University of Reading

Department of Computer Science

Evolutionary Algorithms for Automatic Parallelization

Kenneth Peter Williams

Parallel, Emergent and Distributed
Architectures Laboratory
(PEDAL) Group

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, University of Reading, Department of Computer Science, Whiteknights Campus, Reading, UK, December 3, 1998.

Abstract

The need for new techniques for use in automatically parallelizing compilers is emphasised by the limited effectiveness of current automatic parallelization tools. This presents a major barrier to the improved use of parallel computers.

This thesis proposes that evolution provides an effective way to parallelize sequential programs. The contributions of this thesis are: description and experimentation with direct and indirect representations of an automatically parallelizing compiler which are manipulated by 6 evolutionary algorithms (EAs) across a set of 5 Fortran-77 benchmark programs. One representation (called GT) naturally gives rise to 5 genetic operators plus 1 heuristic crossover operator, VLX-3. The other (GS) treats compiler transformations as mutation operators.

In this research we present the *R*eading *Evol*utionary *R*estructurer (REVOLVER) system which implements a range of EAs to automatically parallelize sequential Fortran-77 programs for a 12-node Meiko CS-1 message-passing architecture.

Issues involving the application of transformations to code (called decoding) are investigated, three decoding strategies developed, and comparative results produced.

Detailed descriptions of a profiler and performance estimation tool which have been implemented to analyse the message-passing code generated by the EAs are given. Static performance estimation of the code serves as the fitness function of the EAs.

Detailed statistical comparisons are made between the two representations, the three decoding strategies, and the six genetic operators. Results show that EAs using the GS representation consistently produce more optimally parallelized code than that produced by EAs using the GT representation. Most important result is that the EAs were able to find a parallelization strategy that would not have been obvious to a human programmer using an interactive tool - therefore showing that EAs have the ability to find novel automatic parallelization strategies.

Acknowledgements

Firstly I would like to thank my thesis supervisor Dr. Shirley A. Williams for providing an excellent environment in which this research could be performed and without whose continuous support, input and encouragement it would never have been completed.

I would also like to thank Professor Graham M. Megson, for founding the Parallel, Emergent and Distributed Architectures Laboratory (PEDAL) group within which this research was conducted.

I'd like to thank the developers of the *Sage*$^{++}$ restructuring compiler libraries (Dennis Gannon, et al., University of Indiana), and of the `Omega` dependency analysis libraries (William Pugh, et. al., University of Maryland) for making their work freely available in this way.

At various stages I've been fortunate to benefit from the extensive knowledge of Peter Bradbeer (Napier University, UK), Mike Denham (Dept. of Applied Statistics, University of Reading), Mark Harman (Goldsmiths College, University of London, UK), Dieter Kranzlmüller (University of Linz, Austria), Andy Nisbet and Jon MacLaren (University of Manchester, UK), and Paul Walsh (University College Cork, Ireland),

Throughout the course of my research I have also benefited from feedback given by other members of the PEDAL group, most notably, David Corne enlightened me on the finer points of evolutionary computation, while Graham Fagg (now at University of Tennessee at Knoxville), Professor Roger W. Hockney, Dr. Roger Loader, and Dave Clarke provided ample support for numerous stimulating discussions on various aspects of parallel computing. I'd also like to thank Tom Lake and Ben Sloman (both of InterGlossa Ltd) for their useful comments in the early stages of this work.

I am also happy to acknowledge the technical (and sometimes not so technical) assistance of fellow Reading students James Andrew Fryer, Juliet Short, Chris Hines, Rob Fish, and Martyn Lewis (Department of Cybernetics). Thankyou all.

This work was financed under grant 94701308 from the Engineering and Physical Sciences Research Council (EPSRC) of the UK, and also, in its later stages, by the Department of Computer Science, University of Reading.

I also wish to acknowledge the influence of the late Professor Geoff Sullivan who persuaded me of the challenges (and rewards) encountered in research work and thereby lead to me starting this Ph.D program.

Last, but by no means least, I'd like to thank my parents for their consistent support and understanding.

"Our problems are man-made, therefore they may be resolved by man. And man can be as big as he wants. No problem of human destiny is beyond human beings".

- John Fitzgerald Kennedy,
Address to the American University,
Washington D.C., 10th June, 1963.

"The reasonable man adapts himself to the world;
The unreasonable man persists to adapt the world to himself.
Therefore, all progress depends on the unreasonable man.
The future belongs to the unreasonable man, who looks forward, not back,
Who thinks the unthinkable, and is certain only of uncertainty".

- George Bernard Shaw

"Study the Science of Art, and the Art of Science,
Develop all your senses, particularly how to See,
Study all the above in the light of the continuing knowledge,
That everything connects, in some way, to everything else".

From the Notebooks of
Leonardo da Vinci,
(dated around 1513)

"Think sideways!"

Edward de Bono
("Lateral Thinking")

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Automatic Parallelization for Supercomputers

There are at least three ways to solve a problem using a parallel computer. You can write from scratch an original parallel program, you can adapt an existing parallel program that solves a similar problem, or you can detect and exploit any implicit parallelism inherent in an existing sequential program. Each method has its place. Unless you are the first person trying to solve a particular problem by using a computer, a sequential algorithm already exists. To avoid re-inventing the wheel, so to speak, you may be able to make use of the work of others. Perhaps the sequential program can be made parallel in a straightforward manner.

Writing programs for parallel computers is a time-consuming and error-prone task. To implement the program to run efficiently the programmer needs to have detailed knowledge of the underlying machine architecture. Features such as the memory hierarchy, inter-connection topology, bandwidth size, processor architecture, task scheduling and processor allocation, all have to be taken into account in the design and implementation of the program.

Manual translation of existing sequential programs into parallel form can be even more tedious. Along with the features of the machine architecture the programmer also needs a clear understanding of the functionality of the sequential program before being able to rewrite the program in a suitable parallel form. If the sequential implementation was poor or particularly complicated, then identifying areas of implicit parallelism can be equally difficult.

Interactive parallelizing tools are available but here the programmer also needs to know about parallelizing transformations, how to combine them, and the affects they have on program performance - all within the constraints imposed by the program dependencies. In short, if the program is to be parallelized to achieve an acceptable level of performance it needs to be customized for each individual architecture it is to run on. This presents a major barrier to producing portable parallel software and

hence increased use of parallel computing generally.

Source to source program restructuring allows a single incarnation of a sequential program to be transformed at source code level so that it is more able to take advantage of the capabilities offered by any given target architecture. In this thesis we lay the foundations for an evolutionary approach to automatic parallelization. We specify how the problem of automatic parallelization can be encoded into a form suitable for manipulation by an evolutionary algorithm. We develop genetic operators to transform programs and - more importantly, show how problem-specific information can be incorporated to guide the evolutionary algorithm so as to produce efficient, parallel code, fast.

1.2 Model of Automatic Parallelization

In order to define the problem of automatic parallelization we first define a number of terms and symbols used throughout this thesis.

S	The sequential program to be parallelized
S^i	The i^{th} restructured (but still sequential) version of S, where $i \in \mathbf{Z}^+$.
P^i	The parallelized version of S^i, $i \in \mathbf{Z}^+$.
P	The final parallelized version of S.
AP	The automatic parallelization function.
Res	The restructuring (but not parallelizing) function.
Res^i	The i^{th} particular sequence of restructuring optimizations and transformations, $i \in \mathbf{Z}^+$.
Par	The parallelizing (but not restructuring) function.
$Arch^S$	Source sequential architecture.
$Arch^T$	Target parallel architecture.
p	A generic computer program (parallel or sequential).
EXT	Estimated execution time of a program p.
AXT	Actual execution time of a program p.

SPE	Static Program Estimation function.
$p \equiv q$	Asserts the *semantic equivalence* of programs p and q.
$p \neq q$	Asserts the *semantic inequality* of programs p and q.
PDA	The Program Dependency Analysis function.
PDG	The Program Dependency Graph.
$STAT(p)$	The set of all statements in a program p

A number of optimisations are considered 'standard' in a modern restructuring compiler. Consequently we need to define a catalogue which consists of a set of one or more optimisations and transformations which are available to a restructuring compiler.

CAT A compiler catalogue of one or more optimisations and transformations.

It is also worth noting that CAT will, in practice, be a subset of the set of all possible program optimisations and transformations ($ALLTFs$).

$$CAT \subseteq ALLTFs$$

Hence, we are able to use the letters 'TF' to refer to a single instance of an undefined element of $ALLTFs$ (i.e. a single transformation $TF \in ALLTFs$).

We are now able to outline the typical automatic parallelization process using the functions and programs we have defined (Fig. 1.1).

A single sequence of restructuring transformations (Res) is an ordered sequence which may consist of zero, one or more transformations (TF).

$$Res^i := TF^*$$

(We use the star '*' to mean "zero or more instances of"). At any stage of applying a sequence of transformations to a sequential program S^i, we may apply the parallelize function to produce code for execution on a particular parallel architecture:

$$P^i := Par(S^i, Arch^T)$$

This code may then have its performance statically estimated by applying the static program estimation function:

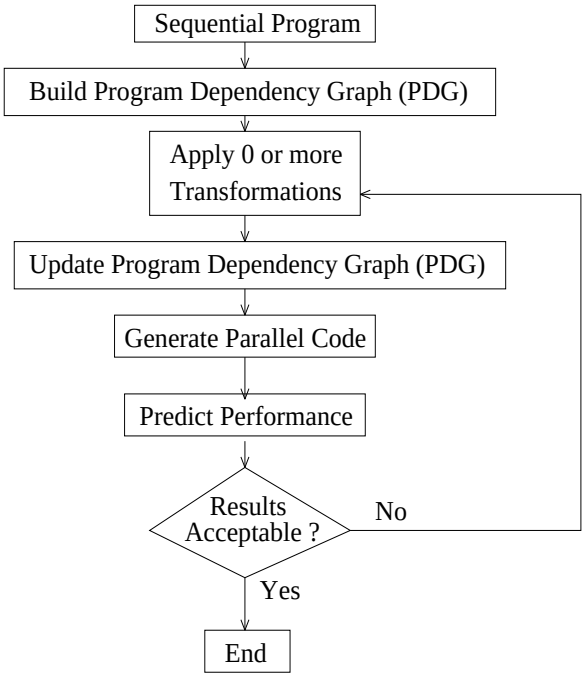


Figure 1.1: Typical Automatic Parallelization Process

$$EXT := SPE(P^i, Arch^T)$$

1.3 Problem Definition

The problem of automatic parallelization can now be defined as follows:

Definition 1.3.1. Automatic Parallelization. Given an initial sequential program S the problem of automatic parallelization (AP) is to apply some sequence of restructuring optimisations and transformations (Res^i) to S to produce some parallelized version (P) of S, such that $S \equiv P$ is $TRUE$, and that we have minimised $AXT(P, Arch^T)$, subject to the constraints imposed by the program dependency graph ($PDG(S)$) as determined by the program dependency analysis (PDA). Namely:

$$AP(S, Arch^T) = \text{MIN } AXT(P, Arch^T)$$

where : $P \in Par(S^i, Arch^T)$

$$S^i := Res^i(S)$$
$$Res^i(S) := Res(S^i)$$

such that: $PDG(S^i) \subset PDG(S)$ (essentially, the PDG of each of the restructured versions of program S, is a sub-digraph of the original program S.)

From this definition it is clear that the purpose of the AP function is not only to parallelize S into P but also to *minimize* the cost of AXT. Most notably our cost function here evaluates only the *execution time* of the resultant program, other users may wish to take other criteria into account (such as memory usage) in the formulation of their own cost function.

This definition of how to evaluate the performance of a program executing on a given architecture is clearly subject to the users' requirements and the domain of the application. Programs parallelized for different architectures may be evaluated for different features depending on the features of the architecture concerned. Perhaps the most common way to measure the performance improvement of a parallelized program is to measure the performance speed up factor:

Definition 1.3.2. Performance Speedup. In an ideal world, a program being transferred from running on a sequential machine (i.e with one processor) to a parallel machine with N processors will see the program run N-times faster. Performance speed up is often calculated as below:

$$\text{Speed Up} = \frac{\text{Execution time on a sequential machine}}{\text{Execution time on a parallel machine}} \tag{1.1}$$

In practice however, due to overheads such as scheduling of tasks, synchronisation, and communication between tasks (see section 1.3.1) the performance speed-up is usually less than perfect. Why this should be so is related to the architectural features of parallel machines and the limitations on parallelizing sequential algorithms. These are explained in the next section.

1.3.1 Hardware Architectures

Computer architectures may be classified in many different ways by focusing on different features of the architectures (such as number of processors, processor inter-connection topology, memory hierarchy and location, etc). Overviews of many of the wider parallel computing issues involved are presented in [138, 108]. Of the many classifications made (see [3, 64, 117]) it is *Flynn's classification* [49] which is the best known classification scheme for serial and parallel computer architectures. The model uses the twin concepts of instruction stream and data stream. An instruction stream is defined as a sequence of instructions to be performed by a computer; a data stream is a sequence of data items to be manipulated by an instruction stream. Flynn classifies an architecture by the capability of the hardware used to manipulate instruction and data streams. "The multiplicity is taken as the maximum possible number of simultaneous operations (instructions) or operands (data) being in the same phase of execution *at the most constrained component of the organisation*", (Flynn's emphasis). Four classes of computers result from this arrangement.

- *Single Instruction, Single Data (SISD).* Most sequential computers (including personal computers) come into this category. Although instruction execution may be pipelined (as in some workstations), computers in this category decode only a single instruction in unit time. A SISD computer may have multiple functional units, but these are under the direction of a single control unit.

- *Single Instruction, Multiple Data (SIMD).* Examples of SIMD computers include vector and processor arrays. A typical processor array for example, executes a single stream of instructions but on many processing elements, each element operating on its' own particular data items. The instructions are executed in discrete time steps, as dictated by a central processor clock, with each processor capable of fetching and processing its own data. The Thinking Machines CM-200, and the ICL Digital Array Processor (DAP) are both SIMD architecture

computers.

- *Multiple Instruction, Single Data (MISD).* There is no clear agreement about what computers fall into the MISD category since it is difficult to visualise how a computer can have one data stream being manipulated by multiple instruction streams, however, some people think pipelined and systolic array computers fit into this category. The term "systolic" is used by analogy to describe how a systolic array works - in the same way that the heart rhythmically contracts to pump blood, so a systolic array pumps data across arrays of processors (usually arranged in two dimensions). Each processor may modify the data before passing it on to the next processor which may be performing different operations on any data it receives.

- *Multiple Instruction, Multiple Data (MIMD).* This category covers most multiprocessors systems. The term MIMD is usually applied to computers containing multiple processing elements which are joined by connections (such as a *bus*) specially designed to transport data between processors and memory at high speeds, thus enabling the processors to work together on one program. The Sequent Symmetry, TC2000, n CUBE-2, Paragon XP/S, Meiko CS-1 and CS-2, Connection Machine CM-5, and SGI Onyx-2 are all MIMD computers.

Despite the obvious attractions of MIMD machines there are a number of overheads associated with their use as outlined below:

1. *Scheduling* is the task of organising N software *processes* to run efficiently across M hardware *processors*. This may be done *statically* at compile-time if the number of processes is known (or can be determined) in advance, or *dynamically* at run-time if new processes may be spawned during program execution.

2. *Synchronisation* is the task of two (or more) processes suspending their execution in order to send/receive a message (data) to/from each

other. One implication of this is that architectures with a coarser granularity* will require less time for synchronisation.

3. *Communication.* Communication costs are the amount of time processors spend storing and retrieving information to and from memory and/or other processes. Its costs are usually quantified as the amount of work that could have been done if the processors didn't have to communicate with memory or with each other. Communication is usually the largest overhead of a parallel computation. Importantly, the amount of communication within a program will more often be determined by the application and the chosen algorithm than by the architecture. As Amdahls' Law [4] states, most problems have some parts that simply *must* be executed sequentially.

Given a brief overview of the architectures of modern high performance computers, one soon realises that Flynn's classifications are gross, in the sense that many machine architectures contain elements of more than one category.

Alongside Flynn's classification, further distinctions can been made on the structure of parallel computers. (Flynn's classification was notably extended by Quinn [116, 117]).

Definition 1.3.3. Amdahl's Law. If s is the fraction of operations in a computation that must be performed sequentially, and p is the fraction of operations in a computation that must be performed in parallel, then the maximum speedup that can be achieved with N processors is

$$\text{Max Speed Up} = 1/(s + p/N) \tag{1.2}$$

In other words,

$$\text{Parallel Execution Time} = \frac{\text{Parallel part}}{\text{No. of Processors}} + \text{Sequential part} \tag{1.3}$$

*Granularity refers to the size of a processing elements sub-task: 'fine grain' is a small task, 'coarser grain' is a larger task involving more computations.

The consequences of Amdahl's Law [4] are far-reaching. Suppose you have a parallel program which has a section that must execute sequentially. Say this sequential part takes up only 5 % of the total execution time. Amdahl's Law says that *no matter how many processors you use* the largest speed up factor you can achieve is only 20. Hence, this law represents an upper bound on the speed up that may be achieved with your current program and expresses the fact that the inherent sequentiality of an algorithm is the ultimate limiting factor for its performance on any machine.

Definition 1.3.4. Gustafson-Barsis' Law. Gustafson-Barsis' Law [60] takes an alternative view to expressing achievable speed up - it states that a problem *scales* with the number of processors available. Consider a program that consists of two parts s and p which now represent the times spent executing the sequential and parallel parts respectively on a *parallel* machine, i.e.

$$\text{Parallel Execution Time} = s + p \tag{1.4}$$

Given N processors we can see that executing the program on a sequential machine will take time $s + Np$.

$$\text{Scaled Speed Up} = s + Np \tag{1.5}$$

Hence the parallel part of the program has scaled linearly with the number of processors used. Unlike the steep curve defined by Amdahl's Law, the *Scaled Speed Up* metric defined by Gustafson-Barsis' Law is a moderately sloped, straight-line function.

