拡張可能プリプロセッサを用いた Java 上でのデータ並列プログラミング

イブ・ルージエ　一杉裕志
roudier@etl.go.jp　ichisugi@etl.go.jp
(STA フェロー)

電子技術総合研究所

本論文では拡張可能 Java プリプロセッサ EPP と、その応用例として実装した Tiny Data-Parallel Java について述べる。EPP は Java プログラムに対するプリプロセッサで、エンドユーザは Java のソースコードの先頭に EPP plugin を指定することで Java の文法を拡張し新しい言語機能を利用することができる。Tiny Data-Parallel Java は、Java の上で簡単なデータ並列プログラミングを可能にするもので、EPP を使った translator と runtime system から構成される。Translator が出力するコードおよび runtime system は純粋な Java プログラムであるため、高い portability が保証される。現在シングルスレッド版、マルチスレッド版、分散版の 3 種類の runtime system を実装中である。

Java Data-parallel Programming using an Extensible Java Preprocessor

Yves ROUDIER　Yuuji ICHISUGI
roudier@etl.go.jp　ichisugi@etl.go.jp
(STA Fellow)

Electrotechnical Laboratory

We describe the extensible Java preprocessor EPP and a data-parallel extension of Java implemented with EPP. EPP is a preprocessor for the Java language which can be extended by incorporating EPP plugins. Tiny Data-Parallel Java is an example of EPP plugin which consists of a translator and runtime system. High portability is guaranteed because the translated code and the runtime systems are pure Java code. We are implementing three versions of the runtime libraries, single-threaded, multi-threaded and distributed, running on any platform supporting Java.

1 Introduction

The Java language [GJG96] has recently become very popular among programmers. Java has socket and thread libraries which can be used on various platforms. In the near future, many JAVA VM(virtual machine) interpreters will support shared memory type multi-processors and will then become a better platform for parallel programming.

On the other hand, Java lacks facilities for language extension, which are adopted by other object-oriented languages. For example, C++ has a macro preprocessor, operator overloading and template facilities. Smalltalk and CLOS have closures and metaclass facilities. These features extending language constructs and operators supplement the inheritance mechanism which extends data types. Without such extension possibilities, it is difficult to make parallel libraries which can be employed easily by the end-users.

In order to supplement the defect of Java, we developed an extensible Java preprocessor kit, EPP [Ich]. EPP can be used to introduce new language features, possibly associated with new syntax.

In this paper, we will describe Tiny Data-Parallel Java, an extension that is implemented as an example application of EPP. It is based on the same language model as Data-Parallel C [HQ91]. The source code of a Tiny Data-Parallel Java application is translated to standard Java code by a translator implemented using EPP. Because the runtime system is written in standard Java, high portability is guaranteed. We are implementing three versions of these runtime libraries : single-threaded, multi-threaded and distributed which run on any platform supporting Java. Using the multi-threaded version and a multiprocessor workstation, applications can be executed in parallel if the VM interpreter supports parallel execution of thread libraries. The distributed version uses the HORB system [Hir97] to implement communications between each node. Using the distributed version, applications can be executed on heterogeneous distributed environments.

Tiny Data-Parallel Java does not have sufficient language features to support very high performance parallel programs; however, we think it shows rather convincingly the effectiveness of the combination of the high extensibility of EPP and of the high portability of Java and their interest for parallel programming. Many data-parallel extensions are currently being proposed for Java (see for example [NPA] or [LP96]), but none of these framework has been introduced by means of a general extension mechanism.

2 An extensible preprocessor kit: EPP

2.1 Existing tools

To implement new languages or extend existing ones, preprocessors or translators are often used rather than native compilers. Many language extensions and source level optimization tools for C/C++ are implemented as translators to C/C++. Recently, several extensions for the Java language have also been proposed which are implemented as preprocessors. Because of the simplicity and object-orientation of Java, it is also relatively easy to provide extension support libraries.

The merits of this style of implementation are easiness and high portability. Instruction level optimization can be delegated to the compiler of the target language.

Although there are potentially many useful language extension systems, the users have to select only one extended language for their own projects. Generally, it is impossible to merge several language extensions or eliminate harmful features from the extended system.

Systems with a compile-time meta-object protocol (MOP) such as MPC++ [Ish94], OpenC++ [Chi95] or JTRANS [KK97] have solved this problem. These systems allow the implementation of language extensions as modules that can be selected by the users. Yet, extensibility of syntax is slightly restricted in these systems.

2.2 Description of EPP

The extensible Java preprocessor kit, *EPP*, is an application framework for preprocessor type language extension systems. Basically, EPP is itself a source to source preprocessor of Java. The parser of EPP is written by recursive descent style and provides many hooks for extensions. By using these hooks, the extension programmer can introduce new features, possibly associated with new syntax without editing the source code of EPP. Because all grammar rules are defined in a modular way, it is also possible to remove some original grammar rules from standard Java.

Once the parsed program has been transformed into a tree, the preprocessor programmers can easily manipulate it from their program. The usefulness of such kind of tool has already been proven by Lisps and adapted to C++ by various systems like Sage++ [ea94], MPC++ [Ish94] or OpenC++ [Chi95].

```
#epp load "swap"
public class test {
  public static void main(String[] argv){
    int a = 1, b = 2;
    swap(int, a, b);
  }}
```

Figure 1: A program using a swap plugin.

EPP enables preprocessor programmers to write an extension as a separate module. We call the extension modules *EPP plugins*. If plugins are written in a certain manner [Ich97, IHT+96], multiple plugins can be incorporated into EPP simultaneously. The users of the preprocessor can select any plugin that fits the characteristics of their projects.

High composability of EPP plugins can be realized thanks to a description language, *Ld-2* [Ich97]. Ld-2 is an object-oriented package implemented in Common Lisp [Ste90]. The inheritance mechanism of object-oriented languages makes it easy to implement extensible applications because all methods of objects can be considered as hooks for extensions. In addition to the traditional inheritance mechanism, Ld-2 provides a novel feature called *system mixin* which supports extensible and flexible softwares.

Although the current target of EPP is only Java, the architecture of EPP is applicable to other programming languages.

2.3 An Example of EPP plugin

The users of the preprocessor can specify *EPP plugin files* at the top of each Java source code. Fig. 1 is a simple example program using an EPP plugin that defines a swap macro. The plugin file is actually an Ld-2 program file. The plugin file will be dynamically loaded by EPP before starting the preprocessing. The user can specify multiple EPP plugins at the top of the source file.

3 Tiny Data-Parallel Java

3.1 General principle

As an example of EPP plugin, we are implementing the *Tiny Data-parallel Java plugin* using the same translation technique as Data-Parallel C [HQ91]. Some methods of Java objects are treated as data-parallel methods. From the programmer's point of view, the methods are executed on a large number of virtual processors. Currently, there are many restrictions on how to write the data-parallel method, and no optimization is done. The specifications and restrictions of data-parallel methods are briefly described below.

- Data-parallel method.
  The `run` method of class `parallel` is *the data-parallel method* which is executed in parallel by virtual processors. EPP translates this method into a standard Java code which uses thread libraries and synchronization primitives. Other methods are not changed by the translator.

- Creating virtual processors.
  `parallel p = new parallel(N); p.run();` generates and starts `N` virtual processors.

- Mono- and poly-variables.
  Static variables defined in the `parallel` class are considered as *mono*-variables and instance variables are considered as *poly*-variables. Mono-variables are shared by all virtual processors and poly-variables are owned by each virtual processor. Mono- and poly-variables can be used in arbitrary expressions in the data-parallel method. No local variable can be declared within the data-parallel method.

- Remote access to poly-variables.
  Each virtual processor can access poly-variables on another virtual processor by using the expression `x@[i]`, where `i` is the ID of a virtual processor and `x` is the name of a poly-variable. The expression `x@[i]` is only permitted on the left hand side or the right hand side of assignment expressions which are at the toplevel of blocks. It is not permitted inside of any other expression (for instance, in `if(x@[i])...`)

- Reduction.
  Currently, only one reduction function, `set_sum` is provided. `set_sum(mono_var, exp)` calculates the value of `exp` on each virtual processor, and assigns the sum of these results to the mono-variable `mono_var`.

- Control structures.
  `if` and `while` statements can be used and nested arbitrarily. `for`, `switch`, `break` and `continue` statements and exception handling are not allowed in the data-parallel method. The semantics of `if` and `while` statements are the same as in Data-Parallel C; virtual processors execute each statement synchronously.

```
x = VPID();
System.out.println("VPID = " + x);
```

```
for (vpi = (0); (vpi) < (vpn); vpi++) {
  (vp_x)[vpi] = (((PE_i) * (vpn)) + (vpi));
  System.out.println(("VPID = ") + ((vp_x)[vpi]));
}
sync();
```

Figure 2: Source code and translated code of a Tiny Data-Parallel Java program.

3.2 Translator

The EPP plugin of Tiny Data-Parallel Java adds an extra pass to EPP, before the code-emitting pass. At this pass, the source code of Tiny Data-Parallel Java is expanded into plain Java code that contains `for` statements simulating virtual processors, thread or process creation codes and synchronization codes.

Fig. 2 shows some source code using the virtual processor ID (VPID) and its translation into plain Java. Virtual processors can be allocated to several Java threads, `PE_i` being the ID of each thread; `vpn` indicates the number of virtual processors allocated to threads (possibly throughout several machines). `vpi` is a loop counter used to simulate allocated virtual processors. Each poly-variable is represented as an array whose size is `vpn`. Therefore, a poly-variable `x` is translated to `(vp_x)[vpi]`. The method invocation `sync()` executes a barrier synchronization. This synchronization call is inserted in several places such as before and after remote data access or reduction function calls.

3.3 Single-threaded version

The first implementation of Tiny Data-Parallel Java is a single-threaded version. In this version, virtual processors are only emulated by `for` loops: data-parallelism is only used as a logical model. The method `sync()` does absolutely nothing.

3.4 Multi-threaded version

We designed a single-process, multi-threaded version of our system that we are integrating within EPP. This version is working on uniprocessors, but the most interesting feature of this type of scheduling is that it will enables us to fully exploit multiprocessor machines.

In this multi-threaded version, a thread is normally created for each virtual processor required. However, it is possible to make use of the previous (single-threaded) implementation inside of threads, thus keeping a reasonable number of threads.

Thread synchonization is realized at each step of the computation. Method `sync` implements a synchronization barrier stopping all threads arrived at this method before others.

The synchronization scheme of that version already adopts a distributed point of view; the only difference with a really distributed architecture is that notifications of synchronization phases are achieved by writing to shared variables instead of sending real messages. Specifically, if one thread exits a synchronization phase first and gets a long enough time slice to enter the next synchronization phase, it must not modify the value of a variable needed by other threads to exit the previous synchronization. To avoid such locking of threads, the update of synchronization shared variables has to be very carefully done, either through a three-phase scheme or by locally keeping track of the current synchronization phase of the thread and storing the state for two successive synchronizations instead of one.

The introduction of synchronization is achieved by interleaving normal instructions with `sync()` method invocations.

3.5 Distributed version

We are currently implementing a distributed version of Tiny Data-parallel Java. For that purpose, we make use of the HORB system [Hir97] for Java. Our choice of HORB is due to its better performance figures compared to Sun's RMI (Remote Method Invocation system). HORB transparently provides us all the facilities needed for data transmission. However, implementing our system on top of RMI would be very straightforward as well; it also remains possible to use a different communication strategy, for instance an interface to PVM or MPI.

Our architecture defines a *virtual processor* as the combination of a HORB server and a HORB client (see Fig. 3). Schematically, the server part performs message reception while the client part performs the real computation and can send messages to other virtual processors. These two objects run simultaneously on different threads: the client part is always running, and HORB skeletons (which are invisible to the HORB user), which also run on threads, receive

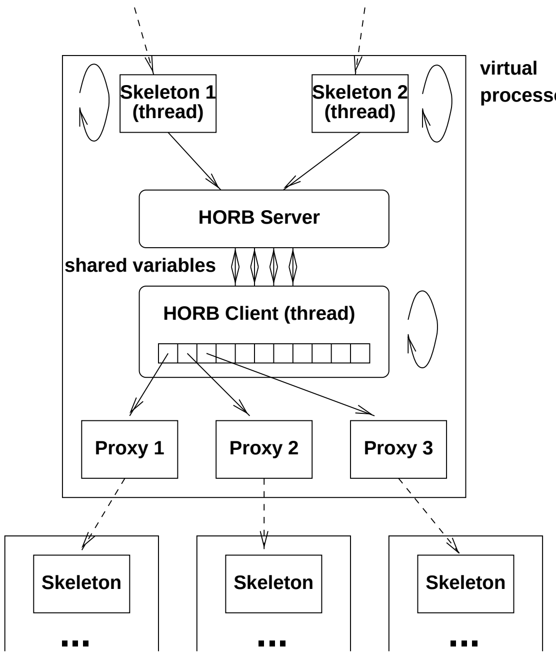

Figure 3: Distributed version

messages and activate the server part. Server and client communicate through shared variables.

As previously, it remains possible to embed threads or sequential loops inside of these more heavy-weight processes. That design would give virtual processors a much finer granularity and would suppress waits during communications or I/O operations (interleaving of activities).

Before starting the program, a list of machines needs to be provided in a configuration file and in the current state of the implementation, we are only thinking of cyclic block allocation on these machines.

3.6 Current limitations

Currently, our data-parallel system has several restrictions. First, the current implementation performs no optimization.

Writing a data-parallel method also implies following some guidelines. For example, remote variable access, i.e. access to a parallel variable on another virtual processor, is only allowed at left hand side or right hand side of assignment. In other words, remote variable access is not allowed in argument of any expressions or method invocations.

We will probably not achieve a high speedup from the very beginning, especially on really distributed memory machines, because of the very fine-grained parallelism of the model. Indeed, we will need precise benchmarks of our machines (costs of communication and scheduling, speed, etc.); only then will we be able to chose the correct trade-off between the different policies (sequential, multi-threaded, distributed) so as to efficiently map the program on the selected architecture.

However, this Java extension can also be useful for other applications. For programs that need a big memory, all one needs to do for getting more memory with this plugin is to add a new machine. It is also especially suited for programs that need to achieve a constant synchronization.

In spite of its current restrictions, this tiny data-parallel Java has many interests:

- all Java libraries including AWT can be used from the program.
- The implementation is as portable as Java. It can make use of the Java thread library, thus achieving much better speedups on multiprocessors (although, at the time, there are unfortunately only few platforms which can execute Java threads really concurrently).
- Other EPP plugins remain applicable.

This extension is also a good demonstration of the suitability of EPP to language extension, in particular in the field of parallel programming.

3.7 Example: calculating π

Fig. 4 shows a program which calculates the value of π. In that program, we launch one thousand virtual processors at the same time, each calculating a part of the surface giving us the value of π. The only synchronization between these threads is the global sum at the end of the computation.

4 Conclusion and future work

We described the extensible preprocessor kit EPP, an application framework for preprocessors, and its use. The preprocessor programmer can implement EPP plugins which are reusable extension modules and which can add new grammar rules to Java.

EPP is currently written in the Ld-2 language which provides the system mixin feature for combining plugins; Ld-2 is itself implemented in Common Lisp. We are currently implementing an EPP plugin which adds system mixin features to Java. After this work is completed, we plan to rewrite EPP directly in Java with this plugin. It is worth noting

```
#epp load "datap0"

public class calc_pi {
  public static void main(String argv[]){
    int num = 1000;  // number of virtual processors
    parallel.width = 1.0 / num;
    parallel p = new parallel(num);
    p.run();
    System.out.println(parallel.result
                         * parallel.width);
  }}

class parallel {
  static double result;
  static double width;
  double x;

  void run(){
    x = (VPID() + 0.5) * width;
    set_sum(result, 4.0 / (1.0 + x * x));
  }}
```

Figure 4: A program which calculates π.

that features supporting plugin programming such as pattern matching and Lisp backquote macro can themselves be implemented as EPP plugins. We are also studying a possible port of EPP to C++.

