E—A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs

Martti Forsell
Computing Platforms
VTT Electronics
Box 1100, FIN-90571 Oulu
Finland

Abstract: As systems on chip are evolving to networks on chip (NOC) providing a unified communication infrastructure for a number of computational resources, being able to easily implement computational tasks as a parallel program that can be efficiently executed by multiple resources together is becoming increasingly important. Recent advances in thread-level parallel (TLP) architectures have made it possible to implement efficiently an easy-to-use synchronous shared memory programming model (Parallel Random Access Machine, PRAM) on a NOC. In this paper we describe a novel programming language, called e, for fine-grained TLP programming on synchronous shared memory NOC architectures realizing the PRAM model. The language uses a familiar c-like syntax and provides support for shared and private variables, arbitrary hierarchical groups of threads, and synchronous control structures. This allows a programmer to use various advanced TLP programming techniques like data parallelism, divide-and-conquer technique, different blocking techniques, and both synchronous and asynchronous programming style. We will also shortly experiment the e-language with real parallel programs using our experimental e-compiler and scalable Eclipse NOC architecture.

Key-Words: Thread-level parallel programming languages, parallel random access machine, networks on chip

1. Introduction

The two main challenges in designing future systems on chip are managing the increasing complexity and redesigning architectures and related design methodologies due to the radical change in silicon properties with future technologies. Among the most promising attempts to solve these problems is the network on chip (NOC) paradigm providing a unified communication infrastructure for a set of computational resources and proposing a new type of design methodology aiming to support reuse and parallelism at all levels [1]. Unfortunately this is not a trivial task and current NOC proposals more or less fail in supporting properly parallelism at the thread-level. Up to certain degree they do support currently very popular type of parallelism, instruction-level parallelism (ILP), in which multiple instructions are tried to execute simultaneously in multiple functional units (FU) belonging to a single processor. ILP can not be a main type of parallelism in NOCs because large amounts of ILP is hard to extract [2]. ILP lacks the possibility of exploiting control parallelism, the complexity of an efficient ILP machine increases quadratically in respect to the number of FUs due to need for forwarding, and the clock cycle of an ILP NOC will inevitably become very slow because signal propagation from an edge to another on a chip will take even tens of clock cycles with future silicon technologies [1]. Thread-level parallelism (TLP), i.e. executing a program consisting of subtasks in multiple processors to solve a computational problem, avoids most of these problems by inherently supporting control parallelism, being much easier to extract, and being resistant for long latencies. Efficient exploitation of TLP requires determining how the functionality and data related to subtasks are distributed to the threads (or processors), how threads communicate with each other, and how synchronicity is maintained between the subtasks. TLP architectures can be classified to message passing architectures and shared memory architectures according to the main method of communication. In message passing architectures (MPA) threads communicate by sending messages to each others and a programmer is responsible for explicitly defining communication, synchronizing subtasks, and describing data and program partition-
ing between threads making MPAs difficult to pro-
gram. In shared memory architectures (SMA) com-
munication happens just by referring to the shared
memory, data and program partitioning happens by
allocating data structures from the shared memory
making programming much easier. Unfortunately
most SMAs consist of multiple interconnected proces-
sor-cache pairs making cache coherency and syn-
tchronicity maintenance very expensive. Recent
advances in TLP architectures [3, 4], however, suggest
that it possible to implement a cacheless and synchro-
nous SMA realizing the parallel random access
machine (PRAM) model [5] on a NOC.

In this paper we describe a novel programming lan-
guage, called e, for fine-grained TLP programming on
synchronous SMA NOCs realizing the PRAM model.
The language uses a familiar c-like syntax and pro-
vides support for shared and private variables, arbi-
trary hierarchical groups of threads, and synchronous
control structures. This allows a programmer to use
various advanced TLP programming techniques like
data parallelism, divide-and-conquer techniques, dif-
ferent blocking techniques, and both synchronous and
asynchronous programming style [6]. We made also a
short experimentation with the e-language by imple-
menting a number of familiar TLP algorithms with e,
compiling them with our experimental e-compiler and
executing them in our scalable high performance
Eclipse NOC architecture [4].

1.1 Related work

There exist a few TLP programming languages that
have quite similar properties as the language intro-
duced in this paper. These languages are targeted for
various PRAM models realized either as a simulator
or as an experimental hardware. Among the best
known are:

- Fork A feature-rich parallel programming language
  supporting parallely recursive and synchro-
nous MIMD programming using a c-like syn-
tax [6]. Fork is targeted for arbitrary concurrent read concurrent write (CRCW) PRAM
model with a fixed number of threads and
hardware support for efficient multiprefix
operations. The model is realized e.g. in the
SB-PRAM parallel computer developed in the
University of Saarbrücken [6].