The gloomy scenario created by Amdahl's Law coloured many people's opinion of parallel computing for many years. Yet Gustafson-Barsis' Law shows that although Amdahl's Law cannot be broken - it can be worked around. The definition of Amdahl's Law contains an implicit assumption that most people will try to speed up their program simply by making more processors available to it (thereby reducing p, ideally to zero). Gustafson-Basis points out however that code can be rewritten or *restructured* in such a way so as to also reduce the value of p, and if it can be reduced to zero, then the

speed-up will scale linearly with the number of processors available. This explains the popularity of SPMD programming in recent years.

Shared Memory / Distributed Memory

In a shared memory machine, all processors share one global address space to which they can all read and write data. The advantages are clear, no need for data to be sent between processors since a simple read or write to memory is sufficient, The main drawback is that the shared bus (which carries date between processors and memory) soon becomes overloaded and controlling consistent access to the memory space becomes infeasible. Consequently, most shared memory machines have comparatively few processors. Expensive hardware components have been developed (such as wider bandwidth buses and multi-layered cache memory) to try to overcome this problem but with only limited success.

The distributed memory model, by contrast, allows each processor to have its own private memory module as part of a simple and inexpensive processor array. Data required, which is not in a processors local memory, now needs to be retrieved from across the network - this results in non-uniform memory access times, and the requirement for explicit message-passing communication instructions (sends/receives) to be inserted into software.

Multiprocessors / Multicomputers

The main feature of a multiprocessor (or *loosely coupled*) architecture, is that a logically shared memory model is presented to the programmer. The two multiprocessor models are the UMA (uniform memory access) model, where the memory is physically centralised, and the NUMA (non-uniform memory access) model, where the memory is physically distributed.

Multicomputers have no shared memory. The programmer is presented with a distributed memory model, each processor has its own private memory and processors interact through message-passing.

In accordance with Flynn's classification, it becomes clear that one of the most

important factors affecting a computers performance is its *inter-connection topology* or *inter-connection network*. This is determined by the number of processors the machine has, and the different ways in which they may be connected. The degenerate case is that of the SISD architecture - one processor, no connections to others. With the SIMD model, the processors are often connected in an array (as in Fig. 1.2), where one processor controls the execution of all the others. All the processing elements (PEs) execute the same instruction in synchronised lock-step sequence - but work on different sets of data, which have been sent to them by the controlling processor. Some problems (and the algorithms used for them) do not map well onto this structure, so other SIMD topologies have been developed, such as:

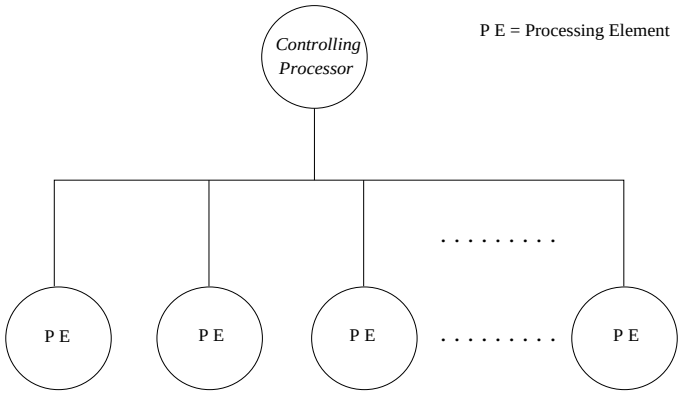


Figure 1.2: SIMD Processor Array Architecture

- *Meshes* (see Fig. 1.3) are regular arrays of processors, which may be connected in 2 or 3 dimensions. They may also have toroidal connections which allow faster transfer of data across the mesh and are often used in simulation, weather prediction, and matrix manipulation type applications.

- *Trees* (see Fig. 1.4) are structures which may vary in their depth (or *height*) and their arity (or *degree*). The arity of a tree is the number of sub-nodes attached to each node (i.e. a binary tree has arity = 2, ternary tree has arity = 3, etc). The trees shown in Fig 1.4 all have depth = 2. The pyramid is a 'hybrid' interconnection network in that

it is a combination of a tree (arity = 4) with a 4 × 4, 2-D mesh.

- *Cubes* (see Fig. 1.5) A 3-D cube is really a variation on the basic 2-D mesh (i.e. it is really just a different way of connecting together 8 processors). The idea has been extended to 4-D hypercubes (16 processors) and to cube-connected cycles.

Other examples of inter-connection topologies (see Fig. 1.6) include:

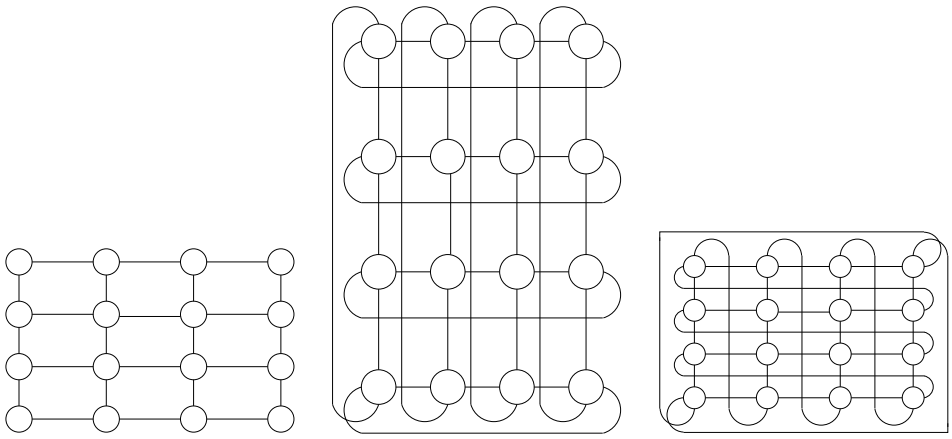
- *Pipeline.* Like a production line in a factory, data flows into the pipeline at one end, is manipulated as it passes along separate stages, before finally emerging in its final form from the final stage of the pipeline. The parallelism comes from the stages of the pipeline which are all executing concurrently but are working (at any one time) on different items of data.

- *Butterfly.* The BBN (Bolt, Beranek and Newman) TC2000 is a NUMA multiprocessor that has upto 128 processor nodes. It is so named, because of its 'butterfly wings' appearance.

- *Ring of Rings.* The first commercial virtual shared memory machine was the KSR-1 (from Kendall Square Research [40]). Its processors were arranged in rings of upto 32 processors each with an extensive cache and memory management system used to present the consistent model of a single memory address space.

Examples of typical processors used in high performance computers include: DEC Alpha's (Cray-T3D), IBM RS/6000 (IBM SP-2), Sun SPARC processors (Meiko CS-2), Intel i860-XP (Intel Paragon), Motorola MC68000 (BBN Butterfly), and SGI MIPS R10000 (SGI Onyx-2).

One increasingly popular new architecture is the Symmetric Multiprocessor (SMP) model (see Fig. 1.7). This model [110] consists of multiple processors connected by a simple bus, one shared-memory address space and one point of input/output (I/O). The symmetry comes from each processor having essentially the same 'view' of the

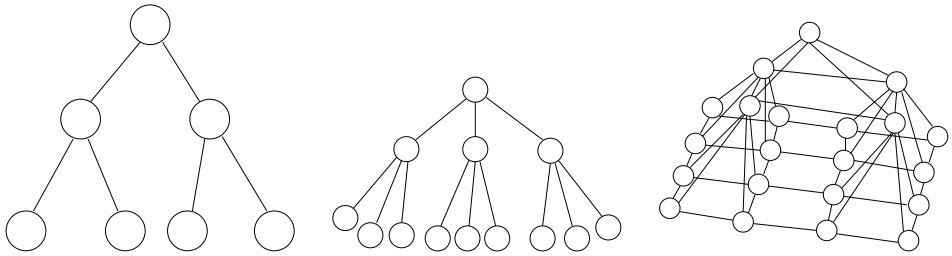


Simple

plus Same ROW/COL Wraparound

plus Adjacent ROW/COL Wraparound

Figure 1.3: Examples of 2-D Mesh Architectures.

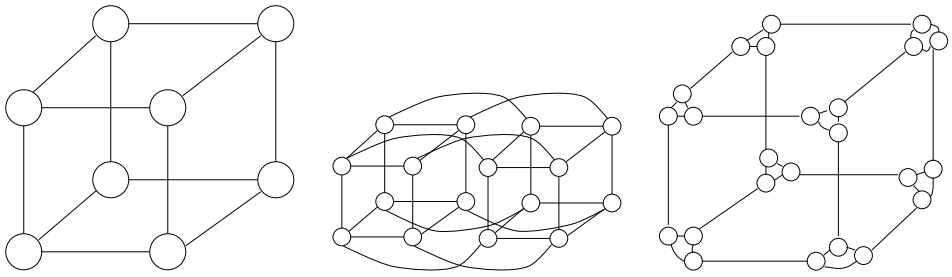


Binary Tree

Ternary Tree

Pyramid

Figure 1.4: Variants of Tree Architectures.



Cube

4-D Cube

Cube-Connected Cycles

Figure 1.5: Variants of Cube Architectures.

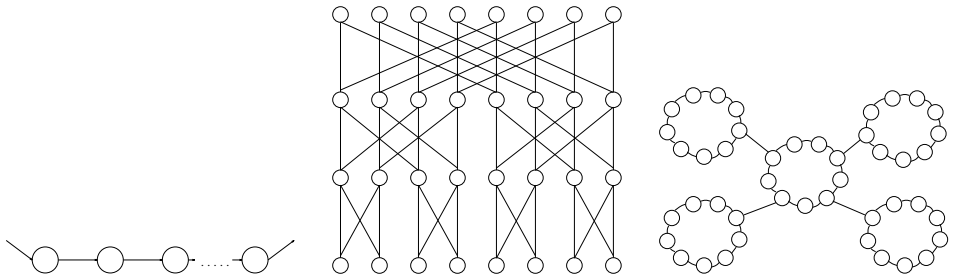


Figure 1.6: Examples of Other Architectures.

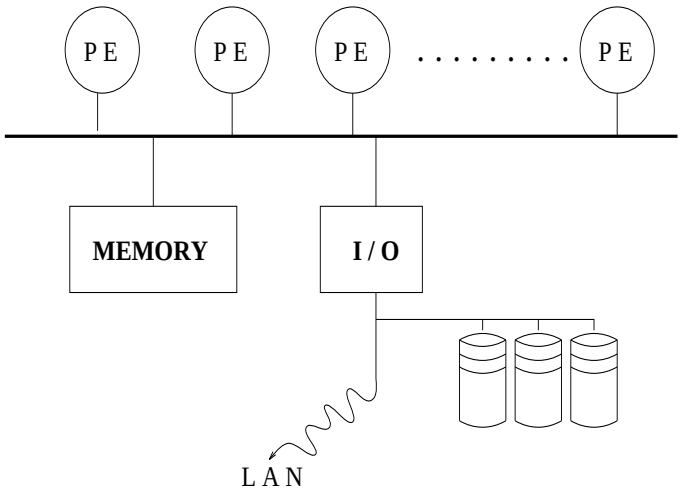


Figure 1.7: Symmetric Multiprocessor (SMP) Architecture

system resources - i.e. each has equal access to/from memory (there are no private processor memory modules, other than local caches) and I/O requests are served on a first-come first-served basis (i.e. no parallel I/O devices are used). The attraction is the modularity of being able to add/remove processors as necessary - the downside (as with most UMA architectures) is the difficulty of maintaining cache coherency.

1.3.2 Software Architectures

To exploit the full capacity of a parallel machine it is necessary to map the software model onto the hardware architecture with a good (ideally one-to-one) correspondence between software *processes* and hardware *processors*. For many problems, several parallel algorithms may exist, each works according to some model of parallelism (e.g.

pipelined, mesh, master/slave). In order to achieve best efficiency however, the algorithm should match the architecture. This ensures that all processors are utilised and good efficiency is achieved. For example, there are several algorithms to multiply two matrices together, but an algorithm that performs efficiently on one architecture may not be the most efficient for another. This may be due to the different communication capabilities of the two architectures, the synchronization mechanisms may also be different, and also the size of the architectures may have an effect.

In the same way that hardware architectures may be classified, so too may software models of computation be classified along similar lines. Indeed many analogous models of hardware architectures may be found in software (and vice-versa). The software *master-slave* model of computation, with its controlling master process and its cloned slave processes all executing the same program but working on different data (see Fig. 1.8), is almost the direct equivalent of basic SIMD hardware architecture depicted in Fig. 1.2.

In some ways, so too is the Single Program Multiple Data (SPMD [36]) model of computation. In this model, each process runs the same executable program, without any controlling master process, however, the processing may execute different statements by taking different branches within the program.

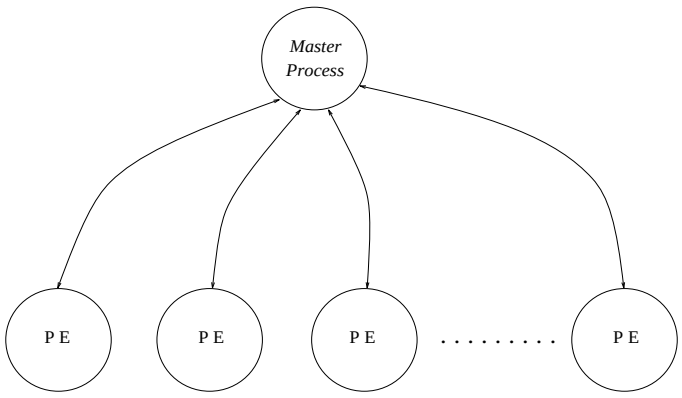


Figure 1.8: Master/Slave Software Architecture

Cluster Computing / Networks of Workstations (NOWs)

Another recent development in computing architectures has been the appearance of cluster computing. This involves using specially created software libraries to allow networks of workstations (clusters) to communicate and work together as a single computational resource. Hence, these libraries (such as Parallel Virtual Machine (PVM) [18], and Message Passing Interface (MPI) [96]) can give the user the apparent performance of a supercomputer by using spare cycles on under-used workstations. PVM efficiently runs compute-intensive programs such as those often found in matrix and numerical processing [134], Monte Carlo simulations [137, 136], evolutionary algorithms [66, 53], and other engineering and scientific applications.

Desirable Features of Parallel Programs

Across the diversity of parallel computing architectures that now exist, it is still possible to identify generally desirable features of a 'good' parallel program. These include:

- Minimise communication costs in terms of both the *number* of messages sent (since message start up times are usually expensive), and also their *sizes* so as to minimise network traffic.
- Workload is as evenly balanced as possible within the constraints of the results of the dependency analysis. If a parallel version is suitable, then ideally each processor should perform the same amount of work.
- Efficient scheduling of processes - ideally map one process to each unique processor available (as in the SPMD model of parallelism). Ensure as far as possible that no processors are wasted or underused.
- Efficient exploitation of the memory hierarchy - where suitable cache usage and data-distribution techniques may be employed to reduce overheads incurred by extensive memory usage.
- Minimise synchronisation costs. No process should have to wait for a message to be received from another process any longer than necessary (and ideally not at all).

The challenge for the evolutionary approach is to identify where and how all this problem knowledge can be incorporated into an evolutionary algorithm making it capable of evolving sequential programs into an optimised parallel form.

1.4 Thesis Methodology

In this section we explain the motivation and objectives of this research, and the performance metrics used.

1.4.1 Criticisms of Current Techniques

Automatic parallelization is an important and difficult problem [23, 43]. It is essentially an optimisation problem where the objective is to find a sequence of program transformations which translate a program from sequential into parallel form and minimize a set of overheads (such as communication, synchronization, scheduling, etc) associated with that program. Current methods however lack an effective organising framework.

Selection of transformations to apply is usually done by either (i) a restructuring compiler using predefined heuristics (ii) a programmer using an interactive restructuring tool, or (iii) a combination of both. Hence the effectiveness of the process depends on the heuristics hard-coded into the compiler, and / or the programmer understanding dependency analysis information, the effects of loop restructuring transformations, and having extensive knowledge of the functioning of the program being restructured and of the underlying target architecture.

A large number of program restructuring transformations are available. Which ones should be applied, in what order, and to which parts of the program? Deciding this is particularly difficult in large applications.

Some transformations have little or no impact on the program's performance themselves (indeed they may even have a negative effect) however, they may enable other transformations to be applied which do have a positive effect on program performance. These *enabling transformations* are particularly difficult to identify.

Most current parallelizing translators restructure loops in isolation of each other (except nested loops). As such they may miss opportunities to discover further parallelism

within the program. The evolutionary technique allows us to approach parallelizing a program *holistically*, that is, to restructure the whole program in the search for the optimal parallel version of the program.

Other problems with restructuring compilers identified in the literature include:

- Loop transformations not being applied to the most important loops in the program (important here usually meaning loops in which the program spends most of its execution time).
- The application of one transformation may prevent the application of other, possibly more beneficial transformations, the *interference problem*.
- The heuristics used by the compiler may incorrectly decide that a transformation is not worth applying.
- The inappropriate application of some transformations may have negative effects on the performance of the code produced.
- Timeouts in the compilers' search for the optimal sequence of transformations

1.4.2 Motivation for Evolutionary Parallelization

There are powerful reasons for taking an evolutionary approach to automatic parallelization. The motivation for the development of an evolutionary algorithm (EA) into an evolutionary parallelizing compiler (EPC) can be summarised as follows:

- No interaction with the user is required, and the user needs no specialist knowledge of the architecture of the target machine, the function of the program being restructured, or any of the processes involved in automatic parallelization. Requirement of such specialized knowledge has always been a big obstacle to the increased use of parallel computing.

- Using the evolutionary approach the restructuring compiler is more likely to find '*enabling transformations*' (i.e. transformations which by themselves have little, no, or possibly even a degrading effect of program performance but which allow other transformations (or sequences of transformations) to be made which *do* produce significant improvements in program performance. Conceptually, such transformations represent non-linearities in the search-space of the AP function. These transformations can be easily missed by a programmer using an interactive tool.

- Extensibility: any new or future analysis, optimization, restructuring or parallelizing techniques that are developed can be simply added to an evolutionary parallelizing compiler. Consequently, any future computer architectures developed may consequently be targeted by the development of such appropriate transformations.

- The evolutionary framework can be extended to incorporate *any* program optimising or restructuring transformation - such as loop restructuring transformations, data-flow optimisations, vectorising transformations, memory/cache usage optimisations, and data-layout transformations, and others. The evolutionary approach is high-level enough not to be restricted to any particular language, application, or architecture. The REVOLVER system for example, is targeted at a message-passing machine - other work is already underway to target shared-memory architectures ([101, 102]).

- A minimal lower-bound can be put on the quality of the solution produced by the evolutionary parallelizing compiler. Like any search technique, its performance can be improved by the use of heuristic information and the incorporation of existing heuristics into the evolutionary parallelizing compiler can guarantee that the solution found will be at least as good as that found by existing techniques. Extensive research

into automatic parallelization over the years has lead to the development of a large body of heuristic and analytical techniques which may be included into the EPC.

- Using an EA allows the EPC to automatically search a wider area of the solution space, faster than possible compared to a programmer using an interactive tool. It is also less likely to be trapped by local minima since it is based on the knowledge of more than one person.

- Like all evolutionary algorithms (EAs), and unlike existing parallelizing compilers, an evolutionary parallelizing compiler can execute in parallel or sequentially.

- Evolutionary algorithms have proved effective at optimizing constrained functions with multimodal, discontinuous solution spaces, similar to those characteristic of the automatic parallelization function.

- Despite the highly irregular solution space of the problem of automatic parallelization it should be noted the EA need only find *one* good solution - there is no requirement for the overall population to be highly fit, or to converge towards a global maxima.

- There is no need for the User to wait for the EPC to run to completion. The EPC may be configured to give them the best solution it has found so far after some set time limit, or number of generations. The EPC may then be terminated or continue to run as a 'background process' if the User wishes.

- The use of clusters of workstations connected by a communications library (such as (PVM [18]) or (MPI [69, 96])) to run the Revolver compiler illustrates how programs can be restructured "off-line" away from the target architecture, thereby saving on the use of expensive computing resources.

- The evolutionary parallelizing compiler can be used as a test-bed for developing new techniques for automatic parallelization. Detailed analysis of the effectiveness of the a new technique, alone, and in combination with other techniques can then be quickly performed.

- The EPC could be enhanced with *rule inference* techniques so as to act as the front-end for a knowledge acquisition tool and as such attempt to identify particular 'types' of loops and develop rules on how best to parallelize them for the given architecture. Such rules would encapsulate this information and could be added to improve the performance of smaller, non-evolutionary parallelizing compilers.

Other issues include:

- Evolutionary approach relies on parallel program performance prediction techniques to guide improvements (i.e. the 'fitness function'). This may prove difficult when trying to automatically parallelize programs for multi-user architectures (e.g. clusters of workstations) but this problem also exists for all existing restructuring compilers.

- No existing work on restructuring compilers has taken an evolutionary approach. Most work has taken a deterministic approach, often adapting linear programming techniques.

- Executing the EPC (sequentially or in parallel) will take longer, and use more computational power than existing parallelizing compilers. However (i) the time requirement can be offset by the user receiving intermediate 'best solutions so far' found by the EPC (ii) the computational requirements are not essential (like any program the EPC will simply run faster the more memory and processors, etc, the machine has available). In order to receive higher quality solutions the EPC will have to execute for a longer period of time than most compilers - however, it is felt that most users will accept this, so long as they know the final parallel code produced will be efficient.

This thesis does not argue that evolutionary algorithms are suitable for all machine translation tasks, it does however take the stand that automatic parallelization is a special case because of the great importance attached to restructuring code to achieve the highest degree of performance possible and the expensive costs of failure to do so.

1.4.3 Thesis Objectives

This thesis has several objectives which span the fields of evolutionary algorithms, compiler design, and automatic parallelization. The main objective is to determine whether an evolutionary approach could be used to effectively translate sequential Fortran programs into parallel form.

Other aims include:

- To investigate in what ways can the forces of evolution be brought to bear on the problem of automatic parallelization - and determine which ways are the most effective.

- To identify what representations may be used in the evolutionary algorithm to encode the problem?

- To characterise the shape of the solution-space for typical automatic parallelization functions - is it unique for every program, and what factors influence its appearance?

- To experiment with new hybrid-operators and evaluate each for their effectiveness.

- To identify points in the evolutionary process where restructuring heuristics can be included to the best effect.

- To identify what conditions make it easy / difficult for the EA to parallelize sequential code.

1.4.4 Performance Metrics

The main performance metric was the 'quality' of the parallel code produced. In simple terms this was an estimated execution time of one parallelized version of the sequential program compared against one or more others. Whichever program had the estimated shortest execution time was taken as performing better than the others.

Other metrics include:

- The 'robustness' of the hybrid-EA produced, that is, the ability of the algorithm to consistently parallelize efficiently a large number and wide variety of Fortran programs.
- The 'efficiency' of the hybrid-EA. The time to produce quality code should not be prohibitive[†] and the EA should not get trapped in local optima when searching the solution-space.
- The size and type of the Fortran application should not prevent it being parallelized using evolutionary techniques (any more than for conventional techniques).
- Identification of useful subsequences of transformations which can be useful to programmers using interactive restructuring tools.
- Finally, we wanted to see how well the EA worked in comparison to traditional parallelizing techniques.

1.5 Thesis Outline

In chapter 2 we introduce the basic concepts used in automatic parallelization, defining terms, program analysis techniques and optimisations with examples where necessary. The chapter then moves on to the related aspects of program profiling and performance estimation techniques.

[†] Although the User can set a maximum time limit the automatic parallelization process is allowed.

Chapter 3 gives an introduction and overview of the basic concepts in evolutionary computation. Evolution strategies and genetic algorithms are outlined and all related terms are defined, as necessary.

Chapter 4 presents how we combined the ideas of automatic parallelization and evolutionary computation into an evolutionary parallelizing compiler (EPC) and describe the features of the Revolver EPC system.

In chapter 5 we provide details of experiments and the raw results which were carried out with the Revolver system and perform some statistical analysis of the results.

In chapter 6 we present a more high-level interpretation and discussion of the results and discuss alternative evolutionary techniques which may also be applied to automatic parallelization and give indications of the extensive future research that needs to be performed in the field of evolutionary parallelization.

Chapter 7 summarises this research project, formulates important conclusions based on our results and ends with reference back to our original research aims and objectives.

2 Automatic Parallelization

2.1 Introduction

In this chapter we present a high level description of both sequential and message passing parallel programs. We define the basic concepts used to describe the work in this thesis and outline the subset of the Fortran-77 programming language used as the input language in the parallelization process. We then provide a comprehensive overview of the automatic parallelization strategy used, and conclude with a detailed summary.

2.2 Basic Concepts

Definition 2.2.1. Program Data Space. We define, a set D to be the data space of all declared arrays and scalar variables that are accessible within a given scope of a program p. Formal parameters are not considered part of D. Each element of an array is associated with exactly one $d \in D$ as are scalar variables.
We can further define D to be the union of two subsets AD the set of all array data accessible within a given scope of program p, and similarly for SD the set of all scalar data. Hence D can be further defined as $D = AD \cup SD$.

Definition 2.2.2. Universal Domain. Let U represent the set of all possible values (integers, reals, chars, logicals, complex, etc), including $UNDEF$ for the undefined value, that a data space element $d \in D$ (i.e. a variable) may take in program p.

Definition 2.2.3. Program State. Using the concept of a 'universal domain' we can say that as program p executes it passes through a number of states, each state recording a value $val \in U$ for each $d \in D$ in p. We define $STATES(p)$ be the set of states program p passes through as it executes.

Definition 2.2.4. Array Index Domain. The index domain Dom of an array A in dimension $i \in \mathbf{Z}^+$ is represented by a non-empty, linearly ordered set of integers of the form $Dom_i = [l_i : U_i]$ where $l_i \leq U_i$ and $[l_i : U_i]$ denotes the sequence of integers $(l_i, l_i + 1, \ldots, U_i)$. Hence, l_i and U_i represent the lower and upper bounds of array A in dimension i.

Definition 2.2.5. Index Domain. Let $A \in AD$ be a declared array in a program p. We can make the following assertions;

- A will be associated with i index domains, where $i \in \mathbf{Z}^+$ is the number of dimensions of array A.
- The elements of A are scalar objects.
- For every array element $e \in \mathbf{Z}^+$ in a dimension of A, $indexA(e)$ reduces to the index of e in that dimension of A.

Definition 2.2.6. Tokens, Symbols and Operators. When performing linear analysis of a program, a *token* is a sequence of characters which have a collective meaning [1]. Tokens may be symbols, (such as numeric values, variable names, or function/subroutine names), or keywords of the language (such as `IF`, `for`, `DO`), punctuation symbols required by the language (such as ';'), or else binary or unary operators (such as ':=', '==', '+', '**', etc). The exact meaning assigned to an operator is language dependent. Operators connect *expressions* to form more complex expressions in ways as determined during parsing. Keywords and some operators connect expressions to form whole *statements*.

Definition 2.2.7. Expressions. An expression is any combination of tokens, symbols (variables), parentheses, and arithmetic operators which may be evaluated in accordance with the precedence rules of normal algebra.

Example 1. Some examples of arithmetic expressions are:

6 (A single variable or number is an expression)

$a + b$

a * ((3.75 + b) ** 2 - c)

Our definition, in accordance with Fortran-77 grammar rules, allows for embedded function calls.

$x = func(a)$

Notably, since our input language is a subset of Fortran-77, no nesting of assignment expressions is allowed (unlike `C`). Hence:

$x = a$ * ((3.75 + b) ** 2 - c)

... is a *statement*, which consists of an assignment operator, in between two expressions. Other examples of statements include `CALL` statements, `DO` statements, variable declaration statements, `IF` statements, etc.

Definition 2.2.8. Affine Expressions. An affine function is a linear function plus a constant value. Where a linear function $f : \mathbf{R}^m \longrightarrow \mathbf{R}^n$, may be represented in matrix form as $f(x) = Fx$ an affine function may be represented as $f(x) = Fx + f_0$. In particular, when analysing array subscript expressions, we want to find the maximum and minimum values that an integer affine function can take for integer valued arguments in a bounded region $\mathbf{R}^m$ where the bounded region may be characterised by a system of integer, linear inequalities. In general this is equivalent to the integer programming problem.
Our definition here concentrates on *affine subscript expressions*, namely, subscript expressions that are affine functions of the loop induction variables. These are sometimes called *linear subscripts* since the expressions are linear combinations of the induction variables.

In the presence of a *non-linear subscript expression* a compiler may choose to conservatively assume the worst case for that dimension, or use a specialised case solver. Fortunately, non-linear subscript expressions do not occur frequently, although they are common in certain types of application (e.g. wavefront applications).

Example 2.

Given integer induction variables i and j, examples of linear and non-linear subscripts are:

A(6)	Linear (numeric constant)
A(i)	Linear
A(i-j+5)	Linear
A(i*j)	Non-linear (a product of two induction variables)
A(i/j)	Non-linear (a ratio of two induction variables)
A(MOD(i,5)+2)	Non-linear (a non-linear function)
A(B(i)-1)	Non-linear (an indexed subscript function)
A(10*i-1, 2*j)	Linear in both dimensions
A(2*i-1, i*j)	Linear, non-linear

Definition 2.2.9. Statement Instantiation. Let $inst(st)$ define an instance of the execution of statement $st \in STAT(p)$ in $STATES(p)$.
Hence each $st \in p$ (i.e. each statement in program p) is unique, and each $inst(st) \in STATES(p)$ (i.e. each instance of execution of statement st, within the execution of program p, as it passes through its states) is also unique.
This definition extends naturally to expressions in program p.

Definition 2.2.10. Memory Reference (or Access). An access is a particular appearance of a variable in a program p. The access may be of a scalar variable, or an element of an array in which case one or more subscript expressions must appear to identify the array element(s) being accessed. An access retrieving a value from memory is called a USE. An access storing a value to memory is called a DEF (definition).

Definition 2.2.11. USE / DEF Sets. Let $USE(st) \subset STAT(p)$ be the set of all Uses and $DEF(st) \subset STAT(p)$ be the set of all Definitions of all references in a particular statement st, where $st \in STAT(p)$.

Definition 2.2.12. Conditional Statement. Let $C \in STAT(p)$, where $STAT(p)$ is the set of all statements in a program p, be a statement such that C contains an expression which can be evaluated and assigned a truth value of $TRUE$ or $FALSE$ only. The result of the evaluation will cause the flow of control to jump to one of two other statements in program p, neither of which need be the lexicographically next statement in following C. Statement C is hence called a conditional statement and is in $COND(p)$ which is the set of all conditional statements in program p. Statement C may be assigned a $TFRatio$ under program analysis.

Example 3. Arithmetic, computed and logical `IF` statements in Fortran as well as `ELSE IF` statements, `if` and `switch` statements in C, and `CASE` statements in PASCAL and MODULA-2, are all examples of conditional statements. Note, an unconditional 'jump' statement, (such as `goto` in C or Fortran), is not a conditional statement.

Definition 2.2.13. TRUE / FALSE Ratio. Let $C \in COND(p)$ where $COND(p)$ is the set of all conditional statements in program p. Then $TFRatio$ is the probability that C evaluates to $TRUE$ in any given execution of p, such that;

$$TFRatio \in \mathbf{R}_0^+ \text{ and, } 0.0 \leq TFRatio \leq 1.0$$

Example 4.

Assume a program p contains statement st which contains a conditional expression and during the program run of p the statement st is executed N times, of which the conditional expression evaluates $TRUE$ a total of T times.

```
        ....
st:     IF (i .eq. j) THEN
              ....
        ENDIF
        ....
```

We can immediately infer that the condition evaluated false $F = N - T$ times and can calculate the $TFRatio$ accordingly:

$$TFRatio(st) = \frac{T}{N}, \text{ where } F, T, N, \in \mathbf{Z}_0^+$$

Definition 2.2.14. Frequency Count. Let $s \in STAT(p)$ then s is assigned a value $execs \in \mathbf{Z}^+$ recording the number of times statement s is executed (i.e the frequency) during the execution of program p.
The value of $execs(s)$ will be the same for all other program statements in the same basic block as s.

Definition 2.2.15. Basic Block. Let st_{enter} and st_{exit} be elements of $STAT(p)$. Then, a basic block B is a consecutive sequence of statements in which control flow may only enter at st_{enter} (the lexicographically first statement in B) and may only exit at st_{exit} (the lexicographically last statement in B) without halting or possibility of branching, except at the end. (Statement st_{enter} is sometimes called the '*leader*' statement of the basic block).
This definition of a basic block implies that all statements in B will have the same frequency count $freq(B)$.

Definition 2.2.16. Loop Iteration Statement. Let $L \in LOOPS(p)$ where $LOOPS(p)$ is the set of all loops in program p be a statement such that L may contain a body (or block) of statements which may be executed repeatedly in a counted format (the count condition being specified in L) or while a particular condition evaluates to $TRUE$ (i.e. a non-counted loop, such as a `WHILE` loop). Statement L is hence called a loop-iteration statement (or simply a loop statement). As with all statements in program p, L may be instantiated more than once in an execution of p.

Notably the number of iterations performed between instantiations of L may vary. This is because the number of iterations performed is dependent on the type of loop bounds L has. Loop bounds may be:

1. *Constant*: that is, integer numbers hard-coded into the program, or terms defined within the program as being constant and unalterable.

2. *Symbolic.* If a loop bound is symbolic it may be either *fixed* or *variable*;
   and also be a function (linear, or non-linear) of some expression.

3. Some combination of the above types.

Loop bound types are important since the type of bounds a loop has will have a major influence on how the loop dependency analysis is performed and in determining whether or not the loop can be legally parallelized. Examples of loop bound types are in Fig. 2.1.

```
L1:  DO i = 1, 10                    L1 - Fixed.
       ....
L2:    DO j = 1, N                   L2 - Symbolic (N),
         ..                          N may be either fixed or variable.
       ENDDO
L3:    DO j = i, N                   L3 - Symbolic (i and N),
         ....                        Lower bound i varies with each
                                     instantiation.
       ENDDO
     ENDDO

L4:  DO i = 1, 10 * (N / 2)          L4 - Linear symbolic
       ....                            Expression may evaluate
       ....                            to fixed or variable values.
     ENDDO

L5:  DO i = 1, MAX(-3, (21 - N * 5))   L5 - Non-linear
       ....                                upper bound.
     ENDDO
```

Figure 2.1: Examples of Some Loop Bound Types

Definition 2.2.17. Loop Nests and Perfect Loop Nests. If a loop statement is used within another loop statement a so-called loop nest results. If no other statements appear between individual loop statements then the whole loop is called a *perfectly nested loop*. If other statements do appear between individual loop statements then the loops are *non-perfectly nested*. Below a loop nest of depth d is illustrated:

```
DO I_1 = L_1, U_1
  ....
    DO I_d = L_d, U_d
```

```
      ....
    ENDDO
  ....
ENDDO
```

Definition 2.2.18. Loop Iteration Space. The loop iteration space for an N-deep loop nest is a sequential ordering of the iterations executed in a particular instantiation of a loop. Each iteration may be attributed with an identifying number, unique within that instantiation, which indicates the iterations position within the sequence of iterations that are executed. If the loop is in normalised form the control variable will start with a value of 1, and have a step-size of 1.

Loop iteration spaces may be represented visually, a single loop for example may be mapped onto a single directed line graph. Two or three-deep nested loops may be mapped into two or three dimensional discrete Cartesian space - the loop iteration variables provide the coordinates for each iteration.

Example 5. The loop iteration space for the following 2-D loop nest may be mapped into 2-D space (see Fig. 2.2) as a directed line. The loop index variables form the axes of the graph, the direction of the arrow indicates the execution order of the iterations.

```
DO i = 2, 10
  ....
    DO j = i - 1, 2 * i - 1
      ....
    ENDDO
  ....
ENDDO
```

The iteration space may also be written as a system of linear integer inequalities:

$$2 \leq i \leq 10$$

$$i - 1 \leq j \leq 2 \times i - 1$$

Which may be rewritten for easier manipulation in the following matrix form, (where the number of rows will be $2 \times D$, and columns $= D + 1$:

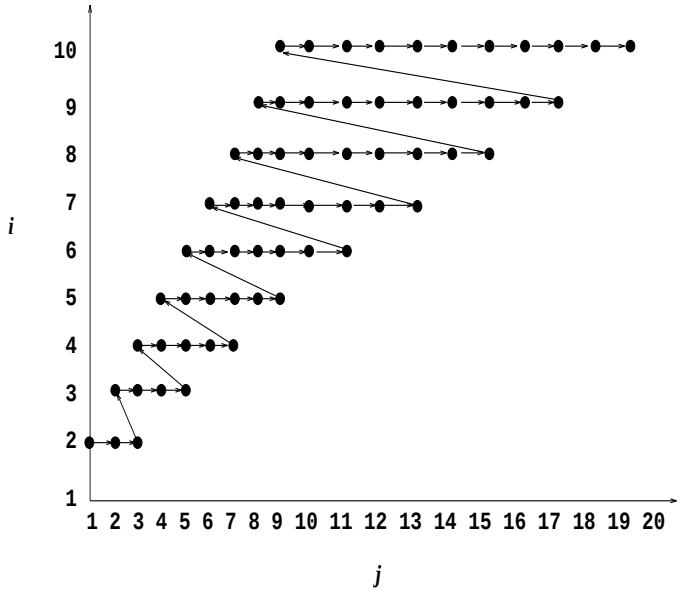


Figure 2.2: Loop Iteration Space of a 2-D Loop Nest

$$\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ -2 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} -2 \\ 10 \\ 1 \\ -1 \end{pmatrix}$$

The complete set of inequalities may be written in a single augmented matrix:

$$\left(\begin{array}{cc|c} -1 & 0 & -2 \\ 1 & 0 & 10 \\ 1 & -1 & 1 \\ -2 & 1 & -1 \end{array}\right)$$

Definition 2.2.19. Loop Iteration Count. Let L be a loop in $LOOPS(p)$, the set of all loops in program p, then a value $iters$ is assigned to the L which records the average number of executions of the body of L across all instantiations of L.

Example 6. Given a loop L with constant loop bounds and with a step-size of 1, then $iters(L) = UB - LB + 1$ (where LB and UB are constant valued expressions). If

the loop bounds are variable, then $iters(L)$ will equal the average number of iterations made across all instantiations of L.

Definition 2.2.20. Control Flow Graph (CFG). A *control flow graph (CFG)* (as described in [1]) is a directed graph G augmented with a unique *entry* node START and *exit* node STOP such that each node in the graph has at most two successors. We assume that nodes with two successors have attributes "T" (True) and "F" (False) associated with these outgoing edges. We further assume that for any node N in G there exists a path from START to N and a path from N to STOP.

Definition 2.2.21. Dominance. Given a control flow graph (CFG) (see definition 2.2.20), we say a node d in CFG *dominates* another node n, if every path from the initial node of the flow graph goes to n through d. Using this definition, every node dominates itself, and the entry node dominates all nodes.

2.3 Program Analysis

A typical parallelizing compiler uses three general techniques to analyze the source program.

- *Dataflow Analysis* is used to compute the values of program variables at different points of execution in the program.

- *Dependency Analysis* is used ensure the original behaviour of the program is preserved under restructuring transformations. This is done by identifying the underlying correct ordering of operations in the source program. This correct ordering can be encapsulated in graph form and imposes constraints upon the restructuring process since most restructuring transformations use dependency analysis information to determine whether a particular application of a transformation is legal.

- *Interprocedural Analysis* is used to encapsulate the read / write (or variables) information without having to inline the procedure itself.

The ultimate goal of this analysis is to be able to determine *exactly* the value of any variable at any point in the program execution (without executing the program itself). From this information, loop iteration counts, and True/False ratios of conditional statements can be computed and used to optimise the code produced by the restructuring process. Full representation of the exact flow of data through the program however is not always completely possible and where the compiler can only compute *inexact* information it must make conservative assumptions about the application of restructuring transformations.

2.3.1 Dataflow Analysis

Dataflow analysis involves the computation of *IN* and *OUT* sets for each statement (or basic block) in the source program. From here, further information (reaching definitions, *DEF / USE* sets, loop iteration counts, general flow of control information, etc) can be inferred, thus enabling a number of dataflow optimisations (such as common subexpression elimination, constant propagation, code motion, induction variable elimination, etc) to be performed.

The *IN* set of a statement S^1 is the set of memory locations (usually referred to by variable names) that *may* be read or used by this statement.

The *OUT* set of a statement S^1 is the set of memory locations (usually referred to by variable names) that *may* be written to or defined by this statement.

Note, that these sets include all memory locations that *may* be read or written to, and as such may be conservatively large in size.

Early development of dataflow analysis frameworks is discused in [100]. A comprehensive treatment of dataflow analysis frameworks is presented in [1]. SSA (static single assignment) forms are presented by Cytron et. al in [35], and Factored Use-Def chains (FUD chains) are discussed in [29, 144].

2.3.2 Dependency Analysis

Implicit in any computation is the idea that the computation will be performed in some particular sequence or order. If an input to one of these sub-computations is dependent on the output on another sub-computation then a dependency exists between the two sub-computations and they may not be executed in parallel. In short, dependence relations impose constraints on parallelism. The analysing of computer programs to identify these constraints is called *dependency analysis*. Dependencies come in two forms, (i) data dependencies, where memory accesses are required such that the ordering of operations must be maintained to preserve program correctness, and (ii) control dependencies, where the flow of control is conditional on the result of the evaluation of some boolean expression in the program.

Data Dependencies

Analysis of the data dependencies is a vital component of any parallelizing compiler. For example, given the block of code in Fig.2.3:

```
S1   i = 2
S2   j = i
S3   k = 2 * (l + i)
S4   l = 5
```

Figure 2.3: Code for Data Dependency Analysis

The compiler must be very careful in reordering these statements. Moving statement S^2 above S^1 would result in j being assigned some "old" (probably different) value of i. Similarly, executing S^4 before S^3 would result in the assignment to k possibly being performed with a different value of l. Notably however, statements S^2 and S^3 can be interchanged without changing the original programs' behaviour. The concept of *dependence* is used to encapsulate the fundamental ordering constraints of the program.

Definition 2.3.1. Data Dependency. Assuming a statement S^2 is *reachable* from S^1, the IN and OUT sets can be used to compute the data dependencies between the two statements.

$$
\begin{aligned}
OUT(S^1) \cap IN(S^2) &= S^1 \delta^t S^2 \quad \text{(True Dependency)} \\
IN(S^1) \cap OUT(S^2) &= S^1 \delta^a S^2 \quad \text{(Anti Dependency)} \\
OUT(S^1) \cap OUT(S^2) &= S^1 \delta^o S^2 \quad \text{(Output Dependency)}
\end{aligned}
$$

A fourth kind of dependence (an input dependence) imposes no constraints on the restructuring process but the information is sometimes useful to the compiler when performing certain optimisations.

$$
IN(S^1) \cap IN(S^2) = S^1 \delta^i S^2 \quad \text{(Input Dependency)}
$$

Example 7. A restructuring compiler typically represents data dependency information in the form of a directed graph (the *data dependency graph*, or DDG). Here, each node may represent an operation, a statement, or a basic block of code. Arcs connecting the nodes represent dependencies between the nodes it connects and is annotated to indicate the *type* of dependency that exists as well as indicating the *direction* of the dependency. Note also that this is a multigraph (i.e. two nodes may be connected by several arcs). The data dependency graph for the code in Fig. 2.3 is shown in Fig. 2.4.

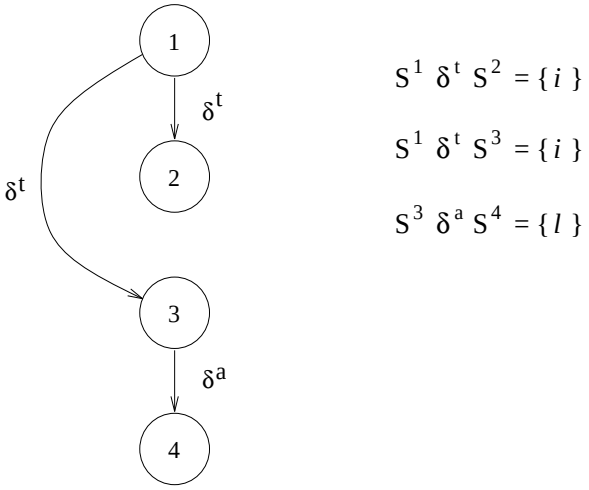


Figure 2.4: Data Dependency Graph and Sets

Control Dependencies

Where the flow of control of a program is determined by the evaluation of some conditional expression, control dependencies also must be calculated.

Definition 2.3.2. Control Dependency. If the execution of a conditional statement S^1 determines whether or not a statement S^2 is to be executed, then we say statement S^2 is *control dependent* on S^1 and the dependency is denoted by $S^1 \delta^c S^2$.

Example 8. Given the following block of code;

```
S1   ....
S2   IF (a .neq. 0) THEN
S3      d = a + 2
S4   ELSE
S5      d = c
S6   ENDIF
S7   ....
```

The execution of statements S^3, S^4, and S^6 are dependent on the conditional evaluation of S^2, hence control dependencies exist from S^2 to all these statements, denoted as: $S^2 \delta^c S^3$, $S^2 \delta^c S^4$, $S^2 \delta^c S^6$.

A restructuring compiler typically represents control dependency information in the form of a directed graph (the *control dependency graph*, or CDG). This is analogous to the data-dependency graph. In the CDG, each node may represent an operation, a statement, or a basic block of code. Arcs connecting the nodes represent control dependencies between the nodes it connects, as well as indicating the direction of the dependency.

Program Dependency Graph (PDG)

Once control and data dependencies have been computed these constraints can be combined into a single graph program abstraction called the *program dependence graph* (PDG). This is a directed multigraph which is typically calculated and then updated after each transformation is applied. It can be shown that the restructuring process is equivalent to finding isomorphisms to the PDG of the original sequential program.

Importantly, it should be noted that only true dependencies cannot be eliminated from programs. The other dependencies are the result of using the same location of

memory and can be eliminated by the use of temporary variables, hence, these are sometimes known as *artificial dependencies.*

A restructuring compiler will construct the PDG as part of the initial program analysis. It will then apply a number of program transformations to attempt to eliminate dependencies while preserving the behaviour of the original program, in order to expose implicit parallelism in the sequential code. It will also have to update the PDG incrementally after each transformation has been applied.

Dependence Analysis for Loops

So far we have only considered dependency analysis of individual memory locations. Dependency analysis of loops that iterate over arrays is the most fruitful area of analysis for exposing implicit parallelism.

Loop iterations can be executed in parallel without explicit synchronisation if and only if there are no dependencies between instructions belonging to different iterations, and the machine allows such access to arrays. To record this information, the data dependency graph (DDG) is annotated with information about the relative iterations in which dependencies occur.

Definition 2.3.3. Loop Carried Dependence. A data dependency which exists between two statement instantiations in different iterations of a loop is called a loop-carried dependence.

Example 9. A simple example of a *loop-carried* dependence is in the following piece of code:

```
        DO i = 2, N
S1          A(i) = A(i) + C
S2          B(i) = A(i-1) * B(i)
        ENDDO
```

There is no dependence between statements S^1 and S^2 within any given iteration of the loop but there is one between any two successive iterations. When $i = k$, S^2 reads the value of $A(k-1)$ which is written by S^1 in iteration $k-1$. To track loop-carried dependencies the compiler must analyze the subscript expressions in each array reference. To discover if there is a dependence in the loop nest it is sufficient to simply

determine whether any of the iterations can write a value that is read or written by any of the other iterations. Many loop dependence algorithms require that loops have only unit increment (i.e. a step size of 1). When they do not, the compiler may be able to normalise them to fit the requirements of the analysis.

Definition 2.3.4. Loop Independent Dependence. A data dependency which exists within one iteration of a loop is called a loop independent dependence.

Example 10. A simple example of a *loop-independent* dependence is in the following piece of code:

```
        DO i = 2, N
S1          A(i) = A(i) + C
S2          B(i) = A(i) * B(i)
        ENDDO
```

Here, the definition of `A(i)` in S^1 forms a *true dependency* with the use of `A(i)` in S^2. However, there are no dependencies which are carried across iterations of this loop.

Definition 2.3.5. Loop Dependency Distance and Direction Vectors. Distance and direction vectors may be used to characterize data dependences by their access pattern between loop iterations. If there exists a data dependence for $\alpha = (\alpha_1, \ldots, \alpha_n)$ and $\beta = (\beta_1, \ldots, \beta_n)$ then the *distance vector* $\mathbf{D} = (D_1, \ldots, D_n)$ is defined as $\beta - \alpha$. The *direction vector* $\mathbf{d} = (d_1, \ldots, d_n)$ of the dependence is defined by the equation:

$$d_i = \begin{cases} < & ,if \quad (\alpha_i < \beta i) \\ = & ,if \quad (\alpha_i = \beta i) \\ > & ,if \quad (\alpha_i > \beta i) \end{cases}$$

The elements are always displayed left to right, from the outermost to the innermost loop in the nest.

Example 11. For example, consider the following loop nest:

```
DO I = 1, X
   DO J = 1, Y
      DO K = 1, Z
```

```
            A(I+1, J, K-1) = A(I, J, K) + Z
          END DO
        END DO
      END DO
```

The distance and direction vectors for the true dependence between the definition and use of array `A` are (1,0,-1) and $(<,=,>)$ respectively. Since several different values of α and β may satisfy the dependence equations a set of distance and direction vectors may be needed to completely describe the dependences arising between a pair of array references.

Direction vectors, introduced by Wolfe [141], are useful for calculating loop-carried dependences. A dependence is carried by the outermost loop for which the element in the direction vector is not an '=' (i.e. an inequality). Additionally, direction vectors are also used to determine the safety and profitability of a loop interchange transformation. Distance vectors are more precise versions of direction vectors that specify the actual number of loop iterations between two accesses to the same memory location. They are often employed by transformations to exploit parallelism and the memory hierarchy.

Burke and Cytron [22] present a framework for dependence analysis that defines a hierarchy of dependence vectors and allows flow and anti-dependences to be treated symmetrically. An anti-dependence is simply a flow dependence with a negative dependence vector.

During analysis of a loop nest the compiler has to decide if two array references are might refer to the same element in different iterations. In examining a loop nest, the compiler:

- tries to prove that all iterations are independent by applying various tests to subscript expressions. These tests almost always rely on the fact that the expressions are linear.
- If dependencies are found, attempt to describe them with distance or direction vectors.
- However, if the subscript expressions are too complex, then the compiler must assume that dependencies do exist and that restructuring

is not possible.

Dependence Tests in Loops

It is infeasible to prove independence directly, even for linear subscript expressions because finding dependencies is equivalent to the $\mathcal{NP}$-complete problem of finding integer solutions to systems of linear diophantine equations [15]. However, two general and approximate tests are:

- The GCD Test [126].
- Banerjee Inequality Test [9].

A simple example of the *gcd test* follows:

Example 12. Test the outer loop for dependencies, given the two array accesses identified in the inner loop.

```
DO i = 1, N
  ....
    DO j = 2, 25
      A(i + 3 * j) = .....
      .... = A(i + 3 * j - 1)
    ENDDO
  ....
ENDDO
```

The normalised iteration vectors for the array accesses are constructed, as below:

$$\begin{aligned} 5 + i_1^d + 3j_1^d &= 4 + i_1^u + 3j_1^u \\ i_1^d + 3j_1^d - i_1^u - 3j_1^u &= -1 \end{aligned}$$

If $gcd(a, b, \dots) \mod k = 0$ then a dependency exists on the outer loop.

$$gcd(1, 3, -1, -3) = 1 \bmod -1 = 0 = TRUE$$

Hence, a dependency exists on the outer loop for the array accesses given. The result can be confirmed by unrolling the outer loop and computing the USE/DEF sets for the first few iterations.

Note that the gcd test tells us nothing about the type, direction, or distance of the dependency. Also, in practice, the `gcd` of a number of integer coefficients regularly reduces to 1, which divides exactly into any integer, so the test is usually not very effective. Details on more advanced tests can be found in section 2.3.3.

2.3.3 Dependency Analysis Research

Dependence relations are used in the restructuring process to identify the reordering constraints placed upon the execution of statements. These constraints are required to preserve the behaviour of the program as it undergoes restructuring to ensure that any parallelized program produced is semantically equivalent to its sequential original.

The efforts of dependency analysis research have increased over time directly in proportion to the increase in complexity of parallelizing compilers. Bernstein [20] used set theoretic notation to identify dependency relationships between basic blocks of Fortran code. The next advance came with the development of the 'hyper-plane' method of Lamport [82] for parallelizing Fortran `DO` loops. Perhaps the most influential contribution to dependency analysis came from Wolfe [107, 141] whose work included; identification of five basic dependency types (true, anti, output, input and control), development of the delta (δ) notation used to describe them, and a discussion of how code transformations could be used to alter the dependencies within a program in order to identify implicit parallelism in the code (in particular algorithms to eliminate anti and output dependencies).

The successful combination of the control dependency graph (CDG) and the data dependency graph (DDG) into the single program dependency graph (PDG) was first presented by Ferrante *et. al* in [47]. This paper showed how to compute control dependencies from data dependency information and went on to demonstrate how optimisations involving both control and data dependency information could be efficiently handled by the PDG.

It also became clear that more complex tests for dependencies within loops were needed and a number of loop dependency tests were devised; separability test, GCD-test [9], Fourier-Motzkin elimination*[133]. Additionally, there are a large number of

exact tests which exploit some subscript characteristics and can be used to determine whether a particular *type* of dependence exists. These include (i) the Single Index Test [9, 141], and (ii) the Delta (δ) Test [52] is a more general strategy that examines some combinations of dimensions.

