Automatic Cost Estimation of Data Parallel Programs

Arjan J.C. van Gemund

Dept. of Information Technology and Systems
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft, The Netherlands
email: a.j.c.vangemund@its.tudelft.nl

October 2001

Abstract

In this document we describe the implementation of a symbolic cost estimator for data parallel programs that has been developed as part of the Joses project. The cost estimator comprises (1) a Pamela code generator engine, built within the Timber compiler that compiles Spar/Java (a data parallel Java dialect) for distributed-memory machines, and (2) a Pamela compiler that translates the generated Pamela model into a symbolic cost expression. The whole proces of compiling parallel programs to symbolic performance models is in the order of seconds. Due to the symbolic simplification engine within the Pamela compiler the solution cost of the symbolic models is in the order of milliseconds. Four test programs (ADI, MATMUL, GAUSS, and PSRS) demonstrate that even with a simple machine model the prediction error is less than 10 %. This accuracy is more than enough to enable scalability analysis as well as a correct selection of the best coding or data partitioning strategy. This report superseedes the final Joses deliverable [4] by presenting and discussing experimental results that are based on a more consistent machine model, a new Pamela simplifier, and a new case study (GAUSS). These results came available after the deliverable was submitted in August 2001. The fact that these new results provide compelling evidence in favor of the Pamela approach has been the major motivation of this new "final" report.

Contents

1 Introduction **3**
1.1 A Quick Tour 4

2 Modeling Engine **11**
2.1 Implementation 11
2.2 Modeling Approach 13
2.3 Machine model interfacing 16
2.4 Modeling Algorithm 17
2.5 Modeling Demo 21

3 Pamela Compiler **26**
3.1 Introduction 26
3.2 Implementation 28
3.3 Compilation 29
3.4 Parameterization 31

4 Case Studies **34**
4.1 Gpp Machine Model 35
4.2 ADI 37
4.3 MATMUL 37
4.4 GAUSS 38
4.5 PSRS 38
4.6 Summary 42

5 Conclusion **43**

A ADI Pamela Model **47**
A.1 Partitioning along j axis 47
A.2 Partitioning along i axis 52

B GPP Machine Pamela Model **54**

C DEMO Pamela Model **57**

D MATMUL Source **64**

E GAUSS Sources **66**

F PSRS Sources 70
F.1 PSRS 70
F.2 PSRS1 77

Chapter 1

Introduction

The JOSES project (1997 - 2001) aims at extending existing CoSy compiler technology to allow the rapid development of Java compilers for embedded multiprocessor systems. A (small) part of the project is concerned with the development of a symbolic cost estimator (hereafter called Cost Estimator) for data parallel programs written in a data parallel Java dialect called Spar/Java [7]. The Spar/Java programs are compiled by the Timber compiler to the DAS distributed-memory machine [1], while producing symbolic cost models as a side result. These models, which evaluate in milliseconds, are typically used to select among various code implementation or data partitioning alternatives, as well as being used for other interactive program/machine parameter studies, such as scalability assessment.

The deliverables of the Cost Estimator project comprise the following:

- a document describing the theory behind our symbolic cost estimation approach [3]
- a document describing the design of the symbolic Cost Estimator [2]
- the Cost Estimator software itself, comprising the Modeling Engine and the PAMELA compiler.
- a document describing the validation results based on initial results of the Cost Estimator [4].

The current report superseedes the final deliverable [4] by presenting the results of a much more elaborate validation that took place after the deliverable was submitted. In this document we describe our implementation of the Cost Estimator and report on the experiments performed to validate the Cost Estimator functionality. For a thorough background of the concepts and terminology used throughout the report the reader is referred to [3, 2].

The Cost Estimator has been implemented conform the architecture described in [2]. Figure 1 shows how the Cost Estimator has been integrated into the Timber compiler. The whole Spar/Java compilation process now involves the following steps (marked by numbers in the figure):

1. parsing the Spar/Java source into Vnus IR [6]. Although all explicit owner (task parallelism) and distribution (data parallelism) pragmas have been processed resulting in statement and expression processor ownerships, the original, shared-memory program form is still preserved.
2. transforming the IR into message-passing SPMD form
3. generating C++ code which is to be compiled and linked to the parallel machine libraries for eventual execution.
4. converting the shared-memory Vnus IR into a PAMELA program model, performed by the Cost Estimator module known as the Modeling Engine.

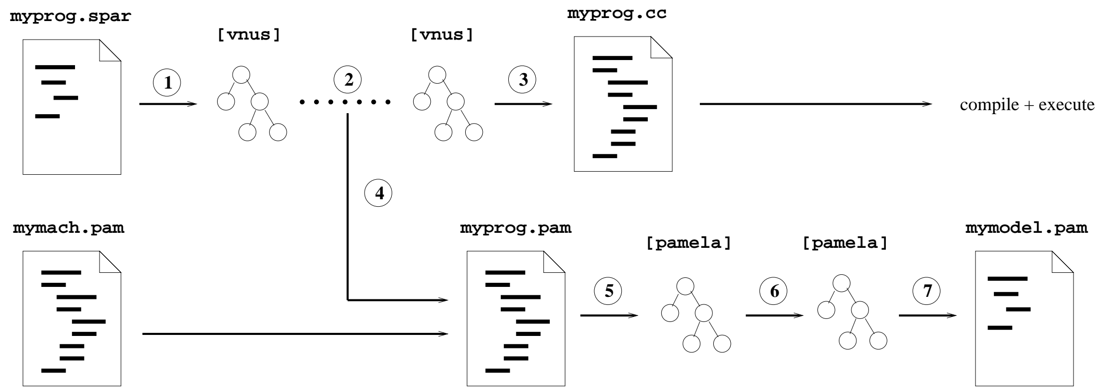

Figure 1.1: Integration of Cost Estimator.

5. parsing the PAMELA model into Vnus IR.
6. transforming the PAMELA process models into PAMELA execution time models according to the calculus explained in [3], subsequently simplifying the resulting expressions.
7. unparsing the predictions in terms of PAMELA source code, which is (a) human readable, and (b) allows interactive parameter study by recompilation.

The Cost Estimator comprises the Modeling Engine (step 4), and the PAMELA compiler (steps 5, 6, and 7).

Due to budget restrictions, only a prototype Cost Estimator has been implemented that does not support the Spar/Java language to its fullest. However, the prototype does provide a convincing proof of concept, as will be demonstrated by the validation experiments.

The report is organized as follows. The remainder of this chapter presents a brief introduction that guides the reader through the main features of the Cost Estimator. Chapter 2 describes the implementation of the Modeling Engine, while Chapter 3 describes the PAMELA compiler. Chapter 4 presents three program case studies, demonstrating what quality the prototype Cost Estimator delivers when producing execution time predictions. Chapter 5 draws a number of conclusions regarding the feasibility of annotation-driven symbolic cost estimation of parallel (embedded) programs.

1.1 A Quick Tour

In order to introduce the general features of the Cost Estimator, we will use the ADI program [3, 2] as introductory example, similar to the discussion in [2], but different in the sense that all codes are now real codes as either programmed or generated by the Modeling Engine and PAMELA compiler.

The actual code `adi.spar` that is accepted by the Timber compiler (and thus our modeling engine) is given by

```
globalpragmas <$
        ProcessorType = ((Gpp "Gpp.pam")),
        Processors = ((Gpp gpp[64])),
        Pamela = (numeric "N")
$>;
```

```
public class adi
{
        static int N = 100;
        static int P = 20;

        static double A[*,*]
        <$ on = (lambda (i j) gpp[(block j (@N / @P))]) $> = new double[N,N];

        <$ Pamela = (@N N),
           Pamela = (@P P_gpp) $>

        public static void main()
        {
                for (i :- 1 : N-1)
                        <$ independent,
                           Pamela = parallel $>
                        foreach (j :- 0 : N)
                                adi.f(i,j);
        }

        public static void f(int i, int j)
        {
                A[i,j] = (A[i-1,j] + A[i+1,j]) / 2.0;
        }
}
```

The resulting PAMELA source `adi.pam` generated by the modeling engine is listed in Appendix A.1.

Apart from the usual Spar/Java code, there are a few pragmas necessary for correct operation of the modeling engine. As explained in [2] the Spar/Java global pragma mechanism is used to link the PAMELA program model to a PAMELA machine model. Apart from minor syntax changes, the `ProcessorType` pragma is similar to [2], which causes the Modeling Engine to generate an include statement

```
#include "Gpp.pam"
```

to include the machine model `Gpp.pam` (discussed later on). This also applies to the `Processors` pragma, which generates the lines

```
numeric gpp(i) = i
numeric parameter P_gpp = 20
```

as explained in [2].

In order to allow the Spar/Java variable `N` to be used as parameter in PAMELA, the so-called PAMELA pragma `(numeric "N")` generates the parameter declaration

```
numeric parameter N
```

which will later be bound to the Spar/Java equivalent (described later on). The specific syntax of the Spar/Java PAMELA pragmas is due to the current syntax rules implemented in the Timber frontend. The ADI code in `class adi` is based on the problem size parameter `N` and number of processors parameter `P`, which are used in both the declaration and partitioning of `A` and the loop bounds.

The PAMELA model of the Spar/Java procedure `main` is directly named after the Vnus identifier, i.e., `adi_main_0`. At this point, the PAMELA pragmas `(@N N)` and `(@P P_gpp)` are processed first, which generate the PAMELA definitions

```
numeric adi_N_0 = N
numeric adi_P_0 = P_gpp
```

as can be seen at the bottom of the `adi.pam` model. This enables the internal Vnus identifiers `adi_N_0` and `adi_P_0` (corresponding to Spar/Java variables `N` and `P`, respectively) to be controlled in terms of the PAMELA parameters `N` (declared earlier) and `P_gpp`, respectively. Thus the PAMELA model is controlled by the more convenient, and standard parameters `N` and `P_gpp` (who are independent of the Spar/Java program that is being modeled).

While some of the generated code in `adi.pam` will be explained later on, at this point it is illustrative to note that both Spar/Java procedures `main` and `f` are compiled to PAMELA processes `adi_main_0` and `adi_f_0` respectively. Apart from some additional control flow generated by the Vnus frontend, the original control structure is fully preserved, the i loop being compiled to a `seq` loop, the j loop being compiled to a `par` loop as a result of the PAMELA `parallel` pragma (motivated in Section 2.2).

In general, the generated code conforms to the specifications described in [2], with a few modifications which will be described in detail later on. In particular, the extra first parameter `p` that is added next to the original function parameters serves to pass the owner processor index of the procedure call. This is particularly useful to model the effect of task parallel Spar/Java procedure owner annotations, an example of which will be discussed in Chapter 4.

At present, a small, but reasonably sufficient number of computation operators and communication operators are supported, including local moves (`Gpp_move_op`), computation (e.g., `Gpp_minus_op`, `Gpp_plus_op`), and the global communication operators `Gpp_Gpp_move_op` and `Gpp_Gpp_bcast_op` (both are not present in this particular case, due to the choice of j partitioning). All these operators are, of course, defined in the machine model `Gpp.pam` (discussed later on).

The PAMELA model `adi.pam` is then compiled by the PAMELA compiler. From the resulting PAMELA model we only show the model of the `main` function which constitutes the symbolic cost estimate of the entire ADI program. In order to illustrate the derivation process we first show the internal result of the transformation rules, described in the PAMELA symbolic analysis calculus, *before* automatic simplification (multi-line expressions folded for readability).

```
numeric parameter t_c
numeric parameter t_l
numeric parameter t_g
numeric parameter P_gpp = 20
numeric parameter N
...
< many models for program functions such as adi_main_0() and adi_f_0() >
...
numeric T_main =
        ((0 + (t_c / 1)) +
        sum (i_0 = 1, ((N - 1) - 1)) {
                (0 + max(max (j_0 = 0, (N - 1)) {
                        (0 + (((((((0 + (t_c / 1)) + 0) +
                                          (t_c / 1)) + 0) +
                                          (t_c / 1)) +
                                          (t_c / 1)) + 0) + 0))
                },max(sum (j_0 = 0, (N - 1)) {
```

```
                    (0 + ((((((((0 +
                          ((t_c / 1) *
                           unitvec(((j_0 div (N div P_gpp)) + 1)))) + 0) +
                          ((t_c / 1) *
                           unitvec(((j_0 div (N div P_gpp)) + 1)))) + 0) +
                          ((t_c / 1) *
                           unitvec(((j_0 div (N div P_gpp)) + 1)))) +
                          ((t_c / 1) *
                           unitvec(((j_0 div (N div P_gpp)) + 1))))
                          + 0) + 0))
                })))
        })
numeric phi_main =
        ((0 + (t_c / 1)) +
        sum (i_0 = 1, ((N - 1) - 1)) {
                (0 + max (j_0 = 0, (N - 1)) {
                        (0 + ((((((((0 + (t_c / 1)) + 0) +
                                        (t_c / 1)) + 0) +
                                        (t_c / 1)) + (t_c / 1)) + 0) + 0))
                })
        })
numeric delta_main =
        ((0 + ((t_c / 1) * [ 0, 1 ])) +
        sum (i_0 = 1, ((N - 1) - 1)) {
                (0 + sum (j_0 = 0, (N - 1)) {
                        (0 + ((((((((0 +
                              ((t_c / 1) *
                               unitvec(((j_0 div (N div P_gpp)) + 1)))) + 0) +
                              ((t_c / 1) *
                               unitvec(((j_0 div (N div P_gpp)) + 1)))) + 0) +
                              ((t_c / 1) *
                               unitvec(((j_0 div (N div P_gpp)) + 1)))) +
                              ((t_c / 1) *
                               unitvec(((j_0 div (N div P_gpp)) + 1))))
                              + 0) + 0))
                })
        })
numeric omega_main =
        max(((0 + ((t_c / 1) * [ 0, 1 ])) +
        sum (i_0 = 1, ((N - 1) - 1)) {
                (0 + sum (j_0 = 0, (N - 1)) {
                        (0 + ((((((((0 +
                              ((t_c / 1) *
                               unitvec(((j_0 div (N div P_gpp)) + 1)))) + 0) +
                              ((t_c / 1) *
                               unitvec(((j_0 div (N div P_gpp)) + 1)))) + 0) +
                              ((t_c / 1) *
                               unitvec(((j_0 div (N div P_gpp)) + 1)))) +
                              ((t_c / 1) *
```

```
                              unitvec(((j_0 div (N div P_gpp)) + 1))))
                             + 0) + 0))
                 })
         }))
```

Because of the PAMELA substitution mechanism, all models of the functions called by the `main` function (including `adi_main_0` and `adi_f_0`) are already included in the `main` model. Note that also a very large number of other, intermediate symbols have disappeared due to the same substitution mechanism, the only symbols remaining being the ones declared as `parameter`. The parameters `N` and `P_gpp` have already been defined at program source level. The `t_c` parameter is defined in the machine model `Gpp.pam` and represents the `Gpp` processor's time delay of some single scalar computations such as addition or subtraction (recall, that the machine models we use presently are kept extremely simple).

Note, that each expression in the above model can be significantly reduced to a very simple expression. The final result of the PAMELA compiler *after* the automatic simplification is (currently) given by

```
    numeric parameter t_c
    numeric parameter t_l
    numeric parameter t_g
    numeric parameter P_gpp = 20
    numeric parameter N
    numeric T_main =
            (t_c + (((2 * -(1)) + N) * max((2 * (2 * t_c)),(2 * (2 * (t_c *
            max (v =(max(0,((-(1) + N)-(2 * (N div P_gpp)))) div (N div P_gpp)),
                    ((-(1) + N) div (N div P_gpp))) {
                            (min(((1 + v) * (N div P_gpp)),N) -
                             max((v * (N div P_gpp)),0))
            }))))))
    numeric phi_main =
            (t_c + (((2 * -(1)) + N) * (2 * (2 * t_c))))
    numeric delta_main =
            ((t_c * [ 0, 1 ]) + (((2 * -(1)) + N) * (2 * (2 * (t_c *
            sum (v = (0 div (N div P_gpp)), ((-(1) + N) div (N div P_gpp))) {
                ((min(((1 + v) * (N div P_gpp)),N) -
                  max((v * (N div P_gpp)),0)) * unitvec((v + 1)))
            })))))
    numeric omega_main =
            max(((t_c * [ 0, 1 ]) + (((2 * -(1)) + N) * (2 * (2 * (t_c *
            sum (v = (0 div (N div P_gpp)), ((-(1) + N) div (N div P_gpp))) {
                ((min(((1 + v) * (N div P_gpp)),N) -
  max((v * (N div P_gpp)),0)) * unitvec((v + 1)))
            }))))))
```

This result still reflects the prototype stage of the current simplify engine. However, the `T_main` expression, which is the focus of our current simplification effort, has already a time complexity of $O(1)$ (upper bound of the `max` loop is $O(1)$), and evaluates in 330 μs (see Section 4.6).