- ll A parallel language supporting parallely
  recursive and synchronous MIMD program-
  ming using Pascal-like syntax [7]. ll allows
  for a (virtual) processor to spawn new ones
  executing the loop body in parallel giving bet-
ter support for the theoretical PRAM model
  with unbounded set of processors.

Some other PRAM languages, like pm2 [8] and
Modula-2* [9], are more data parallelism oriented and
provide no support for explicit group concept. None of
these, however, can be implemented with a standard c-
compiler nor they are able provide direct support for
the two-component programming model of the
Eclipse combining both ILP and TLP exploitation in
an efficient way [10].

1.2 Organization of the paper

The rest of the paper is organized so that in section 2
we describe the novel e-language for fine-grained TLP
programming on synchronous SMA NOCs. Our short
experimentation implementing a number of real paral-
el algorithms with e-language, compiling them with
our experimental e-compiler and executing them on
the different configurations of the Eclipse architecture
is described in section 3. Finally in section 4 we give
our conclusions.

2. E-language

E-language is a novel TLP programming language
created by the author especially for synchronous
shared memory NOC architectures, but it can be used
also for other (multichip) synchronous shared memo-
ry architectures like the IPSM [11]. It can be used as
an integral part of the application development flow in
which computational problems are transformed to ILP
and TLP optimized TLP binaries for the Eclipse archi-
tecture [10] (see Figure 1). The syntax of e-language
is an extension of the syntax of familiar c-language. E-
language supports parallely recursive and synchro-
nous MIMD programming for various PRAM models
including the exclusive read exclusive write (EREW)
PRAM model, the TLP programming model of
Eclipse.
2.1 Variable declaration and referencing

Variables in e-language can be shared among a group of threads or they can be private to a thread. Private variables are expressed in a similar way than variables in c-language. Shared globals are expressed with adding a "_" to the end of their identifier. For example declaration

```plaintext
int source_[1024];
```

defines a shared global source_, which is a 1024 element table of integers. Shared locals are declared after private locals within the brackets `begin_shared_locals_def` and `end_shared_locals_def`, e.g.

```plaintext
begin_shared_locals_def
int table[65536];
char c;
int i;
end_shared_locals_def
```

and should be followed immediately by a block in which the shared locals are used. This block is declared with the brackets `begin_shared_locals_block` and `end_shared_locals_block`. Referencing to shared locals happens through a shared stack pointer `locals` casted to type `Shared`, e.g.

```plaintext
(Shared)locals->X
```

where X is the identifier or expression referencing to a shared variable, e.g. `table[20]` assuming the declaration above. Shared variables cannot be used as modal parameters or as a result value of a function. If an actual parameter is a shared variable, private copies of value or reference will be used in the function execution.

2.2 TLP expressions

To support high-level TLP expressions threads are automatically numbered from 0 to the number of threads - 1 as new groups are created. The thread numbering can be accessed by the built-in variables

```plaintext
_thread_id
_number_of_threads
```

Sometimes a programmer may prefer a static thread numbering, which is invariant across the group boundaries. The static numbering can be accessed by the built-in variables

```plaintext
_absolute_thread_id
_absolute_number_of_threads
```

2.3 Thread groups and control structures

E-language supports hierarchical groups of threads. In the beginning of a program there exists a single group containing all threads. A group can be divided into subgroups so that in each thread of the group is assigned into one of the subgroups. A subgroup may be split into further subgroups, but the existence of each level of subgroups ends as control returns back to the corresponding parent group. Dividing the current group into two subgroups happens by using the statement

```plaintext
_if_else_(c,s1,s2);
```

in which a thread will be assigned to subgroup s1 if condition c holds for it otherwise it will be assigned to...
subgroup s2. As a subgroup is created, variables _thread_id and _number_of_threads are updated to reflect the new situation. As the subgroups join back to the parent group in the end of the statement the old values of these variables are restored.