Other tests that are more expensive consider multiple subscript dimensions simultaneously, such as: (i) the Lambda (λ) Test [86, 27], (ii) a multi-dimensional Banerjee test [9], and (iii) the Power Test [143],

One of the most advanced of these tests is the Omega Test devised by Pugh [114, 113, 115] whose development is ongoing as part of the Omega Project [71]. The `Omega` test library provides routines for analysing constraints over integer variables constructed using linear inequality constraints, set theoretic operations (union, intersection, etc), logical connectives (not, and, or), and existential quantifiers ($\forall$, $\exists$). This is known in the literature as Presburger arithmetic. The tightest upper bound on the complexity of verifying the satisfiability of Presburger formulae is $2^{2^{2^{O(n)}}}$ but in practice the heuristics used have proved very efficient, and the worst case rarely arises.

A library to perform operations on $\mathbf{Z}$-polyhedra has been developed (as part of the ALPHA [85] system and made available by Wilde [132]. The library implements geometric operations (union, intersection, simplification, etc), on structures called domains which consist of finite unions of convex polyhedra. These represent loop iteration spaces for manipulation by the library operations, from which restructured code can then be generated.

2.4 Program Optimisations and Transformations

A major goal of optimizing compilers for high-performance architectures is to discover and exploit parallelism in the original program. The way this is done is by applying transformations and optimisations to the program. The structure of a typical parallelizing compiler is shown in Fig. 2.5, this diagram is adapted from Bacon *et al*, [6]. There are numerous optimisations and transformations that can be applied, the most

*Fourier first proposed this method of solving a system of linear inequalities in 1826. It was reintroduced by Theodore Motzkin in 1936 and hence is usually referred to as Fourier-Motzkin elimination.

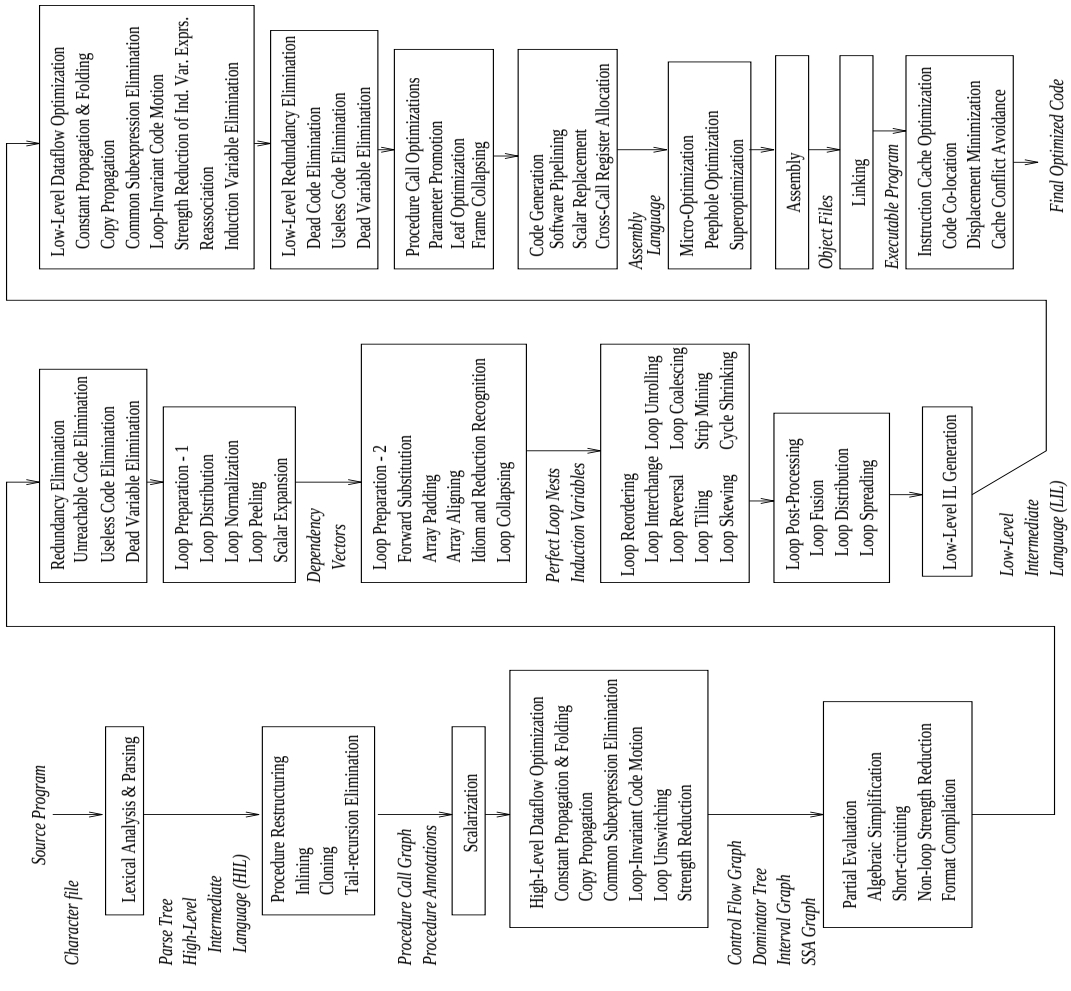


Figure 2.5: High Level Structure of a Typical Parallelizing Compiler (adapted from Bacon *et al* [6]).

common ones can be divided into the following categories:

- Dataflow Optimisations / Partial Evaluation
- Loop Transformations
  - Loop Reordering Transformations
  - Loop Restructuring Transformations
  - Loop Replacement Transformations

  Other areas, as well as loop, have major effects on program performance. These areas may also undergo extensive analysis before optimisation takes place. Such optimisations include:

- Memory Access Optimisations
- Procedure/Function Call Transformations
- Data Layout and Decomposition Transformations

The rest of this section examines these categories in more detail.

2.4.1 Dataflow Optimisations / Partial Evaluation

Most classic dataflow optimisations are based largely on the analysis techniques described in 2.3.1 and include *Constant Propagation*, *Copy Propagation*, *Algebraic Simplification*, and *Strength Reduction*. Partial evaluation refers to to the general technique of performing part of a computation at compile time. Unreachable code as created by transformations that have left "orphaned" code behind may also be eliminated using dataflow analysis techniques. Most of these techniques are employed in some form in nearly all production compilers (sequential and parallel).

Perhaps the most important for automatic parallelization is the *statement reordering* transformation. Simply put, this transformation consists of reordering one or more statements within the dependency constraints. The actual implementation however can come in many disguises: which and how many statements are to be reordered - and into what order? This transformation is also highly important for 'massaging' loops

up and down the program lexicographically, thereby enabling many more powerful transformations (particularly loop-fusion) to be applied.

2.4.2 Loop Transformations

Loops represent the richest source of potential parallelism within a sequential program, hence in order to exploit this parallelism many loop transformations have been created. All the major loop transformations may be categorised into one of three types:

- *Loop Reordering Transformations* change the execution order of the iterations of a loop nest (or nests). They are used to expose potential parallelism and improve data locality. Examples include loop-interchanging, loop-fusing, and loop-splitting.

- *Loop Restructuring Transformations* change the structure of a loop but without changing the operations performed within an iteration or the execution ordering of iterations. Examples include loop-normalisation, and loop-unroling.

- *Loop Replacement Transformations* aim to simply replace the entire loop with more efficient (usually architecture specific) code. This is done by attempting to identify the fundamental purpose of the loop (loop performing a summation, or reduction operation for example) and then using a more convenient operation in its place.

Definition 2.4.1. Loop Normalisation. The object of the loop normalisation function is to restructure a loop into a standard 'normalised' format which makes further restructuring and analysis of the loop easier. In simple terms, convert the loop so that the induction variable has a start value of 1, and the step size is 1 - whilst ensuring that the loop executes the same number of iterations and in the same order and that any array accesses within the loop that are dependent on the loop induction variable must have their subscript expressions amended so as to preserve the original

BEFORE:

```
DO i = expr1, expr2, expr3
  ....
  A(i) = ...
ENDDO
```

AFTER:

```
DO i = 1, (expr2 -  expr1) / expr3 + 1, 1
  ....
  A(expr1 + (i-1) * expr3) = ...
ENDDO
```

Figure 2.6: Generalised Loop Normalisation Transformation

BEFORE:

```
DO i = 2, n+1
  ....
  B(i) = A(i-1) + B(i)
ENDDO
```

AFTER:

```
DO i = 1, n
  ....
  B(i + 1) = A(i) + B(i + 1)
ENDDO
```

Figure 2.7: Loop Normalisation Transformation Example

array accesses. Since loop normalisation is a purely syntactic change it may be applied to any F77 loop.

Example 13. The generalised form of the loop normalisation function restructures any given loop (see Figs. 2.6 and 2.7).

Definition 2.4.2. Loop Reversal. This transformation simply reverses the order in which a loop executes its iterations by swapping the upper and lower bounds of the loop. This has the effect of reversing any dependency relations in which the loop took part (in nested loops, for example). This may often enable other transformations to be applied (particularly loop-interchange and loop-fusion). The only legality constraint is that it may only be applied to loops that have no loop carried dependencies - otherwise the transformation would be illegal and produce invalid code.

Definition 2.4.3. Loop Distribution. (also known as loop-splitting or loop-fission). Ideally, for a loop consisting of N statements, we want the loop-distribution transformation to produce N loops, each containing only one of the statements from within the original loop. (This makes the loops easier to manipulate). However, all statements that belong to a dependency cycle (called a π-block by Kuck [78]) must be placed in the same loop. This can be accomplished by using Tarjan's algorithm [123] to determine cycles (or *strongly connected components*, (SCC's)) within a dependency graph. Any statements which belong to a SCC must be kept together under loop-distribution and form a single loop produced by the transformation. Indeed, if every statement in the original loop is part of the same SCC then the loop-distribution transformation cannot be applied at all.

Once the SCC regions of the dependency graph have been computed we know how many loops will be output by the transformation: 1 for each SCC region, plus 1 for each other statement within the original loop. We then need to determine in which order the new loops should appear in the program. This is done by firstly, computing the acyclic condensation [147] of the program dependency graph. Simply, each SCC is subsumed under a super-node, the resultant graph will be acyclic. Secondly, a topological sort [76] is then performed of the graph nodes. This produces an ordering of the nodes, which is legitimate within the constraints of the dependency analysis. A simple linear traversal of the graph is then all that is needed to produce the transformed code. A simple example of loop-distribution is shown in Fig. 2.8.

Definition 2.4.4. Loop Fusion. The inverse function of loop distribution is loop-fusion[†] (see Figs. 2.9 and 2.10). Two loops may be fused together if (i) they have the same loops bounds, and (ii) there is no statement in the first loop that, when fused with the second loop, results in a backward loop-carried dependence in the fused loop. (This indicates that the reordering of operations is illegal since the execution of the loop may result in producing a different output).

[†] Also known as loop-jamming.

```
BEFORE:
                DO i = 2, n+1
                  A(i) = A(i) * 2 / 5
                  B(i) = A(i) + B(i)
                  C(i) = B(i) + z
                ENDDO

AFTER:
                DO i = 2, n+1
                  A(i) = A(i) * 2 / 5
                ENDDO

                DO i = 2, n+1
                  B(i) = A(i) + B(i)
                ENDDO

                DO i = 2, n+1
                  C(i) = B(i) + z
                ENDDO
```

Figure 2.8: Loop Distribution Example

```
BEFORE:
                DO i = 2, n+1
                  A(i) = A(i)
                ENDDO

                DO j = 2, n+1
                  B(j) = A(j - 1) * 2
                ENDDO

AFTER:
                DO i = 2, n+1
                  A(i) = A(i)
                  B(i) = A(i - 1) * 2
                ENDDO
```

Figure 2.9: Illegal Loop Fusion Example

BEFORE:

```
DO i = 2, n+1
  A(i) = A(i)
ENDDO

DO j = 2, n+1
  B(j) = A(j + 1) * 2
ENDDO
```

AFTER:

```
DO i = 2, n+1
  A(i) = A(i)
  B(i) = A(i + 1) * 2
ENDDO
```

Figure 2.10: Legal Loop Fusion Example

Definition 2.4.5. Loop Interchange. Loop interchange involves exchanging the position of two or more loops in a loop nest (we only consider interchange of perfect loop nests, in this thesis). Loop interchange is legal when the loop bounds form a rectangular iteration space and the original loop dependencies are preserved. Loops with triangular and trapezoidal iteration spaces (such as those introduced by loop skewing) require additional restructuring techniques and are also not considered here.

To preserve dependencies, loops with opposing dependency relation directions cannot be interchanged (since anti-dependencies would become true dependencies, and vice-versa). A loop nest of two loops for example, cannot be legally interchanged if it has both forward *and* backward loop carried dependencies (as in Fig. 2.11). Otherwise the interchange permutation is legal. Program efficiency may also be affected significantly (sometimes better, sometimes worse) when a loop nest has undergone interchanging.

```
DO  i = 2, n
  DO j = 1, n - 1
    A(i, j) = A(i - 1, j + 1) * 2
  ENDDO
ENDDO
```

Figure 2.11: Loops cannot be Interchanged in the presence of both forward *and* backwards loop-carried dependencies

Definition 2.4.6. Loop Replacement. A loop replacement transformation aims to simply replace the entire loop with more efficient (usually architecture specific) code. The first stage in this is for the compiler to attempt to identify the fundamental operation of the loop execution (sometimes called *loop-idiom recognition*). This is often an operation such as the summation of an array or a reduction (apply a single operation to all the elements of an array and return a single number as the result). Once identified, the loop may then be replaced by more efficient code. Depending on the target architecture, the transformation may be a simple replacement of a sequential loop by a single vector operation, or may it may be a more subtle transformation.

For example, we may wish to parallelize the sequential dot-product computation (essentially a summation operation) performed by the loop in Fig. 2.12 for a distributed memory architecture running MPI [96].

```
C Sequential dot vector multiplication subroutine.
      real function sequentialDot( X, Y, Size)
      real X(Size), Y(Size)
      integer Size
      integer i
      real sum

      sum = 0.0

      DO i = 1, Size
         sum = sum + X(i) * Y(i)
      ENDDO
      return sum
      end
```

Figure 2.12: Code Before Loop Replacement Transformation

Assume we have p processors and the value of `Size` is exactly divisible by p. We may distribute the vectors `X` and `Y` across the processors so that each processor gets an equal size of each vector (call this `newSize`). Using the SPMD model of computation, we now add function `parallelDot` to our code (see Fig. 2.13). This calls the `sequentialDot` function now only to perform a 'local' reduction operation. (The global result will be returned to process 0 - as is usual in MPI). Since the original loop now only iterates

```
C Parallel dot vector multiplication subroutine.
      real function parallelDot( localX, localY, newSize)
      real localX(newSize), localY(newSize)
      integer newSize
      real localDot, Dot

      localDot = sequentialDot( localX, localY, newSize )

      MPI_Reduce( localDot, Dot, 1, MPI_FLOAT, MPI_SUM,
                                    0, MPI_COMM_WORLD)
      return Dot
      end
```

Figure 2.13: Code After Loop Replacement Transformation

over a subset of the original data, the loop has effectively been replaced by the single `MPI_Reduce` operation.

Recent advances in parallelization of reduction operations for distributed memory architectures are described in [87].

2.4.3 Memory Access Optimisations

There is an increasing disparity between the speed of CPU chips and memory. When it is necessary to process lots of data at high speed a memory system that is large, yet at the same time fast is required. Possible approaches are:

- Every memory system component can be made individually fast enough to respond to every memory access request.
- Slow memory can be accessed in a round-robin fashion (hopefully) to give the effect of a faster memory system.
- The memory system design can be made "wide" so that each transfer contains many bytes of information.
- The system can be divided up into faster and slower portions and arranged so that the fast portion is used more often than the slow one.
- The instructions and data could be segregated so that the demand for instructions is decoupled from the demand for data.

The most popular (and coincidentally, economically viable) approach is to use some combination of the last three or four options. The arrangement of memory hierarchy in a typical processor is shown in Fig. 2.14.

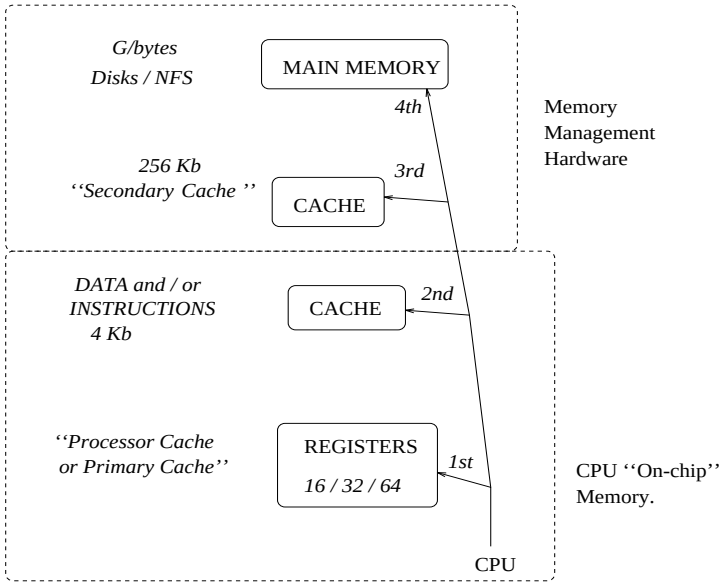


Figure 2.14: Memory Hierarchy in a typical processor

This arrangement (sometimes called the *Harvard Memory Architecture*) is found in many single-processor and workstation machines. The general aim of any memory system is to retrieve as much data as possible with as few cache misses as possible. It should be remembered that a 'page' of memory is really an operating system entity (typically anything from 1Kb to 16Kb in size). Caches generally manipulate information in the form of physical 'blocks' and store it in themselves in the form of 'lines'.

When compilation is optimised for *speed* much of the source code is 'in-lined' so that it runs faster but with the effect that it takes up more space in memory. Furthermore, the program will also have to make longer 'jumps', thus increasing the likelihood of cache misses.

When optimised for *memory*, the program will use less memory with the drawback that procedure and function calls will take longer due to incurring overheads (e.g. stack manipulation, etc). Hence, the optimal compilation for a program is a trade-off between the speed requirements of the user, and memory requirements of the program.

2.4.4 Procedure/Function Call Transformations

Attempts to minimize (and possibly eliminate) the overheads associated with function and procedure calls can be placed into one of four categories:

- Remove the call entirely
- Optimize execution of the called procedure‘s body
- Remove some of the entry/exit overhead
- Where the behaviour of the procedure call is known or can be inferred, take action to reduce it’s overheads.

Using much of the data gathered during the inter-procedural analysis phase (2.3) it may be possible to perform a number of transformations including: procedure in-lining, procedure cloning, parameter register optimization, and recursion elimination.

2.4.5 Data Decomposition Transformations

When a compiler is targeted at a multiprocessor, one of the most important decisions it has to make is how best to decompose the data across the processors available. Since communication is one of the major overheads of any parallel computation, the criteria for good data decomposition for a distributed memory architecture must include minimising necessary communication. However, data decomposition is also important for shared-memory architectures where communication overheads still have a major impact on performance even though communication is performed implicitly.

Regular Array Decomposition

The most common strategy for decomposing an array `A` across a set of processors is to map each dimension of the array on to the processors in a regular pattern (see [6, 67, 68, 73, 77]). The four most common patterns are:

1. *Collapsed.* This is the degenerate case. Usually denoted by the asterisk symbol, ‘`*`’, it means the array is *not* distributed or whole dimension(s) of the array are mapped to a single processor. The computations using

those dimensions of the array are then performed on that processor, accordingly.

2. *Serial.* The dimension is replicated across the set of processors.

3. *Block.* The dimension is segmented into n blocks, allocate block n to the nth processor. The advantage of block decomposition is that adjacent array elements are usually on the same processor. Because a loop that computes a value for a given element often uses the values of neighbouring elements the blocks have good locality and reduce the amount of communication (hence, this arrangement is particularly suited to distributed-memory architectures).

4. *Cyclic(n).* The first n elements are allocated to the first processor, the second n elements are allocated to the second processor, and so on. Allocation of the array elements may wrap-around the processors until all the elements have been allocated. If unspecified, the n parameter usually assumes a default value ($n = 1$). Cyclic decomposition has the opposite effect of blocking, it has poor locality for nearest neighbour based communication but has the advantage of spreading the load more evenly. The maximum benefits of cyclic decomposition are to be achieved when the expensive iterations are spread across the processors. Setting the block-size parameter has the effect of achieving a *block-cyclic* type of distribution which strikes a good compromise between the good data locality of block-decomposition and the load balancing benefits of simple cyclic decomposition.

In High Performance Fortran (HPF) [67, 68], these 'data mapping' directives may be declared to specify the data decomposition of an array across the target architecture.

Example 14. Given a one dimensional array `A`, a two dimensional array `B`, and a shared-memory target architecture consisting of 4 processors (i.e. `NProcs`), the various directives specify the following data distributions:

```
      C       Variable Declarations.
              PARAMETER (NProcs = 4)
              PARAMETER (SIZE = 32)
              INTEGER A(SIZE)
              INTEGER B(SIZE, SIZE)
```

`DISTRIBUTE A(*)`	Elements `A(1:32)` are allocated to a single processor.
`DISTRIBUTE A(SERIAL)`	Elements `A(1:32)` are replicated to all processors.
`DISTRIBUTE A(BLOCK)`	Blocksize = `SIZE` / `NProcs` = 32 / 4 = 8
	Elements `A(1:8)` are allocated to processor 0.
	Elements `A(9:17)` are allocated to processor 1.
	....
	Elements `A(25:32)` are allocated to processor 3.
`DISTRIBUTE A(CYCLIC)`	Elements `A(1:29:4)` are allocated to processor 0.
	Elements `A(2:30:4)` are allocated to processor 1.
	....
	Elements `A(4:32:4)` are allocated to processor 3.
`DISTRIBUTE A(CYCLIC(4))`	Elements `A(1:4)` and `(17:20)` are allocated to processor 0.
	Elements `A(5:8)` and `(21:24)` are allocated to processor 1.
	....
	Elements `A(13:16)` and `(29:32)` are allocated to processor 3.

Figure 2.15: Effects of Simple Array Decomposition Directives

If `SIZE` is not exactly divisible by `NProcs` then blocks of size $b = \lceil SIZE/NProcs \rceil$ are allocated to the first $\lfloor SIZE/NProcs \rfloor$ processors, the remaining ($SIZE \mod NProcs$) elements form a small block which is allocated to the next processor - no elements are allocated to any remaining processors.

2.4.6 Automatic Data Decomposition

In the general case, the automatic determination of data-distributions (and re-distributions) has been shown to be NP-complete [89, 90]. Hence, the main focus of much research has been on semi-automatic distribution tools.

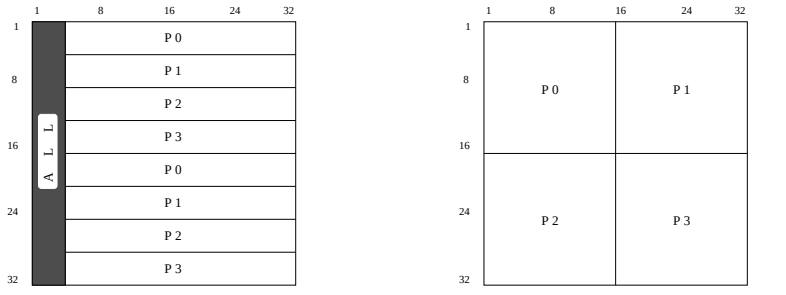
The decomposition of an array across the target architecture is further complicated by the idea of *aligning* arrays across the processors available. For certain applications the improved locality of data and reduced communication can bring substantial improvements in performance. However, languages like HPF rely on the programmer to declare how arrays are to be aligned and what form(s) of decomposition are to be

`DISTRIBUTE B(*,*)`	Elements `B(1:32,1:32)` are allocated to a single processor.
`DISTRIBUTE B(BLOCK,SERIAL)`	Elements `B(1:8,1:32)` are allocated to P0. Elements `B(9:17,1:32)` are allocated to P2. .... Elements `B(26:32,1:32)` are allocated to P3.
`DISTRIBUTE B(SERIAL,CYCLIC(4))`	Elements `B(1:32,1:4)` and `(1:32,17:20)` are allocated to P0. Elements `B(1:32,5:8)` and `(1:32,21:24)` are allocated to P1. .... Elements `B(1:32,13:16)` and `(1:32,29:32)` are allocated to P3.
`DISTRIBUTE B(BLOCK,BLOCK)`	Elements `B(1:16,1:16)` are allocated to P0. Elements `B(1:16,17:32)` are allocated to P1. Elements `B(17:32,1:16)` are allocated to P2. Elements `B(17:32,17:32)` are allocated to P3.

Figure 2.16: Effects of 2-D Array Decomposition Directives

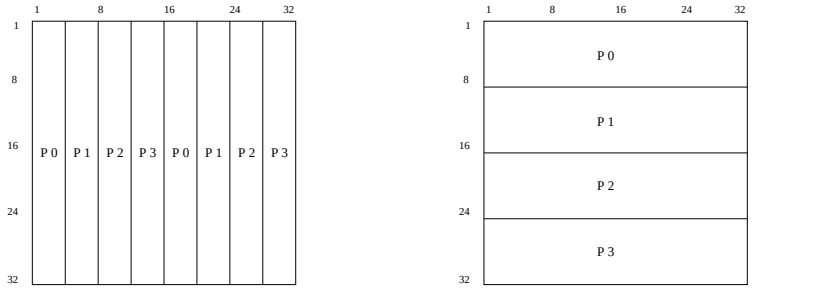
used (BLOCK, CYCLIC, etc) and programmers can find it difficult to choose a good set of declarations, especially since the optimal decomposition can change as the program executes. Research projects have attempted to determine if the optimal data decomposition can be determined automatically without programmer assistance.

The general approach for automating data decomposition and alignment is to express the behaviour of the program in a representation that either captures communication explicitly or allows it to be computed. The compiler then performs operations that



`DISTRIBUTE B(SERIAL, CYCLIC(4))` `DISTRIBUTE B(BLOCK, BLOCK)`

Figure 2.17: 2-D Array Decompositions.



DISTRIBUTE B(CYCLIC(4), SERIAL) DISTRIBUTE B(CYCLIC(4), BLOCK)

Figure 2.18: 2-D Array Decompositions.

reflect different decomposition and alignment choices; the goal being to maximize parallelism while minimizing communication. Two transformations for this purpose are:

- *Scalar Privatization.* Each processor is given its own copy of variables where dependencies allow.
- *Array Privatization.* Analogous to Scalar Privatization, each processor is given its own copy of an array to reduce communication overheads and expose additional parallelism by removing unnecessary dependences caused by storage reuse.

Research into automatic data decomposition techniques is ongoing. Notably Crooks [34] describes the Fort-Translation System (FTS) - a fully-automatic tool for computing near-optimal data distributions and re-distributions for programs written in a subset of Fortran-90. The tool combines pattern-matching templates with a look-up table of distributions and a tree-based approach to dynamic data re-distribution. An extensive literature survey is also presented.

2.4.7 Program Optimisations

One way to improve parallelism on an architecture with asynchronously executing processors is to reduce communication by dividing the program into separate computations (where dependencies allow). Rather than just executing iterations of a loop in parallel this involves program-level analysis to identify parts of the program that can be exe-

cuted separately as sub-tasks. Three common techniques employed to exploit task-level parallelism are:

- *Program Splitting.* Simply identify which parts of the program may execute in parallel *and* are worth partitioning into separate sub-tasks [50]. Taking into consideration the additional communication, scheduling and synchronization overheads that may be incurred by creating another process the granularity of the sub-task (i.e. the amount of work it will do) has to be enough to make the program split worth while.

- *Graph Partitioning.* Data flow languages often expose parallelism by converting the program into a graph that explicitly represents computations at the operator-level. Scheduling such small amounts of work is usually impractical so researchers normally transform the data-flow graph into larger blocks that are executed atomically.

- *Procedure Call Parallelization.* The compiler uses information gathered during the inter-procedural analysis phase to determine if a subroutine call (or calls) may be executed in separately (i.e. in parallel) at some stage of the main program execution.

2.5 Automatic Parallelization Research

In 1987 the problem of automatic parallelism was identified as one of the "grand challenges" of high performance computing [23, 43]. The recognition of the importance of automatically identifying and exploiting implicit parallelism in sequential programs can be seen from the extensive literature available and research efforts made (see Fig. 2.19).

Early experiments in parallelizing Fortran programs were done in the late 1960's and early 70's by Kuck as part of the ILLIAC IV project at the University of Illinois.

The first automatic restructuring compiler was the Parafrase compiler [106] also developed at the University of Illinois. The system has had a profound affect on the

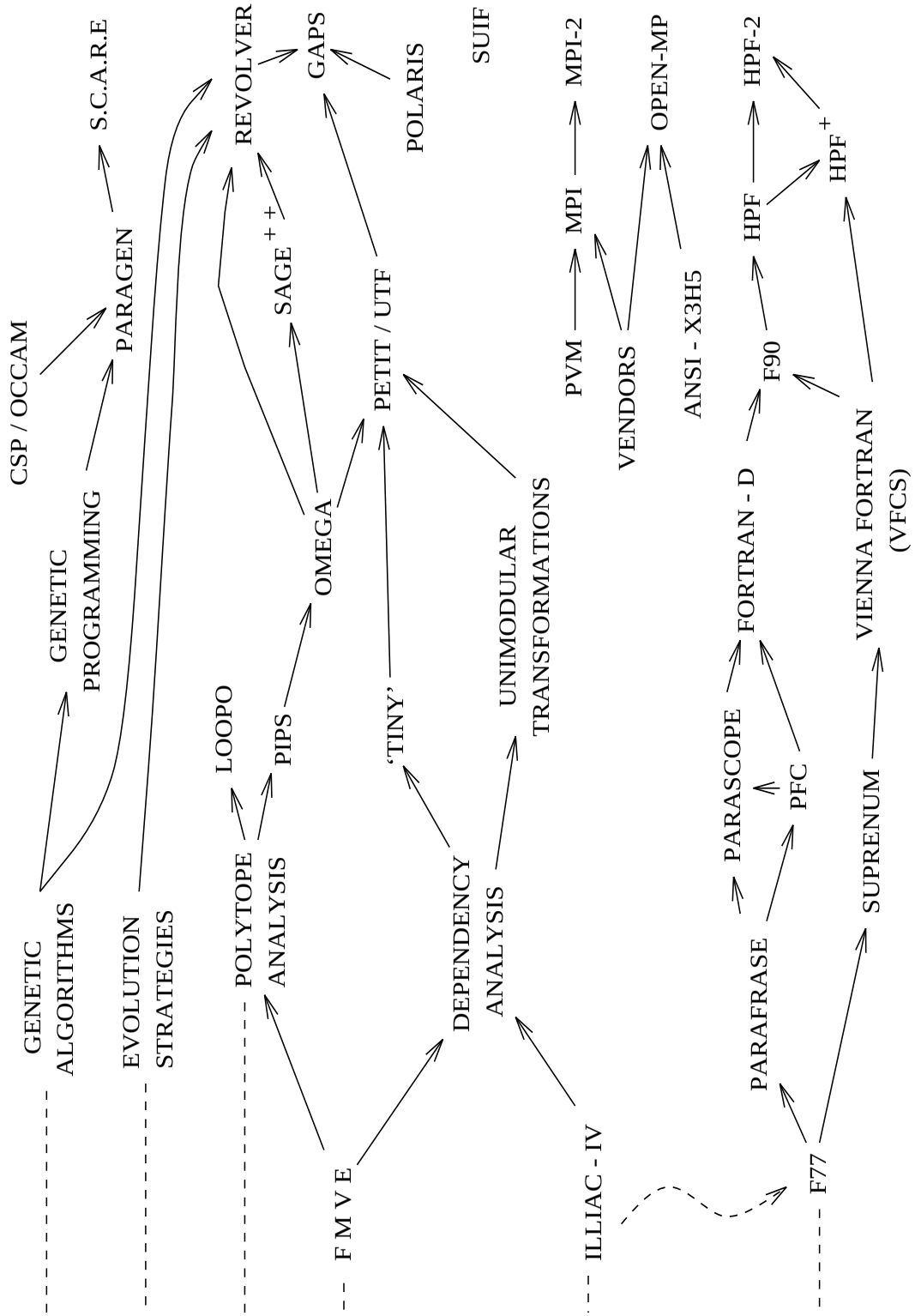


Figure 2.19: Overview of Automatic Parallelization Research

later research. Parafrase could be retargeted at vector, array-processor, and shared-memory multiprocessor architectures. Its features included automatic construction of data dependency graphs, construction of strongly connected components (π-block) graphs, and a large (over 100) program transformation catalogue to improve parallelism. The input to Parafrase consisted of a sequential Fortran program and a *pass list* - a list of pass names and related switch settings - that specified the tasks to be performed. Each pass name identified a procedure that may read and/or write the program: passes were applied in the order specified in the list and each one performed a source-to-source transformation. Timing tables were then used to predict the expected speed up of the restructured program. This organisation is highly flexible: it enables the construction of pass lists that are optimized towards specific architectures and makes it easy to extend the system by including new passes.

The Parafrase-2 compiler (being developed at Illinois and Fukuda Laboratories, Japan) is also capable of restructuring C and Pascal programs.

The PFC (Parallel Fortran Converter) developed at Rice University [2, 7] by Ken Kennedy and his group was based on the Illinois Parafrase compiler. PFC was the first to incorporate the idea of IF-conversions, where IF statements are converted into a guarded assignment statement thereby allowing control and data dependencies to be treated uniformly. A major strength of PFC was that unlike Parafrase, where each transformation operates directly on the source code, PFC transformations manipulated an internal representation resulting in a significant speed-up in restructuring times.

The ParaScope tool [74] (Rice University) incrementally updates the source code and analysis as the skilled user interactively applies a range of reordering, memory optimizing, and dependence breaking transformations.

The SUPERB (Suprenum Parallelizer Bonn) parallelizer was an interactive source-to-source restructuring tool for Fortran programs developed at the University of Bonn from 1985-89 by Hans Zima and coworkers as part of the SUPRENUM project [147]. SUPERB restructured Fortran-77 programs into a data parallel SPMD form (SUPRENUM-Fortran).

The VFCS (Vienna Fortran Compilation System) is a sophisticated, commercially

available restructuring compiler developed at the University of Vienna [19]. It parallelizes Vienna Fortran, Fortran-77, or HPF [67] programs and generates message-passing code (machine-native or MPI[96]) for a variety of distributed memory architectures (including networks of workstations), generating code based on the SPMD model of parallelism. Its extensive range of analysis tools include an advanced profiler[45], runtime library support, and a retargetable static parallel program performance predictor [44, 46]. These allow the user to use VFCS in either command-line, interactive, or fully automatic modes of operation.

The future directions of HPF (namely HPF 2.0 and HPF$^+$) including new features such as (i) data-distribution directives for irregular distributions (ii) distribution of data to subsets of processor arrays (iii) distribution of sub-objects and derived types, and (iv) increased flexibility in expressing work distribution, have recently been detailed by Delves and Zima in [38].

A family of retargetable restructuring tools called KAP (Kuck and Associates Parallelizer), have been developed for commercial use [80]. The CEDAR Fortran restructuring compiler developed as part of the CEDAR project at Illinois was also developed by Kuck and Associates [79].

The TINY loop restructuring tool was a research tool developed at the Oregon Graduate Institute by Wolfe [142] and later integrated into the extensive `Petit` dependency analysis libraries by the Omega Project group at the University of Maryland by Pugh, Kelly and coworkers [71].

The SUIF (Stanford University Intermediate Form) compiler group [139] is an ongoing project where the source code has been made publicly available to encourage research into optimising and restructuring compiler techniques. Notably the SUIF compiler has been selected to be part of the 3-year National Compiler Infrastructure Project [81].

Lengauer and Griebl, *et. al* have made available the LooPo parallelizer. This is a prototype restructurer for testing different space-time mappings for loop nests. It admits perfect or imperfectly nested loops from a number of imperative languages, derives data dependencies from the source code, provides a number of scheduling and

iteration placing strategies, and generates synchronous or asynchronous code for shared or distributed memory architectures. Notably, work on in the LooPo project has also included extensive restructuring analysis of non-counted (i.e. `WHILE`) loops [57, 58].

The PIP parallelizing compiler [48] from Laboratoire PRiSM is a linear, integer, parametric programming system which has been developed to investigate issues of static program analysis. It makes extensive use of polytope analysis and solving systems of affine inequalities (using Lisp-like expressions as input and output through which code can be entered and generated).

A wide overview of applied parallel computing issues (including restructuring compilers) is presented in Perrott [109].

Other restructuring compilers projects include the McCat compiler [62], the POLARIS restructurer for shared-memory multiprocessors architectures [24], and the EVE compiler [103] developed by the PS parallelizing compiler group lead by Nicolau.

In [26] an interesting graph representation is described which facilitates inter-procedural dataflow transformations.

Another factor affecting parallelization is granularity. Partitioning programs into large grain processes with little communication between them can produce some of the most impressive speed up figures. It has been argued in [28] that restricting parallelization of loops to those without loop-carried relations reduces parallel program performance to unacceptable levels.

Automatic Parallelization Approaches

As machine architectures became more complex a number of researchers devised more restructuring transformations to aggressively rewrite the source program in order to expose potential parallelism. Many of these transformations were aimed at 'zapping' dependencies (i.e. rewriting code to remove dependencies) whilst preserving the semantic behaviour of the original program. A comprehensive survey of these techniques is given by Bacon *et al.* [6].

As the number of transformations available increased, many researchers looked for ways to combine them into a unified framework. In [130], Soffa and Whitfield describe

an axiomatic framework of pre and post conditions which is used to determine those interactions among transformations that can create conditions and those that can destroy conditions for applying other transformations. The technique can encompass dataflow optimisations (dead-code elimination, constant propagation, code motion, etc), as well as loop transformations. No results are presented.

The unimodular loop-transformation technique of Banerjee [10, 11, 12, 13, 14] is a framework for applying sequences of transformations on perfectly nested loops.

A single loop transformation can be represented by an $n \times n$ unimodular matrix, where n is the depth of the loop nest the transformation is to be applied to. A sequence of loop transformations can be represented by multiplying together their represective unimodular matrices to create a single $n \times n$ transformation matrix (called T). The bounds of the loops in the loop-nest can be coded up as a system of integer linear inequalities and represented in a matrix L. Multiplication of L by transformation T generates a new matrix of integer inequalities, these new inequalities can then be reduced by applying Fourier-Motzkin variable elimination to determine the new bounds of the new loops. The inverse of the transformation matrix T gives the original loop index variables in terms of the transformed variables and is used in the generated code to update the array subscript expressions.

Example 15.

Given the following perfect 2D loop nest:

```
DO i = 2, 10
    DO j = 1, 10
      a(i, j) = a(i-1, j) + a(i, j)
    ENDDO
ENDDO
```

Dependency analysis reveals only the outer loop carries a dependency with a distance of $^{+}1$ iteration, hence the dependency vector is $D = \binom{1}{0}$. We now encode a sequence of transformations by multiplying their respective matrices to create a transformation matrix $\mathbf{T}$. Suppose we encode a sequence of transformations: Loop-Interchange followed by Inner-Loop Reversal we have

$$TF = LIC \times OuterLRV$$

$$TF = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$TF = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

We are now able to test for the *legality* of the transformations: $TF \times D = \binom{1}{0}$ hence the transformation sequence is legal. (If the dependency vector contained a negative value this would indicate that a dependency had been violated and that the transformations would be illegal).

We now encode the loop bounds as described in section 2.2

$$\begin{matrix} & A & & & B \\ \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} & \begin{pmatrix} i \\ j \end{pmatrix} & \leq & \begin{pmatrix} -2 \\ 10 \\ -1 \\ 10 \end{pmatrix} \end{matrix}$$

The new set of index variables P can be computed (using the inverse of TF) by $P = TF^{-1}$

$$\begin{pmatrix} p \\ q \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ -i \end{pmatrix}$$

The new bounds are computed by applying the inverse transformation matrix to the original bounds, namely $TF^{-1} \times A$. This gives us

$$\underset{A'}{\begin{pmatrix} 0 & -1 \\ 0 & 1 \\ 1 & 0 \\ -1 & 0 \end{pmatrix}} \begin{pmatrix} p \\ q \end{pmatrix} \leq \underset{B}{\begin{pmatrix} -2 \\ 10 \\ -1 \\ 10 \end{pmatrix}}$$

Applying Fourier-Motzkin variable Elimination (FMVE) to the system leaves the following constraints

$$-10 \leq p$$
$$2 \leq q$$
$$p \leq -1$$
$$q \leq 10$$

From which we generate the transformed code

```
DO p = -10, -1
    DO q = 2, 10
      i = q
      j = -p
      a(i, j) = a(i-1, j) + a(i, j)
    ENDDO
ENDDO
```

The code can be further optimised by subsequent compiler transformations. Inner-Loop Reversal was applied to the original innermost loop. The iteration space of the original loop nest has been transformed by the linear transformation defined by unimodular matrix *TF*. Unimodular transformations preserve the volume of the original iteration space. Applying transformations represented by *non-unimodular* matrices will scale the iteration space by some factor k. Generated code must then have an additional step-size value k to preserve correctness.

In [140] Wolf and Lam describe how to apply unimodular transformations to optimise data locality.

A thorough analysis of the theory of unimodular loop transformations is presented in Dowling [41]. Interestingly, automatic parallelization is defined as a "*scheduling problem with precedence constraints*". Dowling also notes that even drastically over-simplified scheduling problems with only a finite number of processors are $\mathcal{NP}$-complete. Only if the number of processors can be considered infinite, can such scheduling problems can be solved polynomially. Searching for optimal transformations is exponentially expensive - i.e. one can expect to spend much more time trying to find the best transformation rather than just finding a good one.

Linear loop transformations have long been used in the systolic array community to map regular algorithms onto a regular processor ensemble (see Karp *et. al*, [70]). Megson and Chen[93] developed the SARACEN system for parallelizing a class of numerical algorithms onto systolic architectures. Nonlinear and piecewise linear transformations are described in [88] by Lu.

Fourier-Motzkin Variable Elimination (FMVE)

Fourier-Motzkin elimination is a technique for finding the solutions (if any) to a system of linear inequalities, in much the same way that Gaussian elimination is used to find solutions (if any) to a system of linear equations. It's relevance in restructuring compilers is (i) it is used to determine new loop bounds after restructuring with unimodular transformations (as above), and (ii) as a technique for determining if dependencies exist between loop iterations (by formulating loop bounds and array access equations as systems of integer inequalities). As such we are generally only interested in solving systems of *integer* inequalities (and integer solutions).

For further details the reader is referred to the literature [12, 21, 144].

Unified Transformation Framework (UTF) / `Petit`

The main advantages of unimodular transformations is that entire sequences of transformations can be quickly composed and legally applied, and that a number of transformations can be conveniently represented as unimodular matrices. The two main restrictions are that (i) they can only be applied to perfect loop nests, and (ii) the

transformations are applied to all statements in the body of the loop nest.

Using the high-level time-space mapping representation borrowed from the systolic array community, Kelly [72] has devised a generalisation of unimodular-transformations called a Unified Transformation Framework (UTF). This is aimed at overcoming the two restrictions of unimodular-transformations. Computations are mapped onto a 1-d array of processors. Rather than mapping iterations of perfect loop-nests, UTF maps instances of individual statements in accordance with the time-mapping specified for that particular statement (or statement block). The time-mapping representations can be manipulated by rules which represent a wide-range of transformations (including loop-fusion, loop-distribution, and statement reordering).

The rules may be applied in any order to a time-mapping, as such any sequence of traditional transformations can be easily represented, and its legality tested.

Space-mapping allow arrays to be distributed across a 1-d array of processors in `BLOCK`, `CYCLIC`, `BLOCK-CYCLIC`, or `ZERO` forms. Program time and space mappings are then searched to find a 'good' and legal solution from which code may be generated. An algorithm is supplied to generate SPMD code for shared-memory multiprocessors only.

The representation allows powerful sequences of transformations to be tested and applied. Analysis of conditional statements and procedure calls is not considered. The system has been made freely available as part of the the `Petit` and `Omega` libraries package [71] from `ftp://ftp.cs.umd.edu/pub/omega/petit/`.

Knowledge-Based Approaches

The knowledge-based approach to automatic parallelization involves using heuristic information to define a rule-base to guide the restructuring process. This approach has been investigated by Wang and Gannon [129], and in the SAVER project developed by Brandes at the University of Marburg [25]. Tenny [124], also researched this approach using a 'transformation planning' model where fatally flawed (i.e. semantically invalid transformations that changed the behaviour of the program) had to be discarded by back-tracking. This was found to be a time and memory consuming process.

2.6 Summary of Automatic Parallelization

So far in this chapter we have defined some of the basic concepts in automatic parallelization with particular attention to dependency analysis techniques and illustrated with many examples, some of the most common compiler optimisations for loop restructuring, and data decomposition. In the rest of this chapter we look at how the effects of applying these transformations can be measured by detailing techniques to enable static estimation of the restructured and parallelized codes' performance.

2.7 Static Performance Estimation

In this section we describe the profiling, benchmarking and static performance estimation techniques used in the Revolver parallelising compiler.

The need for performance estimation can be illustrated by the following question: having applied one or more restructuring transformations, how do we find out what effect they have had on the performance of the source program?

The immediate answer to this question is to compile and run the restructured program and time how long it takes to execute. However, this approach is clearly undesirable since the original program may have taken several hours / days to run (and the restructured program may take even longer !). Basically, the user has to wait too long to receive feedback of the effects of the changes made. What is needed is some way of estimating the execution time of the restructured program without the need compile to execute it. The techniques of static performance estimation described here fall into the following three categories (although other approaches are possible).

- *Profiling.* A single profiling run of the original sequential program (on a typical set of data) is extremely useful in estimating the performance of a restructured version of the program. This is because information gathered in a profiling run (such as how many times a particular loop executed, how many times some condition evaluated $TRUE$, etc) cannot always be accurately determined by simply analysing the source code. A database of such information leads to significantly more ac-

curate estimations and can be used to guide the restructuring effort towards the most computationally intensive parts of the program.

- *Benchmarking Operations.* Benchmarking an operation means to assign an average time-cost value to a particular operation by timing how long the operation takes to execute under different conditions and on different types of data (on the target architecture). In practice, this usually means executing the operation many thousands of times, timing each execution and determining the mean average execution time, in order to obtain an accurate time-value cost for the operation. The sort of operations benchmarked usually include arithmetic, assignment, relational and logical operations, using different types of data (integers, reals, complex, chars, etc) of both variable and constant declaration. It can be however, extended to include more complex operations, such as message start-up times, send / receive costs, data locality and network communication costs, and more.

- *Cost Model Analysis.* Using information gathered from the profiling run and from operation benchmarking, each statement in the restructured source code can now be analysed and attributed with a time-cost value. The accumulated sum of the time-cost values of each statement in the program is then the estimated performance execution time of the whole program.

The static performance estimator used in REVOLVER accumulates program analysis data in an initial profiling run of the sequential program. Benchmarking information on the time costs of arithmetic, assignment and communications operations on the target architecture are then accumulated in a manner similar to the training set method described in Balasundaram *et. al* [8]. These sets of information are then used in a summation algorithm to produce a total time cost estimate for the whole program.

The rest of this section will examine the techniques used in these stages.

2.7.1 Profiling

A single profiling (or *instrumented*) run of the original sequential program (on a typical set of data) is extremely useful in estimating the performance of a restructured version of the program. The data accumulated includes, loop iteration counts, *TRUE/FALSE* ratios of conditional statements, and frequency counts of base-blocks - which can be used to derive further information such as procedure/function call counts. A single typical profiling run is usually enough to accumulate all necessary information for accurate performance estimation of the restructured program.

Definition 2.7.1. Program Profile Time. Let *exec(p)* define a particular instance of the execution of a sequential program p, then *proftime*, is a value $proftime \in \mathbf{R}_0^+$ which is assigned to *exec(p)* such that *proftime(exec(p))* defines the measured execution time of that particular execution of p.

We now illustrate the instrumentation code inserted into the program prior to a profiling run.

```
BEFORE:
                    ....
                    S1
                    ....
                    Sn
                    ....

AFTER:
                    ....
          I:        $BB = $BB + 1
                    S1
                    ....
                    Sn
                    ....
```

Figure 2.20: Frequency Instrumentation Code

Figure 2.20 illustrates how a frequency count of a basic block is determined. Where `I` is an instrumentation statement inserted before the profiling execution run. If, before instrumentation, S_1, is a labelled statement, the label is moved and attached to `I`.

Note also, that `$BB` is an integer variable initialised to zero at the start of the

profiling run (as are all instrumentation variables). The frequency count of the basic block, $freq(B)$, in accordance with definition 2.2.14 is defined by the value of `$BB` at the end of the profiling run. All profiling data accumulated during the run is saved to a profiling data file for later use.

```
BEFORE:
                ....
        L:      DO i = lb, ub
                  ....
                ENDDO
                ....

AFTER:
                ....
        L:      DO i = lb, ub
        I1:      $bb = $bb + 1
                  ....
                ENDDO
                ....
```

Figure 2.21: Loop Iteration Count Instrumentation Code

Figure 2.21 illustrates how the loop iteration count function *iters(L)* of a given loop L is determined. Instrumentation variable `bb` is set to zero at the start of the profiling run. The value of *iters(L)*, in accordance with definition 2.2.19, is defined as *iters(L) = ub - lb + 1*.

```
BEFORE:
                ....
        C:      IF (expr) THEN
                  ....
                ENDIF
                ....

AFTER:
                ....
        C:      IF (expr) THEN
        I:        $T = $T + 1
                  ....
                ENDIF
                ....
```

Figure 2.22: *TRUE/FALSE* ratio Instrumentation Code

Where `I` is an instrumentation statement and `$T` is set to zero at the start of the profiling run. The value of *TFRatio(C)*, in accordance with 2.2.13, is defined as $TFRatio(C) = \frac{T}{N}$, where N is the number of times C was evaluated and T is the number of times C evaluated to $TRUE$.

Clearly the intrusion of inserting instrumentation code will have an impact on the execution time of the source code program, *proftime(p)* will be greater than the execution time of the original, uninstrumented version of p. In automatic parallelizing systems where profile timing information is used to guide the automatic parallelization effort the impact of this intrusion is minimized by the use of optimizations which exist for hoisting instrumentation code, however, since profile timing information is *not* used to guide the automatic parallelization effort in REVOLVER , these are not performed. (As part of the evolutionary approach, we do not wish to guide the EA towards restructuring any particular part of the program, rather we leave it up to the EA to decide which parts of the program need restructuring in order to bring about significant improvements).

An example of an attributed section of code *after* a REVOLVER instrumented run is shown in Fig 2.23. The figures in the 'Number of Executions' column represent the execution frequency of each statement in the program, as recorded during the sequential profiled run of the program. This information helps us to determine the execution frequency of basic blocks of code, and hence, the number of iterations each loop makes. For instance, we can tell that loop L_1 executed 10 iterations, despite the symbolic upper bound (k). Importantly, we can also use this information to compute the TRUE/FALSE ratios of conditional statements. We know for instance, that condition C_1 was evaluated 10 times in the execution, and that the lexicographically following statement was executed 4 times, hence we can infer that condition C_1 evaluated to $TRUE$ $4/10 = 0.4$ times. Similarly, condition C_2 evaluated $TRUE$ $5/6 = 0.83$ times. This lead to loop L_2 being instantiated on 5 separate occasions over which it executed a sum of 22 iterations.

Deriving TRUE/FALSE ratios in this way is more accurate than simply assigning default probabilities of 0.5, and precludes any need for user intervention.

```
                                                  A T T R I B U T E S
                                                  -------------------------
                                                  Number of       TRUE/FALSE
                                                  Executions        Ratio
        PROGRAM attributed                            -               -
        integer A(100), B(100)                        -               -
        integer i, j, k                               -               -
        real X(100), Y(100), Z(100)                   -               -

        . . . . . .

        k = 10                                        1               -

L1:     DO i = 1, k                                  10               -
         A(i) = 0                                    10               -
         Z(i) = 0                                    10               -
         B(i) = irand(100)                           10               -
         X(i) = B(i) * 5 / 9 + 32                    10               -

C1:      IF (X(i) .gt. 100) THEN                     10             0.4
           Z(i) = Z(i) + X(i)                         4               -
C2:      ELSEIF (X(i) .lt. 100) THEN                 6              0.83
L2:        DO j = i, k                              22               -
             A(i) = A(i) + Z(i)                     22               -
           ENDDO
C3:      ELSE                                        1
           A(i) = 0                                  1               -
         ENDIF
        ENDDO

        . . . . . .

        END

(j = i to 10, : 1,3,5,6,7)
```

Figure 2.23: Example Attributed Code

2.7.2 Benchmarking Operations

Benchmarking an operation means to assign an average time-cost value to a particular operation by timing how long the operation takes to execute under different conditions and on different types of data (on the target architecture). For example, to obtain a benchmark value for the assignment operation of an integer constant to an integer variable, such as:

```
integer i
 ....
i = 10
```

We executed and timed this operation 250,000 times on our target architecture (a Meiko CS-1) and took a mean average time-cost for this operation (assignment) on these types of data (from integer constant to integer variable) - namely 23.272704 $\mu secs$[‡]. A complete set of tables of timings for other legitimate combinations of operations and data types is given in appendix A.

.

In the program data structures these timings were encoded as a look-up table. Such a look-up table was used to accumulate costs for sequences of operations and hence compute costs for whole statements. The time cost of each statement being eventually attributed to each statement (see Fig. 2.23) in the program.

