Portable Parallelizing Fortran Compiler

appeared in Concurrency Practice and Experience, Vol. 8(2), pp. 91-123, March 1996.

A. Averbuch, R. Dekel, E. Gabber
School of Mathematical Sciences
Department of Computer Science
Tel Aviv University
Tel Aviv 69978, Israel
e-mail: amir@math.tau.ac.il

Abstract

The Portable Parallelizing Fortran Compiler ($PPFC$) is an additional component for the *portable programming environment* developed in Tel-Aviv University[21]-[24] for scientific code. This environment supports portable and efficient programming of diverse MIMD multiprocessors, both distributed and shared memory.

Till now this environment consisted of two tools: the Virtual Machine for MultiProcessors (VMMP) ([23], [24]) and the Portable Parallelizing Pascal Compiler (P^3C) ([21], [22]). We added the $PPFC$ which is an automatic parallelizer compiler for the Fortran language. The compiler is fully automatic (does not require additional declarations to assist parallelization), which is characterized by loops operating on regular data structures, produces efficient and portable code for a variety of multiprocessors from the same serial code.

The parallel implementation uses the VMMP which is a software package that provides a coherent set of services for explicitly parallel application programs running on diverse MIMD multiprocessors. VMMP is intended to simplify parallel program writing and to promote portable and efficient programming.

The $PPFC$ parallelized 12 out of the 24 Livermore Loops [41]. It was also applied to parallelize all the 14 Fortran application programs that where parallelized by the P^3C and achieved the same speedups and efficiencies. In most examples the $PPFC$ achieved high speedups and efficiencies on all target multiprocessors.

The $PPFC$ emphasizes efficiency and code portability. Although $PPFC$ employs a relatively simple data flow analysis, it produces an efficient code for various widely used application programs.

1 Introduction

The Portable Parallelizing Fortran Compiler ($PPFC$) compiler translates serial programs written in Fortran, which is essentially standard Fortran-77 with minor extensions, into an explicitly parallel SPMD (Single Program Multiple Data) [19] code for VMMP ([23], [24]).

The $PPFC$ is derived from the P^3C ([21], [22]) so that it preserves all the features of the P^3C and also adds some extensions.

Figure 1 illustrates the portable environment structure that contains the $PPFC$, P^3C and VMMP.

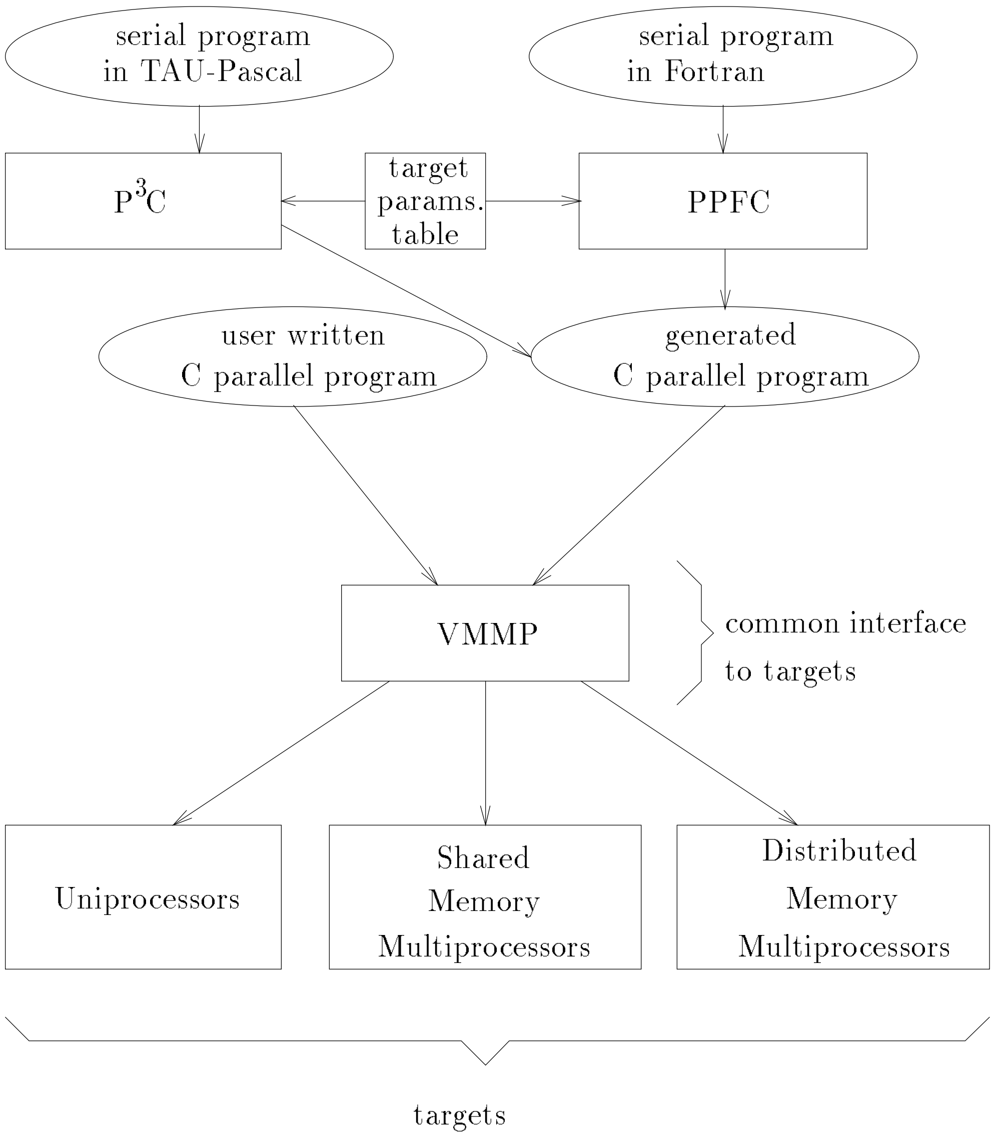


Figure 1: Portable Environment Structure

PPFC generates optimized code from the same source program for all supported targets. It is intended to generate parallel programs with comparable efficiency to manual coded programs. *PPFC* is intended to parallelize "scientific code", which is characterized by loops operating on regular data structures.

The separation of the environment into a static part (*PPFC* and P^3C) and a runtime part (VMMP) simplifies the design and enhances portability. In this way, the machine independent task of parallelization extraction is performed by *PPFC* or P^3C , while the specific details of the implementation are hidden inside VMMP. Moreover, VMMP code may be regarded as a common intermediate language, much like the intermediate language used between compiler front end and back ends.

PPFC employs a relatively simple parallelization algorithm, which recognizes parallel loop by a combination of data dependency analysis and pattern matching. All of the variables in the parallel loops must belong to one of the permitted access categories, as described in section 4. *PPFC* adds new techniques to the methods used by P^3C, for parallelizing loops such as loop that contains weak crossing signal index subscripts (WCSIS) data dependency.

PPFC performs an automatic data partitioning, which distributes the data among the processors according to several strategies. The data partitioning is a crucial step in the parallelization. Good partitioning reduces communication overheads and balances the workload between the processors. The data partitioning is dictated by the variable access pattern in the parallel loops. The generated code is modified according to the *owner computes* rule [47], which states that each processor updates only the data it owns.

PPFC computes an accurate execution time estimation of the generated parallel code, which is performed by a static analysis of the program at compile time with reference to the target parameters table. The time estimation is used mainly to avoid generating parallel code when the synchronization and data distribution overheads are larger than the savings in computation speed. It is also used in the generation of a run time test that chooses between parallel and serial execution of certain loops.

The parameters table contains the execution time of all primitive constructs appearing in the serial program, such as arithmetic operations, array indexing, data movement, `if` overhead, loop test overhead, etc. The execution time is expressed relative to the time of one integer addition. The parameters table also contains the coefficients of the cost functions describing various VMMP services.

Although any VMMP application program will run on every target machine supported by VMMP, it will probably run inefficiently, because the efficiency of VMMP services changes on different targets. *PPFC* ensures the code efficiency by generating parallel code only when the execution time estimation indicates that the parallel code will actually reduce execution time (when taking into the account the associated overheads). In cases of uncertain estimation of loop execution time, *PPFC* generates a run time test to choose between serial and parallel executions based on the loop's bounds and pre-computed thresholds.

PPFC is fully automatic, and does not require additional programmer declarations to assist the parallelization. In particular, data distribution declarations are not needed.

The goal of the *PPFC* was to implement an automatic parallelizer which emphasize portability and efficiency. Although *PPFC* employs a relatively simple data flow analysis comparing to current parallelizing compilers, such as PTRAN [1], it produces an efficient code for various widely used application programs.

PPFC was developed by using a combination of YACC, LEX and Prolog [17]. *PPFC* may be used together with a vectorizing C compiler, because *PPFC* parallelizes the outer loops, while the vectorizer may handles the inner loops.

PPFC currently supports two shared memory multiprocessors (Sequent Symmetry and MOSIX) and a distributed memory multiprocessor (a network of Transputers).

1.1 Results

The *PPFC* was applied to parallelize the Livermore Loops [41], which are widely used as a benchmark for numerical intensive calculation in Fortran, and succeeded to parallelize 12 out of 24 loops. It was also applied to parallelize all the 14 Fortran application programs that previously where parallelized by the P^3C (Pascal version) and achieved the same speedups and efficiencies. It was also applied to parallelize examples that contain weak crossing signal index subscripts (WCSIS) loops.

The parallel code produced by *PPFC* achieved high speedups on all supported targets. For example, the following results where obtained on a 26 processors Sequent Symmetry (only 25 processors used), eight processors MOSIX and a network of eight Transputers. Matrix inversion of a 250 by 250 matrix using Gauss-Jordan reduction achieved speedups (and efficiencies) of 20.32 (81.3%), 6.54 (81.7%) and 6.41 (80.1%) on the Sequent Symmetry, MOSIX and Transputers network, respectively. The Livermore Loop 3 achieved speedups (and efficiencies) of 24.32 (93.76%), 7.66 (95.81%) and 7.83 (97.9%) on the same targets.

1.2 Related Works

There are currently several parallelizing compilers that generate parallel MIMD code from a FORTRAN serial program without data partitioning declarations. Some of these parallelizers are PTRAN [1], Parafrase [45], ParaScope [38] and the work by Hudak & Abraham [36]. These compilers perform an extensive control and data dependency analysis in order to identify potential execution parallelism. For example, the PTRAN compiler builds the program data dependency and control dependence graphs in addition to performing constant propagation analysis, linear induction variable analysis and interprocedural analysis. It attempts to increase parallelism by introducing private copies of global variables. PTRAN generates a nested parallel code, which satisfies the data dependencies by control dependencies. These compilers translate serial FORTRAN programs to some dialect of parallel FORTRAN. However, most parallel FORTRAN dialects assume a shared memory architecture.

Gupta & Banerjee [29] describe an automatic data partitioning strategy for arrays in distributed memory multiprocessors. Their method chooses the data partitioning strategy that obeys the constraints computed from the source code. The work by Ramanujam & Sadayappan [46] describes a method for determining the optimal data partitioning strategy that minimizes communication in distributed memory multiprocessors.

Only a few parallelizing compilers emphasize portability, such as Parafrase-2 and the CSRD Delta system [31].

In contrast with these compilers, $PPFC$ emphasizes compiler and code portability in exchange for less extensive parallelization efforts. Although $PPFC$ employs a relatively simple data flow analysis comparing to current parallelizing compilers, it produces an efficient code for various widely used application programs.

None of the parallelizing compilers mentioned above was applied to parallelize Livermore Loops, or loops that contains WCSIS data dependency.

Several tools use execution time estimation to enhance code generation. PTRAN [1] uses a cost function and an estimation of the parallel and serial execution times in order to determine the most effective compile time partitioning of the program graph. Another tool to determine the program execution time and its variance using run time traces is described in [49].

The static performance estimator combined with ParaScope ([12], [38]) helps the programmer to choose the most efficient data distribution strategy by estimating the execution and communication times of the program. This estimator is similar to the $PPFC$. They both compute an estimation of the execution time using the program parse tree and a table containing the basic CPU operation times and communication times. The ParaScope performance estimator uses a "training set" of programs to compute the performance table. However, the ParaScope performance estimator is used only to aid the programmer and not as a part of an automatic tool for data partitioning, such as $PPFC$.

Our algorithm for estimating the execution time is given in section 6.1.

It is difficult to compare directly the efficiency of $PPFC$ generated code with other parallelizing compilers due to the different applications and target machines used. Some representative speedups of automatically generated parallel programs are described below. The 1990 Gordon Bell Prize winners in compiler parallelization achieved a speedup of 1,900 on a 2,048 floating point processors Connection Machine [18]. They used the KAP Fortran 77 to Fortran 90 preprocessor to generate efficient SIMD vector code from nested loops. The 1988 Gordon Bell Prize honorable mention achieved a speedup of 350 on a 1,024 processors N-Cube/10 distributed memory multiprocessor by running a financial modeling program parallelized by the Crystal compiler [13]. The code was written in the Crystal functional language.

All of the above results where obtained on multiprocessors with different architectures and much larger number of processors than the target machines of *PPFC*. Thus, it is difficult to compare these results directly with *PPFC* results.

1.3 Semi-Automatic Parallelizers

Many compilers perform semi-automatic data and process partitioning based on user directives or annotations. These compilers are mainly intended for generating code for distributed memory multiprocessors. Some of these compilers are High Performance Fortran[32], ParaScope [38], the work by Callahan & Kennedy [15], BLAZE [42], Pandore [2], Id Nouveau [47], Kali [39] and the knowledge based system [53]. In [40] there is a proposal for language extensions for data distribution conformant with both Fortran 77 and 90.

These compilers require the user to specify the data partitioning by special data distribution statements, such as `distribute` or `decompose`. In addition, the parallel loops must also be specified as `forall` loops. The generated code conforms to the *owner computes* rule, which means that the processor owning the data is responsible to update it. References to non-local data are replaced by message passing. Many of the compilers optimize the communication by message vectorization and pipelining [33]

Some of the tools generate SPMD (Single Program Multiple Data) code from serial programs, such as Pandore [2] and Kali [39].

There are some semi-automatic parallizers that do not require forall loops to parallelize a program such as FORTRAN D [33] and Vienna FORTRAN [16][54]. Oxygen [48] does not use the owner computes rule, and several other relax the rule in particular circumstances (such as reductions).

1.4 Comparison

PPFC differs from existing compilers that generate parallel MIMD code in a variety of ways:

It emphasizes portability: Current compilers extensively analyze control and data dependencies to identify potential areas for parallel execution, translating serial Fortran programs to parallel code. *PPFC* emphasizes compiler and code portability in exchange for less extensive parallelization efforts. Although *PPFC* employs a relatively simple data flow analysis comparing to current parallelizing compilers, it produces an efficient code for various widely used application programs. Most of the compilers are primarily for shared-memory architectures.

It is fully automatic: *PPFC* differs from other compilers by performing a fully automatic data partitioning from a serial program. In many compiler such as FORTRAN D [33] data and process partitioning are only semiautomatic. You must declare the parallel loops and the requested partitioning strategy, which the compilers use to generate the optimized code that will carry the computation on distributed-memory multiprocessors. Other methods use data-parallel languages like C^* to support data-parallel execution model, in which each data element in an array may compute its value concurrently with the other elements. *PPFC* requires no additional declarations in or modifications to the code. Another difference between sem-automatic and the *PPFC* is the degree of optimization. The semi-automatic systems generally perform much more compile-time optimization, such as loop transformations, than *PPFC*.

Its accurately estimates execution time: Several tools use runtime estimation to enhance code generation. *PPFC* is concerned with estimating exact execution time and takes into account many architecture compiler details [21, 25].

The rest of the paper is organized as follows: Section 2 describes the Fortran input language for *PPFC*, which is essentially standard Fortran-77 with minor extensions. Section 3 gives a short description of VMMP. Section 4 describes *PPFC* parallelization and data partitioning. *PPFC* attempts to parallelize the outermost `do` loop that is eligible for parallelization and its parallel execution time estimation is less than its serial execution time estimation. Parallel loops must contain only permitted access patterns to data. The permitted access patterns are implemented by corresponding parallel data distribution strategies. Section 5 compares between P^3C and *PPFC* implementations, and describes the work needed to convert the P^3C to *PPFC*. Section 6 describes the performance measurements of *PPFC* generated code. This section presents the performance results of *PPFC* generated code on shared memory and distributed memory multiprocessors.

Wide range applications that illustrates the operation of *PPFC* by presenting the source listings of serial application programs and its generated parallel code is given in [4].

2 The Fortran Language for *PPFC*

The *PPFC* input Fortran language, is a subset of Fortran-77 with minor extensions for scientific computation.

2.1 Added Features

The added features of *PPFC* Fortran are wrapped arrays, zero assignment to array and an `exit` statement.

2.1.1 Wrapped Arrays

Wrapped arrays allow access outside the declared bounds. Any access outside the array bounds is wrapped around and converted to an access inside the array bounds by a modulo operation. Wrapped arrays are especially useful in physical simulations of periodic domains.

Wrapped arrays are defined similar to regular arrays, except that their indices are enclosed by `square brackets` "[...]" instead of `regular brackets` "(...)". Note that the `square brackets` are used only in the declaration of the wrapped array and not in the usage of the array. For example:

```
REAL a[N,M]
```

Given the above definition, the access to the element `a(N+1, -2)` of this array is equivalent to `a(1, M-2)`. Generally the access to element `a(i,j)` of the array is equivalent to:

```
a(1 + mod(i-1,N) , 1 + mod(j-1,M)).
```

2.1.2 Zero Assignment to Array

Zero assignment to array variables is allowed, provided that the elements of the array are numeric. For example, if v is an array of integers then the assignment `v = 0` will clear all elements in v. The zero assignment is especially useful for array initialization.

2.1.3 Exit Statement

The `exit` statement is used to leave the enclosing loop. It is similar to the `exit` statement in Ada and `break` statement in C.

The following loop scans the elements of an array a and exits as soon as a zero element is found.

```
      DO 10 i = 1, N
          IF a(i) = 0 THEN
              pos = i
              exit
          END ID
10    CONTINUE
```

2.2 Unsupported Features

This section describes the declarations, statements and intrinsic functions of standard Fortran-77 which are not supported by the current implementation.

2.2.1 Unsupported Declarations and Statements

The current implementation does not supports the following declarations and statements which are supported by standard Fortran-77.

- `character` type.
- `complex` type.
- `data` declaration.
- `dimension` declaration. There are two possibilities to declare arrays in Fortran. The first is by simple variable declaration associated by it's indices list, for example:

  ```
  INTEGER  a(10)
  ```

  The second is by explicit dimension declaration, for example:

  ```
  INTEGER  a
  DIMENSION a(10)
  ```

  The current implementation supports only the first possibility.
- `equivalence` declaration.
- `named common` declaration which associates a block of shared storage to a list of variables. The current implementation supports only the `unnamed common` declaration.
- `format` statement.

The list of the unsupported intrinsic functions is detailed in [4].

3 Virtual Machine for MultiProcessors (VMMP)

The VMMP (Virtual Machine for MultiProcessors)[23, 24] is a software package, which provides a coherent set of services for explicitly parallel application programs running on diverse MIMD multiprocessors, both shared memory and message passing. It is intended to simplify parallel program writing and to promote portable and efficient programming. VMMP ensures high portability of application programs by implementing the same services on all target multiprocessors. It may also serve as a common intermediate language between very high level language compilers and various target multiprocessors.

VMMP provides high level services, such as the creation of cooperating set of processes (called *crowd computations* in VMMP), creation of a tree of lightweight processes (called *tree computations*), data distribution and collection, barrier synchronization, topology independent message passing, shared objects memory, etc. All of these services may be implemented efficiently on a wide variety of target machines. These high level services ensure that their implementation details can be hidden effectively from the user. VMMP does not provide low level services, such as semaphores, because they are difficult to implement efficiently on all target machines (e.g. on a message passing multiprocessor).

VMMP may be used directly by the application programmer as an abstract parallel machine to aid portability without sacrificing efficiency. VMMP may also be used as a common intermediate language between high level programming tools, such as parallelizers and specification language processors, and the target machines. In particular, VMMP is used as the intermediate language in P^3C and in the current implementation of $PPFC$, which translates serial programs in Pascal or Fortran to explicitly parallel code for a variety of target multiprocessors. VMMP usage is illustrated in figure 1.

VMMP supports two programming models: a *virtual tree machine* for *tree computations*, and a cooperating set (*crowd*) of processes for *crowd computations*.

Tree computations are a general computation strategy, which solves the problem by breaking it into several simpler subproblems, that are solved recursively. The solution process looks like a tree of computations, in which the data flows down from the root into the leaves and solutions flow up towards the root. Some examples to tree computations are divide and conquer algorithms, game tree search and combinatorial algorithms.

Crowd computations are used for algorithms that are defined in terms of a set of cooperating processes. Each of these processes is an independent computational entity, which communicates and synchronizes with its peers. The processes communicate and synchronize mainly by message passing. A crowd may create a tree computation and vice versa.

A basic tool used by both models is the *shared (memory) object*. Shared objects provides an asynchronous and flexible communication mechanism in addition to parameters and message passing, for all computations. Shared objects may be regarded as virtually shared memory areas, which contain any type of data, and can be accessed only by secure operations. Shared objects can be created, deleted, passed by reference, read and written.

The shared objects are intended to distribute data, collect results and collect global results by some associative and commutative operations. Other uses of the shared objects, such as random updates to locations of a large object, are discouraged, because the implementation of these operations incur high overhead on a message passing multiprocessor. The user must manipulate the shared objects only through VMMP services, which guarantee the correctness of the operations.

VMMP also supports several data distribution strategies using shared objects or by special crowd operations.

VMMP provides the abstract machine interface for the convenient use of tree computations, crowd computations and shared memory objects. As such, VMMP does not support base level services like semaphores and barriers. The use of its services ensures the efficient implementation on a wide variety of target machines, and the ability to effectively hide implementation details from the user or the application library developer.

The VMMP services may be used to create, synchronize and terminate crowd processes. These processes can communicate using message passing services, or through the use of the above-mentioned shared objects. Shared objects may be created, initialized, updated and deleted. VMMP supports several data distribution strategies by some crowd operations and by shared objects operations.

3.1 VMMP Implementation

VMMP has been implemented on the following multiprocessors:

Sequent Symmetry A 26 processors Sequent Symmetry running the DYNIX operating system.

MMX An experimental shared bus multiprocessor containing 4 processors running a simple run time executive.

ACE An experimental multiprocessor developed at IBM T.J. Watson Research Center running the MACH operating system.

MOS A shared memory multiprocessor containing 8 NS32532 processors running MOSIX, a distributed Unix.

Transputers A distributed memory multiprocessor consisting of a network of 8 T800 Transputers running the HELIOS distributed operating system.

VMMP simulator is available on SUN3, SUN4, IBM PC/RT and VME532 serial computers.

The same application program will run unchanged on all target machines after a recompilation.

VMMP was used in the implementation of many other problems, such as parallel generalized in frequency FFT (GFFT)[6], comparsion of the implementation on all the above machines[8], parallel solution of underwater acoustic IFD (Implicit Finite Difference) schemes[9], using multigrid to solve Poission equations on all the above target machines[7], solving Topelitz matrices[5], parallelizing BLAS 3 [10], and in a parallel debugger. All the above implementations were tested also on shared memory and message-passing multiprocessors.

The portable programming environment can be ported to a new multiprocessor in approximately 2-4 man months.

4 *PPFC* Parallelization and Data Partitioning

This section describes the *PPFC* parallelization algorithm.

The *PPFC* inherits from the P^3C all the parallel loop detection and data partition methods, and adds new techniques to enable parallelizing loops such as those that contain WCSIS data dependency.

PPFC attempts to parallelize the outermost `do` loops that are both eligible for parallelization and whose estimated parallel execution time is less than the serial execution time.

The loops eligible for parallelization are detected by classification of the loop variables according to their access patterns in the loop body. The access pattern determines the variable's partitioning strategy in the parallel code.

The rest of this chapter describe $PPFC$ computation model and parallel loop detection that already exists in the P^3C.

4.1 $PPFC$ Computation Model and Process Decomposition

$PPFC$ creates an explicitly parallel target code that executes in SPMD mode (Single Program Multiple Data) [19]. SPMD is a restricted form of parallel execution in which the same program runs on all processors. The parallel parts of the SPMD program are executed by all processors running the same code but manipulating distinct parts of the common workload. The initialization and termination parts of the program (outside all parallel loops) are executed by a single processor.

$PPFC$ produces SPMD code because it can be generated by relatively simple techniques from parallel loops. Although SPMD code is more restricted than a general parallel code, its performance is good for numerical algorithms that manipulate large arrays of data.

$PPFC$ creates one process for each physical processor in the multiprocessor. This process operates only on the data mapped to the processor by the data partitioning. The process may also access some private variables. Because each processor executes a single crowd process, the terms "processor" and "process" will be used interchangeably in the following chapters.

$PPFC$ reduces parallel code overhead by replicating the execution of the outer serial loop in all processors and distributing the work of the inner parallel loops between the processors. In this way, the outer replicated loop does not require any synchronization nor data movements, because all processors execute the same outer loop code on the same data and produce the same results. Synchronization is required only at the end of the parallel loops in order to exchange data between the processors and to combine partial results. The synchronization step ensures that the same results will be distributed to all processors. The structure of the generated code is illustrated in figure 2.

Parallel loops are divided among the processors, so that each processor executes approximately the same amount of work. Each processor executes a distinct subset of the loop iterations, which is chosen according to the data partitioning strategy of the loop variables.

The data exchange at the end of the parallel loop is a potential bottleneck. When the ratio of computation/communication increases (the size of the problem in each processor increases) then the scalability phenomenon and the achieved performance is getting better. Application of efficient concurrent algorithms enables to achieve good performance if most of the computational work is done within each processor and the communication and synchronization overheads is reduced. In numeric computations, in order to get good performance (speedup and scalability), we have to reduce the time needed for data exchange compared to the loop computation time.

$PPFC$ computation model may not scale up for a larger number of processors (> 100) as the result of the increased overhead of the data exchange step. A larger number of processors may also cause an uneven work distribution, which reduces computation efficiency.

4.2 Parallel Loop Detection and Data Partitioning

Parallel loop detection and data partitioning are closely coupled in $PPFC$. A parallel loop is recognized when all of the loop's variables belong to one of the permitted access categories described later in this section. The variable's access category dictates its partitioning strategy, which ensures that the parallel execution will compute the correct results. In other words, a parallel loop is

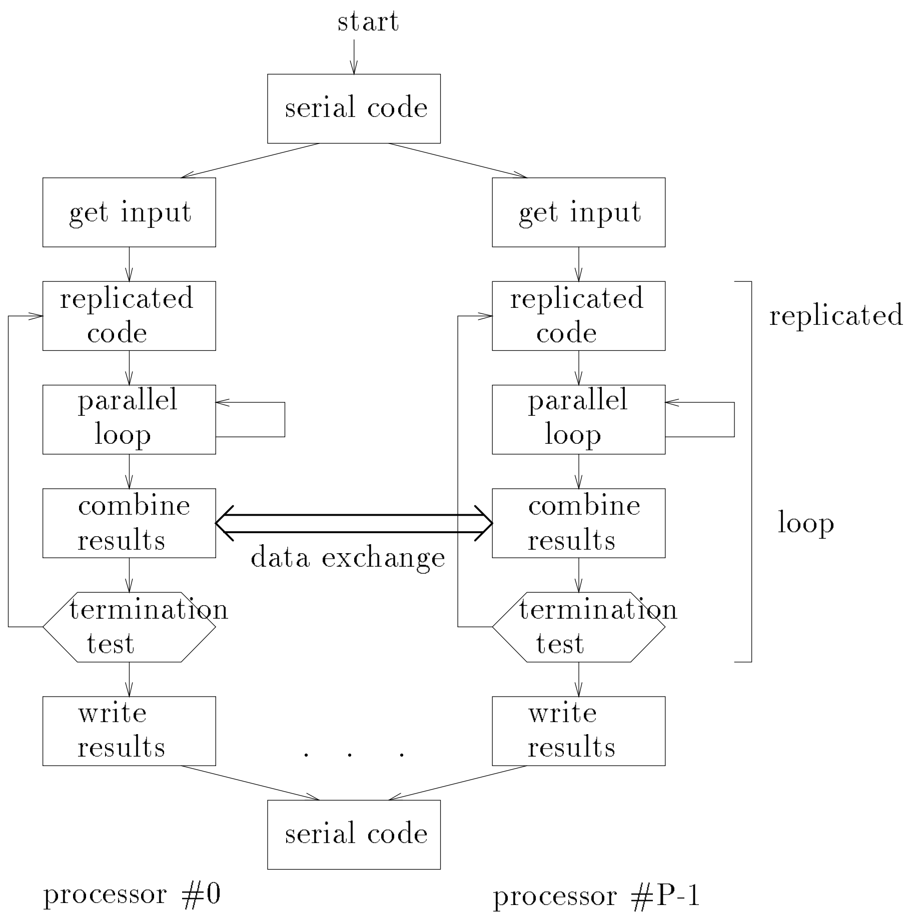


Figure 2: $PPFC$ generated code structure

recognized when all of its variables can be partitioned by one of the supported data partitioning strategies.

PPFC attempts to parallelize the outermost loops that are both eligible for parallelization and reduce total execution time. *PPFC* does not generate parallel code for a loop when the execution time estimation indicates that the parallel code execution time including the data exchange is larger than the serial code time. The loops that fail the parallelization criteria are considered serial. The `do while` loops are also considered serial. *PPFC* also recognizes a parallel loop nested inside a serial loop.

The data partitioning strategies supported by *PPFC* are illustrated in figure 3. This figure illustrates the partitioning on four processors numbered P#0 – P#3.

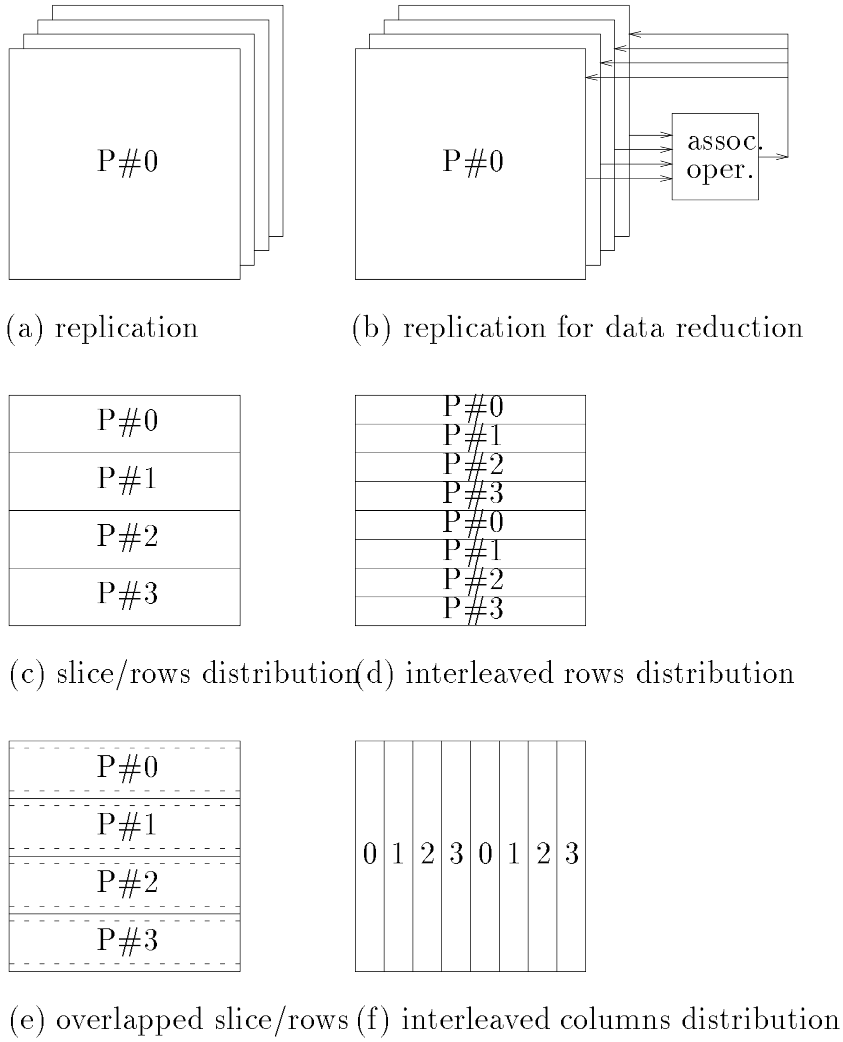


Figure 3: Supported Data Partitioning Strategies

The distribution strategy of a variable inside the replicated loop must be the same for all enclosed parallel loops. Otherwise, the variable data must be re-distributed repeatedly inside the replicated loop, which causes a significant performance penalty when the data size is large.

PPFC attempts to balance the workload among the processors by a suitable allocation of the parallel loop indices. Each processor works only on the data allocated to it. The body of the parallel loop is modified to access the data allocated to the processor. *PPFC* inserts special code at the end of the loops in order to distribute common results.

4.2.1 Permitted Access Categories

A parallel loop is recognized if and only if all of the variables accessed inside the loop body belong to one of the following access categories. These categories do not introduce loop dependencies and are eligible for efficient data distribution. Each of the access categories is implemented in the parallel code by a corresponding data distribution strategy.

The permitted access categories and their associated data distribution strategies are described in the following sections. The access categories are recognized by a combination of data flow analysis and pattern matching. All other access patterns are considered to cause dependencies between loop iterations and inhibit loop parallelization.

Read only variables are used inside the loop but not changed. They are implemented by replication among the processors. Implementing read-only variables by replication can cause problems in achieving scalable computation and has the potential to use incredible amounts of memory. See figure 3-a.

The value of a local variable is used only in the same iteration which had set it. A private variable does not cause any dependency between iterations, because its value is not passed between iterations. For example:

```
DO i = 1, N
    v = expression
    ... v is used here ...
END DO
```

Local variables are partitioned by replication. See figure 3-a.

Loop index variables are changed only by the loop control statement and not inside the loop body. They are implemented by replication, but each processor executes only a subset of the loop iterations according to the index allocation scheme.

Distributed variables are arrays that are changed in the loop, so that each iteration changes a different array element. This array element may be used later in the same iteration. For example:

```
DO i = 1, N
    v(i) = expression
    ..... v(i)  used here
END DO
```

The current implementation identifies distributed arrays only by their first index, which must be a loop index variable.

Distributed variables are partitioned either by rows distribution (figure 3-c) or by interleaved rows distribution (figure 3-d). The particular distribution strategy is chosen according to the estimation of the loop body execution time. One dimensional arrays are partitioned by slice and interleaved distributions, respectively, which are equivalent to the two dimensional partitioning. Section 4.4 describes the determination of the partitioning strategy.

Grids

Grids are a special case of *distributed* variables. Each iteration of the loop computes a single element of the grid using a combination of corresponding elements from a similar grid. The newly computed grid is later assigned into the old grid. The grid is recognized only when the first index of all grid accesses in the loop are of the form $i \pm c$, where i is the loop index and c is a constant. c may be omitted.

The following example illustrates grid access.

```
real  x(N), new_x(N)
....
DO WHILE (.NOT. converged)
   DO i = 2, N-1
      new_x(i) = (x(i-1)+x(i+1))/2
   END DO
   x = new_x    Grid assignment
END DO
```

In this example, both `x` and `new_x` are grids and have the same distribution strategy, because the grid assignment statement requires that both variables should have the same distribution strategy. This example illustrates the code for the solution of a one dimensional wave equation PDE.

Grids are partitioned by overlapped rows distribution. See figure 3-e. Each processor is allocated a slice of the grid with overlapping boundary elements. The processor computes the new value of its local slice and exchanges the boundary elements with its neighbors at the end of each loop. The amount of overlap between the processors is $\max(c_1, c_2, \ldots, c_n)$, where $c_1, c_2, \ldots, c_n$ are the offsets from the loop index used in the grid element computation. For example, the amount of overlap in the last example is one element.

The boundary overlap must be greater than zero. A zero overlap indicates that this grid is actually a *distributed* variable.

Data Reduction

A data reduction variable is updated inside the loop by an assignment statement of the form $v = v\ op\ expr$, where *op* is an associative and commutative operation, such as `+`, `*`, `.and.`, `.or.`, or $v = func(v, expr)$, where *func* is `max` or `min`. The value of the data reduction variables is not used elsewhere inside the loop except in its update operation. For example:

```
DO i = 1, N
    sum = sum + expression
END DO
```

The final value of data reduction variables may be computed at the end of all loop iterations by combining the partial values supplied by the participating processors. We ignore the effect of rounding errors, which may produce different answers depending on the evaluation order.

The operation *op* or the function *func* must be both associative and commutative, because the parallel code computes the final result by a combination of the partial results. The partial results may be combined in any order as the result of the parallel execution.

Data reduction variables are partitioned by replication, and the partial results computed by each processor are combined at the loop termination to produce the final value. See figure 3-b.

The data reduction variable may also be an array, but in this case both the left hand side and the right hand side index expressions must be the same. For example:

```
DO i = 1, N
```

```
    pos = particle_x(i)
    cell(pos) = cell(pos) + particle_charge(i)
END DO
```

In this example the `cell` array is *data reduction* and the arrays `particle_x` and `particle_charge` are *distributed*. The example illustrates the code of a particle in cell (PIC) algorithm [35], which simulates the movement of charged particles.

A data reduction array, whose index expressions consists of the loop index variable, is considered *distributed* and not *data reduction*. The reason is that the implementation of distributed variables is more efficient than data reduction variables. For example:

```
DO i = 1, N
    sum(i) = sum(i) + expression
END DO
```

Interleaved Columns Array

Interleaved columns arrays are a special case of two dimensional arrays, which are accessed along a single column and modified only by matrix row operation, which are described below. Moreover, the column and row access loops must be enclosed inside a serial loop, whose index variable specifies the access column. These restricted access patterns regularly appear in linear algebra algorithms.

The row access loops must match one of the following templates, in which row, row_1 and row_2 are the row indices and s is a scalar variable.

- Computing a linear combination of row_1 and row_2:

  DO $\quad i = low\ ,\ high$
  $\qquad a(row_1, i) = a(row_1, i) \pm s * a(row_2, i)$
  ENDDO

- Scaling row by the scalar s:

  DO $\quad i = low\ ,\ high$
  $\qquad a(row, i) = a(row, i) * s$
  ENDDO

- Exchanging row_1 with row_2:

  DO $\quad i := low\ ,\ high$
  $\qquad s = a(row_1, i)$
  $\qquad a(row_1, i) = a(row_2, i)$
  $\qquad a(row_2, i) = s$
  ENDDO

Code 4 illustrates interleaved column access to the array `a`. It is a part of a program for performing Gaussian elimination. The matrix elements below the diagonal are eliminated by row operations.

```
      REAL     a(N, N)
      INTEGER  i, j, k, pivot
      REAL     scale, t

      DO i = 1, N
C         Find pivot row - serial loop
          pivot = i
 10       DO j = i+1, N
              IF (abs(a(j,i)) .GT. abs(a(pivot,i)))  pivot = j
          END DO

C         Row exchange
          DO k = i, N
              t = a(i,k)
              a(i,k) = a(pivot,k)
              a(pivot,k) = t
          END DO

C         Pivot row scaling
          scale = 1/a(i,i)
          DO k = i, N
              a(i,k) = a(i,k)*scale
          END DO

C         Elimination
          DO j = i+1, N
              scale = a(j,i)
C             row operation in a parallel loop
              DO k = i+1, N
                  a(j,k) = a(j,k)-scale*a(i,k)
              END DO
          END DO
      END DO
```

Figure 4: Interleaved Columns Access

In this example, the loop at label 10 performs column access on column number `i`, which is the index variable of the enclosing serial loop. This column will be called "the current column" in the following discussion. Each of the row exchange, pivot row scaling and elimination loops match a permitted template, so they are eligible for parallelization.

Interleaved column arrays are partitioned by interleaved column distribution. See figure 3-f. The current column is broadcasted at the beginning of each iteration of the enclosing serial loop from processor ($i \bmod \#P$) to all other processors. The loops that access the rows are modified to access only the elements allocated to the processor. The column access code is not modified, because the current column is available in all processors after the broadcast.

4.3 Inhibited Parallel Loops

The parallel loop may call procedures and functions only when these routines have no side effects. In other words, these routines should only access their parameters and no global data. *PPFC* identifies the procedures and functions that may potentially cause side effects by a separate data dependence analysis prior to the analysis of the main program body.

The random number generator functions that use the internal seed may called in the parallel loop (e.g. `rand`, `irand` and `rnor` functions). If the loop is parallelized, the parallel execution may produce a different sequence of random numbers than in the serial execution, because each processor is using an independent seed. If the sequence of random numbers should be repeated in the parallel execution, then the random number generator must be called with an explicit seed (e.g. `rand2`, `irand2` and `rnor2` functions). When an explicit seed is specified, the call to the random number generator inhibits the parallelization, because it creates a data dependency on the seed.

An input or output statement inhibits the parallelization of the enclosing loop. An input statement is permitted only outside the replicated code. An output statement is allowed in the replicated code. In this case, only a single processor will execute the output statement, because all processors have the same data values in the replicated loop.

The `clock` routine, which returns the current time, also inhibits the loop parallelization, because it may return different results in a concurrent execution on different processes. `Clock` may be called only outside the replicated loop.

The `exit`, `stop`, `goto` and `computed goto` statements also inhibits the parallelization, because they may cause a premature termination of the loop.

4.4 Index Allocation in Parallel Loops

PPFC generates either a consecutive or interleaved index allocation for parallel loops according to the loop body execution time and the variables access patterns.

The index allocation schemes are illustrated by the following generic loop:

```
DO i = low, high
    ... body
END DO
```

Consecutive index allocation denotes that the processor executes a slice of consecutive loop iterations. Processor i is allocated iteration numbers from $i \times \lfloor \frac{N}{\#P} \rfloor$ until iteration number $(i+1) \times \lfloor \frac{N}{\#P} \rfloor - 1$, where N is the total number of iterations, $\#P$ is the number of processors and the processor number i is in the range $0 \leq i < \#P$. In the above loop, N is $high - low + 1$. The actual index value is $low + \#iter \times step$, where $\#iter$ is the iteration number in the above range.

Interleaved index allocation denotes that the processor i is allocated non-consecutive loop iterations. Processor i is allocated iteration numbers i, $i + \#P$, $i + 2 \times \#P$, ... The last iteration number is less than N.

The partitioning strategy of the loop variables must be compatible with the index allocation scheme. Certain partitioning strategies are limited to specific index allocation schemes. The *grid* access pattern requires consecutive index allocation, because each processor is assigned a consecutive slice of elements with overlapping boundaries. The interleaved index allocation can not support the required locality of reference inside the slice. The *interleaved columns* access pattern requires interleaved index allocation, because each processor is assigned interleaved columns of the array. All other access patterns are supported in both index allocation schemes.

Consecutive index allocation is generated when the loop body execution time is constant or when the loop contains grids. The *distributed* access pattern is implemented by row distribution in this allocation.

Interleaved index allocation is generated when the loop body execution time is expected to vary from iteration to iteration. It is detected when the loop body contains interior `do while` or `do` loops with unknown number of iterations. It is also detected when the loop body contains calls to procedures and functions, whose execution time may vary. Interleaved allocation is generated also when the loop performs row operations on an interleaved columns array. Interleaved index allocation is the default index allocation when the loop does not access arrays. The *distributed* access pattern is implemented by interleaved distribution in this allocation.

The interleaved index allocation improves the load balancing between the processors when the amount of work in each iteration may vary and the number of iterations is large. Interleaved allocation is also called "wrapped allocation" [43] or vertical allocation.

The index allocation scheme is sufficient to ensure that the parallel loop code accesses only data elements allocated to the processor. Some other statements, such as grid assignments and loops performing row access to interleaved arrays, are modified to fit the data distribution.

4.5 Weak Crossing Signal Index Subscripts (WCSIS)

Loops that contains WCSIS ([14],[27]) data dependency have the following general structure:

DO $\quad i = low \ , \ high$

$\qquad A(c * i) = A(high - c * i + low) \ op \ exp$

END DO

where op and exp are any eligible operation and expression respectively in Fortran language, A is a one dimensional array and c is a constant.

This loop cause cross data dependency, i.e. dependence exists between the definition (writing) and the usage (reading) of array A, when crossing iteration $(high + low)/2$. For instance, consider the loop:

```
DO i = 1, 10
    A(i) = A(11 - i)
END DO
```

At iteration 6, $A(6)$ is defined and $A(5)$ is used, but $A(5)$ was defined at iteration 5, therefore dependence exists between iterations 5 and 6. In the same way we can prove that iteration exists between iterations 4 and 7, and between iterations 3 and 8, etc.

4.5.1 Splitting the WCSIS Loop

These dependencies may be eliminated by splitting the loop at iteration $(high + low)/2$ into two loops as follows:

DO $\quad i = low \ , \ (high + low) \ / \ 2$

$\qquad A(c * i) = A(high - c * i + low) \ op \ exp$

END DO

DO $\quad i = (high + low) \ / \ 2 + 1 \ , \ high$

$\qquad A(c * i) = A(high - c * i + low) \ op \ exp$

END DO

each of these loops contains no data dependency, and therefore can be parallelized.

The first loop contains no data dependency since it defines in each iteration a different element from $A(low)$ to $A((low + high)/2)$, and uses in each iteration different element from $A((low + high)/2)$ to $A(high)$. The second loop contains no data dependency since it defines in each iteration different element from $A((low + high)/2)$ to $A(high)$, and uses in each iteration different element from $A(low)$ to $A((low + high)/2)$. Note that the second loop must not begin before the total end of the first loop.

4.5.2 How *PPFC* deal with WCSIS Loop

When the *PPFC* recognizes loop that contains WCSIS, it performs `loop splitting` transformation, and then tries to parallelize each of the resulting loops. The loops will not be parallelized if their estimated parallel execution time is more than the serial execution time.

The array `A` is partitioned by slice distribution (figure 3-c), i.e. each process gets two distinct slices: one for definition and the other for usage.

4.6 Independent Iterations

The permitted access categories ensure that the loop iterations are independent or contains only loop carried dependencies caused by data reductions.

The *read only*, *index* and *local* access categories clearly do not cause any dependencies between loop iterations. The *distributed* category does not cause any dependencies between iterations, because each iteration access and modify a different array element.

The *grid* category is a special case of *distributed*, so it also does not cause any dependencies. Note that the new grid values are written on a separate array than the original grid, which is *read only* in the loop.

The *data reduction* category causes a loop carried dependence. However, the operation can still be performed in parallel, because the data reduction variable is used only in update statements of the form $v = v \;\; op \; expr$ or $v = func(v, expr)$.

The *interleaved column* category is more subtle. The column access does not cause any dependency, because this is a read operation and the current column is duplicated among the processors at the start of each iteration of the enclosing serial loop. The matrix row operations either access a single element from one row or two elements from the same column in two rows. In both cases the loop iterations can be performed in parallel because the array is distributed by columns. Each processor can either access the complete column or none at all.

In the WCSIS category, after the `loop splitting` transformation, each of the resulting loops does not cause any dependency, since the definition and usage of array `A`, are performed on two distinct parts of the array.

The *PPFC* loop parallelization is much simpler than corresponding algorithms in other parallelizing compilers, such as PTRAN [1] or Parafrase [45].

4.7 Parallelization Example

The loop parallelization and data partitioning are illustrated by code 5, which is taken from a program for the solution of a system of n linear equations in n unknowns $Ax = b$ by Jordan iterations. The matrix A contains the coefficients, x is the solutions vector and b is the right hand side coefficients vector. Each iteration computes a better approximation to the solution vector x by computing: $new\_x_i = (b_i - \sum_{j \neq i} A_{i,j} x_j)/A_{i,i}$

The nested loop structure in the above example is typical for scientific code. The outer `do while` loop is inherently serial, while the inner `do` loops are parallel.

The `do` loop at label 10 is eligible for parallelization, because all of its variables belong to one of the permitted categories: `i` is *index*, `sum` and `j` are *local*, `a`, `b` and `x` are *read only*, `new_x` is *distributed* and `error` is *data reduction*.

The parallel code will be generated only when `N` is large enough to make the parallel execution time estimation less than the serial execution time.

```
      REAL  a(N,N), b(N), x(N), new_x(N)
      ...
      { initialization code }
      error = 1
      DO WHILE (error .GE. EPS)
         error = 0
10       DO i = 1, N
            sum = 0
            DO j = 1, N
               sum = sum + a(i,j)*x(j)
            END DO
            new_x(i) = x(i)+(b(i)-sum)/a(i,i)
            error = error MAX abs(b(i)-sum)
         END DO
         x = new_x
      END DO
      ...
      { termination code}
```

Figure 5: Parallelization Example

5 *PPFC* Implementation

PPFC consists of the same four passes that are used in the P^3C:

Front End - accepts the Fortran language input and performs the syntax analysis.

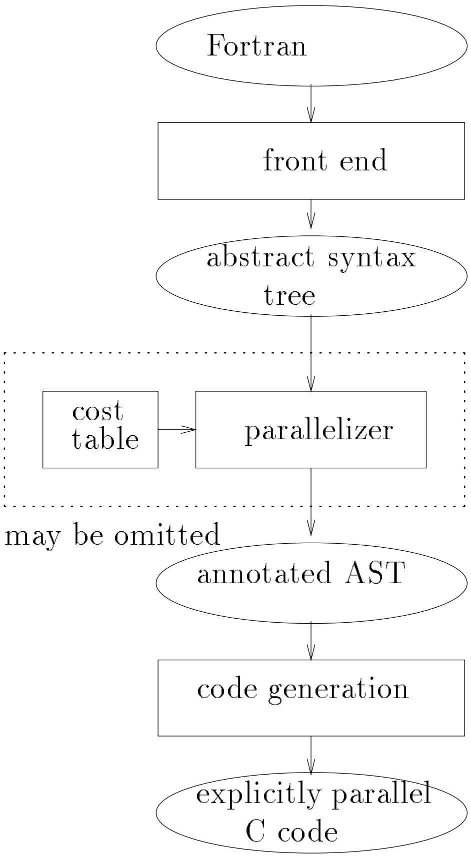
Semantic Check - accepts the program parse tree and performs the semantic analysis.

Parallelizer - accepts the program in the form of a typed parse tree and converts it into an optimized explicitly parallel program.

Code Generation - accepts a program in the form of a typed parse tree and emits the equivalent C code.

Like the P^3C, the *PPFC* front end was implemented in a combination of YACC, and LEX tools, which are the standard Unix compiler development tools. The rest of *PPFC* was implemented in Prolog, because Prolog can express complicated symbolic data manipulations clearly and concisely. Prolog also offers the flexibility needed for experiments. In retrospect, the decision to use Prolog was fully justified.

Figure 6 illustrates *PPFC* internal structure. The rectangles represent the compiler passes and the ovals represent intermediate data files. Note that the parallelizer pass is optional. If it is omitted, *PPFC* will translate Fortran to serial C without modifications. The output of *PPFC* is combined with VMMP run-time library to form an executable code, as shown in figure 1.



æ

Figure 6: *PPFC* components

5.1 Differences between implementations of $PPFC$ and P^3C

The major differences between P^3C and $PPFC$ are in the front end and semantic check passes. The parallelizer and the code generator passes of $PPFC$ preserves all the features of the P^3C, and adds more features.

The rest of this sections describes in details the changes required in each pass.

5.1.1 Front End Pass changes

The $PPFC$ front end had to be written from scratch in order to accept the Fortran language. The structure of the parse tree generated by $PPFC$ front end is identical to the structure of the parse tree generated by P^3C front end, but it contains different terms.

For example Code 7 illustrates Fortran code excerpt which is input for the $PPFC$ front end pass, and Code **??** illustrates the parse tree generated by the $PPFC$ front end pass. Note that each of the terms contains the corresponding source line number.

```
      REAL  a, b
      REAL  c(10)
      INTEGER i

      DO 10 i = 1, 10
          READ ,a,b
          IF (a .GT. b)  THEN
              c(i) = a - b
          ELSE
              c(i) = b - a
          END IF
 10   CONTINUE
      END
```

Figure 7: Fortran example

5.1.2 Semantic Check Pass Changes

The $PPFC$ semantic check pass required a radical change, since it's input parse tree contains different terms, but it's concept remains the same as of the P^3C semantic check pass. It accepts the program parse tree and performs all semantic checks, and it also builds the program symbols table, which contains the type of each declared variable.

The parse tree generated by $PPFC$ semantic check includes the parse tree generated by P^3C semantic check, but it also contains additional terms. These additional terms are due to statements and sentences in Fortran that don't have meaning in TAU-Pascal like: `common`, `goto`, `computed goto`, `stop`, `continue`, etc.

5.1.3 Parallelizer Pass Changes

The $PPFC$ parallelizer preserves all the features of P^3C parallelizer and adds some more features.

The added features are:

- Recognizing the WCSIS loop (see section 4.5), splitting this loop into two loops that contains no data dependency, and then parallelizing each of the resulting loops.
- Recognizing sentences in Fortran that inhibits parallelization like: `goto`, `computed goto`, `stop`, `continue exit`. If one of these sentences exists in a loop, then this loop cannot be parallelized since they may cause a premature termination of the loop.

5.1.4 Code Generation Pass Changes

The $PPFC$ code generator preserves all the features of P^3C code generator and adds some more features.

Some of the added features are:

- Generating the C `goto` statement which implements the Fortran `goto` statement.
- Generating the C `switch` statement which implements the Fortran `computed goto` statement.
- Generating the C `label` declaration which implements the Fortran `label` declaration.
- Generating the C `exit(0)` statement which implements the Fortran `stop` statement.
- Generating the C types `double` and `float` which implements Fortran types `double precision` and `real`, respectively.
- Generating C `global variables` declaration which implements Fortran `common` declared variables.

6 Performance Measurements

This section describes the performance measurements of $PPFC$ generated code.

The $PPFC$ was applied to parallelize the Livermore Loops [41], which are widely used as a benchmark of numerical calculations in Fortran, and succeeded to parallelize 12 out of 24 loops. It was also applied to parallelize all the 14 application programs.

These examples compiled by $PPFC$ and run on the following four target machines:

- $MOSIX$ shared memory multiprocessors, contains 8 processors.
- $SequentSymmetry$ shared memory multiprocessors, contains 26 processors (we used 25).
- $Transputers$ distributed memory multiprocessor (a network of T800 Transputers), contains 8 Transputers.
- $SUN3/50$ and $SUN4/390$ workstations uniprocessors.

The speedup is computed relatively to the serial execution time, which is obtained by running the C code generated by $PPFC$ without the parallelization step. The serial C code is the direct translation of the Fortran serial program.

All floating point computations were performed in double precision.

The same programs were measured on all targets. The programs' data size was limited by the computer with the smallest memory, which is the Transputers network.

6.1 Estimate Excution Time

The parallelizer estimates execution time by examining the program's parse tree and consulting a table of target machine's parameters[25]. The parameters table contains the execution time of all primitive constructs appearing in serial program, such as arrithemtic operations, array indexing, data movement, and various overhead (if, looptest, and so on).

Execution time is expressed relative to the time needed for an integer addition. The parameters table also contains the coefficients of the functions of VMMP services. Execution time is computed by a bottom-up traversal of the program parse tree. The execution time of a straight code sequence is the sum of the execution times of all primitive operations. The execution time of an if statement is the condition execution time plus the weighted average of execution times for the then and else parts.

More detailed explanations are given in [21, 25]. All the performance measurements in section 6 which are summerized in tables 1-6 indicate that the speedup estimation on all the targets is very close to the actual speedup. In the case of the *livermore loops* it is within 2-4% of the measured speedup.

6.2 Livermore Loops Performance

The Livermore Loops [41] are widely used as a benchmark of numerical calculations in Fortran, especially those involving vector operations, and their results are available for a wide range of machines.

Unfortunately, most familiar Fortran parallelizing compilers like: PTRAN [1], Parafrase [45], the work by Hudak & Abraham [36], the work by Gupta & Banerjee [29], the work by Ramanujam & Sadayappan [46], and the work by Callahan & Kennedy [15] didn't use the Livermore Loops to measure theirs ability and performance.

There are 24 Livermore Loops. 9 loops were parallelized by the $PPFC$ without any modification. 3 loops required manual modification in order be parallelized by the $PPFC$. The other 12 loops cannot be parallelized by $PPFC$.

See [4] for the complete source code and the parallel code generated for each of these loops.

6.2.1 Parallelized by $PPFC$ without modification of Livermore Loops

The $PPFC$ succeeded to parallelize 9 Livermore Loops without any manual modification except of preceding code to initial the arrays used by the loop, by random numbers generator.

Code 8 illustrates classic Livermore Loop that can be parallelized by the $PPFC$ without any modification. This code contains 4 nested loops. Each of these nested loops is parallel by it's own, since it does not cause data dependencies between iterations. The $PPFC$ will parallelize the l loop at label 1 since this loop is the outermost parallel loop.

H

```
        start = Vclock();
        Vcrowd(... task1 ...);
        Vwait();

     task1()
     {
      Vcrowd_part(...low1,high1,...);
      for(l = low1+1; l <= (high1+1); l++){
              for(i = 0; i <= 19; i++){
                      for(k = 0; k <= 19; k++){
                              for(j = 0; j <= 19; j++){
                                      px[i][j] = px[i][j]+(vy[i][k]*cx[k][j]);
                                }
                        }
                }
        }
     }
```

Figure 9: Livermore Loop 21: the parallelized C code generated by *PPFC*. The code that is running in each processor

```
 1      DO l = 1, N
            DO i = 1, M
                DO k = 1, M
                    DO j = 1, M
                        px(i,j)= px(i,j) +vy(i,k) * cx(k,j)
                    END DO
                END DO
            END DO
        END DO
```

Figure 8: Livermore Loop 21: parallelized by *PPFC* without modification

6.2.2 Parallelized by *PPFC* with modification of Livermore Loops

Livermore Loops 8, 9 and 10 required the following manual modification in order to be parallelized by the *PPFC*.

In Livermore Loop 8, a three dimensional array was replaced by several two dimensional arrays.

In Livermore Loops 9 and 10, a two dimensional array was replaced by several one dimensional arrays.

All these modifications required since the current implementation of the *PPFC* recognizes distributed arrays only by their first index, which must be the loop index variable.

For example Code 10 and Code 11 illustrates the original and modified code, respectively, of Livermore Loop 10. Since the *PPFC* recognizes distributed arrays only by their first index, which must be the loop index variable, it will not recognize *px* array of the original Livermore Loop 10 as distributed array. Hence this loop cannot be parallelized by *PPFC*. In the modified code of Livermore Loop 10, *px* array was replaced by several one dimensional arrays i.e. $px4$, $px5$, etc. Each of these one dimensional arrays will be recognized by the *PPFC* as distributed array, and hence this loop will be parallelized by the *PPFC*.

```
DO l = 1, N
    DO i = 1, M
        ar      =      px(4,i)
        br      = ar - px(4,i)
        px(4,i) = ar
        cr      = br - px(5,i)
        px(5,i) = br
        ar      = cr - px(6,i)
        px(6,i) = cr
        br      = ar - px(7,i)
        px(7,i) = ar
        cr      = br - px(8,i)
        px(8,i) = br
        ar      = cr - px(9,i)
        px(9,i) = cr
        br      = ar - px(10,i)
        px(10,i)= ar
        cr      = br - px(11,i)
        px(11,i)= br
        px(13,i)= cr - px(12,i)
        px(12,i)= cr
    END DO
END DO
```

Figure 10: Original code of Livermore Loop 10 - cannot be parallelized by *PPFC*

```
DO l = 1, N
    DO i = 1, M
        ar      =      px4(i)
        br      = ar - px4(i)
        px4(i) = ar
        cr      = br - px5(i)
        px5(i) = br
        ar      = cr - px6(i)
        px6(i) = cr
        br      = ar - px7(i)
        px7(i) = ar
        cr      = br - px8(i)
        px8(i) = br
        ar      = cr - px9(i)
        px9(i) = cr
        br      = ar - px10(i)
        px10(i)= ar
        cr      = br - px11(i)
        px11(i)= br
        px13(i)= cr - px12(i)
        px12(i)= cr
    END DO
END DO
```

Figure 11: Livermore Loop 10 after modification - can be parallelized by $PPFC$

6.2.3 Livermore Loop which is not parallelized by $PPFC$

The other 12 Livermore Loops cannot be parallelized by $PPFC$, since the data partition in these loops are not supported by the $VMMP$. Each of these loops contains at least one array which is not corresponding to any of the permitted access categories mentioned in 4.2.1.

Most of these loops contain the following array access example:

```
DO i = 1, N
    k = ix(i)
    v(k) = expression
    ..... v(k)  used here
END DO
```

In this example the array v is not corresponding to any of the permitted access categories used by $PPFC$.

Code 12 illustrates the serial code of Livermore Loop 13. This loop cannot be parallelized by the $PPFC$, since the arrays `e`, `f`, `h`, `y` and `z`, are not corresponding to any of the permitted access categories mentioned in 4.2.1.

```
DO l = 1, N
    DO ip = 1, M
        i1 = p(0,ip)
        j1 = p(1,ip)
        i1 = mod2n(i1,64)
        j1 = mod2n(j1,64)
        p(2,ip) = p(2,ip) + b(i1,j1)
        p(3,ip) = p(3,ip) + c(i1,j1)
        p(0,ip) = p(0,ip) + p(2,ip)
        p(1,ip) = p(1,ip) + p(3,ip)
        i2 = p(0,ip)
        j2 = p(1,ip)
        i2 = mod2n(i2,64) - 1
        j2 = mod2n(j2,64) - 1
        p(0,ip) = p(0,ip) + y(i2+32)
        p(1,ip) = p(1,ip) + z(j2+32)
        i2 = i2 + e(i2+32)
        j2 = j2 + f(j2+32)
        h(i2,j2) = h(i2,j2) + 1.0
    END DO
END DO
```

Figure 12: Livermore Loop 13 which is Not Parallelized by the PPFC

6.2.4 Livermore Loops Results

Tables 1, 2 and 3 contain the speedup measurements and estimated speedup of 12 Livermore Loops, parallelized by $PPFC$, on the Sequent Symmetry, MOSIX and the Transputers network, respectively.

The tables present the following information for each program:

Elapsed The elapsed time is in seconds. It includes all data distribution and collection overheads of the parallel code.

Speedup The speedup of the parallel code is computed by: $speedup = \frac{serial\ time}{parallel\ time}$. The speedup is computed relative to the serial program execution time. The serial program is the direct translation of the Fortran program to C, which is obtained by running $PPFC$ without the parallelization pass.

Eff.% The efficiency is computed by the formula: $efficiency = \frac{speedup \times 100}{\#processors}$.

E. Spd. The estimated speedup. It is computed by the time estimation algorithm of the $PPFC$.

6.2.5 Livermore Loops Observations

11 out of 12 Livermore Loops, parallelized by $PPFC$, achieved a very high speedup and efficiency on all targets.

Some of them even achieved efficiency of more then 100% on the Sequent and the MOSIX. The reason for the super linear efficiency is that these computers contain cache memory for each

Program	Data	Sequent Processors							
		serial	1	2	5	10	15	20	25
Loop1	Elapsed	521.38	577.18	289.94	116.51	58.83	39.22	29.57	23.25
	Speedup		0.90	1.80	4.47	8.86	13.30	17.63	22.42
	Eff.%		90.33%	89.91%	89.50%	88.63%	88.63%	88.16%	89.69%
	E. Spd.			1.94	4.85	9.52	14.12	18.94	24.13
Loop3	Elapsed	371.77	384.04	189.64	75.62	38.54	25.73	19.56	15.86
	Speedup		0.97	1.96	4.92	9.65	14.45	19.00	23.44
	Eff.%		96.80%	98.02%	98.32%	96.47%	96.32%	95.02%	93.76%
	E. Spd.			1.90	4.77	9.51	14.92	19.46	24.32
Loop7	Elapsed	360.67	413.59	206.38	82.53	41.46	27.87	21.33	17.10
	Speedup		0.87	1.75	4.37	8.70	12.94	16.91	21.09
	Eff.%		87.20%	87.38%	87.41%	86.99%	86.26%	84.54%	84.37%
	E. Spd.			1.94	4.85	9.04	14.35	18.24	23.74
Loop8	Elapsed	371.53	319.72	159.61	64.15	32.36	21.92	16.51	14.46
	Speedup		1.16	2.33	5.79	11.48	16.95	22.50	25.69
	Eff.%		116.20%	116.39%	115.83%	114.80%	113.00%	112.48%	102.78%
	E. Spd.			1.98	4.96	9.85	14.91	19.96	24.93
Loop9	Elapsed	539.63	590.89	295.31	118.61	59.55	39.98	30.36	23.91
	Speedup		0.91	1.83	4.55	9.06	13.50	17.77	22.56
	Eff.%		91.32%	91.37%	90.99%	90.62%	89.99%	88.87%	90.02%
	E. Spd.			1.95	4.84	9.26	14.11	18.98	24.01
Loop10	Elapsed	743.10	749.29	366.76	140.44	71.43	46.76	35.21	28.27
	Speedup		0.99	2.03	5.29	10.40	15.89	21.11	26.29
	Eff.%		99.17%	101.31%	105.82%	104.03%	105.95%	105.53%	105.14%
	E. Spd.			1.99	4.99	9.98	14.99	19.99	24.97
Loop11	Elapsed	345.08	405.62	208.89	83.29	42.05	28.13	21.28	16.62
	Speedup		0.85	1.65	4.14	8.21	12.27	16.21	20.07
	Eff.%		85.07%	82.60%	82.86%	82.06%	81.79%	81.07%	83.05%
	E. Spd.			1.94	4.85	9.11	13.14	18.64	22.45

Table 1: Measurements of $PPFC$ generated code for Livermore Loops on the Sequent Symmetry

Program	Data	Sequent Processors							
		serial	1	2	5	10	15	20	25
Loop12	Elapsed	350.45	409.06	204.66	82.12	41.90	28.64	21.42	16.32
	Speedup		0.86	1.71	4.27	8.36	12.24	16.36	21.47
	Eff.%		85.67%	85.62%	85.35%	83.63%	81.58%	81.82%	85.89%
	E. Spd.			1.88	4.72	9.14	13.99	18.32	23.81
Loop18	Elapsed	61.63	67.21	52.47	45.91	38.30	32.59	29.01	25.18
	Speedup		0.92	1.17	1.34	1.61	1.89	2.12	2.45
	Eff.%		91.70%	58.73%	26.85%	16.09%	12.61%	10.62%	9.79%
	E. Spd.								
Loop21	Elapsed	154.43	160.65	80.42	32.24	18.13	13.34	9.12	7.56
	Speedup		0.96	1.92	4.79	8.52	12.51	16.95	20.42
	Eff.%		96.13%	96.02%	95.80%	85.18%	83.43%	87.75%	81.70%
	E. Spd.			1.99	4.83	9.92	14.59	19.43	24.62
Loop22	Elapsed	1791.19	1818.29	928.46	371.82	185.75	124.37	93.82	68.15
	Speedup		0.99	1.93	4.82	9.64	14.40	19.09	26.28
	Eff.%		98.51%	96.46%	96.35%	96.43%	96.01%	95.46%	105.13%
	E. Spd.			1.97	4.93	9.91	14.92	19.93	24.94
Loop24	Elapsed	738.71	755.66	387.52	153.34	76.08	51.20	38.35	29.42
	Speedup		0.98	1.91	4.82	9.71	14.43	19.26	25.11
	Eff.%		97.76%	95.31%	96.35%	97.10%	96.19%	96.31%	100.44%
	E. Spd.			1.86	4.66	9.64	14.12	19.31	24.57

Table 1: (Cont.) Measurements of $PPFC$ generated code for Livermore Loops on the Sequent Symmetry

Program	Data	MOSIX Processors					
		serial	1	2	4	6	8
Loop1	Elapsed	219.80	210.96	106.13	52.90	35.46	26.80
	Speedup		1.04	2.07	4.16	6.20	8.20
	Eff.%		104.19%	103.55%	103.88%	103.31%	102.52%
	E. Spd.			1.99	3.99	5.87	7.945
Loop3	Elapsed	106.00	104.63	52.33	27.03	18.23	13.83
	Speedup		1.01	2.03	3.92	5.81	7.66
	Eff.%		101.31%	101.28%	98.04%	96.91%	95.81%
	E. Spd.			1.99	3.98	5.86	7.90
Loop7	Elapsed	167.33	166.63	83.36	41.8	27.93	21.1
	Speedup		1.00	2.01	4.00	5.99	7.93
	Eff.%		100.42%	100.37%	100.08%	99.85%	99.13%
	E. Spd.			1.99	3.99	5.87	7.94
Loop8	Elapsed	160.23	171.4	85.83	43.00	28.76	21.93
	Speedup		0.93	1.87	3.73	5.57	7.31
	Eff.%		93.48%	93.34%	93.16%	92.85%	91.33%
	E. Spd.			1.99	3.99	5.88	7.95
Loop9	Elapsed	218.00	262.96	131.53	65.93	44.1	33.26
	Speedup		0.83	1.66	3.31	4.94	6.55
	Eff.%		82.90%	82.87%	82.66%	82.39%	81.93%
	E. Spd.			1.99	3.98	5.75	7.88
Loop10	Elapsed	196.43	205.8	99.46	48.6	32.2	24.03
	Speedup		0.95	1.97	4.04	6.10	8.17
	Eff.%		95.45%	98.75%	101.04%	101.67%	102.18%
	E. Spd.			1.99	3.99	5.87	7.98
Loop11	Elapsed	143.43	143.03	68.7	35.73	24.06	17.96
	Speedup		1.00	2.09	4.01	5.96	7.99
	Eff.%		100.28%	104.39%	100.36%	99.36%	99.83%
	E. Spd.			1.99	3.99	5.85	7.96

Table 2: Measurements of *PPFC* generated code for Livermore Loops on the MOSIX Multiprocessor

Program	Data	MOSIX Processors					
		serial	1	2	4	6	8
Loop12	Elapsed	139.56	135.4	67.73	34.6	23.53	17.66
	Speedup		1.03	2.06	4.03	5.93	7.90
	Eff.%		103.07%	103.03%	100.84%	98.85%	98.78%
	E. Spd.			1.99	3.99	5.86	7.96
Loop18	Elapsed	37.7	35.1	31.26	29.23	28.33	26.47
	Speedup		1.07	1.21	1.29	1.33	1.42
	Eff.%		107.41%	60.30%	32.24%	22.18%	17.80%
	E. Spd.						
Loop21	Elapsed	44.1	44	22	11.134	7.433	5.567
	Speedup		1.00	2.00	3.96	5.93	7.92
	Eff.%		100.23%	100.23%	99.02%	98.88%	99.02%
	E. Spd.			1.99	3.99	5.91	7.93
Loop22	Elapsed	1952.23	1939.76	969.26	485.13	323.76	244.03
	Speedup		1.01	2.01	4.02	6.03	8.00
	Eff.%		100.64%	100.71%	100.60%	100.50%	100.00%
	E. Spd.			1.99	3.99	5.91	7.99
Loop24	Elapsed	274.93	280.36	140.46	69.9	47.36	35.6
	Speedup		0.98	1.96	3.93	5.81	7.72
	Eff.%		98.06%	97.87%	98.33%	96.75%	96.53%
	E. Spd.			1.99	3.99	5.89	7.97

Table 2: (Cont.) Measurements of *PPFC* generated code for Livermore Loops on the MOSIX Multiprocessor

Program	Data	# Transputers					
		serial	1	2	4	6	8
Loop1	Elapsed	218.13	219.34	109.26	54.61	36.52	27.39
	Speedup		0.99	1.99	3.99	5.97	7.96
	Eff.%		99.4%	99.8%	99.8%	99.5%	99.5%
	E. Spd.			1.99	3.98	5.99	7.99
Loop3	Elapsed	150.29	150.57	75.72	37.83	25.35	19.17
	Speedup		0.99	1.98	3.97	5.92	7.83
	Eff.%		99.8%	99.2%	99.3%	98.8%	97.9%
	E. Spd.			1.99	3.97	5.98	7.98
Loop7	Elapsed	134.78	135.28	67.71	34.16	22.94	17.27
	Speedup		0.99	1.99	3.94	5.87	7.80
	Eff.%		99.6%	0.99%	98.5%	97.9%	97.5%
	E. Spd.			1.99	3.98	5.97	7.97
Loop8	Elapsed	120.59	121.66	60.84	30.78	20.61	15.68
	Speedup		0.99	1.98	3.91	5.85	7.69
	Eff.%		99.1%	99.1%	97.9%	97.5%	96.1%
	E. Spd.			1.99	3.98	5.99	7.99
Loop9	Elapsed	172.06	172.33	86.21	43.15	29.23	22.13
	Speedup		0.99	1.99	3.98	5.88	7.77
	Eff.%		99.8%	99.7%	99.6%	98.1%	97.1%
	E. Spd.			1.99	3.97	5.96	7.96
Loop10	Elapsed	242.89	246.02	123.30	62.74	41.22	31.07
	Speedup		0.98	1.96	3.87	5.89	7.81
	Eff.%		98.7%	98.4%	96.7%	98.2%	97.7%
	E. Spd.			1.99	3.98	5.97	7.97
Loop11	Elapsed	169.61	170.38	85.46	42.90	28.43	21.91
	Speedup		0.99	1.98	3.95	5.96	7.74
	Eff.%		99.5%	99.2%	98.8%	99.4%	96.7%
	E. Spd.			1.99	3.98	5.97	7.96

Table 3: Measurements of *PPFC* generated code for Livermore Loops on the Transputers Network

Program	Data	# Transputers					
		serial	1	2	4	6	8
Loop12	Elapsed	170.05	170.27	85.54	42.93	28.82	21.96
	Speedup		0.99	1.98	3.96	5.90	7.74
	Eff.%		99.8%	99.4%	99.0%	98.3%	96.7%
	E. Spd.			1.99	3.98	5.98	7.98
Loop18	Elapsed	63.19	63.71	51.67	46.81	41.12	38.84
	Speedup		0.99	1.22	1.34	1.54	1.66
	Eff.%		99.2%	61.1%	33.7%	25.6%	20.7%
	E. Spd.						
Loop21	Elapsed	88.18	88.71	44.43	22.61	15.22	11.51
	Speedup		0.99	1.98	3.90	5.79	7.66
	Eff.%		99.4%	99.2%	97.5%	96.5%	95.7%
	E. Spd.			1.99	3.97	5.97	7.97
Loop22	Elapsed	1462.89	1465.31	731.51	365.92	244.85	183.91
	Speedup		0.99	1.99	3.99	5.97	7.95
	Eff.%		99.8%	99.9%	99.9%	99.5%	99.4%
	E. Spd.			1.99	3.99	5.98	7.98
Loop24	Elapsed	349.01	349.31	174.69	87.47	58.81	43.74
	Speedup		99.9	1.99	3.99	5.93	7.97
	Eff.%		99.9%	99.8%	99.7%	98.9%	99.7%
	E. Spd.			1.99	3.99	5.98	7.98

Table 3: (Cont.) Measurements of *PPFC* generated code for Livermore Loops on the Transputers Network

processor, and when the program is executed on some processors, the data is distributed among the processors, so that each one of them has to process a small amount of data that can reside in it's cache memory, and so each processor will execute it's part faster since most of the time it will access the cache memory.

The speedup estimation on the all targets is very close to the actual speedup. It is within 2 – 4% of the measured speedup.

The only exception is **Loop18** which achieved a very poor efficiency on all computers. Code 13 illustrates the body of Livermore Loop 18.

```
 1     DO l = 1, N
           DO k = 2, kn
 2             DO j = 2, jn
                   za(j,k)= (zp(j-1,k+1)+zq(j-1,k+1)-zp(j-1,k)-zq(j-1,k))
    *                       *(zr(j,k)+zr(j-1,k))/(zm(j-1,k)+zm(j-1,k+1))
                   zb(j,k)= (zp(j-1,k)+zq(j-1,k)-zp(j,k)-zq(j,k))
    *                        *(zr(j,k)+zr(j,k-1))/(zm(j,k)+zm(j-1,k))
               END DO
           END DO

3          DO k = 2, kn
               DO j = 2, jn
                   zu(j,k)= zu(j,k)+s*(za(j,k)*(zz(j,k)-zz(j+1,k))
    *                   -za(j-1,k) *(zz(j,k)-zz(j-1,k))
    *                   -zb(j,k)   *(zz(j,k)-zz(j,k-1))
    *                   +zb(j,k+1) *(zz(j,k)-zz(j,k+1)))
                   zv(j,k)= zv(j,k)+s*(za(j,k)*(zr(j,k)-zr(j+1,k))
    *                   -za(j-1,k) *(zr(j,k)-zr(j-1,k))
    *                   -zb(j,k)   *(zr(j,k)-zr(j,k-1))
    *                   +zb(j,k+1) *(zr(j,k)-zr(j,k+1)))
               END DO
           END DO

4          DO k = 2, kn
               DO j = 2, jn
                   zr(j,k)= zr(j,k)+t*zu(j,k)
                   zz(j,k)= zz(j,k)+t*zv(j,k)
               END DO
           END DO
       END DO
```

Figure 13: Livermore Loop 18

PPFC recognizes that loop at label 1 is replicated, and the loops at labels 2, 3 and 4 are parallel. Since the upper bound of these parallel loops is variable (jn for loop at label 2 and kn for loops at labels 3 and 4), the *PPFC* cannot estimates the serial and parallel time of these loops, so it generates code that decides on run time whether or not to parallelize these loops. For instance, the decision whether to parallelize the loop at label 2 is made by comparing the current value of jn to a constant computed by *PPFC* time estimation algorithm. It seems that usually the run time decision is not to parallelize the loops at label 3 and 4, so that these loops only cause additional overhead that degrade the performance.

6.3 Application Programs Performance

6.4 Programs Description

The 14 application programs that were used to measure the performance of the P^3C were also translated into Fortran and compiled by *PPFC*. The speedups achieved by the *PPFC* were identical to the speedups achieved by the P^3C, since the VMMP code generated by the *PPFC* for these programs was identical to the code generated by the P^3C.

These programs belong to several important domains. Most of them were previously used in the literature for evaluating parallel computers and automatic parallelization. These programs represent a wide selection of algorithms.

The last program illustrates WCSIS loops.

Most of the programs (except **Pi**, **MontePi**, **Mandel** and **MM**) perform data exchange among the processors after each iteration.

Linear Algebra

CG Solve a system of 400 linear equations of 400 unknowns with a dense random coefficients matrix (not positive definite) using the Conjugate-Gradient method [28]. The Conjugate Gradient method is the preferred algorithm for the solution of large systems of equations resulting from Finite Element Analysis.

Invert Invert a 250 by 250 matrix using Gauss-Jordan reduction with partial pivoting. This algorithm is similar to Gaussian elimination and LU factorization. *PPFC* automatically allocates the array by interleaved columns for efficient execution of row operations. This is the same allocation used by manual parallelization of the matrix elimination [26].

PPFC converted the column access to row access on the Sequent in order to avoid memory thrashing due to the large amount of active data pages. This conversion involved transposing all array references. Row access reduces the number of active memory pages relative to column access on a large number of processors.

Linear Solve a system of 400 linear equations in 400 unknowns by Jordan iterations with even/odd update order of unknowns to accelerate convergence. Each iteration involves sharing of last iteration data.

MM Multiply two 230 by 230 matrices and compute the trace of the product matrix. This program exhibits less than linear speedup because the timing include the data distribution and the collection of the results.

Simulations

Fluid Compute a small scale fluid flow by a cellular automata on a 200 by 200 hexagonal grid for 300 time steps. This is the algorithm used by Shell Research for computational fluid dynamics with complex boundary conditions [52]. The structure of this program is similar to **Laplace**, because both programs compute the new values of grid points using some update stencil (neighboring points).

Nbody Simulate the motion of 250 bodies in 3 dimensions attracted by a long range (gravitation) force for 100 time steps. Each step involves the computation of all pair-wise interactions (this is an $O(n^2)$ algorithm). An explicitly parallel version of this program appeared in [50].

Note: there are algorithms with lower complexity for solving this problem, such as the tree algorithm by Appel [3].

PIC One dimensional simulation of two warm electron beams by the Particle in Cell (PIC) method with cloud in cell (CIC) charge assignment. 10,000 particles are moved for 200 time steps using a 256 cells grid. The algorithm is taken from the book [35]. Hiromoto [34] parallelized a two dimensional PIC code by hand.

Ising Simulation of the Ising model using Monte-Carlo technique on a 256 by 256 lattice with periodic boundary conditions for 400 time steps. The program implements the algorithm of Friedberg and Cameron [20].

The structure of this program is also similar to **Laplace**. Apart from a different update scheme for the grid points, this program uses a wrapped grid to simulate a periodic domain instead of a regular grid.

PDE Solution

Laplace Solve the Laplace PDE on a 200 by 200 grid by Simultaneous Over Relaxation (SOR) with Chebyshev acceleration and red/black point ordering.

Wave Calculation of a two dimensional surface wave through a deflector with periodic boundary conditions using a 192 by 192 grid for 500 time steps. The domain is mapped on the unit square. This is the same program and data that used by the 1987 Gordon Bell Prize winners [30].

Cellular Automata

Life Compute the game of Life by a cellular automata on a 300 by 300 grid for 300 iterations. The structure of this program is also similar to **Laplace**.

Miscellaneous (embarrassingly parallel)

Pi Compute the value of π by integration of the function $\sqrt{1-x^2}$ in the interval [0,1] using 2,500,000 steps. This is the program used by Karp and Babb [37] for comparing parallel FORTRAN compilers.

MontePi Compute the value of π by Monte Carlo integration using 1,500,000 random points.

Mandel Compute the Mandelbrot set on a 150 by 150 grid, whose lower left point is at $[-2, -1.25]$ and highest right point is at $[0.5, 1.25]$. This calculation is balanced by interleaved allocation of grid rows to the processors.

Cross Illustrates the parallelization of WCSIS loops mentioned in section 4.5. This example is taken from the Callahan Dongarra Levine vector test suite [14]. The program includes a WCSIS loop of 1000000 iteration.

6.5 Application Programs Results

Tables 4, 5 and 6 contain the speedup measurements and estimated speedup on the Sequent Symmetry, MOSIX and the Transputers network, respectively.

The information presented in the tables is described in 6.2.4.

Program	Data	Sequent Processors							
		serial	1	2	5	10	15	20	25
CG	Elapsed	937.48	953.24	480.98	196.81	102.24	70.91	55.45	46.07
	Speedup		0.98	1.95	4.76	9.17	13.22	16.91	20.35
	Eff.%		98.3%	97.5%	95.3%	91.7%	88.1%	84.5%	81.4%
	E. Spd.			1.94	4.82	9.46	13.74	18.11	22.07
Invert	Elapsed	454.45	473.01	241.71	98.83	50.72	34.65	26.86	22.37
	Speedup		0.96	1.88	4.60	8.96	13.12	16.92	20.32
	Eff.%		96.1%	94.0%	92.0%	89.6%	87.4%	84.6%	81.3%
	E. Spd.			1.98	4.85	9.23	13.45	16.96	20.44
Linear	Elapsed	393.43	398.61	198.45	79.29	40.82	28.85	21.53	17.61
	Speedup		0.99	1.98	4.96	9.64	13.64	18.27	22.34
	Eff.%		98.7%	99.1%	99.2%	96.4%	90.9%	91.4%	89.4%
	E. Spd.			1.93	4.74	9.16	12.69	17.05	20.52
MM	Elapsed	212.50	218.21	108.34	43.61	22.04	15.46	11.72	9.84
	Speedup		0.97	1.96	4.87	9.64	13.74	18.13	21.59
	Eff.%		97.4%	98.1%	97.4%	96.4%	91.6%	90.7%	86.4%
	E. Spd.			1.91	4.76	9.43	13.44	17.76	21.16
Fluid	Elapsed	429.87	432.55	218.31	88.26	44.00	30.31	23.45	19.38
	Speedup		0.99	1.97	4.87	9.77	14.18	18.33	22.18
	Eff.%		99.3%	98.5%	97.4%	97.7%	94.5%	91.7%	88.7%
	E. Spd.			1.96	4.81	9.32	13.83	17.42	21.04
Nbody	Elapsed	635.00	635.96	319.99	131.79	69.73	50.16	40.08	35.30
	Speedup		1.00	1.98	4.82	9.11	12.66	15.84	17.99
	Eff.%		99.8%	99.2%	96.4%	91.1%	84.4%	79.2%	71.9%
	E. Spd.			1.99	4.91	9.51	13.39	16.60	20.08
PIC	Elapsed	582.20	585.90	294.88	121.14	64.92	46.40	37.56	32.01
	Speedup		0.99	1.97	4.81	8.97	12.55	15.50	18.19
	Eff.%		99.3%	98.7%	96.1%	89.7%	83.6%	77.5%	72.8%
	E. Spd.			1.98	4.85	9.26	13.08	16.28	18.83

Table 4: Measurements of *PPFC* generated code on the Sequent Symmetry

Program	Data	Sequent Processors							
		serial	1	2	5	10	15	20	25
Laplace	Elapsed	1413.0	1421.6	708.88	288.02	147.30	101.92	78.66	66.13
	Speedup		0.99	1.99	4.91	9.59	13.86	17.96	21.37
	Eff.%		99.4%	99.7%	98.1%	95.9%	92.4%	89.8%	85.5%
	E. Spd.			1.98	4.86	9.52	13.32	18.10	21.97
Wave	Elapsed	2764.9	2813.2	1405.0	573.42	293.78	192.58	148.55	118.57
	Speedup		0.98	1.97	4.82	9.41	14.36	18.61	23.32
	Eff.%		98.2%	98.4%	96.4%	94.1%	95.7%	93.1%	93.3%
	E. Spd.			1.96	4.81	9.35	14.30	18.49	22.96
Ising	Elapsed	966.83	967.75	495.44	207.44	103.78	73.05	56.10	45.61
	Speedup		1.00	1.95	4.66	9.32	13.24	17.23	21.20
	Eff.%		99.9%	97.6%	93.2%	93.2%	88.2%	86.2%	84.8%
	E. Spd.			1.99	4.90	9.73	13.95	19.12	22.42
Life	Elapsed	888.08	888.96	447.73	179.37	89.58	63.31	45.09	36.92
	Speedup		1.00	1.98	4.95	9.91	14.03	19.69	24.06
	Eff.%		99.9%	99.2%	99.0%	99.1%	93.5%	98.5%	96.2%
	E. Spd.			1.99	4.96	9.84	14.62	19.28	23.81
Pi	Elapsed	96.17	97.18	48.62	19.45	9.76	6.52	4.89	3.93
	Speedup		0.99	1.98	4.94	9.85	14.76	19.65	24.47
	Eff.%		98.9%	98.9%	98.9%	98.5%	98.4%	98.2%	97.9%
	E. Spd.			2.00	4.99	9.98	14.95	19.88	24.77
MontePi	Elapsed	137.58	137.65	68.95	27.62	13.82	9.22	6.93	5.57
	Speedup		0.99	1.99	4.98	9.96	14.92	19.87	24.70
	Eff.%		99.6%	99.8%	99.7%	99.6%	99.5%	99.3%	98.8%
	E. Spd.			2.00	4.99	9.99	14.96	19.92	24.84
Mandel	Elapsed	305.15	305.36	152.66	61.02	31.04	20.96	15.60	12.81
	Speedup		1.00	2.00	5.00	9.83	14.56	19.56	23.83
	Eff.%		99.9%	99.9%	100.0%	98.3%	97.1%	97.8%	95.3%
	E. Spd.			2.00	4.99	9.99	14.95	18.66	24.79
Cross	Elapsed	81.05	81.36	56.76	39.51	12.32	9.67	7.14	6.08
	Speedup		1.00	1.43	2.05	6.58	8.38	11.35	13.33
	Eff.%		99.62%	71.40%	41.03%	65.79%	55.88%	56.76%	53.32%
	E. Spd.			1.75	3.7	7.54	10.27	13.34	16.31

Table 4: (Cont.) Measurements of *PPFC* generated code on the Sequent Symmetry

Program	Data	MOSIX Processors					
		serial	1	2	4	6	8
CG	Elapsed time	245.03	246.13	126.63	67.17	50.00	40.33
	Actual Speedup		0.99	1.93	3.65	4.90	6.08
	Eff.%		99.6%	96.7%	91.2%	81.7%	75.9%
	E. Spd.			1.95	3.67	5.01	6.00
Invert	Elapsed time	111.30	111.86	58.40	30.60	21.53	17.03
	Actual Speedup		0.99	1.91	3.64	5.17	6.54
	Eff.%		99.5	95.3%	90.9%	86.1%	81.7%
	E. Spd.			1.96	3.75	5.26	6.53
Linear	Elapsed time	97.80	103.23	52.50	27.43	19.33	15.73
	Actual Speedup		0.95	1.86	3.56	5.06	6.22
	Eff.%		94.7%	93.1%	89.1%	84.3%	77.7%
	E. Spd.			1.93	3.63	4.96	6.15
MM	Elapsed time	57.33	65.67	29.23	15.17	10.53	8.53
	Actual Speedup		0.87	1.96	3.78	5.44	6.72
	Eff.%		87.3%	98.1%	94.5%	90.7%	84.0%
	E. Spd.			1.97	3.79	5.40	6.87
Fluid	Elapsed time	74.87	75.40	37.90	19.43	13.43	10.27
	Actual Speedup		0.99	1.98	3.85	5.57	7.29
	Eff.%		99.3%	98.8%	96.3%	92.9%	91.2%
	E. Spd.			1.98	3.86	5.66	7.20
Nbody	Elapsed time	410.03	419.93	211.13	108.70	75.23	58.57
	Actual Speedup		0.98	1.94	3.77	5.45	7.00
	Eff.%		97.6%	97.1%	94.3%	90.8%	87.5%
	E. Spd.			1.98	3.84	5.54	6.93
PIC	Elapsed time	154.43	155.68	79.63	41.40	29.67	24.33
	Actual Speedup		0.99	1.94	3.73	5.21	6.35
	Eff.%		99.2%	97.0%	93.3%	86.8%	79.3%
	E. Spd.			1.97	3.73	5.18	6.30

Table 5: Measurements of *PPFC* generated code on the MOSIX Multiprocessor

Program	Data	MOSIX Processors					
		serial	1	2	4	6	8
Ising	Elapsed time	459.60	468.50	238.97	122.50	84.93	64.03
	Actual Speedup		0.98	1.92	3.75	5.41	7.18
	Eff.%		98.1%	96.2%	93.8%	90.2%	89.7%
	E. Spd.			1.99	3.96	5.83	7.71
Laplace	Elapsed time	434.03	438.60	217.03	116.63	82.70	67.40
	Actual Speedup		0.99	2.00	3.72	5.25	6.44
	Eff.%		98.9%	100.0%	93.0%	87.5%	80.5%
	E. Spd.			1.96	3.75	5.38	6.67
Wave	Elapsed time	908.87	909.20	462.33	233.33	158.60	119.60
	Actual Speedup		1.00	1.97	3.90	5.73	7.60
	Eff.%		100.0%	98.3%	97.4%	95.5%	95.0%
	E. Spd.			1.99	3.97	5.89	7.76
Life	Elapsed time	89.50	90.73	46.00	23.80	16.43	12.50
	Actual Speedup		0.99	1.95	3.76	5.45	7.16
	Eff.%		98.6%	97.3%	94.0%	90.8%	89.5%
	E. Spd.			1.99	3.93	5.78	7.42
Pi	Elapsed time	138.67	141.93	71.03	35.53	23.70	17.77
	Actual Speedup		0.98	1.95	3.90	5.85	7.80
	Eff.%		97.7%	97.6%	97.6%	97.5%	97.6%
	E. Spd.			2.00	3.99	5.99	7.99
MontePi	Elapsed time	82.87	82.90	41.43	20.73	13.83	10.40
	Actual Speedup		1.00	2.00	4.00	5.99	7.97
	Eff.%		99.9%	100.0%	99.9%	99.8%	99.6%
	E. Spd.			2.00	3.99	5.99	7.99
Mandel	Elapsed time	129.23	129.37	64.93	32.67	21.87	16.37
	Actual Speedup		0.99	1.99	3.96	5.91	7.90
	Eff.%		99.8%	99.5%	98.9%	98.5%	98.7%
	E. Spd.			2.00	3.95	5.99	7.88
Cross	Elapsed time	27.43	27.17	18.63	10.57	8.67	7.63
	Actual Speedup		1.01	1.47	2.60	3.16	3.60
	Eff.%		100.96%	73.62%	64.88%	52.73%	44.94%
	E. Spd.			1.9	3.34	4.83	6.54

Table 5: (Cont.) Measurements of *PPFC* generated code on the MOSIX Multiprocessor

Program	Data	# Transputers					
		serial	1	2	4	6	8
CG	Elapsed time	412.44	413.01	211.25	110.79	79.33	64.12
	Speedup		1.00	1.95	3.72	5.20	6.43
	Eff.%		99.9%	97.6%	93.1%	86.7%	80.4%
	E. Spd.			1.95	3.72	5.21	6.49
Invert	Elapsed time	234.98	242.44	125.37	65.59	46.06	36.67
	Speedup		0.97	1.87	3.58	5.10	6.41
	Eff.%		96.9%	93.7%	89.6%	85.0%	80.1%
	E. Spd.			1.93	3.71	5.32	6.86
Linear	Elapsed time	175.68	176.08	90.30	47.80	35.70	29.83
	Speedup		0.99	1.95	3.68	4.92	5.89
	Eff.%		99.7%	97.3%	91.9%	82.0%	73.6%
	E. Spd.			1.92	3.62	4.89	5.94
MM	Elapsed time	101.60	101.71	51.85	26.66	18.23	13.90
	Speedup		0.99	1.96	3.81	5.57	7.31
	Eff.%		99.8%	98.0%	95.3%	92.9%	91.4%
	E. Spd.			1.98	3.87	5.69	7.56
Fluid	Elapsed time	326.22	326.55	165.02	83.63	56.53	43.07
	Speedup		1.00	1.98	3.90	5.77	7.57
	Eff.%		99.9%	98.8%	97.5%	96.2%	94.7%
	E. Spd.			1.98	3.91	5.83	7.57
Nbody	Elapsed time	339.58	340.95	172.02	90.35	64.84	53.14
	Speedup		0.99	1.97	3.76	5.24	6.39
	Eff.%		99.6%	98.7%	94.0%	87.3%	79.9%
	E. Spd.			1.98	3.79	5.40	6.65
PIC	Elapsed time	201.97	202.19	104.11	56.10	41.54	35.37
	Speedup		0.99	1.94	3.60	4.86	5.71
	Eff.%		99.8%	97.0%	90.0%	81.0%	71.4%
	E. Spd.			1.95	3.63	4.97	5.94

Table 6: Measurements of $PPFC$ generated code on the Transputers Network

Program	Data	# Transputers					
		serial	1	2	4	6	8
Ising	Elapsed time	882.41	883.29	447.56	227.50	153.65	117.39
	Speedup		1.00	1.97	3.88	5.74	7.52
	Eff.%		99.9%	98.6%	97.0%	95.7%	94.0%
	E. Spd.			1.99	3.97	5.88	7.85
Laplace	Elapsed time	632.66	634.56	323.19	168.50	116.61	91.49
	Speedup		0.99	1.96	3.75	5.43	6.92
	Eff.%		99.7%	97.9%	93.9%	90.4%	86.4%
	E. Spd.			1.95	3.74	5.41	6.78
Wave	Elapsed time	1487.42	1502.45	756.05	380.23	254.53	191.96
	Speedup		0.99	1.97	3.91	5.84	7.75
	Eff.%		99.0%	98.4%	97.8%	97.4%	96.9%
	E. Spd.			1.99	3.97	5.92	7.83
Life	Elapsed time	800.70	803.89	403.37	202.61	136.01	103.84
	Speedup		0.99	1.99	3.95	5.89	7.71
	Eff.%		99.6%	99.3%	98.8%	98.1%	96.4%
	E. Spd.			1.99	3.97	5.91	7.72
Pi	Elapsed time	119.98	120.1	60.25	30.61	20.58	15.63
	Speedup		1.00	1.99	3.92	5.83	7.68
	Eff.%		99.9%	99.6%	98.0%	97.2%	96.0%
	E. Spd.			1.99	3.98	5.96	7.92
MontePi	Elapsed time	106.34	106.45	53.17	26.65	17.81	13.41
	Speedup		1.00	2.00	3.99	5.97	7.93
	Eff.%		99.9%	100.0%	99.8%	99.5%	99.1%
	E. Spd.			1.99	3.98	5.95	7.91
Mandel	Elapsed time	89.92	90.25	45.27	22.83	15.31	11.53
	Speedup		0.99	1.99	3.94	5.87	7.80
	Eff.%		99.6%	99.3%	98.5%	97.9%	97.5%
	E. Spd.			1.99	3.89	5.86	7.64
Cross	Elapsed time	41.28	42.13	22.43	12.08	8.17	6.02
	Speedup		0.97	1.86	3.42	5.12	6.83
	Eff.%		97.6%	93.2%	85.4%	85.4%	85.4%
	E. Spd.			1.89	3.61	4.67	6.53

Table 6: (Cont.) Measurements of $PPFC$ generated code on the Transputers Network

6.5.1 Application Programs Observations

The results shown in tables 4, 5 and 6 indicate that *PPFC* generates high quality code for a wide range of practical scientific computations. Most programs achieved an efficiency of 70% – 98% on the targets. Some programs even achieved close to a linear speedup.

The only exception is the efficiency of **Cross** which achieve relatively bad efficiency on the MOSIX and Sequent, and relatively good efficiency on the Transputers. The reason is that this program contains a WCSIS loop which is splitted by the *PPFC* into two parallel loops. These two parallel loops requires a barrier synchronization at the end of each, and this is what reduces the program performance, since at the barrier synchronization all the processes has to wait to the slowest process. On the Transputers the variance between the execution time of all process is very small, so that the barrier synchronization overhead is less critical.

The *super linear efficiency* phenomenon mentioned in 6.2.5 did not occurr in the application programs, i.e. no one achieved efficiency of more than 100%. The reason is that these programs contain much more data than the most Livermore Loops, so that the cache memory of each processor in the MOSIX or Sequent cannot contains all the data it has to process, even though the data of the program is distributed among the processors.

The speedup estimation on the all targets is close to the actual speedup. It is within 4% of the measured speedup on the MOSIX and the Transputers and within 8% of the measurements on the Sequent. The only exceptions are the estimations of **Nbody** on the Sequent, **Ising** on the MOSIX and **Invert** on the Transputers, which are within 11.6%, 7.3% and 7.0% of the measurements, respectively. The reason for these deviations might be in an inaccurate estimation of VMMP services time. Many estimations are within 2% of the measurements, such as most estimations on the MOSIX.

The accurate speedup estimation is one of the reasons for the high efficiency of the generated code, because the parallelization is performed only on applicable loops. See section 4 for more details.

PPFC generates efficient data distribution code, which is similar to the data distribution chosen by programmers for the same programs. For example, all grid algorithms (**Fluid**, **Wave**, **Laplace**, **Ising** and **Life**) employs grid distribution with overlapping slices. The matrix inversion **Invert** uses interleaved column distribution, which was recommended by [26]. **CG**, **Linear** and **Nbody** distribute the result vectors by slice. The **PIC** program employs several data distributions strategies, which are similar to the distribution suggested by Hiromoto [34].

Many of the application programs are not trivial to parallelize by hand. In some cases, the hand produced parallel code was the subject of a separate paper, such as matrix inversion [26], particle in cell code [34], 2D wave equation [30] and conjugate gradient [11] (for a sparse matrix). In some papers the hand produced code was used because there was no parallelizing compiler. For example, computation of π by integration [37] and N-body problem [50]. *PPFC* generated an efficient parallel code for all of these programs directly from the serial code and employed similar data partitioning to the manual parallelization.

The combination of the high efficiency of *PPFC* generated code and *PPFC* portability suggests that manual parallelization may be replaced by an automatic parallelizer for many programs on most multiprocessors.

7 Summary and Conclusions

The *PPFC* is a small automatic parallelizer which emphasize portability and efficiency.

PPFC produces efficient parallel code for a wide range of scientific applications. Some of these programs were previously parallelized only manually. *PPFC* is fully automatic and does not require any user directives or hints. It generates efficient data distribution code, which is similar to the manual parallelization of many programs.

PPFC operation is based on the target *parameters table*, which contains information on the execution time of primitive operations and VMMP services. *PPFC* computes an accurate estimation of the program's execution time based on the program's parse tree and the parameters table. A measurement program produces most of the parameters table automatically.

This work indicates that efficient parallel code can be generated for many important application programs by a relatively simple parallelization technique. Good data distribution is the key to efficient parallel execution.

The high efficiency of the generated code and *PPFC* portability suggest that manual parallelization can be replaced by an automatic parallelizer for many programs on most target machines. In this way automatic parallelizers, such as *PPFC*, may simplify the use of parallel computers.

References

[1] F. Allen, M. Burke, P. Charls, R. Cytron & J. Ferrante. "An Overview of the PRTAN Analysis Syatem for Multiprocessing". in Proc. of the 1st Int. Conf. on Supercomputing, Lect. Notes in Comp. Sci., Vol. 297, Springer Verlag, New York-Heidelberg-Berlin, June 1987, pp. 194-211.

[2] F. Andre, C. Jard, Jean-Louis Pazat & Henry Thomas, "Data-driven Distribution of Programs", Int. Workshop on Compilers for Parallel Computers, Paris, France (Dec. 3-5, 1990), pp. 237-247.

[3] Andrew W. Appel, "An Efficient Program for Many-Body Simulation," SIAM Journal of Scientific and Statistical Computing, Vol. 6, No 1(Jan 1985), pp. 85-103.

[4] A. Averbuch, E. Gabber & R. Dekel, "PPFC - Portable Parallelizing Fortran Compiler" Technical Report 283/93, Eskenasy Institute of Computer Sciences, Tel-Aviv University, Tel-Aviv, Israel, October 1993.

[5] A. Averbuch, I. Gohberg, I. Koltracht & B. Shoham , "Timing Analysis of a Parallel Algorithem for Toeplitz Matrices on a MIMD Parallel machine", Parallel Computing, Vol. 17, pp. 563-577 1991.

[6] A. Averbuch, E. Gabber, B. Gorodissky & Y. Medan, "A Parallel FFT on an MIMD Machine", Parallel Computing, Vol. 15, No. 1 (Sep. 1990) pp. 61-74.

[7] A. Averbuch, E. Gabber, S. Itzikowitz & B. Shoham, "On the Parallel Elliptic Single / Multi-Grid Solutions about Aligned and Nonaligned Bodies using VMMP", Scientific Programming, Vol. 3(1994), pp. 13-32.

[8] A. Averbuch, E. Gabber, "A Scalable and Portable Implementation of a Parallel FFT on MIMD Multiprocessors", in preparation.

[9] A. Averbuch, S. Itzikowitz, E. Sammsonov & H. Primack "An Efficient Parallel Algorithm for the Solution of Tridiagonal Systems Associated with the Underwater Acoustic Parabolic Equation", to appear.

[10] A. Averbuch, D. Amitai, R. Friedman & E. Gabber, "Portable Parallel Implementation of BLAS", Concurrency: Practice and Experience, Vol. 6, No. 5(1994), pp. 411-459.

[11] C. Aykanat, F. Ozguner, F. Ercal & P. Sadayappan, "Iterative Algorithm for Solution of Large Sparse Systems of Linear Equati ons on Hypercubes," IEEE Trans. on Computers, Vol. 37, No. 12 (Dec. 1988), pp. 1554-1568.

[12] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy & Ulrich Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions" Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Williamsburg, Virginia (Apr. 21-24, 1991), pp. 213-223. Published as SIGPLAN Notices, Volume 26, Number 7, July 1991.

[13] J. Browne, J. Dongarra, A. Karp, K. Kennedy & D. Kuck, "Special Report: 1988 Gordon Bell Prize", IEEE Software, Vol. 6, No. 3 (May, 1989), pp. 78-85.

[14] D. Callahan, J. Dongarra & D. Levine. "Vectorizing compilers: A test suite and results". In Proceedings of Supercomputing '88, Orlando, FL, November 1988.

[15] D. Callahan & K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors", The Journal of Supercomputing, Vol. 2, No. 2 (Oct. 1988), pp. 151-169.

[16] B. Chapman, P. Mehrotra & Hans Zima, "Programming in Vienna FORTRAN", Scientific Programming, 1(1) (1992), pp. 31-50.

[17] W. F. Clocksin & C. S. Mellish. "Programming in Prolog". SPRINGER, 1981.

[18] J. Dongarra, A. H. Karp, K. Miura & Horst Simon, "Special Report: 1990 Gordon Bell Prize Winners", IEEE Software, Vol. 8, No. 3, (May 1991), pp. 92-97, 102.

[19] F. Darema, D. A. George, V. A. Norton & G. F. Pfister. "A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN". Parallel Computing, Vol 7, No. 1 (APR. 1988), pp. 11-24.

[20] Friedberg & Cameron, "Test of the Monte-Carlo Method: Fast Simulation of a Small Ising Lattice," J. of Chemical Physics, Vol. 52, No. 12(June 1970), pp. 6049-6058.

[21] E. Gabber, A. Averbuch & A. Yehudai, "A Portable Parallelizing Pascal Compiler", IEEE Software, pp. 71-81, March 1993.

[22] E. Gabber, A. Averbuch, A. Yehudai, "PPPC: A System for Generating Portable and Efficient Code for Multiprocessors", Advanced Compilation Techniques for Novel Architectures, M. C. Chen & R. Pinter (Editors), Lecture Notes in Computer Science, Springer Verlag, 1992.

[23] E. Gabber, "VMMP: A Practical Tool for the Development of Portable and Efficient Programs for Multiprocessors", IEEE Transaction on Pattern Analysis, Vol. 1 No. 3 (July 1990) pp. 304-317

[24] E. Gabber "VMMP User's Guide", Technical Report 239/92, Eskenasy Institute of Computer Sciences, Tel-Aviv University, Tel-Aviv, Israel, June 1992.

[25] E. Gabber, "Scalebl... ", Ph.D Thesis, Dept. of Computer Science, School of Mathematical Sciences, Tel Aviv University, 1993.

[26] G. A. Geist & C. H. Romine, "LU Factorization Algorithms on Distributed-Memory Multiprocessor Architec tures," SIAM Journal of Scientific and Statistical Computing, .Vol. 9, No. 4(July 1988), pp. 639-649.

[27] G. Goff, K. Kennedy & Chau-Wen Tseng. "Practical Dependence Testing". PROC of the ACM SIGPLAN'91 CONF on Programming Language Design and Implem entation. Toronto, Canada JUN 26-28, 1991. pp. 15-29. Published as SIGPLAN Notices, Volume 26, Number 6, June 1991.

[28] G. H. Golub, C. F. Van Loan, "Matrix Computations", John Hopkins University Press, 1984

[29] M. Gupta & P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers", IEEE Trans. on Parallel and Distributed Systems, Vol. 3, No. 2(Mar.1992), pp. 179-193.

[30] J. L. Gustafson, G. R. Montry & Robert E. Benner, "Development of Parallel Methods for a 1024-Processors Hypercube," SIAM Journal of Scientific and Statistical Computing, Vol. 9, No. 4 (July 1988), pp. 609-638.

[31] L. Harrison, "An Overview of Compiler Research at CSRD" Int. Workshop on Compilers for Parallel Computers, Paris, France(Dec. 3-5, 1990), pp. 37.

[32] High Performance Fortran Forum, "High Performance Fortran Language Specification", version 1.0 edition, May 1993.

[33] S. Hiranandani, K. Kennedy & Chau-Wen Tseng, "Compiling FORTRAN D for MIMD Distributed Memory Machines", Comm. ACM, Vol. 35, No. 8 (Oct. 1992), pp. 66-80.

[34] R. Hiromoto, "Some Issues in Parallel Processing as Encountered on the Denelcor HEP," Parallel Computing, Vol. 3(1986), pp. 111-127.

[35] Hockney & Eastwood, Computer Simulation Using Particles, McGraw-Hill, 1981 .

[36] D. E. Hudak & S. G. Abraham, "Compiler Techniques for Data Partitioning of Sequentially Iterated Parallel Loops", Proc. of the 1990 Int. Conf. on Supercomputing Amsterdam, The Netherlands(June 11-15, 1990), pp. 187-200.

[37] A. H. Karp & R. G. Babb II "A Comparison of 12 Parallel Fortran Dialects," IEEE Software, Vol. 5, No. 5 (Sept. 1988), pp. 52-67.

[38] K. Kennedy, K. S. McKinley & Chau-Wen Tseng, "Interactive Parallel Programming Using the ParaScope Editor", IEEE Trans. on Parallel and Distributed Systems, Vol. 2, No. 3, (July 1991), pp. 329-341.

[39] Charles Koelbel, Piyush Mehrotra & John Van Rosendale, "Supporting Shared Data Structures on Distributed Memory Architectures", Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programm ing (PPoPP), Seattle, Washington (Mar. 14-16, 1990), pp. 177-186.

[40] D. Loveman, "High Performance Fortran: Proposal", In *High Performance Fortran Forum*, Houston, TX, January 1992.

[41] F. H. McMahon. "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range". Lawerence Livermore National Laboratory. (LLNL UCRL-53745).

[42] Piyush Mehrotra & John Van Rosendale, "The BLAZE Language: A Parallel Language for Scientific Programming", Parallel Computing, Vol. 5, No. 3 (1987), pp. 339-361.

[43] David M. Nicol & Joel H. Saltz. "Principles for Problem Aggregation and Assignment in Medium Scale Multiprocessors". ICASE Report No. 87-39, Institute for Computer Applications in Science and Engineering, SEP 1987.

[44] F. Pereira. "C-Prolog User's Manual". SRI International, Memelo Park, 1984.

[45] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung & D. Schouten. "Parafrase-2: An Environment for Parallelizing partitioning, Synchronizing, and Scheduling Programs on Multiprocessors". Proc. of the 1989 Int. Conf. on parallel Processing, Vol. 2, St. Charls, Illinois (Aug. 8-12, 1989), pp. 39 - 48.

[46] J. Ramanujam & P. Sadayappan, "Compile-Time Techniques for Data Distribution in Distributed Memory Machines", IEEE Trans. on Parallel and Distributed Systems, Vol. 2, No. 4(Oct. 1991), pp. 472-482.

[47] A. Rogers & K. Pingali. "Process Decomposition Through Locality of Reference". Proc. of the SIGPLAN'89 Conf. on Programming Languages Design and Implement ation, Portland, Oregon(June 21-23, 1989), pp. 69-80.

[48] R. Ruhl & M. Annaratone, "Parallelization of FORTRAN Code on Distributed-memory Parallel Processors" , Proc. of the 1990 Int. Conf. on Supercomputing, Amsterdam, The Netherlands (June 11-15, 1990), pp. 342-353.

[49] Vivek Sarkar, "Determining Average Program Execution Times and their Variance" Proc. of the SIGPLAN'89 Conf. on Programming Languages Design and Implementa tion, Portland, Oregon (June 21-23, 1989), pp. 298-312.

[50] C. L. Seitz, "The Cosmic Cube," Comm. ACM, Vol. 28, No. 1(Jan. 1985), pp. 22-33.

[51] A. C. Shaw, "Reasoning About Time in Higher-Level Language Software", IEEE Trans. on Software Engineering, Vol. 15, No. 7 (July 1989), pp. 875-889.

[52] J. A. Somers & P. C. Rem, "A Parallel Cellular Automata Implementation on a Transputer Network for the Simulation of Small Scale Fluid Flow Experiments," in Proc. Of Parallel Computing 1988, Shell Conference, G. A. van Zee & J. G. G. van de Vorst, Lecture Notes in Computer Science Vol. 384, Springer-Verlag, 1989, pp. 116-126.

[53] A. E. Terrano, S. M. Dunn & J. E. Peters, "Using an Architectural Knowledge Base to Generate Code for Parallel Computers", Comm. ACM, Vol. 32 No. 9 (Sept. 1989), pp. 1065-1072.

[54] H. Zima et. al., "VIENNA Fortran - A Language Specification Version 1.1", ICASE Report, #189629 (March 1992); (see also reports numbers: 187634 (Sept. 1991), 191451 (April 1993), 194896 (March 1994), 194904 (April 1994), 194913 (May 1994).