Synchronous shared memory NOC machineries, like Eclipse, guarantee synchronous execution of instructions at machine instruction level. In e-language synchronicity through control structures having private enter/exit conditions can be maintained with special versions of control structures if_, if_else_, while_, do_while_ and for_ supporting automatic synchronization at the end of the structure, _if_, _if_else_, _while_, _do_while_ and _for_ supporting automatic automatic subgroup creation, and _if_, _if_else_, _while_, _do_while_ and _for_ supporting automatic synchronization and subgroup creation (see Figure 2). Asynchronous control structures with private enter/exit conditions if, if-else, while, do-while and for (using the conventional c-language syntax) can be used only at the leaf level of group hierarchy. Entering to an asynchronous area happens by using an asynchronous control structure and returning back to the synchronous area happens by an explicit group-wide barrier statement synchronize assuming all threads of the group will reach the barrier.

2.4 Example program

Let us consider a recursive version of the randomized parallel quicksort algorithm [13] providing O(log² N) execution time with a high probability in an EREW PRAM machine. The synchronous e implementation of the algorithm is shown in Figure 3. In an Eclipse this version will execute in O(log N) with a high probability because it utilizes Eclipse specific constant time primitive spread_ and automatic group creation structures _if_else_ and _if_.

3. Experimentation

In order to experiment e-language in real parallel programming we wrote e-language versions of five TLP programs representing widely used primitives of parallel computing [13] (see Table 1). The programs we compiled with our experimental e-compiler with the level 2 optimizations (-O2) and external ILP optimization (-ilp) on [10], and executed in three configurations of our scalable Eclipse NOC architecture with the IPSMSim simulator [14]. The parameters of the configurations are listed in Table 2.

For each benchmark and configuration pair we measured the execution time of the program excluding the initialization of data, the utilization of FUs, the source code size and executable size. The results of our measurements are shown in Figure 4.

The rough scalability of execution time can be seen clearly from the curves of the benchmarks except rsort, in which the size of the input data is linearly dependent on the number of threads. The slight vari-
ance compared to a pure linear scalability is due to the randomized nature of communication and that the number of threads is fixed making larger configurations relatively weaker than smaller ones. The utilization of FUs does not achieve the level measured in our earlier tests suggesting that the quality of the code produced by the gcc is not as good as that of hand compiling. The sizes of executables remained modest although the start-up code and Eclipse-related runtime libraries were included. The reason for this behavior is that we used -mtraps flag in compilation substituting some I/O routines with operating system traps, the benchmarks were not I/O intensive, and processors used VLIW coding.

In general, turning the relatively simple benchmark algorithms to e-programs was as straightforward as we expected. However, it should be remembered that TLP programming is not necessary trivial to those used to sequential programming.

4. Conclusions

We have described a new parallel programming language, called e, for fine-grained TLP programming on synchronous shared memory NOC architectures realizing the PRAM programming model. It uses a familiar c-like syntax and provides a versatile set of control structures, a concept of arbitrary hierarchical thread grouping, shared and private variables as well as support for synchronous and asynchronous programming styles. With these means a programmer is able to express the (parallel) functionality in a sophisticated way, make sure that the application is executed efficiently in the TLP hardware, and avoid pitfalls of current shared memory programming. According to our

---

### Table 1. The benchmark programs.

<table>
<thead>
<tr>
<th>Processors</th>
<th>4 (E4), 16 (E16), 64 (E64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per processor</td>
<td>512</td>
</tr>
<tr>
<td>Functional units</td>
<td>4</td>
</tr>
<tr>
<td>Bank access time</td>
<td>12 processor clock cycles</td>
</tr>
<tr>
<td>Bank cycle time</td>
<td>15 processor clock cycles</td>
</tr>
<tr>
<td>Length of FIFOs</td>
<td>16</td>
</tr>
</tbody>
</table>

### Table 2. The configurations used in evaluations.

- A parallel program that moves a block of integers in the shared memory from a location to another
- A parallel program that applies finite response filter to a table of given integers in the shared memory
- A parallel program that finds the maximum of given table of random integers in the shared memory
- A parallel program that calculates the prefix sums for given table of integers in the shared memory
- A parallel program that sorts given table of T random integers in the shared memory using the iterative randomized parallel quicksort algorithm, where T is the number of threads. (An iterative version of the algorithm shown in Figure 3)
experimentation with an experimental e-compiler and scalable Eclipse NOC architecture, e-language provides ease of use and scalable performance.

Our future work includes refining some details of the language and compiler. We plan also a thorough evaluation of the whole application development scheme being developed for the Eclipse architecture using general purpose parallel applications as well as some digital signal processing domain applications. Finally, we hope to be able to enhance the architecture of Eclipse so that it would provide at least partial support for stronger PRAM models like CRCW. The first steps towards this direction have already been taken [12]. From the e-language side there is no obstacle to use such a strong model even now.

References: