

CHAPTER 1

Introduction

Converting Sequential Programs to Parallel Programs

Although parallel computers can place enormous amounts of computing power in the hands of programmers, writing and porting software to these environments can be quite difficult. From the users standpoint, an automatic parallelizing compiler would solve many problems. Unfortunately, such compilers are beyond the current state of the art and will be required to recognize algorithmic patterns throughout an entire program. Because algorithmic analysis is not currently possible, parallelization tools must work "below the algorithm" and attempt to parallelize the program in its current form. In some cases this approach can produce large performance gains when using parallel computers. On the other hand, certain applications may not yield any performance gain due to algorithmic restrictions. In such cases, the programmer may be required to do an in-depth analysis and re-write parts of the application.

Fortunately, *BERT77* can be used at all levels of the parallelization process. Whether it is an automatic parallelization or a re-coding of the algorithm by the user, *BERT77* can help provide the answer to an important question:

"How efficient is my parallelization ?"

Because *BERT77* attempts to answer this question based on parameters (communication times and computation times) for a specific parallel computing environment (dedicated machine, cluster workstations, etc.), the user can gain an understanding into the parallel efficiency of the algorithm being employed.

Concurrency and Parallelism

Quite often the terms "concurrent" and "parallel" are used to represent the same thing. This mislabeling is often the source of disappointment and misunderstanding when attempting to parallelize programs.

For the purposes of this manual and the *BERT77* parallelization tool the following definitions are provided:

Concurrent - two or more things can be done independently, but not necessarily on different processors. Multiple users on a workstation constitute a concurrent operation.

Parallel - two or more things can be done independently at the same time on different processors.

Although parallel implies concurrent, concurrent does not imply parallel.

This distinction is important because it determines the efficiency of parallel execution. In many cases when some portion of a program is found to be concurrent, it is said to be "parallel". Indeed, it can be executed in parallel, but does it lead to a reduction in computation time? The efficiency of executing something in parallel depends upon the time it takes to communicate with other processors and the time it takes for the other processor to calculate results. If this time is greater or equal than the time required for a single processor, then the parallelization is inefficient. The parallel efficiency of program is hardware dependent. Any attempt to parallelize a program without some insights into computation time and communication time is essentially a guess. The *BERT77* parallelization tool is designed to eliminate "efficiency guessing".

Impediments to Concurrency

Before *BERT77* can determine the efficiency of a parallelization, it must first determine if something is concurrent. The basic impediment to concurrency are recursive variables or loop carried dependencies. These impediments are dependencies that prevent a portion of the program from being divided into concurrent parts. For example, if a loop uses a variable that depends upon previous cycles of the loop, then this loop is not concurrent and can not be parallelized. Distributing this loop across multiple processors is inefficient because each cycle of the loop must wait for the previous cycle to finish before it can begin. Most automatic parallelization tools attempt to determine data dependencies and if possible identify concurrent portions of the program. In practice, very few sequential FORTRAN 77 programs can be converted to parallel programs with out any user intervention because the original authors were not considering concurrent operation. The amount of user input required create concurrency depends upon the program itself. In some cases, simple changes can produce large benefits, in other cases numerous changes to the program and algorithm may be required.

Introduction to *BERT77*

The *BERT77* (or "*BERT*") software tool attempts to automatically and efficiently convert sequential FORTRAN 77 programs for a parallel computing environment. *BERT* produces code that is compatible with the standard message passing model supported by PVM. Like most other conversion tools *BERT* provides two important features:

- 1) detection of concurrent parts of the FORTRAN program,
- 2) re-coding a parallel version of these concurrent parts.

In contrast to many other conversion tools, however, *BERT* examines the concurrent parts of your program to see whether they **should** be executed in parallel. *BERT* understands that overall performance depends upon how efficiently concurrent portions of your program can be executed on separate processors. Without considering computation times and communication times, there is no guarantee that concurrent portions of your program can be efficiently executed in parallel.

To determine parallel efficiency, *BERT* includes a scheduler between the concurrency detection and code generation portions of the conversion process. The following figure shows how the scheduler fits into the generalized conversion process. The *BERT* scheduler requires that any portion executed in parallel provide an execution speed-up. Concurrent portions of your program that are not found to be efficient are not converted to parallel by *BERT* (unless you override *BERT*'s decision).

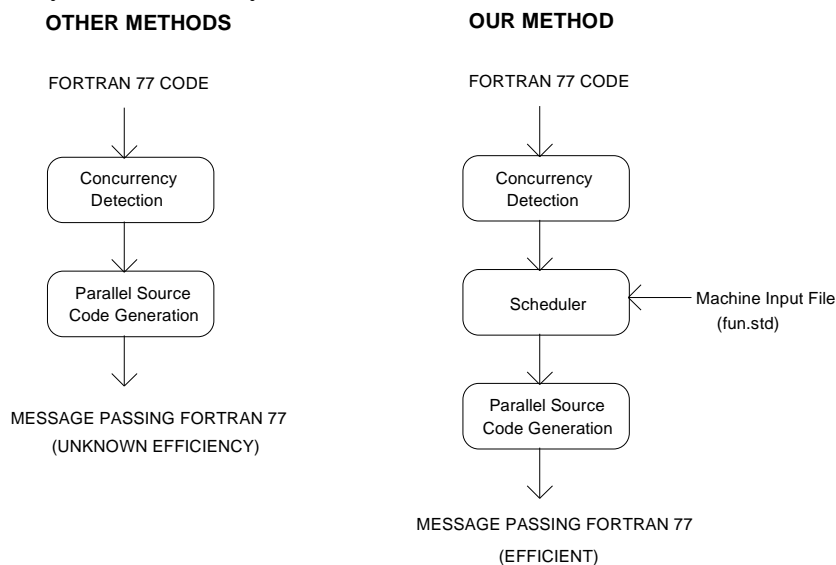


Figure 1-1: Parallelization Methods

BERT makes efficiency decisions by consulting a machine information file for each target host parallel computer. The machine information file contains execution and communication times so that *BERT* can determine whether concurrent portions should be executed in parallel.

Based on previous experience, completely converting a program on the first pass is a rare event. Programs often contain "parallelization inhibitors" (recursive variables or loop carried dependencies) which are detected by *BERT*. Quite often, the program was written without regard to parallelization or parallel efficiency. *BERT* provides a solution to this problem by providing a rapid and cost effective method to analyze and convert sequential FORTRAN 77 applications to **efficient** parallel programs.

The following are some of the features and benefits you can expect from *BERT*

***BERT* FEATURES**

- ☐ Parallelization of loops with procedure calls - using a powerful interprocedural analysis
- ☐ Parallel efficiency analysis of all concurrent and non-current loops
- ☐ Support for task parallelism

- ☐ Pre-runtime performance estimations
- ☐ Determination of parallelization inhibitors

EXPECTED BENEFITS AFTER USING *BERT* FOR SEVERAL HOURS

- ☐ Know if your program can run efficiently under a "dataflow" model and what to do if it will not
- ☐ Know where the parallel and non-parallel loops (inhibited loops) are in your program
- ☐ Know how much each loop contributes to the overall computation time of your program
- ☐ eliminate possible parallelization strategies that will not work
- ☐ Run a converted program
- ☐ Continue to use your original source code on sequential or vector machines

EXPECTED BENEFITS FROM USING *BERT* FOR SEVERAL DAYS

- ☐ Have a firm grasp of the parallel nature of your program
- ☐ Determine if "inhibited loops" can be made parallel
- ☐ Run an efficient conversion of your program in parallel

Performance Estimations

One important benefit gain from efficiency analysis is the ability to estimate performance gains prior to actually running your parallelized program. While actual results can vary due to compiler optimizations, Input and Output, and some run time differences, *BERT77* often has the ability to accurately predict actual parallel speed-up. The figure below is an indication of this ability using a simple Simpsons rule integration program.

The ability to predict parallel performance allows the user to determine the optimum number of processors without actually having the processors available. This prediction can aid in determining the most cost effective number of processor required for a given application

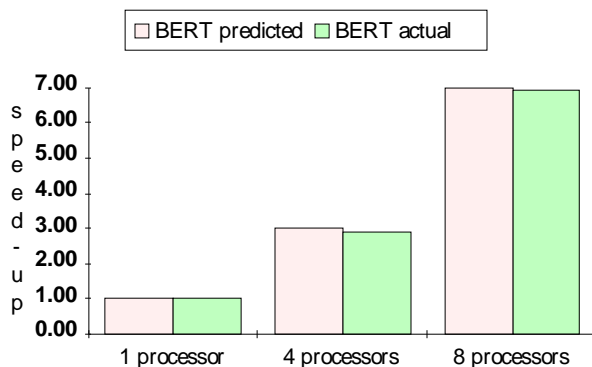


Figure 1-2: Predicted vs. Actual Parallel Performance

BERT77 Requirements

Hardware Requirements

BERT77 requires a workstation host to do the analysis and source code generation. Programs produced by *BERT77* will operate on any number of processors. In general, the maximum number of processors used by *BERT77* is limited by the user's algorithm.

The graphical user interface requires X windows (not included with this version).

The host must have access (either direct or via a network) to a parallel computer or a network of workstations which share a common file system.

Software Requirements

A host library (`libBERT.a`) is required for each hardware platform. This library is included in the *BERT77* distribution and is required by the source code produced by *BERT77*.

A host information file (`fun.std`) is required for a specific hardware platform. This file is included in the *BERT77* distribution and contains computation and communication times for the host parallel computer. Note: the `fun.std` files are encrypted and not user modifiable.

BERT77 requires the presence of a FORTRAN compiler and linker for the target parallel computing platform. The compiler and linker should be supplied by your hardware vendor.

Installation

Please see the platform specific installation instructions provided with manual. The installation notes are also in the file `read.me`.

Programmers View

From the programmers viewpoint, *BERT77* can be considered a language "preprocessor" or even a "parallel compiler" (although a host FORTRAN compiler is required). Essentially *BERT77* will process your FORTRAN program, report on conversion efficiency, and allow you to create an executable or change you program and re-run the analysis.

The following figure describes the user's (programmer's) domain and *BERT77*'s domain. Note that the programmer needs only to concentrate on the FORTRAN 77 source code - not on the details of the conversion. (Although this information is accessible to the programmer.)

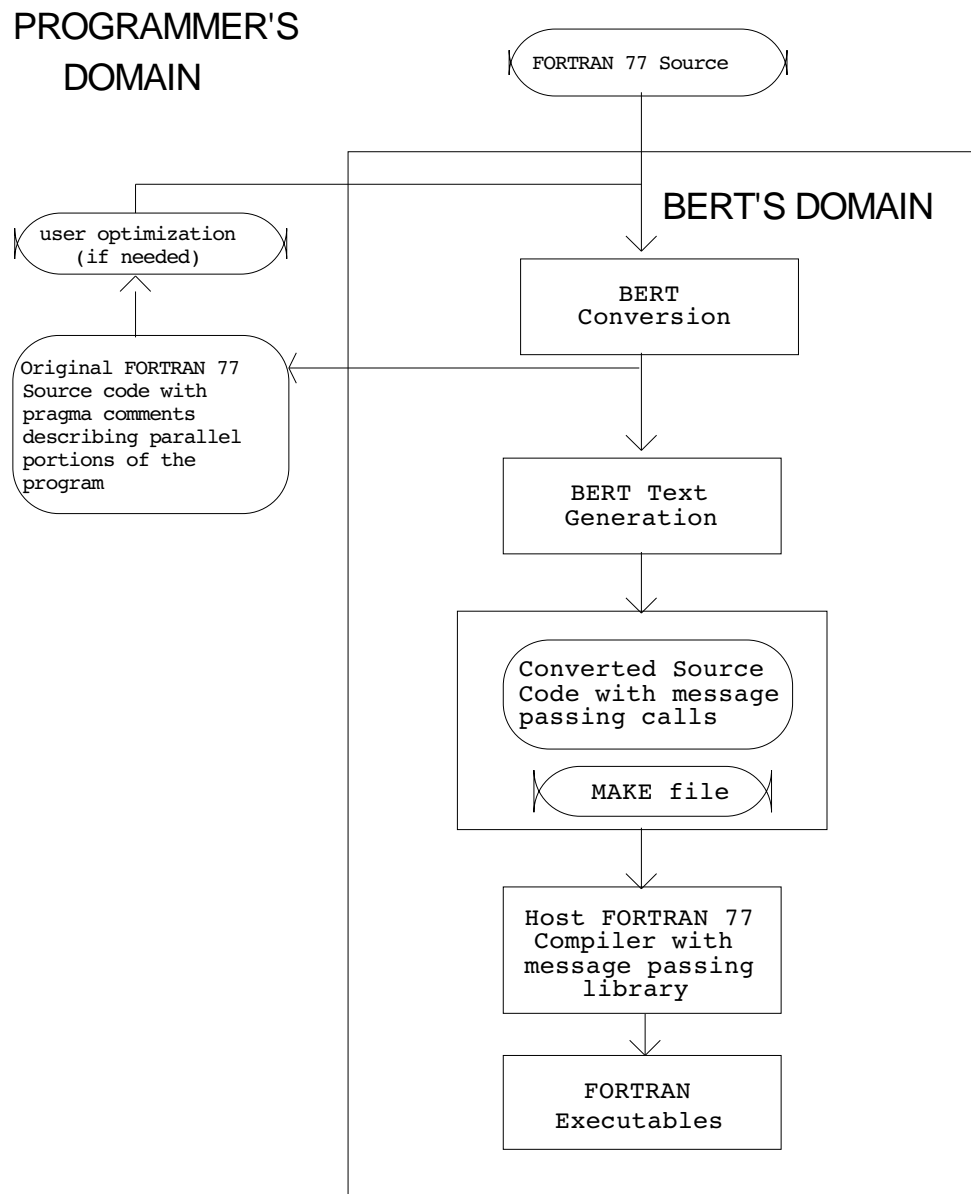


Figure 1-3: Programmer's domain versus BERT 77's domain

Parallization Model and Methods

The following sections discuss background information for *BERT77* and give a general idea behind the parallelization process using *BERT77*.

Introduction

In contrast to most of the parallelizing systems which make a static decomposition of a program, *BERT77* provides a dynamic execution mode. FORTRAN programs are dynamically decomposed, and decisions on parallel execution are made at run-time to achieve better load-balance. The most efficient program constructions for

this approach are DO loops with module calls. Furthermore, if iterations times vary, it does not dramatically slow speedup of a parallel program.

Of course, no automatic system can compare with the power of the human mind. That is why manual control of parallelization is available using pragmas - hints provided to the *BERT77* program. Pragmas always take the form of pseudo comments that do not interfere with standard FORTRAN 77 compilation of the source code. Using pragmas allows the user to concentrate on the program's algorithm without doing a lot of routine work. Pragmas can also be automatically generated by *BERT77* and modified by the user.

A Parallelizing Converter

Many areas of science and engineering require large amounts of computation. To obtain results in a reasonable amount of time a high performance computer is required. The ideal computer for most of the application programmers is a sequential FORTRAN computer with infinite memory and infinite performance. Unfortunately, none of these requirements is realistic. Existing technology has limitations due to the speed of light, molecular level scale of integration, and cooling VLSI components. Distributed memory parallel computers is an alternative to solve these problems, because they allow increased computing power simply by adding more and more relatively inexpensive processors. This architecture, however, requires parallel programs. Users are not very enthusiastic with this requirement because the solution of additional optimization problems is required. An automatic tool can help them to recompile scalar programs onto parallel hardware. Another advantage is that they can give users better understanding of parallel programming without "drowning in a tea-cup". Fully automatic tools are not very practical now, because they can not understand the program from an algorithmic level. Perhaps fully automatic conversion will be possible within 20 years. For the time being, user interaction with the parallelization process provides the best results.

Program Organization

Parallel programs are organized into the data flow master/worker model. Sequential and parallel parts of program are detected. The sequential part is performed by one master (host) processor. Parallel parts are dynamically loaded onto a number of worker (node) processors. When a worker is started, all data required for its execution is sent from master to worker. When the worker is finished, all results are sent back to master. Let's explore an example.

```
CALL S1(X)
DO I = 1, N
  CALL S2(Y(I))
ENDDO
CALL S3(Z)
```

If the subroutine calls have no side effects, the program can be decomposed in the following way.