Although the above model is essentially the end product of the Cost Estimator, the PAMELA compiler can additionally be used to *evaluate* the model for a specific parameter setting (also read Chapter 3 on the properties of PAMELA models and the PAMELA compiler `eval` engine). For instance, modifying the above definitions of the variables `P_gpp`, `N`, and `t_c` according to, e.g.,

```
numeric P_gpp = 20
numeric N = 100
numeric t_c = 1
numeric t_l = 0
numeric t_g = 0
```

will cause the PAMELA compiler to further reduce ("evaluate") the above expressions for `main`. Indeed, if we recompile the above modified cost estimate, the PAMELA compiler will generate the following output:

```
numeric t_c =
        1
numeric t_l =
        0
numeric t_g =
        1
numeric P_gpp =
        20
numeric N =
        100
numeric T_main =
        1961
numeric phi_main =
        393
numeric delta_main =
        [ 0, 1961, 1960, 1960, 1960, 1960, 1960, 1960, 1960, 1960, 1960,
             1960, 1960, 1960, 1960, 1960, 1960, 1960, 1960, 1960, 1960 ]
numeric omega_main =
        1961
```

The `main` models are now reduced to numbers, being our execution time prediction of the ADI program for a 100×100 matrix running on 20 "Gpp"-type processors. Note that the execution time T^l of ADI is predicted to be 1,961. Since we have chosen `t_c = 1` and `t_l = 0` this execution time represents the equivalent to 1,961 scalar computation operations (39,2001 operations in total, corresponding to about 4 operations per matrix element, as can be seen from `delta_main`). The absence of global communication can be observed from the first entry in `delta_main` as in the machine model used in the above experiment all global communication has been mapped to one single resource with index 0 (i.e., `fcfs(0,1)`). The 20 processor resources $i = 1, \ldots 20$ are mapped to resource indices `fcfs(i,1)`. The almost perfect load balance can be also observed from the first through 20th entry in `delta_main`.

In order to demonstrate a typical use of the Cost Estimator in predicting the effect of an alternative partitioning we examine the effect of an i axis partitioning as described in [2]. The i axis partitioning involves changing the `block j ...` into `block i ...` resulting in the following partitioning pragma

```
static double A[*,*]
<$ on = (lambda (i j) gpp[(block i (@N / @P))]) $> = new double[N,N];
```

The resulting PAMELA model is shown in Appendix A.2. Essentially all code is identical to the j-axis model, except the model for `adi_f_0` (i.e., the actual computation on `A`). First, the communication operation is modeled that moves row $i-1$ to $i+1$, such that all required data resides on the processor that owns `A[i+1,*]`. Next, the computation `(A[i-1,j] + A[i+1,j])/2.0` is performed on the same

processor, the result being moved to the processor that owns A[i,j][1]. Using the same parameter settings for N, P_gpp, and t_c the PAMELA model generated by the Cost Estimator is compiled by the PAMELA compiler to following result:

```
numeric P_gpp =
        20
numeric N =
        100
numeric t_c =
        1
numeric t_l =
        1
numeric t_g =
        1
numeric T_main =
        55001
numeric phi_main =
        646
numeric delta_main =
        [ 5700, 2001, 3000, 3000, 3000, 3000, 3000, 3000, 3000, 3000, 3000,
          3000, 3000, 3000, 3000, 3000, 3000, 3000, 3000, 3000, 2800 ]
numeric omega_main =
        5700
```

In the model, two additional communication machine parameters t_g and t_l are defined to a numeric value. The symbol t_g denotes the delay per globally transferred scalar (processor-processor), while t_l denotes the local (processor-memory) transfer delay. In order to illustrate the additional global communication, the model used for Gpp_Gpp_move is temporarily altered from the model used throughout in our validation experiments. This "demonstration" model is given by

```
process Gpp_Gpp_move(p,q) =
        if (p == q)
                use(Gpp(p),t_l)
        else
                use(network,t_g)
```

In this way, all global moves map to the network resource, which allows the number of moves to be observed from the delta vector.

Clearly, the Cost Estimator succeeds in predicting the absence of speedup, as T_adi_main_0 is now roughly equal to the *sum* of the individual processor workloads, rather than to the *maximum* of the workloads as in the previous j partitioning (a few operations overlap, which is the reason why the sum is not perfect). Also note the considerable amount of inter-processor communication (5,700 scalar moves in total).

The diagnostic use of the omega, and phi values with respect to T has already been explained in [3], the delta vector providing insight in resource load distribution.

[1]This particular communication/computation scheme is due to the current optimization heuristics implemented in the Timber compiler.

Chapter 2

Modeling Engine

In this chapter we describe the Modeling Engine that traverses the Vnus intermediate representation of the Spar/Java source, while generating a corresponding PAMELA model.

After presenting an overview of current Modeling Engine functionality, we briefly revisit the "program level" modeling approach we adopt in the Cost Estimator with regard to modeling program parallelism, and describe the consequences with respect to the Modeling Engine procedures. Next, we present the machine modeling interface that is currently supported by the Modeling Engine. Finally, we describe the Modeling Engine algorithms, demonstrating their operation through a simple example Spar/Java program, called `demo.spar`.

2.1 Implementation

The Modeling Engine has been implemented conform the architecture described in [2]. Due to budget restrictions, a prototype Modeling Engine has been implemented that does not support the Spar/Java language in all its (sequential) details. Yet the most essential features are covered, indeed allowing the Modeling Engine to provide a convincing proof of concept. As mentioned earlier, the most important theme of the Cost Estimator is to predict the effects of parallelization and code/data mappings, rather than to accurately predict sequential workload. Consequently, resources have been spent accordingly. The Modeling Engine is implemented in C and measures approximately 4,000 lines of source code, presently organized in terms of one large source file (`vnus2pam.c`).

Before describing the operation of the Modeling Engine in detail (see Section 2.4) we present an account of Modeling Engine functionality according to the structure followed in [2].

- Modeling level:
  According to [2]. More details will be given in Section 2.2.

- Basic Blocks:
  The modeling approach basically conforms to the principles described in [2]. Only a number of most important basic operations have been implemented, according to the list of 20 computation and communication operator models presented in the machine model described in Section 2.3. As the programming paradigm presented by the IR to the Modeling Engine must still be of the shared-variable type (in contrast to message-passing) the IR has only undergone a limited number of compiler engine transformations. As at the point at which the Modeling Engine operates, the IR is still expressed in terms of scalar operations (i.e., no message aggregation), all models are scalar. Presently, no distinction has been made between floating point and integer operations. No transcedental function models are supported either. As no heterogeneous processor declarations

are currently supported by the compiler frontend, the global move models are always according to the `X_X_move_op` style, rather than the `X_Y_move_op` style.

- Flow Control:
  The following flow control constructs are modeled, i.e., `while`, `do .. while`, `if`, `if .. else`, `for`, and `foreach`. Apart from the workload involved in computing index bounds, additional SPMD (loop) overhead as generated by the backend is not taken into account since this is has not yet been generated at the level at which the Modeling Engine operates. Currently, only *one* cardinality per list is recognized, having stride one (both of which are sufficient for our experiments). Goto-like constructs such as `break`, `return`, `continue`, and `switch` are currently not supported, while the occurrence of a Vnus `goto` construct will generate a fatal error. With respect to parallelism few modifications relative to [2] have been made, which are discussed in Section 2.2.

- Performance Annotations:
  The following pragma constructs are recognized: lists, `+`, `-`, `*`, `mod`, `/`, `[]`, lambda abstraction, numbers, booleans, and strings. As the current compiler frontend only supports one processor array (homogeneous architectures), the Modeling Engine also only supports one single processor type and name throughout the modeling process. All pragmas mentioned in [2] have been implemented, with the exception of the size, value, and frequency pragma. As (sequential) data allocation and initialization is not yet modeled there has not yet been the need for size annotation. In our experiments there has not been a need for value and frequency annotation either. Instead, a new `parallel` pragma has been added as mentioned earlier. In contrast to [2], probabilistic truth models such as `Bern` are not supported since the PAMELA compiler evaluation engine does not yet support random functions. The `log` function, however, is supported since this function has been used in complexity expressions with the `cost` annotation.

- Miscellaneous:
  Ownerships are restricted to only one index. Block and cyclic index functions are supported. Index expressions are fully supported with the exception of index arrays. This implies that index mappings involving indirection arrays which turn up in ownership computations will generate a fatal error.

- Vnus IR constructs:
  Although the above discussion presents a good impression of the current functionality of the Modeling Engine in terms of the discussion in [2], the following list provides a detailed overview of what is not fully covered in terms of the Vnus Tm constructs [5] `declaration`, `statement` and `expression`:

  - `declaration`:
    - supported:
      `DeclFunction`, `DeclProcedure`, `DeclExternalFunction` (`default_proc` equation is generated), `DeclExternalProcedure` (`default_proc` equation is generated).
    - not supported (currently not needed):
      `DeclGlobalVariable`, `DeclLocalVariable`, `DeclFornalVariable`, `DeclCardinalityVariable`, `DeclExternalVariable`, `DeclRecord`.
  - `statement`:
    - supported:
      `SmtAssign`, `SmtAssignOp`, `SmtProcedureCall`, `SmtExpression`, `SmtWhile`, `SmtDoWhile`,

SmtFor (with limitations mentioned earlier), SmtIf, SmtBlock, SmtForeach (with limitations mentioned earlier), SmtPrint (only a print() model is generated), SmtPrintln (only a println() model is generated).

    * not supported (typically modeled by delay(0)):
      SmtSwitch, SmtReturn, SmtValueReturn (expression is modeled), SmtGoto, SmtThrow, SmtRethrow, SmtCatch (statement block is modeled), SmtDelete, SmtGarbageCollect.

  - expression:
    * supported (non-zero workload):
      ExprUnop (all operators), ExprBinop (all operators), ExprDeref (currently not compiler-specific), ExprCast (currently not type-specific), ExprIsBoundViolated (generates cmp_op), ExprIsUpperBoundViolated (generates cmp_op), ExprIsLowerBoundViolated (generates cmp_op), ExprCheckedIndex (generates cmp_op), ExprUpperCheckedIndex (generates cmp_op), ExprLowerCheckedIndex (generates cmp_op).
    * supported (zero workload except when subexpressions involved):
      ExprByte, ExprShort, ExprInt, ExprLong, ExprFloat, ExprDouble, ExprChar, ExprBoolean, ExprString, ExprNull, ExprName, ExprReduction, ExprSelection, ExprFlatSelection, ExprFunctionCall, ExprComplex, ExprIf, ExprWhere, ExprNotNullAssert, ExprWrapper.
    * not supported: (except for subexpressions involved)
      ExprField, ExprFieldNumber, ExprShape, ExprRecord, ExprAddress, ExprNewArray, ExprNew, ExprFilledNew, ExprNulledNew, ExprNewRecord, ExprSizeof, ExprGetBuf, ExprGetSize, ExprGetLength, ExprIsRaised, ExprArray.

Although the list of not or partially supported constructs seems large, experiments (Chapter 4) demonstrate that the current functionality already provides a convincing proof of the feasibility of automatic, symbolic cost estimation that requires little interaction on the users part, while providing good prediction accuracy, even with extremely simple machine models.

2.2 Modeling Approach

As mentioned in [3, 2] our approach to modeling parallelism differs from the SPMD paradigm that applies to the code that is actually generated by the compiler. Consider the following Spar/Java code:

```
for (i :- 0:N)
        A[i] = A[i] * alpha;
```

The SPMD code generated by the compiler would be modeled by (optimizations not taken into account for simplicity)

```
par (p = 0, P_gpp-1) {
        seq (i = 0, N-1) {
                if (owner(A[i]) == p)
                        Gpp_mult_op(p)
                ;
                Gpp_move_op(p)
        }
}
```

From the apparent ease with which the Pamela model is expressed, it would seem that a modeling approach, close to the actual SPMD scheme would be quite appropriate. However, consider the following Spar/Java code:

```
for (i :- 0:N)
        A[i] = A[i-1] * alpha;
```

The SPMD code generated by the compiler now contains a message-passing synchronization scheme that cannot be modeled in terms of an above Pamela model, much in the same way as in the case of the ADI example, discussed in [2]. Thus, the modeling approach taken my the Cost Estimator is not to model the actual SPMD parallelism, as in

```
par (p = 0, P_gpp-1) {
        seq (i = 0, N-1) {
                if (owner(A[i-1]) != p)
                        Gpp_Gpp_move_op(p,owner(A[i-1]))
                ;
                if (owner(A[i]) == p)
                        Gpp_mult_op(p)
                ;
                Gpp_move_op(p)
        }
}
```

which would totally disregard the sequentialization across all processors, but to adopt a more program level approach, according to

```
seq (i = 0, N-1) {
        if (owner(A[i-1]) != owner(A[i]))
                Gpp_Gpp_move_op(owner(A[i]),owner(A[i-1]))
        ;
        Gpp_mult_op(owner(A[i]))
        ;
        Gpp_move_op(owner(A[i]))
}
```

Thus, the inherent sequentialism in the program is preserved, yielding a correct prediction, in contrast to the SPMD model.

As mentioned in Chapter 1, following this approach, the Modeling Engine traverses the IR after the alignment engine and part of the SPMD engine have processed all data distribution and ownership pragmas, yielding the appropriate processor ownership fields on all the original code statements and expressions. Thus, the code being traversed is still according to the shared-variable programming model, before the remaining passes of the SPMD engine which transform the IR into a message-passing (SPMD) programming model.

While in our program level modeling approach information on sequentialism is preserved, in a naive implementation of such an approach the first program code would now yield:

```
seq (i = 0, N-1) {
        Gpp_mult_op(owner(A[i]))
}
```

which is clearly at odds with the actual parallelism of the SPMD code generated by the compiler, which effectively should have been modeled by

```
par (i = 0, N-1) {
        Gpp_mult_op(owner(A[i]))
}
```

since the compiler has determined that the loop could be parallelized. Yet, because of the fundamental problems when modeling the SPMD scheme, we have no choice but to take this program level approach. Consequently, we must still determine whether a sequential loop will be parallelized by the compiler. Clearly the loop of the first program code would execute in parallel, while the loop of the second code would not.

The decision whether the compiler parallelizes a sequential source code loop is taken by the SPMD engine, and depends on the absence of loop-carried dependencies, of which, unfortunately, the analysis has not yet been performed at the time the code is processed by our engine. Using the IR after such analysis has been performed is impossible in the present compiler implementation as the IR has already been transformed into SPMD message-passing code, a programming model which is not amenable to our symbolic cost estimation method as mentioned earlier [3, 2].

In order to influence the decision by the Modeling Engine to transform a sequential loop into either a `seq` loop or `par` loop, a PAMELA `parallel` pragma has been introduced. When present, a loop will be modeled by a `par` loop. (Note, that this pragma differs from the "independent" pragma, which does not always cause a loop to be parallelized.) Although such pragma should be one of the first candidates to be rendered obsolete in a better engineered compiler + Modeling Engine, in this way, we can at least avoid having to model (and therefore reimplement) the particular parallelization heuristics of the SPMD engine.

The absence of an explicit `par (p = 0, P_..) { }` SPMD fork in the models generated by the Modeling Engine, essentially implies a program level modeling approach where the original program code is actually regarded as a *sequential* program with a number of possibly (task of data) parallel loops, rather than an SPMD program. For instance, the following code

```
x = 3 * 4;
for (i :- 0:N)
        A[i] = x;
```

is modeled by

```
Gpp_mult_op(0)
;
Gpp_move_op(0)
;
par (i :- 0:N)
        Gpp_move_op(owner(A[i]))
```

where processor 0 is (by arbitrary convention) chosen as the resource on which the sequential thread of control resides. This approach can be observed in terms of index 0 being passed as owner index parameter in the model of the `main` Vnus procedure in the Modeling Engine algorithm shown in the next section.

While in reality the first statement would be executed on all processors we avoid modeling this in terms of

```
par (p :- 0:P_gpp) {
        Gpp_mult_op(p)
        ;
        Gpp_move_op(p)
```

```
}
;
par (i :- 0:N)
        Gpp_move_op(owner(A[i]))
```

because (1) the implicit barrier synchronization of the par is not present in reality, and (2) because the total execution time of the model does not depend on whether all processors are involved in the statement or only processor 0 (i.e., the above two models have equal execution time).

Although seemingly motivated by practical considerations, also this modeling approach actually reflects a pure interpretation of a shared-variable, data parallel program as if it were a parallel algorithm written down by the programmer, rather than its implementation by the compiler, through either an SPMD implementation or some other implementation.

