors (a ring), which can be easily extended to deal with other typical topologies as a grid or a torus. After this introductory example, we present how to define a general topology dealing with any graph of connections amongst processors.

3.1 Defining Simple Skeletons

Let us remark that process abstractions in Eden are not just annotations but first class values which can be manipulated by the programmer (i.e. passed as parameters, stored in data structures, and so on). This facilitates the definition of skeletons as higher order functions. The most classical and simple skeleton is `map`. Given a list of inputs `xs`, and a function `f` to be applied to each of them, the Haskell specification is as follows:

```
map f xs  =  [f x | x <- xs]
```

This can be trivially parallelized in Eden using a different process for each task:

```
map_par f xs =  [pf # x | x <- xs]
                where pf = process x -> f x
```

The process abstraction `pf` wraps the function application `(f x)`. It determines that the input parameter `x` as well as the result value will be transmitted on channels.

Let us remark that developing skeletons in Eden is relatively easy. Due to the lack of space, we only show the simplest example, but many other skeletons have already been implemented, and in most of the cases their source code fit in half a page. Details about their implementation can be found in [12].

3.2 Ring Topology

A ring is a well-known topology where each process receives values from its left neighbor and sends values to its right one. Additionally, the first and last processes are also

[2] Currently, we have tested it on Beowulf clusters of up to 64 processors running Linux, on clusters of workstations running Solaris, and on a shared memory UltraSparc machine running also Solaris.

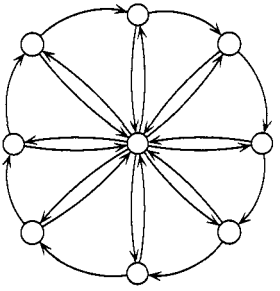

Fig. 1. A ring topology

considered to be neighbours, to form a real ring. In addition to that, all the processes can communicate with the main one — See Figure 1. This topology is appropriate for uniform granularity algorithms in which the workers at the nodes perform successive rounds. Before the first round, the main process sends the initial data to the workers. After that, at each round, every worker computes, receives messages from its left neighbour, and then send messages to its right neighbour. Eden's implementation uses lists instead of synchronization barriers to simulate rounds.

In order to create the topology, the skeleton receives two parameters: the worker function `f` that each of the processes will perform, and the initial list of `inputs` that will be provided initially to the processes. Let us remark that the length of such list will be the same as the number of processes in the ring. Let us also remark that the function `f` receives an initial datum from the parent and a list of data from the left neighbour, and it produces a list of data for its neighbour and a final result for its parent. In the following piece of code (that includes the whole skeleton), the `ring` function creates the desired topology by properly connecting the inputs and outputs of the different `pring` processes. As we want processes to receive values from its previous process, it is only necessary to shift the outputs of the list of processes before using them as inputs of the same list. Each `pring` receives an input from the parent, and one from its left sibling, and produces an output to the parent and another one to its right sibling:

```
ring f inputs = outsToParent   where
    outs    =  [(pring f) # outA' | outA' <- outs']
    (outsToParent,outsA) = unzip outs
    outsA' = last outsA : init outsA
    outs'  = zip inputs outsA'

pring f = process (fromParent, inA) -> out
    where out = f (fromParent, inA)
```

The previous definition can be optimized by using the lower level constructions of the language, that is, the *dynamic channels*. Fortunately, it is not neccessary to repeat the design, as we can take advantage of the methodology defined in [12] to automatically

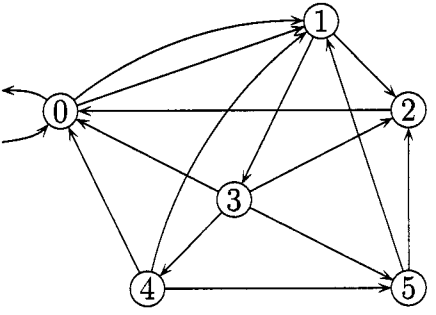

Fig. 2. A graph topology

transform high level definitions into lower level ones. The main idea of such transformation (not shown due to lack of space) is that a new dynamic channel is created by each process desiring to receive a value, and the name of such channels are to be sent to the producers of the values. Thus, by applying the transformation, now each `pring` receives an input from the parent, and a channel name to be used to send values to its sibling, and produces an output to the parent and a channel name to be used to receive inputs from its sibling, as shown below.

```
pring f = process (fromParent,outChanA) -> out
   where out = new (inChanA, inA)
                   let (toParent,outA) = f (fromParent,inA)
                   in outChanA !* outA par (toParent,inChanA)
```

Let us remark that it is trivial to extend the ring skeleton to deal with two-dimensions. In fact, in [12] it can be found the definition of several topologies, like a grid or a torus.

3.3 A General Graph

In order to proof the expressive power of our approach, we will now describe the most general topology: a graph of processes. Let us remark that conventional skeleton-based languages do not provide such a topology, even though it is becoming more and more important. Traditionally, parallel computers only used specialized topologies, and a general graph was not very unuseful. However, nowadays due to the wide implantation of the Internet, parallel computation is not restricted to specialized expensive architectures: several standard computers connected to the web can cooperate in the computation of a single problem. So, the actual topology they are using is a general graph. Thus, a parallel language should facilitate the use of such topologies.

As in the previous examples, in order to define a graph, we need to take care of two tasks: Defining the behaviour of the processes, and defining the connection topology. In this case, we will directly present the optimized implementation using dynamic channels, which has been obtained taking as basis the corresponding definition using only process abstractions and instantiations, as in the ring case.

Each process will be identified by a unique number, and its behaviour will be parameterized by the function to be computed, and by the list of identities of the processes that have direct connections with it. The source code is the following, where the process uses `generateChannels` to create as many dynamic channels as input connections are needed, while `sendValues` is used to actually send the output values:

```
pabs i f senders = process chansOut -> sends
  where channelsIn = generateChannels (length senders)
        (namesIn,valuesIn) = unzip channels
        valuesOut = f valuesIn
        sends = sendValues (zip ChansOut valuesOut)
                           (zip namesIn senders)
```

Let us remark that there must be an initial process in the graph, as shown in Figure 2. The difference with the other processes is that it will receive the inputs of the problem, and it will return the outputs. That is, it will be the interface of the topology with the outside world. The definition of this initial process is done in the same way as a normal one, but receiving an extra input `ins` and producing an extra output value `outs`:

```
pabs0 f senders = process (chansOut,ins) -> (sends,outs)
  where channelsIn = generateChannels (length senders)
        (namesIn,valuesIn) = unzip channels
        (outs:valuesOut) = f (ins:valuesIn)
        sends = sendValues (zip ChansOut valuesOut)
                           (zip namesIn senders)
```

Once the basic processes have been defined, we only need to properly connect them. In order to do that, we need as parameters the list of functions `fs` to be computed by each process, the list of connections `c` amongst processes, and the inputs `ins` of the initial process. In order to access to the function that process i should perform, it will be enough to use the predefined operator `!!`, that extracts the i-th element from a list. The connection topology only need to create $n-1$ *normal* processes, and one extra *initial* process. After doing that, function `reorganize` uses the information encoded in the list of connections in order to establish them in the right way:

```
graph fs c ins = outs       where
  sendss = [(pabs i (fs!!i) (senders i)) # (receivers!!i)
              | i <- [1..n-1]]
  (sends0,outs) = (pabs0 (fs!!0) (senders 0)) # (receivers!!0,ins)
  senders i = map fst (filter ((== i).snd) c)
  receivers = reorganize (sends0:sendss)
  reorganize xss =  toList2 . sort2 . concat
  n = length fs
```

Let us remark that, even though it could seem that the previous program is complex to understand, it is not really important to understand the details of it. The important thing is that it can be written in a compact way and, most importantly, it can be used without understanding it: It will only be necessary to pass as parameters the list of connections, and the list of behaviours.

Let us also remark that it is completely trivial to nest several subgraphs. That is, in case several graphs have been defined to solve a set of problems, it is straighforward to create a new graph connecting all the subgraphs. In order to do that, it is enough to consider each subgraph as a function of the list `fs`. When doing that, each subgraph will be a node of the overall graph, obtaining the desired topology. The reason why this can be done so easily is that Eden processes are first-class citizens in a higher-order language. Thus, they can be used as parameters of other functions or processes.

4 Measuring an Application: Interactions amongst Particles

In this section we present the actual results obtained for one example using the previous topologies. For the sake of clarity, we have chosen a not too complex example, using a simple method to compute the forces amongst a set of particles, but we have already used our approach to parallelize other complex examples, like a simulator of the evolution of the economy of a country. Unfortunately, this simulator is too complex for explaining it briefly, but the speedups we have obtained with it are similar to those described in the following example.

The experiments have been performed in a 64-processor Beowulf cluster at the University of St. Andrews. Nodes are 450MHz Pentium II running Linux RedHat 5.2, with 348MB of DRAM and connected through a CISCO 2984G full duplex 100Mb/s fast Ethernet switch, being the latency $\delta = 142\mu s$. So, it is a low cost environment with high latencies. Eden RTS was running on top of PVM 3.4.2. Due to administrative reasons, it has not been possible to use all the processors in the tests.

Let us assume that we want to determine the force undergone by each particle in a set of n atoms. The total force vector f_i acting on each atom x_i, is

$$f_i = \sum_{j=1}^{n} F(x_i, x_j)$$

where $F(x_i, x_j)$ denotes the attraction or repulsion between atoms x_i and x_j.

This constitutes an example of pairwise interactions. For a parallel algorithm, we may consider n independent tasks, each devoted to compute the total force acting on a single atom. Thus, task i is given the datum x_i and computes $\{F(x_i, x_j) \mid i \neq j\}$. It is however inconceivable to have a separate process for each task when dealing with a large set of particles, as it is usually the case. Therefore, we distribute the atoms in as many subsets as the number of processors available. We use a ring structure, so that all the data can flow around. In the first iteration, each process will compute the forces between the local particles assigned to it. Then, in each iteration it will receive a new set of particles, and it will compute the forces between its own particles and the new ones, adding the forces to the ones already computed in the previous iterations.

In the next program, function `force` solve the problem by creating a ring of processes. The list of particles `xs` is split into `np` parts, being `np` the number of processors of the ring. Afterwards, the forces are computed by using the sequential function `force'`:

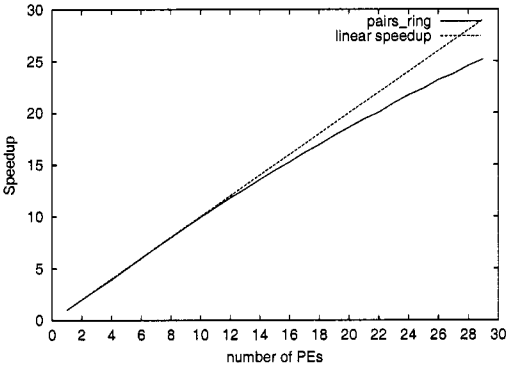

Fig. 3. Speedups of pair interactions

```
force xs = ring (force' np) (splitIntoN np xs)
force' np (local,ins) =  (total,outs)
  where outs          = take (np - 1) (local : ins)
        total         = foldl1' f forcess
        f acums news  = zipWith addForces acums news
        forcess       = [map (faux ats) local | ats <- (local:ins)]
        faux xs y     = sumForces (map (forcebetween y) xs)
        sumForces l   = foldl' addForces nullvector l
```

Let us remark that it has not been necessary to deal with processes in the definition of the example. Let us also remark that the definition of `force'` could have been done in other programming language, as C. In fact, the efficiency could be improved not only by using a more efficient language, but also by using more efficient algorithms. The important point is that the main difficulty will be finding the right sequential solution, while parallelizing it will not require much effort.

Figure 3 shows the speedups obtained using 7000 particles, the sequential execution time being 194.86 seconds. As we pointed out when presenting the language, the speedups obtained are *acceptable*. In fact, in this particular example the speedups are quite good, although in other examples with less inherent parallelism the speedups are not so high. Anyway, our results are always *competitive* with those obtained by using C+MPI: Although slightly slower when running, our applications are developed much faster.

5 Conclusions

In this paper we have presented a framework that facilitates the task of parallelizing large applications. By using our language, the programmer can concentrate on the development of the sequential applications, while the parallelization effort will be minimized. The main advantage of our language is the combination of high-level facilities (that enable fast development) and lower-level constructions (that improve the efficiency).

The experiments we have performed so far are encouraging. As it can be seen in [5], our efficiency is always comparable to that obtained by using C+MPI: Although our results are always slightly worse, the programming effort needed is much smaller. It is important to remark that in [5] we have already compared the efficiency of Eden and other two parallel languages: GpH [14] and PMLS [7]. The first one represents the state of the art in parallel functional programming, while the second one is a good representative of the skeletons community. The results obtained were encouraging: Even though we are still working on optimizing our Eden compiler, the runtimes obtained were better (on average) than those obtained with the other (more mature) languages.

References

1. G. H. Botorog. *High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms*. PhD thesis, RWTH-Aachen, January 1998.
2. M. Cole. *Algorithmic Skeletons: Structure Management of Parallel Computations*. MIT Press, 1989. Research Monographs in Parallel and Distributed Computing.
3. A. Geist, Ad. Beguelin, J. Dongarra, and W. Jiang. *PVM: Parallel Virtual Machine*. MIT Press, 1994.
4. S. L. Peyton Jones, C. V. Hall, K. Hammond, and W. Partain. The Glasgow Haskell Compiler: a Technical Overview. Department of Computer Science, University of Glasgow, December 1992.
5. H. W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, Á. J. Rebón Portillo, S. Priebe, and P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-Order and Symbolic Computation*, 2003. To appear.
6. R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*, pages 95–128. Springer-Verlag, 2002.
7. G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Journal of Parallel Algorithms and Applications*, 16:181–206, August 2001.
8. S. Pelagatti. *Structured Development of Parallel Programs*. Taylor and Francis, 1998.
9. R. Peña and F. Rubio. Parallel Functional Programming at Two Levels of Abstraction. In *PPDP'01*, pages 187–198. ACM Press, September 2001.
10. S. L. Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *ESOP'96 — European Symposium on Programming*, volume 1058 of *LNCS*, pages 18–44, Linköping, Sweden, April 22–24, 1996. Springer-Verlag.
11. S. L. Peyton Jones and J. Hughes. Report on the Programming Language Haskell 98. Technical report, February 1999. `http://www.haskell.org`.
12. F. Rubio. *Programación Funcional Paralela Eficiente en Eden*. PhD thesis, Dpto. Sistemas Informáticos y Programación. Universidad Complutense de Madrid (Spain), November 2002. In Spanish.
13. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1996.
14. P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *Programming Language Design and Implementation (PLDI'96)*, pages 79–88. ACM Press, 1996.