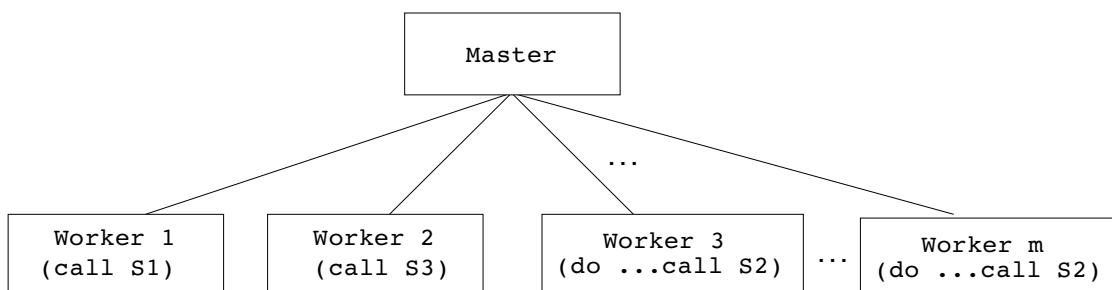


Figure 1-4: Master/Worker Diagram

Here m is the number of parts into which the DO loop is decomposed, provided there are that many node processors. If there are not, the jobs are distributed among the available processors.

Conditions for Parallelism

The basic condition for concurrency and possible parallelization is determined by the data dependencies in data processing. Additional limitations are based on the computational model. In this model, two sets of variables correspond to every program fragment.

An *IN Set* is the set of all variables which values are used for computations in given fragments.

An *OUT Set* is the set of all variables which values are computed in given fragments.

The basic rule is: If the OUT data of one fragment belongs to the IN data of another fragment, then the two fragments are not concurrent and cannot be executed in parallel. For example:

```
X = Y + 1
Z = X + 1
```

The two FORTRAN statements are not concurrent, because variable X, produced by the first statement, is used in the second. An example of concurrent instructions follows:

```
X = Y + 1
Z = Y + 2
```

The *BERT77* master/worker computational model requires further limitations. Data can be produced by parallel fragments in any order. Therefore, no IN set can be intersected with any OUT set for parallel fragments and no OUT sets can be intersected.

```
Z = X + 1 ! Not concurrent, because X value can be computed
X = Y + 1 ! before use
```

```
X = Y + 1 ! Not concurrent, because X values can be computed
X = Y + 2 ! in any order
```

File operations can not be parallelized, because they use modified file pointers. Parallel read/write will require support from the operating system. GOTO's and

STOP instructions cannot be executed in parallel. Distributed memory allows conflicts with local data to be avoided.

```
DO I = 1, N           !All iterations can be executed
  TMP = X(I) * Y(I)   !in parallel
  Y(I) = TMP - 1.
ENDDO
```

Even if the value of TMP is used later it will be broadcast to the master from the last iteration. *BERT77* automatically computes IN and OUT sets by considering module calls and side effects while checking for conditions that inhibit concurrency. The principle of automatic parallelism detection is: *all that is not allowed is prohibited*. When *BERT77* cannot determine the intersection of sets, it assumes the worst case. When concurrent segments are detected it does not mean that they can be efficiently executed in parallel. Overhead caused by data transfer may limit or exceed the gain from parallelization. For example:

```
DO I = 1, 1000
  X(I) = X(I) + 1.
ENDDO
```

Parallelizing this DO loop is inefficient, because in all existing distributed memory systems, transfer time of one data element will be greater than the time of its computation. Consider the loop:

```
DO I = 1, N
  DO J = 1, I
    X(I) = X(I) + Y(J)
  ENDDO
ENDDO
```

Here the efficiency of the DO loop depends on the value of N. Typically if $N = 10$, then parallelizing the DO loop is usually inefficient because of the cost of distributing the loop. If $N = 1000$ the loop is usually efficient. Another limitation is that currently a worker can not organize its own local workers. If we have nested parallel DO loop, only one of them can be parallelized. *BERT77* makes efficiency estimations for parallel constructions and globally selects which of them will be executed in parallel. If it is more efficient to parallelize the inner loop, then the outer loop will remain sequential.