2.3 Machine model interfacing

In the current Cost Estimator version only the commonly used Vnus computation operators are supported, along with two external Vnus print functions. As mentioned earlier, all models are scalar. Thus, vectorization capabilities of computation or communication resources are currently not modeled. Including the standard resource definitions, and the three communication operations, this implies that the machine model must support the following instructions:

```
% resource definitions:

resource parameter fcfs(i,m)
resource <proc type>(i) = fcfs(<f(i)>,1)

% computation operators:

process <proc_type>_negate_op(p) = <model using <proc_type>(p)>
process <proc_type>_not_op(p) = <model using <proc_type>(p)>
process <proc_type>_uplus_op(p) = <model using <proc_type>(p)>
process <proc_type>_divide_op(p) = <model using <proc_type>(p)>
process <proc_type>_equal_op(p) = <model using <proc_type>(p)>
process <proc_type>_greater_op(p) = <model using <proc_type>(p)>
process <proc_type>_greaterequal_op(p) = <model using <proc_type>(p)>
process <proc_type>_less_op(p) = <model using <proc_type>(p)>
process <proc_type>_lessequal_op(p) = <model using <proc_type>(p)>
process <proc_type>_minus_op(p) = <model using <proc_type>(p)>
process <proc_type>_mod_op(p) = <model using <proc_type>(p)>
process <proc_type>_notequal_op(p) = <model using <proc_type>(p)>
process <proc_type>_or_op(p) = <model using <proc_type>(p)>
process <proc_type>_plus_op(p) = <model using <proc_type>(p)>
process <proc_type>_times_op(p) = <model using <proc_type>(p)>

process <proc_type>_deref_op(p) = <model using <proc_type>(p)>
process <proc_type>_cast_op(p) = <model using <proc_type>(p)>

% communication operators:
```

```
process <proc_type>_move_op(p) = <model using <proc_type>(p)>
process <proc_type>_<proc_type>_move_op(p,q) =
        <model using <proc_type>(p) and <proc_type>(q)>
process <proc_type>_<proc_type>_bcast_op(P,q) =
        <model using <proc_type>(q), q = 0..P-1>

% miscelaneous instructions:

process print(p) = delay(0)
process println(p) = delay(0)
```

Appendix B shows the simple example machine model that has been used in our experiments (see Chapter 4).

2.4 Modeling Algorithm

The main algorithms on which the Modeling Engine is based are implemented in terms of the procedures (listed in top-down order) `model_program`, `handle_pragmas`, `model_declaration`, `model_statement`, and `model_expression`. Each line is preceded by a number for reference purposes. The first procedure that is called is `model_program`, which essentially processes the global pragmas, all procedure and function declarations, as well as the `main` procedure.

```
model_program:

01      handle_pragmas(global pragmas)
02      add all Vnus procedure/function identifiers to known_list
03      for all Vnus declarations
04              model_declaration(Vnus declaration)
05      model_statement(Vnus main block, 0)
06      list Pamela symbol table definitions
```

The `handle_pragmas` procedure below interprets those pragmas that are relevant for the Modeling Engine.

```
handle_pragmas(pragmas):

10      if ProcessorType pragma
11              store type in proc_type
12              generate include directive
13      if Processors pragma
14              store name in proc_name
15              generate P_name parameter definition in symbol table
16      if Pamela pragma
17              if definition pragma
18                      add lhs to known_list
19                      generate Pamela definition in symbol table
20              if parallel pragma
21                      store information for later use in
22                      model_statement()
23              if cost pragma
```

```
24                          store rhs information for later use in
25                          model_statement()
26                  if
27                     lower pragma
28                     upper pragma
29                     cond pragma
30                       generate Pamela definition in symbol table,
31                          store rhs information for later use in
32                          model_statement()
33                  if role pragma
34                          determine condition value as function of role
35                          treat as Pamela cond pragma with above value
```

Most pragmas have already been introduced in [2]. The purpose of the `parallel` pragma has been explained in the previous section. A new pragma is the `role` pragma, which is motivated by the following. Many branches are programmed by the user. If the branch condition is not deterministic a Pamela condition (`cond`) pragma is needed to provide a meaningful expression (otherwise a default is substituted). These pragmas are included in the source code. However, in the course of source code translation extra branches are created as part of, e.g., initialization, garbage collection, null pointer checks, etc., which clearly cannot be annotated by the user. In a number of such cases a role pragma is added by the compiler to assist in determining the truth (probability) of such a branch condition. In such a case, the pragma is simply treated as if being a user-supplied Pamela condition pragma. At this moment the `static-init` role is recognized and interpreted according to a false branch condition (initialization of each static variable occurs only once in its lifetime. The test whether initialization is required is a dynamically changing condition, which can only be estimated in a static context).

The `model_declaration` procedure processes all Vnus declarations, of which the functions and procedures are most important.

```
model_declaration(d):

40          handle_pragmas(declaration pragmas)

41          if d = function or procedure
42                  generate Pamela process definition
43                  add parameters to known_list
44                  forall statement in statement block
45                          model_statement(statement, p)

46          if d = external function or procedure
47                  generate Pamela process definition
48                  generate default body
```

Most of the work is concentrated in the procedures `model_statement`, and `model_expression`. The `model_statement` algorithm generates all control flow and communication operations such as `Gpp_move_op`.

```
model_statement(s, owner_index):

50          handle_pragmas(statement pragmas)
51          if owner(s) != owner_index
```

```
52              owner_index = owner(s)

53      if cost pragma
54              generate use(proc_name(owner_index),cost)
                return

55      if s = assignment
56              model_expression(rhs, owner_index)
57              if owner(rhs) = owner_index
58                      generate local_move_op(owner_index)
                else
59                      if owner_index != replicated
60                             generate global_move_op(owner_index,owner(rhs))
                        else
61                             generate global bcast_op(owner(rhs))

62      if s = procedure call
63              if procedure identifier in known_list
64                      generate verbatim Pamela call
                else
65                      generate default procedure call
66              generate first proc arg equal to owner_index
67              for all args in procedure call
68                      model_expression(arg, owner_index)
69                      if arg expression in terms of known_list symbols
70                              generate verbatim arg expression
                        else
71                              generate default arg

72      if s = foreach
73              add index identifier to known_list
74              model_expression(lb,owner_index)
75              model_expression(ub,owner_index)
76              if parallel pragma
77                      generate par
                else
78                      generate seq
79              if lower pragma
80                      generate new lb conforming to lower bound pragma
                else
81                      if lb expression in terms of known_list symbols
82                              generate verbatim lb expression
                        else
83                              generate default lb
84              if upper pragma
85                      generate new lb conforming to upper bound pragma
                else
86                      if ub expression in terms of known_list symbols
87                              generate verbatim ub expression
```

```
                    else
88                          generate default ub
89              model_statement(foreach statement body,owner_index)

90        if s = if or s = ifelse
91              model_expression(condition,owner_index)
92              if cond pragma
93                    generate new cond conforming to lower bound pragma
                else
94                    if cond expression in terms of known_list symbols
95                          generate verbatim cond expression
                      else
96                          generate default cond
97              model_statement(if statement body,owner_index)
98              if s = ifelse
99                     model_statement(else statement body,owner_index)
          ...
```

The `model_expression` algorithm generates the computation operations such as `Gpp_times_op`.

```
model_expression(e, owner_index):

100       if owner(e) != owner_index
101             owner_index = owner(e)

102       if e = name
103             return whether variable in known_list

104       if e = byte, ..., boolean, double
105             return true

106       if e = char, string
107             return false

108       if e = function call
109             if function identifier in known_list
110                   generate verbatim Pamela call
                else
111                   generate default function call
112             for all args in function call
113                   model_expression(arg,owner_index)
114                   if arg expression in terms of known_list symbols
115                         generate verbatim arg expression
                      else
116                         generate default arg
          ...

117       if e = binary times operator
118             model_expression(left,owner_index)
119             model_expression(right,owner_index)
```

```
120                 generate proc_name_times_op(owner_index)
```

2.5 Modeling Demo

We will illustrate the operation of the above algorithms using the following Spar/Java example:

```
globalpragmas <$
        ProcessorType = ((Dsp "Dsp.pam")),
        Processors = ((Dsp cpu[20]))
$>;

public class demo
{
        static int N = 100;

        static double A[*]
        <$ on = (lambda (p) cpu[(block p 5)]) $> = new double[N];

        <$ Pamela = (@N 100) $>

        public static void main()
        {
                <$ independent,
                   Pamela = parallel $>
                foreach (i :- 0:N)
                        A[i] = A[i] * 2.0;

                <$ on = cpu[(local 0)] $>
                demo.f(1,2);
        }

        public static void f(int n, int m)
        {
                double  x;

                x = N / 2;
                <$ Pamela = (lower (@N / 20)) $>
                for (i :- g(x) : N-1) {
                        <$ Pamela = (cond 0) $>
                        if (x == 2)
                                x = 1.0 * 2.0;
                        if (i > n)
                                x = x / 2;
                }

                <$ Pamela = (cost (@N / @m)) $>
                while (x < N) {
                        x = x + m;
                }
```

```
        }
        public static int g(double x)
        {
                return (int) (x / 10);
        }
}
```

The corresponding output is given in Appendix C (only relevant portions are listed).

We now describe the operation of the algorithms while referring to the line number of the algorithm operation involved.

First of all, the global pragmas are processed (01). The `ProcessorType` pragma (10) generates the `#include Dsp.pam` statement (12) and assigns the internal string `proc_type` to `Dsp` (11). This type will be prefixed to all computation and communication operations. The `Processors` pragma (13) generates the number of `cpu` processors parameter `P_cpu` (15) which is defined to the value 20, the `Dsp` processor index array `cpu(i)`, and assigns the internal string `proc_name` to `cpu` (14). This processor index mapping is used to refer to the processor indices in all computation and communication operations [2], as can be seen in all generated operation models such as `Dsp_move_op`, `Dsp_times_op`, etc.

All three functions `demo_main_0`, `demo_f_0`, and `demo_g_0` in the class definition `demo` are modeled by `model_declaration` (03, 04, 40). Before all Vnus declarations are actually processed, all procedure/function indentifiers are inserted in an internal list called `known_list` (02). The purpose of this list is to distinguish symbols that will be defined in the PAMELA domain, from those symbols that will not have a PAMELA equivalent, such as data-dependent variables. Typically, data-dependent symbols such as data variables or control variables, also referred to as *stochastic* symbols, will not appear in the PAMELA models, while symbols such as static procedure names, induction variables, static parameters, also referred to as *deterministic* symbols, and expressions in terms of them, will appear in the PAMELA models *verbatim*. Whenever, the Modeling Engine needs to make a choice whether the symbol (or expression) needs to be copied into the PAMELA model it looks up this list. Clearly, all Vnus procedures/functions need to appear in the PAMELA model (with bodies being PAMELA models of the original procedure/function bodies), which requires all symbols to be inserted in the `known_list` (thus implying that these symbols are *known* to PAMELA). The way this important mechanism works will become clear during the example.

The data distribution pragma for `A` is ignored by the Modeling Engine. However, the effect has been such that by the time the Modeling Engine traverses the Vnus IR, all appropriate statements (bodies) and expressions will have owners (processor index expressions). This immediately determines the processor index that is passed as an argument to all operation models such as `Dsp_move_op`, `Dsp_times_op`, etc.

The `Pamela = (@N 100)` pragma is processed as part of processing `demo_main_0` (04, 40, 17), and is an example of what we call a (PAMELA) definition pragma. The intention is to make it known within the PAMELA model that `@N` (i.e., `demo_N_0` in Vnus terms) has the value 100. Rather than just using an `@N = 100` pragma syntax, we must adopt a list-type style as required by current Timber compiler frontend pragma syntax rules. The result of the pragma is that a PAMELA equation is generated (06) since the lhs (`demo_N_0`) and rhs (`100`) are entered in an internal symbol table (19), which is listed at the end of the modeling process (06) in terms of PAMELA equations (in this case: `numeric demo_N_0 = 100`). Subsequently, the symbol `demo_N_0` is added to `known_list` (18) so that all future references to `N` in the Spar/Java code may be copied into the PAMELA model verbatim as the symbol value is known (e.g., the upper bound `N-1` in the `i` loop in `demo_f_0` can now be simply copied to the upper bound in the corresponding `seq` loop in the PAMELA model).

Before modeling the procedure body, the procedure declaration is processed, which involves generating the corresponding PAMELA process definition `process demo_main_0` (42), the list of formal parameters verbatim, including an extra first parameter `p` that is used to pass the processor owner within the PAMELA model to each of the body submodels (explained later on). As all formal argument symbols are always known within PAMELA, their symbols are added to `known_list` (43).

The body of `demo_main_0` is modeled by a series of calls to `model_statement` (44, 45). Note, that external function/procedure declarations have no body, which causes the Modeling Engine to generate `default_procedure` or `default_function` stubs (48). The `parallel` pragma causes the Modeling Engine to take this into account in the next following statement it will process (21, 22). Consequently, the `foreach` loop statement is modeled (72) in terms of a `par` loop (77). In every loop statement, the induction variable is passed on to the PAMELA domain verbatim, and is consequently added to `known_list` (73). Next, the lower and upper bound computation work load is modeled in terms of PAMELA models (74, 75). As in both cases the expressions are merely symbols (102, 104) no operation models are generated. In this loop, the generation for the control symbols is straightforward. Since there are no lower and upper bound PAMELA pragmas, and since both lower and upper bound expression only involves variables known to PAMELA (lower bound: 104, 81, 82, upper bound: 102, 86, 87), the expressions are simply passed on to the PAMELA domain (with the proper name translation, of course).

The assignment to `A` involves a lhs and a rhs expression. Being a functional (process-algebraic) language, PAMELA has no concept of state. This implies that lhs variables are not modeled (known) in PAMELA. Consequently, the assignment (55) is only modeled in terms of the rhs expression workload (56, 117, 118, 102, 119, 104, 120, dereference ignored), and the subsequent move to the lhs location (58). Note, that for the expression and move models the owner is equal to the statement owner, which is been set to `cpu[block(i_0,20)]` by the previous compiler engines (in PAMELA terms: `cpu(i_0 div 5)`). This owner overrides the default owner `p` (the first parameter of the PAMELA procedure model) which causes (51, 52) to take effect on both move and expression modeling within this statement. Thus the workload is spread across different processors which causes PAMELA to eventually predict speedup for this statement. Note that the move is modeled as local (58) since the owners of lhs and rhs are equal.

The next `on` pragma causes the previous compiler engines to add a new owner to the next statement, which is the procedure call `demo.f`. This procedure call is modeled verbatim (61, 62, 63) since the corresponding PAMELA model `demo_f_0` is in `known_list` (02) as the process model will be generated at some point (03, 04). In this case, also the actual parameter expressions are passed on to the PAMELA call verbatim (69, 70), since all variables involved are known to PAMELA (102, 104). Note, that the actual expressions are modeled (68) prior to generating the PAMELA process call, which yields the `times_op` model (117). As the function is to be executed on `cpu[(local) 0]`, the first argument of `demo_f_0` is set to 0, thus overriding the default setting `p` (51, 52). As in the assignment of `A[i]`, this is an example where a statement is owned by a different processor than the processor that owns the procedure/function currently being modeled (`p`). The former example relates to data parallelism, while the latter relates to task parallelism. In Chapter 4 we will see more examples of the Cost Estimator dealing with both types of parallelism.

While modeling the next procedure `demo_f_0` the formal parameters `n_0` and `m_0` are added to `known_list` for future reference in the course of modeling the procedure body. The first assignment on `x` is modeled in much the same way as the previous assignment. The next loop statement includes a lower bound pragma since the lower bound expression involves `g(x)`, which cannot be passed on to (i.e., computed by) PAMELA. While the symbol `demo_g_0` is known to PAMELA, function calls are not *computed* in PAMELA, only *modeled*. Processing of the loop proceeds exactly as if the loop would have been a `foreach` loop and proceeds as follows. After adding the index `i_1` to `known_list` (73), the lower bound expression workload is modeled (74), which involves modeling the call to `demo_g_0` with argument `x` (108). Since there exists a PAMELA model for the evaluation of the `demo_g_0` workload,

a corresponding call is generated (109, 110). In contrast to the call itself, the actual parameter expression (x) cannot be computed in PAMELA (114) and a default argument expression (arg_0003) is substituted (116). Note, that this procedure is in contrast to the procedure followed for the previous demo_f_0 call, where both arguments were passed verbatim (70). The lack of a valid value for arg_003 (equal to the default value 0), however, is of no consequence, since the argument plays no role in the PAMELA model of demo_g_0 itself. Next, the upper bound expression workload is modeled (75), which yields a Dsp_minus_op model, similar to the generation of the Dsp_times_op model (120). Then, the PAMELA control loop is generated. Due to the lower bound pragma (79), a default lower bound control variable lb_0002 is used for the lower bound. This symbol has been generated prior, during processing of the lower bound pragma where a PAMELA definition numeric lb_0002 = (demo_N_0 div 5) was added to the symbol table (30). The use of the intermediary variable lb_0002 instead of immediately substituting the expression demo_N_0 div 5 is to ease the identification of user influences on the PAMELA model generated by the Modeling Engine. The upper bound control expression is generated in the same way as in the previous loop.

As the next branch statement includes a condition expression that is not known to PAMELA, a condition pragma is included, that generates the definition numeric cd_0004 = 0 in the symbol table (30), subsequently used in the same way as the previous lower bound variable. The condition expression workload itself is modeled (91) by the Dsp_cast_op and Dsp_equal_op models. As mentioned earlier, some of the branches have been added by the compiler frontend instead of the user. An appropriate example is the if (demo_needstatic_init_demo0_0) branch. Clearly, the user cannot supply pragmas for compiler-generated branch conditions. In order to provide some form of information a role pragma is added to these branches, which is handled much in the same way as a condition pragma (35). For instance, the demo_needstatic_init_demo0_0 call in demo_man_0 is annotated by the pragma static-init. This pragma is interpreted by the Modeling Engine in terms of a 0 truth probability of the associated condition (34), which causes the Modeling Engine to generate the condition variable cd_0000 (93) according to the definition numeric cd_0000 = 0 (35).

The next branch condition i > n merely involves variables known to PAMELA which causes the condition to be passed on verbatim (91, 117, 102, 102).

The next statement is a while loop, which cannot be modeled by the Cost Estimatorwithout a bound pragma [3, 2]. For demonstration purposes, however, we choose to feature the use of the cost pragma as described in [2]. The rhs expression of the pragma (24, 25) is used as workload argument of a special Dsp_cost model (53, 54) that takes into account the translation in terms of the DSP bandwidth [2] (for an example see Gpp_cost in Appendix B).

The modeling of the procedure demo_g_0 does not involve anything different to what has already been explained earlier.

While most of the above is caused by the three calls to model_declaration (03, 04), the modeling process proceeds with modeling the main procedure (05), followed by generating the PAMELA symbol table definitions along with some statistics, including the contents of known_list, for diagnostic purposes, as shown in Appendix C.

If we feed the output of the Modeling Engine to the PAMELA compiler the following model for main results:

```
numeric T_main =
        ((t_c * 5) + (50 + (285 * t_c)))
numeric phi_main =
        (t_c + (50 + (285 * t_c)))
numeric delta_main =
        ((t_c * [ 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
                  5, 5, 5, 5, 5, 5, 5, 5 ]) +
```

```
          (((3 * (t_c * [ 0, 1 ])) + ((2 * (t_c * [ 0, 94 ])) +
            (94 * (t_c * [ 0, 1 ])))) + [ 0, 50 ]))
numeric omega_main =
        max(((t_c * [ 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
                          5, 5, 5, 5, 5, 5, 5, 5, 5 ]) +
          (((3 * (t_c * [ 0, 1 ])) + ((2 * (t_c * [ 0, 94 ])) +
            (94 * (t_c * [ 0, 1 ])))) + [ 0, 50 ])))
```

This result is based on a `Dsp.pam` machine model that is similar to the `Gpp.pam` model listed in Appendix B. This implies that local communication cost is 0 and all arithmetic computations have computation cost `t_c`. When removing the `parameter` modifier in the definition of `t_c` (the only symbol left in the model) such that `numeric t_c = 1`, the result for `main` becomes (after recompiling):

```
numeric T_main =
        340
numeric phi_main =
        336
numeric delta_main =
        [ 0, 340, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
numeric omega_main =
        340
```

This result of the Cost Estimator is validated as follows. The predicted execution time accounts for 5 computations for each of the 20 processors in account of the `A[i]` assignment, and an additional 335 computations on behalf of `demo.f(1,2)` executing on `cpu[0]` (index 1 on the `fcfs` resource vector). This number correctly breaks down into 1 division (`x = N / 2`), 1 division (`demo_g_0` call for lower bound `i` loop), 1 subtraction (`N - 1` for upper bound), 94 iterations (lower bound = 5, upper bound is 98) comprising 1 equality test (`x == 2`), 1 greater or equal test (`i > n`), and 1 division (`x = x / 2`), and finally the cost model of the `while` loop which equals 50 (i.e., $1+1+1+94(1+1+1)+50 = 335$).

Chapter 3

Pamela Compiler

In this chapter we briefly describe the Pamela compiler that is used within the cost estimation process. As mentioned in [3, 2] we have built an entirely new (prototype) compiler that has the required symbolic transformation capability, rather than using (and extending / totally revising) the existing compiler which is more geared towards generating simulation code. As a result, the new compiler has a better architecture and has the flexibility to allow extensions that will be required in the future. In this chapter we will focus on the symbolic evaluation and transformation capabilities of the compiler, rather than on the many low level implementation details.

3.1 Introduction

A Pamela model is a list of equations of the form

```
type [parameter] identifier[paramlist] [= expression]
```

which defines an identifier, possibly with argument parameters, of type `numeric`, `process`, or `resource`, by an algebraic expression. The identifier can be annotated as parameter using the optional `parameter` modifier, which is explained later on. In this case, the expression can be omitted, effectively yielding an identifier declaration, rather than an equation. The following list of equations constitutes a valid (but rather meaningless) Pamela model.

```
numeric a = 1.2 * max(b,c(10))                     % binary max
numeric b(n) = sum (i=1,c(n+2)) (i/n)              % function def
numeric parameter c                                % no eq needed
process f(i,j) = delay(i) ; par (i = 1,j) use(r,i) % last i is local
process g = f(max(x = 1,c) (c*c+1),100)            % expr passing
resource x(i) = y(i*10,20)                         % same for resources
resource parameter y                               % same here
```

For each identifier used in the expressions there must be a matching definition / parameter declaration.

For some specific numeric operators the following definitions apply (a, b scalar, v vector, $f(i)$ some expression containing i):

`unitvec(i)`	$= \underline{e}^i$ (e.g., $\underline{e}^2 = (0,0,1)$)
`max(a,b)`	$= \max(a,b)$
`max(v)`	$= \max\limits_i v_i$
`max(i = a, b) f(i)`	$= \max\limits_{i=a}^{b} f(i)$

```
sum(i = a, b) f(i)
```

$$= \sum_{i=a}^{b} f(i)$$

The PAMELA compiler accepts a Pamela model source and translates it to a time domain performance model. For each Pamela process four performance models are generated, i.e., T^l, φ, δ, and ω, all of which are defined in [3]. Thus for some `process` identifier `L` the compiler will generate the identifiers `T_L`, `phi_L`, `delta_L`, and `omega_L`.

Both PAMELA domain and time domain models are expressed in terms of the same language since PAMELA processes already include many time domain expressions. For instance, consider the following Pamela model that models two parallel tasks running on `cpu1` and `cpu2`, respectively:

```
resource parameter fcfs              % always declare fcfs parameter
resource cpu1 = fcfs(0,1)            % map to fcfs resource index 0
resource cpu2 = fcfs(1,1)            % map to fcfs resource index 1
numeric time_expr1 = 1+2             % a time domain expression
numeric time_expr2 = 3+4             % a time domain expression
process x = use(cpu1,time_expr1)     % a process expression
process y = use(cpu2,time_expr2)     % a process expression
process L = x || y                   % a process expression
```

The model already incorporates time domain expressions, either as explicit numeric equations, or within process equations. The corresponding time domain expressions for the four performance models mentioned earlier are given by the following Pamela source as generated by the compiler:

```
numeric time_expr1 =
        3
numeric time_expr2 =
        7
numeric T_x =
        3
numeric phi_x =
        3
numeric delta_x =
        [ 3 ]
numeric omega_x =
        3
numeric T_y =
        7
numeric phi_y =
        7
numeric delta_y =
        [ 0, 7 ]
numeric omega_y =
        7
numeric T_L =
        7
numeric phi_L =
        7
numeric delta_L =
        [ 3, 7 ]
```

```
numeric omega_L =
        7
```

Thus the PAMELA compiler is in fact a source-to-source compiler, transforming all `process` equations into corresponding `numeric` equations (while removing `resource` equations as they are no longer relevant).

The correctness of the above result is easily checked by realizing that `x` and `y` actually proceed in parallel as they are mapped to *different* processor resources (`cpu1` has a different index in the resource array `fcfs` than `cpu2`). Thus `T_L` is simply the maximum of `T_x` (3) and `T_y` (7). Note that `delta_L` illustrates the load imbalance within the resource array `fcfs`.

Note, that some optimizations have been applied in the compilation process. First of all, the numeric additions (1+2 and 3+4) have been performed at compile-time. Second, the max and $\sum$ operations, generated during the transformation (see [3]) have also been computed at compile-time since all arguments in the source were numeric. Hence, the resulting performance models already have numeric form and do not require further evaluation.

Although a user may only be interested in the overall performance model of `L`, the compiler generates performance models for each process equation encountered, as lower level processes may also be the subject of specific performance feedback (e.g., a Timber compiler engine or user may be interested in the performance of some lower level function, rather than only the `main` function). Again, note that the `x` and `y` performance models do not explicitly show up within the `L` performance models as they have already been substituted.

3.2 Implementation

The internal intermediate representation of the compiler is implemented using Tm [5]. The compiler is typically used in terms of the following pipeline

```
pampp | pamparse | pameval | pam2time | pamsimple | pameval | pamprint
```

Apart from the first pipe which carries Pamela source code, all intermediate pipes carry the Tm representation. The compiler is implemented using the C programming language in terms of the following modules (comprising more than 10,000 lines of source code):

- preprocessor `pampp`.
  Supports file inclusion (`include` statement) and removes comments (`%` starts comment until end of line).

- parser `pamparse`.
  Parses Pamela sources, generating Tm representation. The parser is implemented using `flex` and `bison`.

- evaluator `pameval`.
  Implements expression evaluation, which involves substituting expressions where possible as well as numerically evaluating expressions where possible. The evaluator is used at least at two stages of the performance modeling process (see above script): (1) before transformation to time, in order to perform all necessary resource expression substitutions, and (2) to numerically optimize the expressions as symbolic optimization typically introduces further possibilities for reduction.

- analyzer `pam2time`.
  Transforms process and resource equations to numeric equations according to the calculus described in [3].

- simplifyer `pamsimple`.
  Symbolically simplifies PAMELA expressions, using a range of reduction rules such as

```
0+e -> e
e/1 -> e
0*e -> 0
1*e -> e
e+e -> 2*e
(n*e)+e -> (n+1)*e
e1*e3+e2*e3 -> (e1+e2)*e3
max(e,e) -> e
max(e) -> e
max([n]) -> n
max(n*e) -> n*max(e)
max(sum(i=a,b) (unitvec(i))) -> 1
max(i=a,b) c -> c
sum (i) (c*e) -> c * sum(i) e
sum (i=a,b) c -> max(0,b-a+1)*c
sum (i=a,b) if (i > c) e -> sum (i=max(a,c+1),b) e
sum (i=a,b) f(i div c) -> sum (v=a div b, b div c) g(v)
```

  where `n` is a number, `e` is an expression, and `c` is a constant expression independent of `i`.

- unparser `pamprint`.
  Prints PAMELA equations from Tm representation back into pamela source grammar. This is the principle readable form how the results are fed back to the user.

- C generator `pam2c`.
  Generates C source code that computes Pamela numeric equations. In this way, the performance models can be linked into the Timber compiler framework as ready-to-call C functions. This (prototype) module is not used in the JOSES project.

3.3 Compilation

The above example model is compiled as follows. After the first `pameval` pass, the following model results (the `.tm` output shown has been unparsed by `pamprint`):

```
resource parameter fcfs
resource cpu1 =
        fcfs(0,1)
resource cpu2 =
        fcfs(1,1)
numeric time_expr1 =
        3
numeric time_expr2 =
        7
process x =
        use(fcfs(0,1),3)
process y =
```

```
        use(fcfs(1,1),7)
process L =
        {use(fcfs(0,1),3) || use(fcfs(1,1),7)}
```

The time expressions have been evaluated involving the optimizations $1 + 2 = 3$ and $3 + 4 = 7$. In processes `x` and `y` the `cpu` variables have been substituted by their `fcfs` definitions, respectively. The `parameter` modifier within the `fcfs` equation prevents the compiler from substituting the `fcfs` occurrences. Finally, in process `L` the symbols `x` and `y` have been substituted by their evaluated righthand sides. Thus `pameval` performs symbolic evaluation, substituting all symbols that are not declared parameter, and numerically optimizing all non-symbolic expressions.

At this point the process equations are in a suitable form to be transformed into corresponding time equations. After the first `pameval` pass, the following model results (again, the `.tm` output being unparsed by `pamprint`):

```
numeric time_expr1 =
        3
numeric time_expr2 =
        7
numeric T_x =
        (3 / 1)
numeric phi_x =
        (3 / 1)
numeric delta_x =
        ((3 / 1) * unitvec(0))
numeric omega_x =
        max(delta_x)
numeric T_y =
        (7 / 1)
numeric phi_y =
        (7 / 1)
numeric delta_y =
        ((7 / 1) * unitvec(1))
numeric omega_y =
        max(delta_y)
numeric T_L =
        max(max((3 / 1),(7 / 1)),
            max((((3 / 1) * unitvec(0)) + ((7 / 1) * unitvec(1)))))
numeric phi_L =
        max((3 / 1),(7 / 1))
numeric delta_L =
        (((3 / 1) * unitvec(0)) + ((7 / 1) * unitvec(1)))
numeric omega_L =
        max(delta_L)
```

The resource expressions are deleted as they are no longer relevant since all process equations have been replaced by new numeric equations. These new equations are directly conforming to the rewrite rules presented in [3], where `unitvec` is the Pamela language implementation of the $\underline{e}^m$ unit vector notation as used in [3]. Note that the `max` operator is overloaded at language level (used as unary, binary, as well as n-ary max).

In order to further reduce the above time domain equations as much as possible, a simplification pass is performed, yielding

```
numeric time_expr1 =
        3
numeric time_expr2 =
        7
numeric T_x =
        3
numeric phi_x =
        3
numeric delta_x =
        (3 * unitvec(0))
numeric omega_x =
        max(delta_x)
numeric T_y =
        7
numeric phi_y =
        7
numeric delta_y =
        (7 * unitvec(1))
numeric omega_y =
        max(delta_y)
numeric T_L =
        max(7,max(((3 * unitvec(0)) + (7 * unitvec(1)))))
numeric phi_L =
        7
numeric delta_L =
        ((3 * unitvec(0)) + (7 * unitvec(1)))
numeric omega_L =
        max(delta_L)
```

Finally, a second `pameval` pass is performed, in order to numerically evaluate numeric (sub)expressions, resulting in the eventual compiler output shown earlier. Note, that due to all leaf nodes being numeric, all `max` and `unitvec` calls have been resolved.

3.4 Parameterization

An important feature is the `parameter` modifier, that allows the chain of symbol substitution to be broken at some chosen symbol (the parameter). As an example we choose the Machine Repair Model (MRM) described in [3]. Consider the following Pamela code

```
resource parameter fcfs
resource s = fcfs(0,1)

numeric P = 10
numeric N = 10

process main =
        par (p = 1, P)
```

```
                seq (i = 1, N) {
                        delay(tl) ;
                        use(s,ts)
                }

numeric tl = 10
numeric ts = 10
```

Compilation (according to the entire pipeline shown earlier) yields the following time domain model:

```
numeric P =
        10
numeric N =
        10
numeric T_main =
        1000
numeric phi_main =
        200
numeric delta_main =
        [ 1000 ]
numeric omega_main =
        1000
numeric tl =
        10
numeric ts =
        10
```

Clearly the results for `main` are numeric, i.e., all symbols `P`, `N`, `tl`, `ts` have been numerically substituted. In order to obtain a symbolic performance model, this substitution must be disabled. For example, if one is interested in the effect of all above parameters, the `parameter` modifier should be inserted at each variable definition. Compilation (including simplification) yields

```
numeric parameter P
numeric parameter N
numeric T_main =
        max((N * (tl + ts)),(P * (N * ts)))
numeric phi_main =
        (N * (tl + ts))
numeric delta_main =
        (P * (N * (ts * [ 1 ])))
numeric omega_main =
        (P * (N * ts))
numeric parameter tl
numeric parameter ts
```

Note that this automatically generated model is practically equal to the hand-computed result mentioned in [3]. The above result can now be used for further parameter studies. For instance, if one is interested in the effect of `P` when `N = tl = ts = 10`, three of the modifiers need to be removed as in

```
numeric parameter P
numeric N = 10
```

```
numeric T_main =
        max((N * (tl + ts)),(P * (N * ts)))
numeric phi_main =
        (N * (tl + ts))
numeric delta_main =
        (P * (N * (ts * [ 1 ])))
numeric omega_main =
        (P * (N * ts))
numeric tl = 10
numeric ts = 10
```

Compilation now yields

```
numeric parameter P
numeric N =
        10
numeric T_main =
        max(200,(P * 100))
numeric phi_main =
        200
numeric delta_main =
        (P * [ 100 ])
numeric omega_main =
        (P * 100)
numeric tl =
        10
numeric ts =
        10
```

which reveils the effect of `P`. This model can now be evaluated for individual values of `P`. For instance, for `P = 10` the above model compiles to the numeric result mentioned at the start (where `T_main = 1000`). Of course, the parameter study for `P` could have been performed right from the start. Compiling the following input source

```
resource parameter fcfs
resource s = fcfs(0,1)

numeric parameter P
numeric N = 10

process main =
        par (p = 1, P)
                seq (i = 1, N) {
                        delay(tl) ;
                        use(s,ts)
                }

numeric tl = 10
numeric ts = 10
```

would immediately yield the above result.

Chapter 4

Case Studies

In this chapter we demonstrate the operation of the Cost Estimator for four real-world programs. Apart from inspecting the PAMELA models as generated by the Modeling Engine, we numerically compare the actually measured execution time against the results of compiling the PAMELA output by the PAMELA compiler.

In this study into the use of the Cost Estimator we are less interested in the absolute quantitative accuracy of the generated cost models, as this would involve a detailed modeling and calibration effort of the machine that is used for the actual performance measurements. Since this would be a lengthy study in its own right, possibly requiring more man months effort than the entire Cost Estimator project itself, we rather focus on

- how well a PAMELA model is automatically generated by the Modeling Engine from a Spar/Java program, i.e., how much user effort is still required.
- how well the PAMELA model is capable to qualitatively predict the effects of changing data partitionings, and how well application scalability is predicted.
- how well the PAMELA model can be reduced to a model that evaluates extremely fast, compared to, e.g., simulation models.

The first application is the ADI program which has already been mentioned in the introduction of this report. In Chapter 1 we have already shown that the Cost Estimator-generated model adequately predicts the speedup for the j-partitioned version, while predicting absence of speedup for the i-partitioned version. In this section we numerically compare the predictions with actual execution tine measurements.

The second application is a matrix multiplication program, called MATMUL. We show that the Cost Estimator is capable of quite accurately predicting the execution time as function of problem size N and number of processors P. Furthermore, we show that the PAMELA model is amenable to mechanical model reduction, which speeds up model evaluation time by orders of magnitude, demonstrating the huge cost/performance benefit of symbolic cost estimation over (performance) simulation.

The third application is a Gaussian elimination code, called GAUSS. The case study illustrates the use of the PAMELA model in predicting the difference between cyclic and block partitioning, and also demonstrates the importance of modeling cache effects.

While the first three applications are regular and therefore require negligible performance annotations, the third application is the PSRS parallel sorting program discussed as case study in [3]. This application demonstrates the ease with which highly irregular, data dependent control flow is handled by the Cost Estimator. Moreover, we show that also in this case the cost models have high accuracy, clearly distinguishing between an original program version with inferior data partitioning scheme (PSRS) and an improved version (PSRS1).

This chapter is organized as follows. In the next section, we describe the target machine used in our experiments and describe the associated PAMELA machine model that is included by the Cost Estimator. In the following sections we present our results for ADI, MATMUL, and PSRS, respectively.

4.1 Gpp Machine Model

The measurements have been performed on a 64 node partition of the Dutch Distributed ASCI Supercomputer (DAS [1]). Each node is powered by a Pentium Pro board, while the interconnection network is based on Myrinet hardware.

The predictions of the Cost Estimator are based on an extremely simple machine model due to time limitations (developing an accurate machine model is a project on its own). The model, called `Gpp.pam`, is shown in Appendix B. The machine model is divided in three sections.

The first section declares the resources used. Each processor, called `Gpp(p)` is modeled by a simple (but adequate) FCFS resource, by convention denoted `fcfs`. This implies that each process (thread) running on a processor will be scheduled on a non-preemptive First Come First Served basis. For our contention modeling scheme this assumption is fully adequate. The `network` resource is used to account for potential network contention, and maps to to `fcfs(0)`, while the processor resources are consecutively mapped starting from `fcfs(1)`. The use of the resources is discussed in the sequel.

The second section defines the computation operator models. All computation operators are simply mapped to the *same* computation model that simply charges `t_c` time units to the processor resource that performs the operation. Due to the use of comments the section is self-explanatory and closely follows the discussion in [2]. Note, that only "real" scalar computations such as addition, subtraction, multiplication, division have been assigned `t_c` time delay, while other operations such as dereferencing and casting are currently mapped to zero delays.

The third section defines the communication models. As mentioned earlier, local (processor-memory) communication is effectively ignored. Global communication is determined by only one parameter `t_g` that defines the scalar communication delay. Note, that this implies that no latency/bandwidth model is used. Although such a model would be the minimum requirement for adequately modeling communication delay[1], unfortunately the Cost Estimator has no knowledge of any communication vectorization schemes performed at a lower compilation level. Thus, the (usually advantageous) effects of message aggregation on communication delay cannot be taken into account in the cost estimation process. For the applications considered, however, this flaw is dealt with by using appropriate `t_g` values as discussed later on. The global move model includes a test if both peers are indeed different. This test is required as during Cost Estimator model generation time, it is not always known whether the processor index expressions will end up being non-equal or not. Typically, however, the condition will evaluate false. In that case, the move currently maps to a simple use of the sending processor resource. In many cases data is owned by one processor but by all (so-called replicated data), which causes data to be broadcasted to all processors. In such cases the Cost Estimator generates a specific `bcast` instruction, rather than global move operations, in order to provide the flexibility to tailor the machine model to the specific (parallel) broadcast properties of the target machine (or operating/run-time systems). In the `Gpp` machine model a broadcast is modeled by a sequential loop of individual global transfers to all available processors. Unlike the demonstration-version used for ADI in Section 1.1, the communication models do not account for network contention. The fact that communication mainly induces processor workload, rather than network workload has been shown by measurements (discussed in the PSRS case study in Section 4.5), and is due to the software overhead of the current compiler run-time system used.

[1] This abstraction indeed causes a noticible difference in the t_g values that are required to accurately model MATMUL and PSRS.

application	τ_c	τ_l	influential op	τ_g	influential op
ADI	0.007	0.11	stride-1 ld/st	0.16	stride-1 vector bcast
MATMUL	0.007	0.20	stride-N ld	0.16	stride-1 vector bcast
GAUSS	0.007	$0.11 \min(S, 4)$	stride-S st	0.30	stride-N vector bcas
PSRS	0.007	0.11	stride-1 ld/st	6.0	stride-1 scalar bcast

Table 4.1: Machine parameters (μs).

In the following we describe how the machine parameters t_c, t_l, and t_g are determined. The *local communication* parameter t_l is determined by a simple (micro)benchmark according to

```
for (k = 1:K)
    for (i = 1:N)
        for (j = 1:S:N)
            statement
```

where `S` denotes the stride, and `statement` is given by `V[i] = A[i,j]` to measured matrix loads, and `A[i,j] = V[i]` to measure matrix stores. The benchmark is run with $N = 1024$ for strides $S = 1, \ldots, N$ to determine the influence of memory hierarchy. Although simple, the benchmark is representative for all applications considered where matrices are accessed that exceed the cache capacity. As the cache line size equals 4 double precision words, the fraction of cache misses per access is proportional to S for $S = 1, \ldots, 4$. For greater values of S each access generates a cache miss. Measurements show that t_l can be approximated well according to $t_l = 0.11 \min(S, 4)$ μs for matrix stores, while for matrix loads t_l ranges between 0.11 and 0.20 μs for $S = 1, \ldots, N$.

The *local computation* parameter t_c is determined by the same benchmark, where `statement` is extended with various simple arithmetic expressions (additions, subtractions, multiplication). Taking into account the multiplication and addition due to the two-dimensional matrix indexing, the average workload per arithmetic operation is given by $t_c = 0.007$ μs. Note, that for the large matrix operations we consider, the workload per statement is largely dominated by memory hierarchy effects.

The *global communication* parameter is determined using the same benchmark where `A` and `V` are mapped onto different processors. The parameter t_g as measured for point-to-point vector transfers equals $t_g = t_l + 0.10 \mu s$. This implies a communication bandwith of 10 Mword/s, consistent with the fact that the compiler maps to a more efficient Panda communication layer compared to MPI. Furthermore, t_g has also been measured for scalar and vector broadcast operations, which are generated by the compiler for various applications. Unlike point-2-point transfers, broadcasts are only measured for the `V[i] = A[i,j]` statement, where `V` is replicated on all processors. As mentioned earlier, the broadcast costs are approximately linear in P (small efficiency gain for increasing P due to network fanout effects). For vector broadcasts t_g ranges from $t_g = 0.16$ μs for $S = 1$ to $t_g = 0.30$ μs for $S = N$. For scalar broadcasts $t_g = 6.0$ μs, independent of stride.

Table 4.1 summarizes the parameter values used in the machine model. The choice of τ_l and τ_g is determined by the most influential matrix access operation.

In the following sections all measurements ('m' in the plots) and predictions ('p' in the plots) are given terms of absolute execution time instead of speedup to fully demonstrate the accuracy of the prediction technique. The sizes of the four test codes range from 40 (ADI) to 900 (PSRS) lines of Spar/Java code. The sizes of the generated PAMELA models range from 3,000 (ADI) to 5,000 (PSRS) lines of PAMELA code (including the models for run-time functions, and debugging comments). The sizes of the compiled symbolic performance models for the `main` function range from 15 lines (ADI) to 20 lines (GAUSS).

4.2 ADI

In this section we compare the Cost Estimator results with real measurements of the ADI program. The program is similar to the one given in the introduction (Section 1.1). The PAMELA model produced by the Cost Estimator is similar to the model shown in Section 1.1. The prediction for a $1,024 \times 1,024$ matrix, shown in Figure 4.1 clearly distinguishes between the block partitioning on the j-axis (vertical) and the i-axis (horizontal). The prediction error of the vertical version for large P is caused by the fact that the PAMELA model generated by the compiler does not account for the loop overhead caused by the SPMD level processor ownership tests. The maximum prediction error is therefore 77 % but should be attributed to the current PAMELA generator, rather than the PAMELA method. The average prediction error is 15 %.

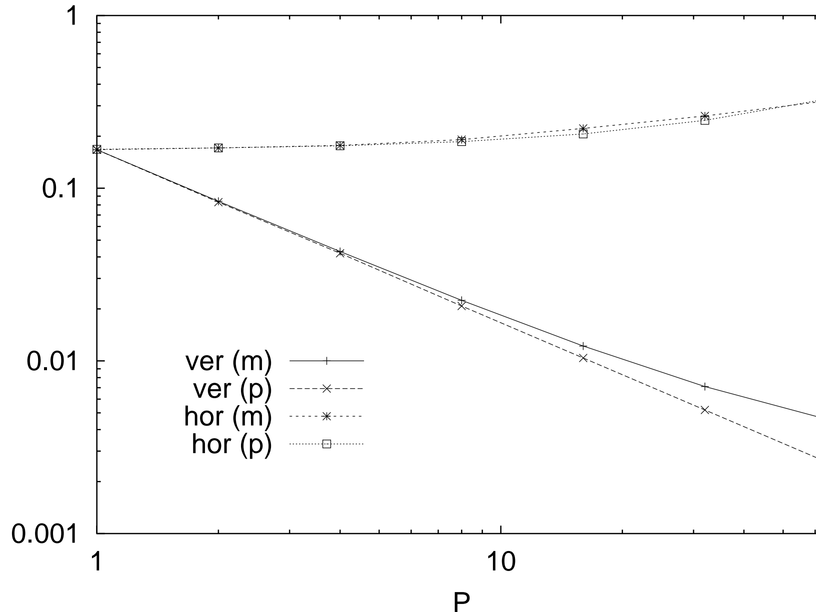

Figure 4.1: ADI (vertical and horizontal data mapping)

4.3 MATMUL

In this section we compare the Cost Estimator results with real measurements for the matrix multiplication program. The program multiplies $N \times N$ matrices A and B the result being stored in C. A is partitioned on the i axis while B and C are partitioned on the j axis. In order to minimize communication, each row of A is broadcast (using the replicated temporary vector `TMP`) to the owners of the columns of B and C that are involved in the subsequent inner product computation. The program code is given in Appendix D. The prediction and measurement results for N = 256, 512, and 1,024 are shown in Figure 4.2. The prediction error is 5 % on average with a maximum of 7 %. Even with the extremely simple machine model as given in Appendix B these results are quite satisfactory.

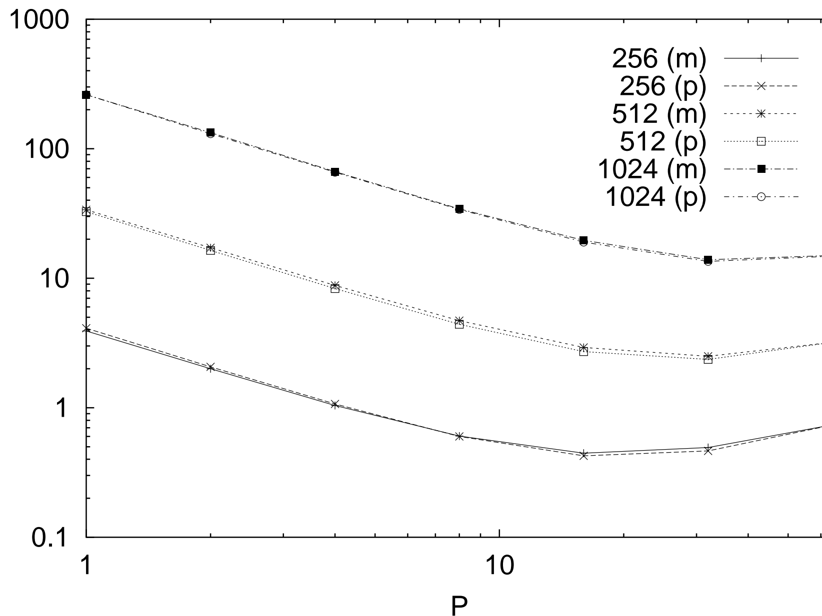

Figure 4.2: MATMUL (for $N = 256$, 512, and 1024)

4.4 GAUSS

The Gaussian elimination code illustrates the use of the PAMELA model in predicting the difference between cyclic and block partitioning, and also demonstrates the importance of modeling cache effects. The 512×512 matrix is partitioned on the j-axis. The submatrix update is coded in terms of an i loop, nested within a j loop. As the SPMD ownership tests apply to the j axis, the $j - i$ loop arrangement minimizes the overhead. The source code is shoiwn in Appendix E. The results shown in Figure 4.3 illustrate the ability of the PAMELA model to predict the superior load balancing of a cyclic partitioning compared to a block partitioning. The prediction error for large P is caused by the fact that individual broadcasts may partially overlap due to the use of asynchronous communication, which is not modeled by *bcast*. Figure 4.4 shows the performance of a slightly modified code using an $i - j$ loop arrangement. The results show that the cache performance improvement as a result of the j dimension traversal outweighs the ownership test overhead. For a cyclic partitioning S scales with P which causes delayed speedup. For a block partitioning it always holds $S = 1$. Indeed PAMELA predicts that for compute-bound settings it is *block* partitioning that produces the best results. The prediction error is 13 % on average with a maximum of 35 %.

4.5 PSRS

In this section we compare the Cost Estimator results with real measurements for the parallel sorting program. PSRS (parallel sorting by regular sampling) is a real-world program that sorts an input array X of length N using P processors, resulting in a sorted array Y [3]. Due to the mix of task and data parallelism, as well as its data-dependent control structure, the program represents an interesting challenge to the Cost Estimator (as well as the Timber compiler). We will consider two versions, called PSRS and PSRS1, differing in the mapping strategy. Both versions are modeled by the Cost Estimator the resulting predictions being compared to the actual execution time measurements.

The Spar/Java code of PSRS is given in Appendix F. X is block-partitioned into P equal parts.

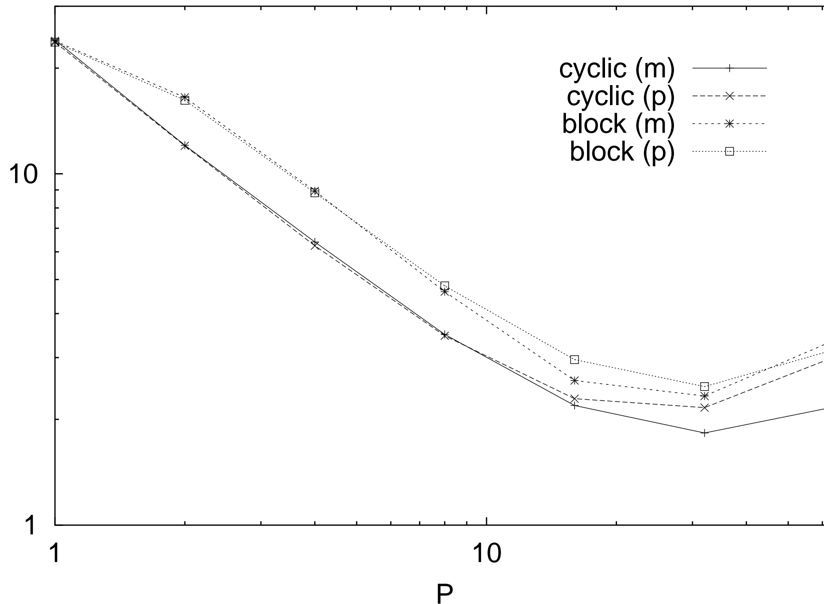

Figure 4.3: Gauss (j-i loop order, cyclic and block mapping)

Each part is sorted in parallel using Quicksort. In contrast to the data parallel programming model used thusfar, this is achieved by applying task parallelism through the `on = gpp[@p]` annotation in the parallel sort loop. To this end, `X` (as well as other arrays) are declared as arrays of vectors, the first index being mapped to the processor index. Thus, `X[p]`, being mapped to `gpp[p]`, is also sorted locally by `gpp[p]`, for all processors in parallel. Next, P^2 regular samples are taken from `X` and copied into a replicated array called `samples` (in function `compute_pivots`). After sorting `samples` (by each processor), $P-1$ pivot values are identified and stored in the replicated array `pivots`. Next, the size of each subpartition delimited by each pivot is determined (in function `disjoin`) by scanning `X` and storing each size in replicated array `zx` (disjoin part 1). Based on `zx`, two other auxiliary (replicated) arrays `sz` and `sy` are constructed (disjoin part 2). The three arrays `zx`, `sx`, and `sy`, are subsequently used to disjoin `X` into `Y` (disjoin part 3). Through this permutation (transposition) of `X` into `Y`, each partition of `Y` contains exactly those elements that conform to a global sort over all processors[2]. Finally, each partition of `Y` is locally sorted (again, task parallel), such that `Y` is completely sorted.

In this version, apart from the input and output vectors `X` and `Y`, all arrays are replicated, which would seem a straightforward mapping approach as all auxiliary information (`zx`, `sx`, `sy`, and therefore `pivots`) is needed by each processor in order to perform the `X`-to-`Y` disjoin. In addition, there is no reason to confine the creation and sorting of `samples` to one processor (processor 0, say) as then `samples` would have to be distributed to all processors (now all processors simply sort `samples` in parallel, thus saving communication).

With respect to the cost estimation a few PAMELA pragmas (`lower`, `upper`, `cond`) have been added to the code. In particular, the sequential Quicksort function (`sort`) has not been modeled at all, due to the fact that the Timber frontend maps the `break` statement to a `goto` statement, which cannot be handled by the Modeling Engine as explained in [2]. Instead, the whole function is modeled in terms of the PAMELA `cost` pragma, of which the coefficient has been determined by separately profiling the

[2]In the code used for the measurements, the three loops are merged into one loop with multiple cardinalities in order for the compiler to produce optimal code. Since the Cost Estimator currently does not support multiple cardinalities (see Section 2.1) the 3-loop version is used for Cost Estimator input.

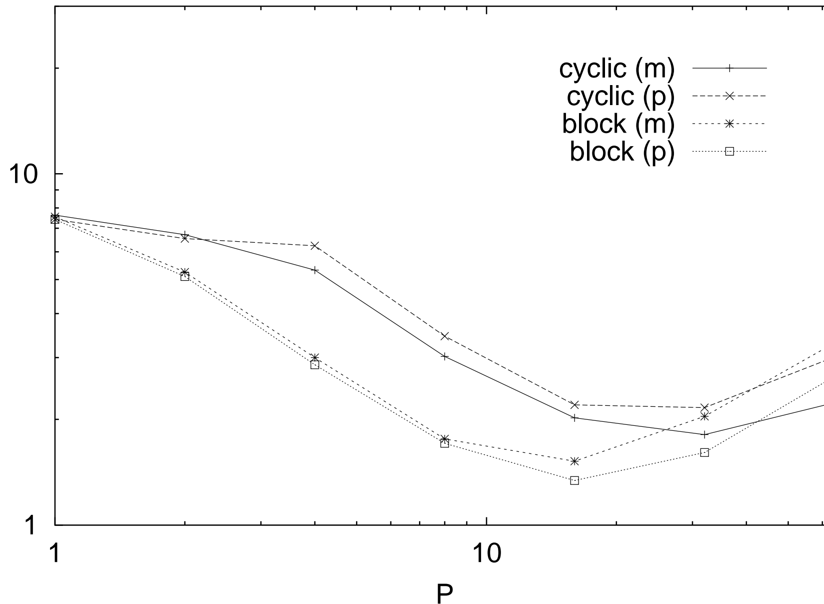

Figure 4.4: Gauss (i-j loop order, cyclic and block mapping)

`sort` function for various vector lengths.

Figure 4.5 shows the execution times for the original (and the improved) version. The plot shows that, again, the Cost Estimator is quite capable of predicting the (poor) speedup behavior of the application. Inspection of the model reveals that the bottleneck is in the first part of the `disjoin` function, i.e., the statement `x = X[i][j]`, which causes the broadcast of in total N `X` values, since `x` is replicated to all processors (`x` is subsequently used to determine `zx` which is also replicated). In the Pamela model this statement is effectively modeled by

```
seq (i_2 = 0, P - 1)
    seq (default_index = 0, N/P - 1)
        Gpp_Gpp_bcast(P,i_2)
```

Given that `Gpp_Gpp_bcast` is modeled by

```
Gpp_Gpp_bcast(P,q) =
        seq (p = 0, P-1)
                Gpp_Gpp_move(p,q)
```

where

```
Gpp_Gpp_move(p,q) =
        if (p == i)
                use(Gpp(p),t_l)
        else
                use(Gpp(p),t_g)
```

the time cost of the statement `x = X[i][j]` is $O(NP)$, which explains the *increase* of T with P, rather than the decrease that was hoped for.

The solution to this problem is by localizing the computation of `zx` through the introduction of a *local* version of `zx` called `zxl`. In the alternative version (PSRS1, see Appendix F, second part) this

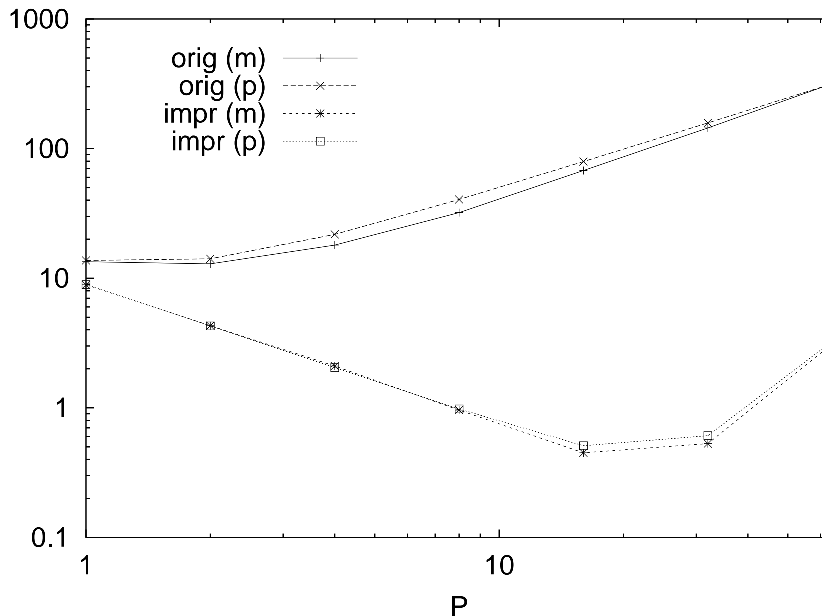

Figure 4.5: PSRS (original and improved data mapping)

array has been added which is partitioned over the processors on the first index (like `X` and `Y`). The first part of the disjoin code is now executed *task* parallel instead of data parallel, exploiting the fact that since the target array `zxl` is now also local to processor `p` no communication is needed at all. The payoff is an extra copy from `zxl` to `zz`, which, however, only adds $O(P^3)$ time cost, as this code is modeled by

```
seq (p_a = 0, P - 1)
        seq (k_2 = 0, P - 1)
                Gpp_Gpp_bcast_op(P,p_a)
```

The effect of the recoding of PSRS into PSRS1 is quite an improvement. The eventual performance drop for large P is due to the $O(P^3)$ communication overhead of the statements

```
samples[p*P+j] = X[p][j*N/(P*P)]
```

(`compute_pivots`), and

```
zx[p][k] = zxl[p][k]
```

(`disjoin`), mentioned earlier. The cost estimates are based on exactly the same parameter values which, again, yield a good fit in the sense that the initial speedup and eventual performance drop for large P are correctly predicted. The prediction error is 12 % on average with a maximum of 26 %.

The PSRS case study also shows the relative validity of the contention-less communication model used for `Gpp_Gpp_move` and `Gpp_Gpp_bcast`. In Section 1.1 global communication was mapped to the `network` resource to allow global communications to be counted. This implied that no parallelism would ever be possible for concurrent communications. However, the almost linear speedup measured in the `X`-to-`Y` translation code in part 3 of `disjoin` proves that the N global moves proceed quicker as more processors are involved. Such speedup would never be possible in the network contention model mentioned earlier.

program	A [-]	CT [s]	ST_1 [Ks]	ST_2 [ms]	err [%]
MATMUL	0	5.9	684	0.82	5
ADI	0	5.7	72	0.33	15
GAUSS	0	14.2	34	16.1	13
PSRS	6	18.1	32	1.84	12

Table 4.2: Prediction statistics (MATMUL: $N = 1,024$).

4.6 Summary

For each of the four test programs Table 4.2 lists the number of application-specific annotations (A), the compilation time (CT) of the PAMELA model generated by the Spar/Java compiler (generation time itself is negligible), the solution time (ST) of the model, and the average prediction error (err). The timing results are expressed in CPU s (450 Mhz Pentium II). In PSRS, a data-dependent, dynamic program, 6 annotations were necessary, only one of which required a few sequential profiling runs (the Quicksort function). Since machine benchmarking, sequential profiling, and PAMELA compilation time is potentially amortized over many parameter settings, modeling cost cost is negligible. The average time to compile a symbolic performance model is 11 seconds. To illustrate the cost reduction potential of symbolic prediction, the solution costs are shown before (ST_1) and after automatic simplification (ST_2). The solution time ST_2 listed in the table reflects the simplification capability of the current PAMELA compiler version. For instance, large strings of maximizations, additions and multiplications of only two or three symbols are not yet properly reduced, as well as certain sum reductions in GAUSS although $O(1)$ solutions exist. The table shows that the average time to obtain the performance plots currently ranges from 330 μs to 16.1 ms per point. As the numerical evaluation of the symbolic model is performed using the PAMELA compiler's internal numeric evaluator, more specialized evaluators may produce further speedup. The overall average prediction error is less than 10 % with a maximum of 77 % due to trivial modeling inaccuracies. Apart from providing a good scalability assessment, the model correctly predicts the best design choice in all cases.

Chapter 5

Conclusion

In this document we have presented the implementation of the Cost Estimator and we have reported on a number of experiments in order to validate the tool. The Cost Estimator has been implemented using the C programming language (12,000 lines of source code in total), with the aid of Tm (for both the Modeling Engine and PAMELA compiler), and Flex and Bison (for the PAMELA compiler).

The Modeling Engine is essentially in a prototype stage. As described in Chapter 2, the Modeling Engine is not yet capable of handling the full Spar/Java language. Nevertheless, the validation experiments have clearly shown that the current functionality serves as a good demonstration of the potential of our approach. At present the PAMELA compiler still has a prototype a symbolic optimization engine. This implies that further simplifications are possible, yielding an impressive prediction cost reduction compared to simulation.

Rather than spending all project resources on the above issues, we have conducted a number of validation experiments using real Spar/Java codes while comparing the actual execution time results on the Distributed ASCI Supercomputer (DAS) with the predictions of the Cost Estimator. All four test programs, MATMUL, ADI, GAUSS, and PSRS have demonstrated that despite the simplicity of the DAS machine model used, as well as the fact that certain aspects are not modeled (ownership testing/index bound computation overhead, latency/bandwidth properties, etc.), the Cost Estimator predictions are of good quality in the sense that they adequately predict the scalability effects of remapping and/or recoding. All examples demonstrate the huge performance potential of symbolic estimation compared to simulation. The MATMUL example demonstrates the accuracy of the cost estimation technique for different N and P. The ADI example illustrates the necessity of our program-level approach to program modeling. The GAUSS example demonstrates the utility of the PAMELA aproach in predicting the effects of partitioning and memory hierarchy. Finally, the PSRS example illustrates the relative ease with which performance annotations are included in order to enable the Cost Estimator to automatically produce its symbolic cost estimates.

In view of the limitations mentioned above, it may seem surprising that the prediction accuracy is as good as it is. This is partly due to the somewhat favorable circumstances under which the experiments have been performed, and partly due to the choice of applications. First of all, as mentioned earlier, the currently generated message passing code consumes more processor load than network load. Hence, a simple processor load model suffices and no - potentially very complex - network delay model is required. It should also be noted, that the Timber compiler fortunately generates the efficient communication schedules as assumed in our approach. Second, the machine model has been derived using a micro benchmark that is relatively close to most of the application kernels. Clearly, the results for t_c, $_l$, and t_g are well calibrated for the job. Third, the applications - in hindsight - are quite amenable to our static prediction approach. Their parallelism structure is static, which implies that one does not need to profile, e.g., parallel loop bounds. Also the sequential loop bounds and the conditional

control flow within most of the sequential modules (PSRS) exhibits a deterministic dependency on N and P that can be effectively captured by our performance pragmas. On the other hand, however, the applications used did not feature any specific properties (and certainly were not selected on any) that would be vital for our approach to succeed (an only property being that the Timber compiler must be able to produce efficient code, which at this point is still quite application-dependent). As mentioned throughout our work, our approach is general in the sense that it applies to any program as long as it has an SP parallel structure. In the worst case one must profile all branches and loop bounds as function of N and P and one must be prepared to take an $O(1)$ accuracy penalty in the case of certain contention scenarios [3]. In practice, however, one would indeed expect good results from most applications. One important class of applications where the universal "profiling problem" as mentioned earlier may induce severe prediction inaccuracies are object-oriented applications of which the function call targets are not known at compile-time (i.e. dynamic classes). In cases where the target addresses are very data-dependent an (e.g., profile-based) annotation approach may fail to produce sufficient accuracy. Currently, however, the Spar/Java application focus is on "classic" high-performance (data) parallel programs, which explains why we have not conducted research in the dynamic, object-oriented area.

Although the current tool provides a convincing proof of concept, the perspective as presented in the above discussion suggests that a number of obvious improvements can still be made. First, the Cost Estimator functionality should be extended to cover the full language, generating Pamela code for each and every Spar/Java construct and/or operation. Second, the interfacing with the Timber compiler should be re-examined. Ideally, the Modeling Engine interfaces at a point at which many of the compiler analyses have been performed (in order to avoid *ad hoc* features such as the `parallel` pragma) and where much of the final code is already generated (in order to account for overhead and message aggregation). In the current Timber compiler organization this would require interfacing at the message-passing level where vital source level control flow information has disappeared. Third, the Pamela compiler should be extended with a better symbolic optimization engine to even better benefit from the potential speedup that symbolic cost estimation offers compared to simulation. Last but not least, considerable validation effort should be directed towards experimentation involving a much larger range of applications where measurements are compared with predictions that are based on machine models that have been calibrated in great detail.

Acknowledgements

Many thanks go to Kees van Reeuwijk, Henk Sips, and especially to Frits Kuijlman, for the many fruitful discussions as well as the excellent support they provided to enable us to conduct all the experiments. An initial version of the `pam2time` engine was coded by DUT Computer Science student Eric ten Voorde. The Distributed ASCI Supercomputer (DAS) was used with the kind permission of the Dutch graduate school Advanced School for Computing and Imaging (ASCI). Needless to say, the financial support from ESPRIT, as well as the friendly collaboration within the Joses consortium is gratefully acknowledged.

Bibliography

[1] H. Bal *et al.*, "The distributed ASCI supercomputer project," *Operating Systems Review*, vol. 34, Oct. 2000, pp. 76–96.

[2] H. Gautama and A.J.C. van Gemund, "Design of performance estimator," Tech. Rep. 1-68340-44(2000)04, Delft University of Technology, Delft, The Netherlands, Aug. 2000. ESPRIT Project 28198 Deliverable D5.3.1/2.

[3] A.J.C. van Gemund and H. Gautama, "Performance estimation for embedded systems," Tech. Rep. 1-68340-44(2000)01, Delft University of Technology, Delft, The Netherlands, Jan. 2000. ESPRIT Project 28198 Deliverable D5.3.1/1.

[4] A.J.C. van Gemund and H. Gautama, "Results of validation performance estimator," Tech. Rep. 1-68340-44(2001)04, Delft University of Technology, Delft, The Netherlands, Aug. 2001. ESPRIT Project 28198 Deliverable D5.3.2/2.

[5] C. van Reeuwijk, "Tm: A code generator for recursive data structures," *Software: Practice and Experience*, vol. 22, Oct. 1992, pp. 899–908.

[6] C. van Reeuwijk, "Vnus language specification 2.1 beta 1," tech. rep., Delft University of Technology, Delft, The Netherlands, Apr. 2000.

[7] C. van Reeuwijk, A.J.C. van Gemund and H.J. Sips, "Spar: A programming language for semi-automatic compilation of parallel programs," *Concurrency: Practice and Experience*, vol. 9, Nov. 1997, pp. 1193–1205.

Appendix A

ADI Pamela Model

A.1 Partitioning along j axis

The following code shows the most important parts of the PAMELA model as generated by the Modeling Engine based on the `adi.spar` source with the j-partitioning as given in Section 1.1. The entire `adi.pam` source file contains 3,381 lines, most of the code being models of internal functions and procedures generated by the Timber compiler for, e.g., exception handling, etc.

```
%-----------------------------------------------------
% Global pragmas (file 'adi.v', line 5)
%-----------------------------------------------------

#include Gpp.pam
numeric gpp(i) = i
numeric parameter P_gpp = 20
% pamela pragma: (numeric N)
numeric parameter N
% value pragma: processors = 20



%-----------------------------------------------------
% Declarations:
%-----------------------------------------------------

...
...

%-----------------------------------------------------
% procedure declaration (file 'adi.v', line 13)
%-----------------------------------------------------

process adi_static_init_adi0_0(p) = {
        % file 'adi.v', line 15:
        % statement: adi_needstatic_init_adi0_0 = FALSE

        Gpp_move_op(gpp(p))
```

```
        % file 'adi.v', line 17:
        % statement: adi_A_0 = NulledNew<SHAPE(..)>
        ;
        Gpp_move_op(gpp(p))

}


%------------------------------------------------------
% procedure declaration (file 'adi.v', line 21)
% pamela pragma: (adi_N_0 N)
% pamela pragma: (adi_P_0 P_gpp)
%------------------------------------------------------

process adi_main_0(p) = {
        % file 'applypre1-g.cc::103', line 1:
        % statement: toplink0 = __gc_reflink_chain

        Gpp_move_op(gpp(p))

        % file 'adi.v', line 24:
        % statement: if (adi_needstatic_init_adi0_0) { ... }
        % role pragma: static-init
        ;
        if (cd_0000) {
                % file 'adi.v', line 26:
                % statement: adi_static_init_adi0_0

                adi_static_init_adi0_0(p)

        }

        % file 'adi.v', line 31:
        % statement: __gc_reflink_chain = toplink0
        ;
        Gpp_move_op(gpp(p))

        % file 'adi.v', line 33:
        % statement: for (i_0 :- 1 : (adi_N_0 - 1) : 1) { ... }
        ;
        Gpp_minus_op(gpp(p))
        ;
        seq (i_0 = 1, (adi_N_0 - 1) - 1) {
                % file 'adi.v', line 35:
                % statement: __gc_reflink_chain = toplink0

                Gpp_move_op(gpp(p))
```

```
            % file 'adi.v', line 38:
            % statement: foreach (j_0 :- 0 : adi_N_0 : 1) { ... }
            % flag pragma: independent
            % pamela pragma: parallel
            ;
            par (j_0 = 0, adi_N_0 - 1) {
                    % file 'adi.v', line 40:
                    % statement: __gc_reflink_chain = toplink0

                    Gpp_move_op(gpp(p))

                    % file 'adi.v', line 42:
                    % statement: adi_f_0(i_0,j_0)
                    ;
                    adi_f_0(p,i_0,j_0)

            }

        }

}


%------------------------------------------------------
% procedure declaration (file 'adi.v', line 48)
%------------------------------------------------------

process adi_f_0(p,i_1,j_1) = {
        % file 'applypre1-g.cc::103', line 2:
        % statement: toplink1 = __gc_reflink_chain

        Gpp_move_op(gpp(p))

        % file 'adi.v', line 51:
        % statement: if (adi_needstatic_init_adi0_0) { ... }
        % role pragma: static-init
        ;
        if (cd_0001) {
                % file 'adi.v', line 53:
                % statement: adi_static_init_adi0_0

                adi_static_init_adi0_0(p)

        }

        % file 'adi.v', line 58:
        % statement: __gc_reflink_chain = toplink1
        ;
        Gpp_move_op(gpp(p))
```

```
        % file 'adi.v', line 60:
        % statement: *adi_A_0[(i_1,j_1)] =
          ((*adi_A_0[((i_1 - 1),j_1)] + *adi_A_0[((i_1 + 1),j_1)]) div 2.0)
        % statement on: gpp[block(j_1,(adi_N_0 div adi_P_0))]
        ;
        Gpp_minus_op(gpp(j_1 div (adi_N_0 div adi_P_0)))
        ;
        Gpp_deref_op(gpp(j_1 div (adi_N_0 div adi_P_0)))
        ;
        Gpp_plus_op(gpp(j_1 div (adi_N_0 div adi_P_0)))
        ;
        Gpp_deref_op(gpp(j_1 div (adi_N_0 div adi_P_0)))
        ;
        Gpp_plus_op(gpp(j_1 div (adi_N_0 div adi_P_0)))
        ;
        Gpp_divide_op(gpp(j_1 div (adi_N_0 div adi_P_0)))
        ;
        Gpp_move_op(gpp(j_1 div (adi_N_0 div adi_P_0)))
        ;
        Gpp_deref_op(gpp(j_1 div (adi_N_0 div adi_P_0)))

}

...
...

%-----------------------------------------------------
% procedure declaration (file 'applypre4-g.cc::35', line 71)
%-----------------------------------------------------

process __rc_main0(p) = {
        % file 'applypre2-g.cc::46', line 8:
        % statement: adi_needstatic_init_adi0_0 = TRUE

        Gpp_move_op(gpp(p))

        % file 'applypre2-g.cc::46', line 9:
        % statement: adi_N_0 = 100
        ;
        Gpp_move_op(gpp(p))

        % file 'applypre2-g.cc::46', line 10:
        % statement: adi_P_0 = 20
        ;
        Gpp_move_op(gpp(p))

...
...
```

```
        % file 'adi.v', line 1138:
        % statement: catch { <block> }
        ;
        {
                % file 'adi.v', line 1139:
                % statement: adi_main_0

                adi_main_0(p)

        }

}



%----------------------------------------------------
% Main statement block:
%----------------------------------------------------

process main = {
        % file 'applypre4-g.cc::38', line 72:
        % statement: __rc_main0

        __rc_main0(0)

}

%----------------------------------------------------
% Deterministically defined symbols:
%----------------------------------------------------

% N
% adi_static_init_adi0_0
% adi_main_0
% adi_f_0

...
...

% __rc_main0
% adi_N_0
% adi_P_0
% cd_0000
% i_0
% j_0
% i_1
% j_1
% cd_0001
```

```
...
...

%-----------------------------------------------------
% Numeric symbol pragma definitions:
%-----------------------------------------------------

numeric adi_N_0 = N
numeric adi_P_0 = P_gpp
numeric cd_0000 = 0.000000
numeric cd_0001 = 0.000000

...
...

%-----------------------------------------------------
% End of automatic Pamela source code generation
%-----------------------------------------------------
```

A.2 Partitioning along i axis

The following code shows the PAMELA model as generated by the Modeling Engine based on the `adi.spar` source with the i-partitioning. Only the code that has changed is listed (some lines folded for readability).

```
%-----------------------------------------------------
% procedure declaration (file 'adi.v', line 48)
%-----------------------------------------------------

process adi_f_0(p,i_1,j_1) = {
        % file 'applypre1-g.cc::103', line 2:
        % statement: toplink1 = __gc_reflink_chain

        Gpp_move_op(gpp(p))

        % file 'adi.v', line 51:
        % statement: if (adi_needstatic_init_adi0_0) { ... }
        % role pragma: static-init
        ;
        if (cd_0001) {
                % file 'adi.v', line 53:
                % statement: adi_static_init_adi0_0

                adi_static_init_adi0_0(p)

        }

        % file 'adi.v', line 58:
```

```
    % statement: __gc_reflink_chain = toplink1
    ;
    Gpp_move_op(gpp(p))

    % file 'applybreakout-g.cc::480', line 1:
    % statement: *adi_A_0[((i_1 - 1),j_1)] =
      (Wrapper) *adi_A_0[((i_1 - 1),j_1)]
    % flag pragma: isAssignToSelf
    % statement on: gpp[block((i_1 + 1),(adi_N_0 div adi_P_0))]
    % expression wrapper, on: gpp[block((i_1 - 1),(adi_N_0 div adi_P_0))]
    ;
    Gpp_minus_op(gpp((i_1 - 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_deref_op(gpp((i_1 - 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_Gpp_move_op(gpp((i_1 + 1) div (adi_N_0 div adi_P_0)),
                    gpp((i_1 - 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_minus_op(gpp((i_1 + 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_deref_op(gpp((i_1 + 1) div (adi_N_0 div adi_P_0)))

    % file 'adi.v', line 60:
    % statement: *adi_A_0[(i_1,j_1)] =
      (Wrapper) ((*adi_A_0[((i_1 - 1),j_1)] +
                 *adi_A_0[((i_1 + 1),j_1)]) div 2.0)
    % statement on: gpp[block(i_1,(adi_N_0 div adi_P_0))]
    % expression wrapper, on: gpp[block((i_1 + 1),(adi_N_0 div adi_P_0))]
    ;
    Gpp_minus_op(gpp((i_1 + 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_deref_op(gpp((i_1 + 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_plus_op(gpp((i_1 + 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_deref_op(gpp((i_1 + 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_plus_op(gpp((i_1 + 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_divide_op(gpp((i_1 + 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_Gpp_move_op(gpp(i_1 div (adi_N_0 div adi_P_0)),
                    gpp((i_1 + 1) div (adi_N_0 div adi_P_0)))
    ;
    Gpp_deref_op(gpp(i_1 div (adi_N_0 div adi_P_0)))

}
```

Appendix B

GPP Machine Pamela Model

The following is the machine model `Gpp.pam` that has been used in the experiments reported in Chapter 4.

```
%-------------------------------------------------------
% Pamela Gpp machine model
%-------------------------------------------------------

%-------------------------------------------------------
% machine resources:
%-------------------------------------------------------

resource parameter fcfs(i,m)

% network contention, use multiple server
% map to resource index #0

numeric parameter M_net
resource network = fcfs(0,M_net)

% Gpp processor array definition:
% map to resource index #1 ...

resource Gpp(i) = fcfs(i+1,1)

% define a default DontCareProc index

numeric DontCareProcIndex = 0


%-------------------------------------------------------
% computation operators:
%-------------------------------------------------------

% first model generic cost function
% c specified in CPU s times some known BW
% on the processor the code was benchmarked
```

```
% so divide by BW_Gpp to convert to Gpp CPU s

process Gpp_cost(p,c) = use(Gpp(p),c/BW_Gpp)
numeric BW_Gpp = 1.0

% next follow all the specific operators
% model them identically for now with t_c delay

numeric parameter t_c
process Gpp_comp(p) = use(Gpp(p),t_c)

process Gpp_negate_op(p) = Gpp_comp(p)
process Gpp_not_op(p) = Gpp_comp(p)
process Gpp_uplus_op(p) = Gpp_comp(p)

process Gpp_divide_op(p) = Gpp_comp(p)
process Gpp_equal_op(p) = Gpp_comp(p)
process Gpp_greater_op(p) = Gpp_comp(p)
process Gpp_greaterequal_op(p) = Gpp_comp(p)
process Gpp_less_op(p) = Gpp_comp(p)
process Gpp_lessequal_op(p) = Gpp_comp(p)
process Gpp_minus_op(p) = Gpp_comp(p)
process Gpp_mod_op(p) = Gpp_comp(p)
process Gpp_notequal_op(p) = Gpp_comp(p)
process Gpp_or_op(p) = Gpp_comp(p)
process Gpp_plus_op(p) = Gpp_comp(p)
process Gpp_times_op(p) = Gpp_comp(p)

process Gpp_deref_op(p) = delay(0)
process Gpp_cast_op(p) = delay(0)


%---------------------------------------------------
% communication operators:
%---------------------------------------------------

% guaranteed local communication operator
% no network contention

numeric parameter t_l
process Gpp_move_op(i) = delay(t_l)


% possibly global communication operator
% network contention is small compared
% to software overhead so only processor workload

numeric parameter t_g
process Gpp_Gpp_move_op(p,q) =
```

```
        if (p == q)
                % local to p
                use(Gpp(p),t_l)
        else
                % xfer p <- q
                use(Gpp(p),t_g)

% global broadcast communication operator
% in HPAM this is a sequential loop of sends

process Gpp_Gpp_bcast_op(P,q) =
        seq (p = 0,P-1)
                Gpp_Gpp_move_op(p,q)


%------------------------------------------------------
% miscelaneous instructions:
%------------------------------------------------------

% Vnus external routines

process print(p) = delay(0)
process println(p) = delay(0)
```

Appendix C

DEMO Pamela Model

The following code shows the most important parts of the PAMELA model as generated by the Modeling Engine based on the `demo.spar` source as given in Section 2.4. The entire `demo.pam` source file contains 3,446 lines, most of the code being models of internal functions and procedures generated by the Timber compiler for, e.g., exception handling, etc.

```
%----------------------------------------------------
% Global pragmas (file 'demo.v', line 5)
%----------------------------------------------------

#include Dsp.pam
numeric cpu(i) = i
numeric parameter P_cpu = 20
% value pragma: processors = 20



%----------------------------------------------------
% Declarations:
%----------------------------------------------------

...
...

%----------------------------------------------------
% procedure declaration (file 'demo.v', line 21)
% pamela pragma: (demo_N_0 100)
%----------------------------------------------------

process demo_main_0(p) = {
        % file 'applypre1-g.cc::103', line 1:
        % statement: toplink0 = __gc_reflink_chain

        Dsp_move_op(cpu(p))

        % file 'demo.v', line 24:
        % statement: if (demo_needstatic_init_demo0_0) { ... }
```

```
        % role pragma: static-init
        ;
        if (cd_0000) {
                % file 'demo.v', line 26:
                % statement: demo_static_init_demo0_0

                demo_static_init_demo0_0(p)

        }

        % file 'demo.v', line 31:
        % statement: __gc_reflink_chain = toplink0
        ;
        Dsp_move_op(cpu(p))

        % file 'demo.v', line 34:
        % statement: foreach (i_0 :- 0 : demo_N_0 : 1) { ... }
        % flag pragma: independent
        % pamela pragma: parallel
        ;
        par (i_0 = 0, demo_N_0 - 1) {
                % statement block on: cpu[cyclic(i_0,20)]
                % file 'demo.v', line 36:
                % statement: *demo_A_0[(i_0)] = (*demo_A_0[(i_0)] * 2.0)

                Dsp_deref_op(cpu(i_0 div 5))
                ;
                Dsp_times_op(cpu(i_0 div 5))
                ;
                Dsp_move_op(cpu(i_0 div 5))
                ;
                Dsp_deref_op(cpu(i_0 div 5))

        }

        % file 'demo.v', line 40:
        % statement: demo_f_0(1,2)
        % statement on: cpu[local(0)]
        ;
        demo_f_0(0,1,2)

}


%------------------------------------------------------
% procedure declaration (file 'demo.v', line 44)
%------------------------------------------------------

process demo_f_0(p,n_0,m_0) = {
```

```
% file 'applypre1-g.cc::103', line 2:
% statement: toplink1 = __gc_reflink_chain

Dsp_move_op(cpu(p))

% file 'demo.v', line 47:
% statement: if (demo_needstatic_init_demo0_0) { ... }
% role pragma: static-init
;
if (cd_0001) {
        % file 'demo.v', line 49:
        % statement: demo_static_init_demo0_0

        demo_static_init_demo0_0(p)

}

% file 'demo.v', line 54:
% statement: __gc_reflink_chain = toplink1
;
Dsp_move_op(cpu(p))

% file 'demo.v', line 56:
% statement: { <block> }
;
{
        % file 'demo.v', line 57:
        % statement: x_0 = (BASETYPE) (demo_N_0 div 2)

        Dsp_divide_op(cpu(p))
        ;
        Dsp_cast_op(cpu(p))
        ;
        Dsp_move_op(cpu(p))

        % file 'demo.v', line 60:
        % statement: for (i_1 :- demo_g_0(x_0) : (demo_N_0 - 1) : 1) ...
        % pamela pragma: (lower ((demo_N_0 div 5)))
        ;
        demo_g_0(p,arg_0003)
        ;
        Dsp_minus_op(cpu(p))
        ;
        seq (i_1 = lb_0002, (demo_N_0 - 1) - 1) {
                % file 'demo.v', line 63:
                % statement: if ((x_0 == (BASETYPE) 2)) { ... }
                % pamela pragma: (cond 0)

                Dsp_cast_op(cpu(p))
```

```
                ;
                Dsp_equal_op(cpu(p))
                ;
                if (cd_0004) {
                        % file 'demo.v', line 65:
                        % statement: x_0 = (1.0 * 2.0)

                        Dsp_times_op(cpu(p))
                        ;
                        Dsp_move_op(cpu(p))

                }

                % file 'demo.v', line 70:
                % statement: if ((i_1 > n_0)) { ... }
                ;
                Dsp_greater_op(cpu(p))
                ;
                if ((i_1 > n_0)) {
                        % file 'demo.v', line 72:
                        % statement: x_0 = (x_0 div (BASETYPE) 2)

                        Dsp_cast_op(cpu(p))
                        ;
                        Dsp_divide_op(cpu(p))
                        ;
                        Dsp_move_op(cpu(p))

                }

        }

        % file 'demo.v', line 79:
        % statement: while ((x_0 < (BASETYPE) demo_N_0)) { ... }
        % pamela pragma: (cost ((demo_N_0 div m_0)))
        % cost pragma found
        ;
        Dsp_cost(cpu(p),((demo_N_0 div m_0)))


    }

}


%------------------------------------------------------
% function declaration (file 'demo.v', line 87)
%------------------------------------------------------
```

```
process demo_g_0(p,x_1) = {
        % file 'demo.v', line 90:
        % statement: if (demo_needstatic_init_demo0_0) { ... }
        % role pragma: static-init

        if (cd_0005) {
                % file 'demo.v', line 92:
                % statement: demo_static_init_demo0_0

                demo_static_init_demo0_0(p)

        }

        % file 'demo.v', line 97:
        % statement: return (BASETYPE) (x_1 div (BASETYPE) 10)
        ;
        Dsp_cast_op(cpu(p))
        ;
        Dsp_divide_op(cpu(p))
        ;
        Dsp_cast_op(cpu(p))
        ;
        delay(0)

}


...
...


%-----------------------------------------------------
% procedure declaration (file 'applypre4-g.cc::35', line 70)
%-----------------------------------------------------

process __rc_main0(p) = {
        % file 'applypre2-g.cc::46', line 8:
        % statement: demo_needstatic_init_demo0_0 = TRUE

        Dsp_move_op(cpu(p))

...
...

        % file 'demo.v', line 1176:
        % statement: __gc_reflink_chain = __gc_globallink
        ;
        Dsp_move_op(cpu(p))
```

```
        % file 'demo.v', line 1177:
        % statement: catch { <block> }
        ;
        {
                % file 'demo.v', line 1178:
                % statement: demo_main_0

                demo_main_0(p)

        }

}



%----------------------------------------------------
% Main statement block:
%----------------------------------------------------

process main = {
        % file 'applypre4-g.cc::38', line 71:
        % statement: __rc_main0

        __rc_main0(0)

}

%----------------------------------------------------
% Deterministically defined symbols:
%----------------------------------------------------

% demo_static_init_demo0_0
% demo_main_0
% demo_f_0
% demo_g_0

...
...

% __rc_main0
% demo_N_0
% cd_0000
% parallel
% i_0
% n_0
% m_0
% cd_0001
% lb_0002
% i_1
```

```
% cd_0004
% x_1
% cd_0005

...
...

%-----------------------------------------------------
% Numeric symbol pragma definitions:
%-----------------------------------------------------

numeric demo_N_0 = 100
numeric cd_0000 = 0.000000
numeric parallel = 1
numeric cd_0001 = 0.000000
numeric lb_0002 = ((demo_N_0 div 5))
numeric cd_0004 = 0
numeric cd_0005 = 0.000000

%-----------------------------------------------------
% Numeric symbol default definitions:
%-----------------------------------------------------

numeric default_expr = 0

numeric arg_0003 = default_expr

...
...


%-----------------------------------------------------
% Default process definitions:
%-----------------------------------------------------

process default_proc = delay(0)



%-----------------------------------------------------
% End of automatic Pamela source code generation
%-----------------------------------------------------
```

Appendix D

MATMUL Source

```
/*----------------------------------------------------------
 * Matrix Multiplication
 *----------------------------------------------------------
 */

globalpragmas <$
        ProcessorType = ((Gpp "Gpp.pam")),
        Processors = ((Gpp gpp[64])),
        Pamela = (numeric "N")
$>;

package spar.testsuite;

public class matmul
{
        static int N = 512;
        static int P = 64;

        static double A[*,*]
        <$on = (lambda (i j) gpp[(block i (@N / @P))])$> = new double[N,N];

        static double B[*,*]
        <$on = (lambda (i j) gpp[(block j (@N / @P))])$> = new double[N,N];

        static double C[*,*]
        <$on = (lambda (i j) gpp[(block j (@N / @P))])$> = new double[N,N];

        static double TMP[*]
        <$on = (lambda (i) gpp[_all])$> = new double[N];

        <$ Pamela = (@N N),
           Pamela = (@P P_gpp) $>

        public static void main() {
```

```
        <$ independent $>
        foreach (j:-0:N)
                <$ independent,
                   Pamela = parallel $>
                foreach (i:-0:N)
                        A[i,j] = 1.0;

        <$ independent $>
        foreach (i:-:N)
                <$ independent,
                   Pamela = parallel $>
                foreach (j:-0:N)
                        B[i,j] = 1.0;

        <$ independent $>
        foreach (i:-:N)
                <$ independent,
                   Pamela = parallel $>
                foreach (j:-0:N)
                        C[i,j] = 0.0;

        foreach (i:-0:N) {

                // cache entire A[i,*]
                <$ independent $>
                foreach (k:-0:N)
                        TMP[k] = A[i,k];

                <$ independent,
                   Pamela = parallel $>
                foreach (j:-0:N) {
                        foreach (k:-0:N) {
                                C[i,j] += TMP[k]*B[k,j];
                        }
                }
        }
    }
}
```

Appendix E

GAUSS Sources

```
/*--------------------------------------------------------
 * Gaussian Elimination
 *--------------------------------------------------------
 */

globalpragmas <$
        ProcessorType = ((Gpp "Gpp.pam")),
        Processors = ((Gpp gpp[64])),
        Pamela = (numeric "N")
$>;


package spar.testsuite;

public class gauss
{
        static int N = 512;
        static int P = 64;

        static final boolean DEBUG = false;

// choose between block or cyclic partitioning:

        static double A[*,*]
        // <$on = (lambda (i j) gpp[(cyclic j)])$> = new double [N,N];
        <$on = (lambda (i j) gpp[(block j (@N / @P))])$> = new double [N,N];

        static double v[*]
        // <$on = (lambda (j) gpp[(cyclic j)])$> = new double [N];
        <$on = (lambda (j) gpp[(block j (@N / @P))])$> = new double [N];

        static double w[*]
        <$on = (lambda (i) gpp[(_all)])$> = new double [N];

        <$ Pamela = (@N N),
```

```
   Pamela = (@P P_gpp) $>


public static void main() {

        int m;

// initialization:

        if (DEBUG) __println(1,"initializing");

        <$ independent $>
        foreach (i :- 0:N)
                <$ independent,
                   Pamela = parallel $>
                foreach (j :- 0:N)
                        A[i,j] = Math.random();
        if (DEBUG) {
                __println(1,"A:");
                dump(A,N);
        }

// computation:

        // __println(1,N);
        for (k :- 0:N-1) {

                if (DEBUG) __println(1,"scale");

                // find pivot index

                m = k;
                v[k] = A[k,k] > 0 ? A[k,k] : -A[k,k];
                for (i :- k+1:N)
                        if (A[i,k] > v[k] || -A[i,k] > v[k]) {
                                m = i;
                                v[k] = A[i,k];
                        }

                // swap rows k and m

                <$ independent,
                   Pamela = parallel $>
                foreach (j :- k:N) {
                        v[j] = A[m,j];
                        A[m,j] = A[k,j];
                        A[k,j] = v[j];
                }
```

```
                        // scale column

                        for (i :- k+1:N)
                                A[i,k] = A[i,k] / A[k,k];

                        if (DEBUG) dump(A,N);



                        // update submatrix

                        if (DEBUG) __println(1,"update");

// i-j loop nesting:

                        <$ independent $>
                        foreach (i :- k+1:N)
                                w[i] = A[i,k];

                        <$ independent,
                           Pamela = parallel $>
                        foreach (j :- k+1:N) {
                                <$ independent $>
                                foreach (i :- k+1:N)
                                        A[i,j] = A[i,j] - w[i] * A[k,j];
                        }


// j-i loop nesting:

//                      <$ independent $>
//                      foreach (i :- k+1:N) {
//                              w[i] = A[i,k];
//                              <$ independent,
//                                 Pamela = parallel $>
//                              foreach (j :- k+1:N)
//                                      A[i,j] = A[i,j] - w[i] * A[k,j];
//                      }


                        if (DEBUG) dump(A,N);

                }
        }

        static public void dump(double [*,*] A, int dim) {

                for (i :- 0:dim) {
                        for (j :- 0:dim) {
```

```
                        __print(1,A[i,j]);
                        __print(1," ");
                    }
                    __println(1,"");
                }
            }
        }
```

Appendix F

PSRS Sources

F.1 PSRS

The following code corresponds to the original version without additional local index array `zxl`.

```
/*-------------------------------------------------------
 * Parallel Sorting by Regular Sampling (Data // version)
 *-------------------------------------------------------
 */

globalpragmas <$
        ProcessorType = ((Gpp "Gpp.pam")),
        Processors = ((Gpp gpp[32])),

        // problem size parameter N
        Pamela = (numeric "N")
$>;


public class psrs
{
        static int N=102400;
        static int P=32;
        static int l;

        static final boolean DEBUG = false;


        static double [*][]
        <$ on = (lambda (p) gpp[(local p)]) $>
                X = new double [P][*];

        static double [*][]
        <$ on = (lambda (p) gpp[(local p)]) $>
                Y = new double [P][*];

        static int [*][]
```

```
<$ on = (lambda (p) gpp[_all]) $>
        zx = new int [P][*];

static int [*][]
<$ on = (lambda (p) gpp[_all]) $>
        sx = new int [P][*];

static int [*][]
<$ on = (lambda (p) gpp[_all]) $>
        sy = new int [P][*];

static double [*]
<$ on = (lambda (p) gpp[(_all)]) $>
        samples = new double [P*P];

static double [*]
<$ on = (lambda (p) gpp[(_all)]) $>
        pivots = new double [P];


static public void main () {

        // export @N as N, surprisingly this prevents N propagation!
        // export P as P_gpp
        <$ Pamela = (@N N),
           Pamela = (@P P_gpp) $>


// initialization:

        if (DEBUG)
                __println(1,"initializing");
        <$ independent $>
        foreach (p :- 0:P) {
                {
                zx[p] = new int [P];
                sx[p] = new int [P];
                sy[p] = new int [P];
                }
        }
        <$ independent,
           Pamela = parallel $>
        foreach (p :- 0:P) {
                <$ on = gpp[@p] $> {
                X[p] = new double [N/P];
                Y[p] = new double [2*(N/P)];
                initialize(X[p]);
                }
        }
```

```
        if (DEBUG) {
                __println(1,"X:");
                for (p :- 0:P)
                        <$ on = gpp[(local @p)] $> dump_d(X[p],N/P);
        }

// task parallel sort of X[p] on gpp[p]:

        if (DEBUG)
                __println(1,"sort X");

        <$ independent,
           Pamela = parallel $>
        foreach (p :- 0:P) {
                <$ on = gpp[@p] $>
                sort(X[p],0,N/P-1);
        }

        if (DEBUG) {
                __println(1,"X:");
                for (p :- 0:P)
                        <$ on = gpp[(local @p)] $> dump_d(X[p],N/P);
        }

// compute pivots:

        if (DEBUG)
                __println(1,"compute_pivots");

        compute_pivots();

        if (DEBUG) {
                __println(1,"samples:");
                dump_d(samples,P*P);
                __println(1,"pivots:");
                dump_d(pivots,P);
        }

// disjoin X into Y:

        if (DEBUG)
                __println(1,"disjoin_parts");
        disjoin_parts();
        if (DEBUG) {
                __println(1,"zx:");
                for (p :- 0:P)
                        <$ on = gpp[(local @p)] $> dump_i(zx[p],P);
                __println(1,"sx:");
                for (p :- 0:P)
```

```
                        <$ on = gpp[(local @p)] $> dump_i(sx[p],P);
                __println(1,"sy:");
                for (p :- 0:P)
                        <$ on = gpp[(local @p)] $> dump_i(sy[p],P);
                __println(1,"X:");
                for (p :- 0:P)
                        <$ on = gpp[(local @p)] $> dump_d(X[p],(N/P));
                __println(1,"Y:");
                for (p :- 0:P)
                        <$ on = gpp[(local @p)] $> dump_d(Y[p],2*(N/P));
        }

// task parallel sort of Y[p] on gpp[p]:

        if (DEBUG)
                __println(1,"sort Y");
        <$ independent,
           Pamela = parallel $>
        foreach (p :- 0:P) {
                <$ on = gpp[@p] $> {
                l = p < P-1 ? sy[p+1][0]-sy[p][0] : N-sy[p][0];

                <$ Pamela = (@l (@N/@P)) $>
                sort(Y[p],0,l-1);
                }
        }
        if (DEBUG) {
                __println(1,"Y:");
                for (p :- 0:P)
                        <$ on = gpp[(local @p)] $> dump_d(Y[p],2*(N/P));
        }

// check results:

        if (DEBUG)
                __println(1,"checking");
        check();


}


static public void initialize(double [] X) {

        // sort vector

        for (i :- 0:N/P)
                X[i] = Math.random();
}
```

```
static public void compute_pivots() {

        // take regular samples from X
        // sort samples and assign regular pivots
        // input samples[*] (proc 0)
        // output pivots[*] (bcast)

        <$ Pamela = parallel $>
        for (p :- 0:P)
                for (j :- 0:P)
                        samples[p*P+j] = X[p][j*N/(P*P)];
        sort(samples,0,P*P-1);
        for (p :- 0:P-1) {
                pivots[p] = samples[(p+1)*P+P/2-1];
        }
}

static public void disjoin_parts() {
        int sx1 = 0;
        int sy1 = 0;
        int m1, m2;
        int j,d;
        double x;

        // compute sizes of individual partitions
        // to be disjoined
        // input pivots, X
        // output zx

        if (DEBUG)
                __println(1,"disjoin part 1");

        for (i :- 0:P) {
                for (p :- 0:P)
                        zx[i][p] = 0;
                d = 0;

                // Side effect so keep it a while loop

                <$ Pamela = (lower 0),
                   Pamela = (upper ((@N / @P) - 1)) $>
                for (j = 0; j <= N/P-1; j++) {

                        x = X[i][j];

                        <$ Pamela = (cond 1) $>
                        if ((pivots[d] >= x) || (d >= P-1)) {
                              zx[i][d]++;
```

```
                        }
                        else {
                                j--;    // Side effect
                                d++;
                        }
                }
        }

        // compute global start indices of partitions
        // to be disjoined
        // input zx
        // output sx, sy

        if (DEBUG)
                __println(1,"disjoin part 2");

        for (p :- 0:P) {
                for (k :- 0:P) {
                        sx[p][k] = sx1;
                        sx1 += zx[p][k];
                        sy[p][k] = sy1;
                        sy1 += zx[k][p];
                }
        }

        // disjoin X to Y
        // by moving X partitions to Y partitions
        // in a sort of Butterfly fashion

        if (DEBUG)
                __println(1,"disjoin part 3");

        for (p :- 0:P) {
                for (k :- 0:P) {

                        <$ Pamela = (upper (@N / (@P * @P) - 1)) $>
                        for (l :- 0:zx[k][p]) {
                                Y[p][sy[p][k]-sy[p][0]+l] =
                                X[k][sx[k][p]-k*(N/P)+l];
                        }
                }
        }
}

static public void sort(double [] g, int glb, int gub) {
        int j,k,l,m;
        double temp;

        <$ Pamela = (cost (0.353e-6 * (@gub-@glb+1) *
```

```
                        (log (@gub-@glb+1)) * BW_Gpp)) $>

        if (! ((gub-glb+1) < 2)) {
                l = ((gub-glb+1) >> 1) + glb;
                k = gub;
                for (;;) {
                        if (l > glb)
                                temp = g[--l];
                        else {
                                temp = g[k];
                                g[k] = g[glb];
                                if (--k == glb) {
                                        g[glb] = temp;
                                        break;
                                }
                        }
                        m = l;
                        j = (l-glb+1)*2 + glb-1;
                        while (j <= k) {
                                if ((j < k) && (g[j] < g[j+1]))
                                        j++;
                                if (temp < g[j]) {
                                        g[m] = g[j];
                                        m = j;
                                        j = glb+((j-glb+1)*2)-1;
                                }
                                else
                                        j = k + 1;
                        }
                        g[m] = temp;
                }
        }
}

static public void dump_d(double [] x, int dim1) {

        for (i :- 0:dim1) {
                __print(1,x[i]);
                __print(1," ");
        }
        __println(1,"");
}

static public void dump_i(int [] x, int dim1) {

        for (i :- 0:dim1) {
                __print(1,x[i]);
                __print(1," ");
        }
```

```
		__println(1,"");
	}

	static public void check() {

		double  d;
		int     m;

		d = 0.0;
		for (p :- 0:P) {
			m = p < P-1 ? sy[p+1][0]-sy[p][0] : N-sy[p][0];
			for (i :- 0:m) {
				if (d > Y[p][i])
					__println(1,"*** Sorting Error");
				else
					d = Y[p][i];
			}
		}
	}
}
```

F.2 PSRS1

The following code corresponds to the improved version with additional local index array `zxl`. Only the parts of the code that have been modified are shown.

```
/*-------------------------------------------------------
 * Parallel Sorting by Regular Sampling (Data // version)
 *-------------------------------------------------------
 */

...
...

	// additional array:

	static int [*][]
	<$ on = (lambda (p) gpp[(local p)]) $>
		zxl = new int [P][*];

...
...

		<$ independent,
		   Pamela = parallel $>
		foreach (p :- 0:P) {
			<$ on = gpp[@p] $> {
			X[p] = new double [N/P];
			zxl[p] = new int [P];
```

```
                Y[p] = new double [2*(N/P)];
                initialize(p,X[p]);
                }
        }
...
...


    static public void disjoin_parts() {
        int sx1 = 0;
        int sy1 = 0;
        int m1, m2;
        int j,d;
        double x;

        // compute sizes of individual partitions
        // to be disjoined
        // input pivots, X
        // output zxl <==

        if (DEBUG)
                __println(1,"disjoin part 1");

        <$ Pamela = parallel $>
        for (i :- 0:P) { <$ on = gpp[@i] $> { // localize X access

                for (p :- 0:P)
                        zxl[i][p] = 0;
                d = 0;

                // Side effect so keep it a while loop

                <$ Pamela = (lower 0),
                   Pamela = (upper ((@N / @P) - 1)) $>
                for (j = 0; j <= N/P-1; j++) {

                        x = X[i][j]; // work around compiler bug

                        <$ Pamela = (cond 1) $>
                        if ((pivots[d] >= x) || (d >= P-1)) {
                                zxl[i][d]++;
                        }
                        else {
                                j--;    // the j side effect
                                d++;
                        }
                } }
        }
        if (DEBUG) {
```

```
                __println(1,"zxl:");
                for (p :- 0:P)
                        <$ on = gpp[(local @p)] $>
                        dump_i(zxl[p],P);
        }


        // copy local zxl to global zx

        for (p :- 0:P) {
                for (k :- 0:P) {
                        zx[p][k] = zxl[p][k];
                }
        }


        // continue with original code (using zx)

...
...


    }
}
```