We also presented the programming of a simple data-parallel framework for Java with EPP. This programming framework demonstrates some of EPP capabilities at integrating new features into Java. We are still currently implementing the EPP tiny dataparallel plugin itself, but we already experimented with a hand-translated multi-thread application. We are also planning to offer different scheduling policies at the same time (for instance, multi-threading and distributed support). There are also other possible models of data-parallel programming that we will be investigating in the future.

Java threads provide sufficient basic mechanisms for multithreading, and in a portable way since any Java Virtual Machine provides this facility. However, generally speaking, the thread model is far too complex and error-prone for large-scale programming. Adding distributed features again adds more complexity to the application. We really believe that transparent, high-level frameworks are necessary for large-scale (should we say real-world ?) software development: this is especially important if a team, and not only a single developer, is involved.

We thus plan to develop other such high-level frameworks in Java for data- as well as task-parallel applications, and also for reactive programming (a more complete presentation of a reactive framework for Eiffel can be found in [CR96]; we now plan a different implementation technique in Java). We will keep on using EPP for this purpose and try to understand its possible limitations; other experiments are needed on that matter.

References

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, volume 30(10) of *ACM Sigplan Notices*, pages 285–299, Austin, Texas, October 1995. ACM Press.

[CR96] D. Caromel and Y. Roudier. Reactive Programming in Eiffel//. In J.-P. Briot, J.-M. Geib, and A. Yonezawa, editors, *Object-Based Parallel and Distributed Computation OBPDC'95*, LNCS 1107, pages 125–147. Springer-Verlag, 1996.

[ea94] F. Bodin et al. Sage++: A Class Library for Building Fortran and C++ Restructuring Tools. In *Proc. of Object-Oriented Numerics Confs., Oregon*, April 1994.

[GJG96] J. Gosling, B. Joy, and Steele. G. *The Java Language Specification.* Java Series. Sun Microsystems, 1996.

[Hir97] S. Hirano. HORB: Distributed Execution of Java Programs. In *Proceedings of WWCA'97 (International Conference on Worldwide Computing and Its Applications), Tsukuba, Japan*, March 1997. see also http://ring.etl.go.jp/openlab/horb/.

[HQ91] P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, 1991.

[Ich] Y. Ichisugi. EPP and Lods home page. http://www.etl.go.jp/etl/bunsan/~ichisugi.

[Ich97] Y. Ichisugi. Ld-2 users manual (DRAFT). Technical report, Electrotechnical Laboratory, 1997. can be obtained from [Ich]. In Japanese.

[IHT+96] Y. Ichisugi, S. Hirano, H. Tanuma, K. Suzaki, and M. Tsukamoto. Compiler Widgets — Reusable and Extensible Parts of Language System —. In *The 11th workshop of object oriented computing WOOC'96, Japan Society of Software Science and Technology*, March 1996. In Japanese.

[Ish94] Y. Ishikawa. Meta-level Architecture for Extendable C++, Draft Document. Technical Report Technical Report TR-94024, Real World Computing Partnership, 1994.

[KK97] A. Kumeta and M. Komuro. Meta-Programming Framework for Java. In *The 12th workshop of object oriented computing WOOC'96, Japan Society of Software Science and Technology*, March 1997.

[LP96] P. Launay and J.L. Pazat. Integration of control and data parallelism in an object oriented language. In *Sixth Workshop on Compilers for Parallel Computers (CPC'96) Aachen, Germany*, December 1996.

[NPA] NPAC. HPJava reference and workshops site. http://www.npac.syr.edu/projects/javaforcse/.

[Ste90] G.L. Steele. *Common Lisp the Language, 2nd edition.* Digital Press, 1990.