Expression Evaluation Order

It is clear that the order in which operations are performed in evaluating an expression has an important impact on the time the expression takes to execute. In order to obtain an accurate time-estimate for the expression the expression has to essentially be 'parsed' in the same order it will be evaluated. This is clear when given a statement:

```
i1 = 2 + 3 * 4
```

Variable `i1` must be assigned `14`, *not* 20, or anything else. (Multiplication takes higher precedence than addition). In order to obtain an accurate time-estimate, we

[‡]That is *micro*-seconds, 1 μsecond = 10^{-6} seconds

must therefore evaluate the time cost of each operation, in the same order which the operations will be executed (and thereby accumulate an accurate time-cost for the whole statement)..

Example 16. Given a statement, as below, where we know variable `r1` to be declared of type `real`, and variable `i1` of type `integer`:

```
r1 = 2 - 3 * (i1 + 4.75) + 1
```

Using values from the tables in appendix A, we compute the time cost for executing this statement as:

(1):	i1 + 4.75	(`int var`+ `real const`)	= 0.1968640
(2):	3 * (1)	(`int const`× `real var`)	= 0.0578816
(3):	2 - (2)	(`int const`- `real var`)	= 0.0615168
(4):	(3) + 1	(`real var`+ `int const`)	= 0.3630080
Total Estimated Execution Time			= 0.6792704 *secs*

Benchmarking Communications

One of the most expensive overheads of any parallel computation is the cost of communication between processes. Processes communicate to send data to and from each other that they need in order to complete their computations. The communications protocol used in REVOLVER is the blocking-synchronous protocol, as described in section 4.10.3). It is essential therefore that the time-cost of any communications made must be taken into account in estimating the performance of a parallel program.

Implementing a general performance estimator for parallel programs (on *any* architecture) is a non-trivial problem. The estimation of communication costs on any distributed memory architecture is particularly problematic. Factors such as network contention, network latency, and message-routing algorithms all have to be taken into account when attempting to accurately benchmark the average cost of a single message communication.

Our performance estimator, takes advantage of the form of parallelism we exploit

in REVOLVER . Indeed, the restricted form of master/slave parallelism we use (section 4.10.5) was partly chosen because the regular communications patterns allow accurate estimations of performance to be made. Our approach to the estimation of communications costs then, is based on the following two observations on the code REVOLVER generates:

1. The master process is *always* placed on processor #0.

2. Slave processes are *always* allocated to processors from 0 increasing up to N-1, within each phase of parallel execution the program enters.

As such, it can be seen that all communications will *only* be between processor #0 and each of the other processors (i.e. *not* between these other processors). Therefore the only communications we need to benchmark are messages of various sizes being sent/received between processor #0 and each of the other processors.

This was done for a range of message sizes (in bytes), namely in the range: 256 bytes, 512, 768, ... , 5120 bytes. Each message was sent 5000 times and a time-cost recorded for each. An average time-cost was then calculated. To prevent the occurrence and propagation of truncation and round-off errors, an arbitrary precision library was implemented for use in these calculations.

These average time-costs were then plotted onto the graphs in appendix A (Figs. A.1 to A.6). A χ-squared line-fitting algorithm [8, 111] was then used to compute an accurate representative function to 'fit' each set of data produced. (Each of the graphs also displays its representative line-fit function). Fortunately, when the data points were plotted, reasonably straight lines were formed, so the line-fit functions achieved a good degree of 'fit' to their data. These functions are then used by REVOLVER to estimate the time cost of individual communications within a program.

2.7.3 Cost Model Analysis

Using the techniques described in 2.7.1 and 2.7.2 we can compute a time cost value for individual operations, statements and message-passing communications. These costs can then be summed in a form of cost model analysis to derive a time cost value for

the whole program (see Figs 2.24, 2.25, 2.26) and Appendix A (pg 222).

```
1  float funct Workload_estimate (FILE*f)
2        var val, cost = 0.0
3          stmt := f- > firstStatement()
4          while (stmt) do
5                  val = getStatementCost(stmt)
6                  val = val * stmt- > NO_OF_EXECUTIONS
7                  cost+ = val
8                  stmt = stmt- > followingStmt()
9          od
10         return cost
11       .
```

Figure 2.24: Workload Estimation Algorithm

The *getStatementCost* function involves parsing any expressions in a statement and accumulating time-costs for all operations (+, -, *, /, etc) using a lookup table of benchmarked time costs (presented in Appendix A, pg 222).

```
1  float funct Communications_estimate (FILE*f)
2        var val, cost = 0.0
3          stmt := f- > firstStatement()
4          while (stmt) do
5                  if (stmt- > type() == SEND)
6                    then
7                          val = lookupMessageCost(stmt)
8                          val = val * stmt- > NO_OF_EXECUTIONS
9                          cost+ = val
10                 fi
11                 if (stmt- > type() == RECV)
12                   then
13                         val = lookupMessageCost(stmt)
14                         val = val * stmt- > NO_OF_EXECUTIONS
15                         cost+ = val
16                 fi
17                 stmt = stmt- > followingStmt()
18         od
19         return cost
20       .
```

Figure 2.25: Communications Cost Estimation Algorithm

Accumulate time-costs for all sends and receives executed. Since, in our model, all communication goes via the host (or Master) process (which is ALWAYS placed on

processor # 0) we need only accumulate time-costs for Sends and Recives performed by the Master process, to have a figure for the program as a whole (Fig 2.25.

```
1  float funct Performance_estimate (CODE − REC*code)
2        var *Master,* Slave, cost = 0.0
3          Master = code− > master()
4          cost+ = Communications_estimate(Master)
5          cost+ = Workload_estimate(Master)
6          Slave := code− > slave
7          while (Slave) do
8                  cost+ = Workload_estimate(Slave)
9                  Slave = Slave− > next
10         od
11         return cost
12       .
```

Figure 2.26: Performance Estimation Algorithm

2.8 Static Performance Estimation Research

Since the REVOLVER system is targeted at a distributed memory architecture, we consider here only research into performance estimation of parallel programs to be executed on distributed memory architectures.

2.8.1 Fundamental Approaches

There are fundamentally four different approaches to static performance estimation of parallel programs:

1. *Statistical.* Using ideas from probability theory, stochastic/Markov processes, and queueing networks, etc, a statistical model of the performance of the program is created with the aim of using this model to compute a single figure or performance equation. Often many simplifying assumptions have to be made.

2. *Simulation.* Discrete event simulations often take the form of graph representations and / or Petri nets (of several varieties). The simulation approach allows quite realistic estimates to be made but often

detailed simulation is difficult and time-consuming.

3. *Analytical.* Rather than executing an expensive, detailed simulation, this approach aims to concentrate on identifying the underlying features of the target architecture which exert the greatest influences on program performance. By characterising these fundamental features in the form of equations an accumulated 'cost' of executing the program can be quickly computed. The big advantage of this technique is that an analytical cost function can be created and fine tuned quickly. The technique however is not suited to estimating 'global' effects (such as resource contention), nor in cases where dynamic program behaviour is complex.

4. *Benchmarking.* This approach involves creating a suite of programs which contain typical features of the programs you wish to estimate the performance of (such as communication patterns, conditional evaluations, loop executions, subroutine calls, etc) and recording the time cost of these features as the programs execute on different problem sizes and on different numbers of processors. In this way, a 'database' of performance feature times can be accumulated. Concise, performance equations or functions may be fitted to these times for efficient use during estimation analysis. The difficulty with this approach is creating the initial representative suite of programs.

Since no one approach is clearly superior to any other a number of researchers have combined several features from each approach to create *hybrid models.*

Some early work on static performance estimation was performed by Cohen [32] whose combined approach of micro- and macro-program analysis to derive detailed time cost functions to estimate the execution time of sequential programs proved a starting point for much further research. The micro analysis phase involved 'mimicking' the actions of a parser in that it recognised the syntactic components of the program, but instead of generating code, a time cost function (which may have included branching

conditions) was constructed instead.

2.8.2 Performance Estimation of Parallel Programs

Compared to static performance estimation of sequential programs, estimation of parallel programs requires much greater program analysis and knowledge of the target architecture to take into account the additional effects of parallelism (scheduling, synchronisation, and communication).

The technique of benchmarking typical patterns of communication to determine their time costs was pioneered by Balasunderam, et al. [8] as part of the ParaScope restructuring tool. Their technique (called the *training set method*) involves benchmarking time-costs for typical communications patterns such as circular shifting, vector movement, broadcasting, and reduction operations. The data accumulated during this phase was then summarised by a line-fitting function (a variant on the chi-squared data-fitting method) and the function used during the estimation phase. During program analysis these costs were summed to derive a total cost value for the program as a whole. No profiling run was used. TRUE/FALSE ratios were assigned default values of 0.5, but could have been amended by the user as necessary. Their system did not handle subroutine calls and assumed constant loop bounds. The results of this approach however were promising in that they were able to accurately identify 'crossover-points' (i.e. decide at which point of data partitioning, one distribution strategy became preferable to another - block vs. column, for example).

A later version of the ParaScope system incorporates subroutine calls into its analysis (see [75]).

Fahringer [44] describes a portable performance predictor which uses a set of "training runs" to identify the relative importance of a set of six criteria, for the target architecture. The system (called the P^3 Tool) forms part of the Vienna Fortran Compilation System (VFCS [19]). A number of techniques are then used [46] to statically estimate the performance of parallelized SPMD Fortran programs with limited control flow. The approach includes the utilisation of information gathered during a single profiling execution of the original sequential program (see [45]) and also the identifica-

tion of six criteria used in computing the performance estimation value, namely: work distribution (i.e. how well the program workload is balanced across the processors); amount of data transferred; number of data transfers; network contention; transfer times; and estimated cache hit ratio. Statements in the program are then attributed with information used in the final summation analysis. Fahringer also shows that these attributes can be correctly maintained and recalculated under program restructuring transformations.

Clement and Quinn [31, 30] describe using statistical inference techniques combined with construction of an analytical model to estimate the performance of Data-Parallel C programs executing on multicomputer architectures. This approach has the advantage of allowing performance prediction without the need of a benchmarking or training run on the target machine.

In [121] Clement and Steed describe their APACHE (Automated PVM Application Characterization Environment) system to predict performance of PVM programs executing across networks of workstations. The system works on C programs with regular communication patterns and uses information accumulated during an initial profiling run.

Nudd *et al.* describe the PACE system [104] which takes an initial user description of a parallel program (the description is written in a scripting language) plus a description of the target architecture, and combine static analysis results with benchmarked times to allow user interrogation of the system to estimate performance across a variety of parallel models and architectures. Accuracy to within 10% of actual execution times are reported.

In [127], van Gemund describes a novel approach (called *serialization analysis*) which involves efficient compile-time construction of parallel program performance models,using a form of process-algebra type language called PAMELA. A number of rules are then applied to attempt to reduce the model to a single estimation number, or else to upper and lower bound estimates on performance, or else ultimately a simulated program execution is performed. The approach is particularly suitable for estimating resource contention.

2.9 Summary

In this chapter we have described the main ideas central to restructuring compiler theory. We have defined the major terms commonly used and reviewed the literature on dependency analysis and automatic parallelization techniques. We have also described the three phases of program analysis (dataflow, dependency, and inter-procedural) with particular emphasis on dependency analysis. This was followed by an extensive section on restructuring transformations and optimisations for loops, memory accesses, procedure/function calls and detailed examples of data decomposition techniques.

Static performance estimation was introduced as being of major benefit in the restructuring process since it is able to provide automatic feedback on the effects of applying a sequence of transformations. This section also contained descriptions of how program behaviour information is gathered through the use of profiling and how operations (including communication) can be benchmarked on a distributed memory architecture. An overview on the literature of static performance estimation was provided. The chapter concluded with algorithms illustrating how to sum the costs of individual statements to obtain a time-cost estimate for the whole program.

In the next chapter we give a detailed overview of evolutionary algorithms and indicate how they can be used to optimise the automatic parallelization process.

3 Evolutionary Algorithms

3.1 Introduction

In this chapter we introduce the idea of evolutionary algorithms. We will describe the mechanics of evolutionary algorithms, the different variations that exist, the features that affect the performance of an evolutionary algorithm (EA), and describe some of their applications.

3.2 Evolutionary Concepts

Survival in a largely hostile world is a problem that confronts all life forms as they compete to prolong their lives, to propagate, and to prosper. This not only applies to individuals but also to entire species. Individuals compete with other individuals, species compete with other species, and all the while the problem of survival never goes away. From the moment we are conceived to the moment we die, we all search for food, water, shelter and a mate. The actual realisation of these needs may vary - lions eat meat, whereas cows eat grass, for example, yet ultimately we are all looking for the same basic things. The fact that so much life exists on this planet, is a great tribute to the problem-solving skills of nature.

It is from this line of reasoning that researchers into problem-solving disciplines began experimenting in the 1960s with ideas based on observations made on how nature solves problems. Research work done in the United States and in Germany concentrated on manipulating chromosomes, but rather than biological threads of genetic structures these chromosomes were linear strings of binary digits. The binary strings were a lot smaller and a lot less complex than their biological counterparts, but just like them they encoded the characteristics of the individual they represented. What was more important for the researchers, was that the environment in which the individuals had to survive could be controlled and manipulated.

In order to solve a problem then, it was necessary to define it as a hostile environ-

ment, create an initial population of possible solutions, apply the laws of Darwinian evolution ('survival of the fittest') and see what nature could come up with. This simple but powerful process forms the basic mechanism of all evolutionary algorithms, both biological and digital. We now attempt a definition of digital evolutionary algorithms:

Definition 3.2.1. Evolutionary Algorithms. Evolutionary algorithms (EAs) are a class of direct, probabilistic search algorithms sharing features (such as a population of individuals, breeding, mutation, and selection of fit individuals) which are analogous to those found in the natural world. The population is made up of individuals, each of whom represents a possible solution to the problem. By applying evolutionary operators - (e.g. mutation, recombination, crossover, etc) - offspring are produced who are then evaluated for their 'fitness' as a solution to this problem. This process is repeated in a sequence of steps (called *generations*) such that the repeated generation of offspring, evaluation of their fitnesses and propagation of the fittest members causes the population as a whole to evolve towards an optimal solution to the problem.

This basic population-based generate-and-test approach has been adapted and applied to many different problem-domains resulting in five broad streams of research, evolutionary programming (EP), evolutionary strategies (ESs), genetic algorithms (GAs), genetic programming (GP), and classifier systems (CFS) (see [53, 37, 98]). Only genetic algorithms and evolutionary strategies will be of interest to us in this thesis.

3.2.1 Stochastic (or Random Mutation) Hillclimbing (RMHC)

Hillclimbing is a form of simple generate-and-test search strategy. Given a function to optimize (called f) and a set of possible solutions (we call S). The set of all possible solutions form the solution-space (sometimes called search-space, or fitness landscape). A hill-climbing search then proceeds in the following way:

1. We take an initial possible solution $s \in S$ and evaluate its fitness by applying f. We make s the 'current solution'.

2. We then make a slight random change to s to produce a new solution s'.

3. If $f(s') \geq f(s)$ then make s' the new current solution and discard s, else simply discard s'.

4. If some termination criteria is $TRUE$ then stop, else return to step 2.

RMHC is computationally efficient and tends to be quite quick, however it is easily trapped by local maxima in the solution space.

Definition 3.2.2. Fitness Landscape (or Solution Space). The concept of fitness landscape is very similar to the idea of a solution-space to a problem. Each node in the space represents a potential solution to the problem, however in a fitness landscape each node is also annotated with a value which would be the 'fitness' of the solution at that point in the space. As such, a fitness landscape is often presented as a three dimensional landscape with peaks and troughs which represent undulations in the solution space. The aim of an EA is to find the globally deepest trough (when minimizing a function), or to find the globally highest peak (when maximizing). If the undulations of the fitness landscape are smooth, then an EA should find a good solution to the problem. However, if the landscape is uneven and discontinuous, then the progress of the EA towards global optima may be hindered.

3.2.2 Steepest Ascent Hillclimbing (SAHC)

This is a very similar technique to RMHC, however it introduces the important idea of a 'neighbourhood'.

1. Take an initial possible solution $s \in S$ and evaluate its fitness by applying f. We make s the 'current solution'.

2. We then generate and evaluate all the solutions in the immediate 'neighbourhood' of s. The immediate neighbourhood of s is the set of all solutions one small change (a mutation, see Fig. 3.1) away from s. We choose the best solution in the neighbourhood of s and call it s'.

3. If $f(s') \geq f(s)$ then make s' the new current solution and discard s, else simply discard s'.

4. If some termination criteria is $TRUE$ then stop, else return to step 2.

SAHC is more computationally expensive than RMHC (especially when the neighbourhood is large) and is still easily trapped by local maxima or local maxima plateaux.

Before 0 0 0 0 0 0 0 0 0 0

Random Mutation point = 4

After 0 0 0 1 0 0 0 0 0 0

Figure 3.1: Uniform Mutation Operation on a Binary String

3.2.3 Simulated Annealing (SA)

Simulated annealing incorporates the ability to 'escape' local maxima and therefore has more chance of finding the global maximal solution. The basic idea here is that small mutations may produce new solutions (offspring) which may sometimes be accepted as the new current solution even if it evaluates worse than the present solution. The chances of this happening however decrease as the search progresses and also decrease as the difference between the two solutions increases.

Definition 3.2.3. Fitness Function. A function applied to the problem variables which the user wishes to optimize (i.e. maximise or minimize) sometimes within a set of constraints. (In classic optimization terminology this is often called the 'objective function'). The *feasible region* of an optimization problem is the set of all points satisfying the constraints (if any) of the problem. For a maximization problem, an optimal solution is a point in the feasible region with the largest objective function value. Similarly, for a minimization problem, an optimal solution is a point in the feasible region with the smallest objective function value.

A fitness function must be devised for each problem to be solved. It takes as input a single chromosome and returns a single numeric value which should be some measure of the 'quality' or 'goodness' of the solution represented by the chromosome. As such it provides the 'feedback' needed to evaluate the quality of the solutions represented by each member of the population.

The distance between any two points in the fitness landscape (that is, any two potential solutions) can be measured in terms of the number of times a genetic operator has to be applied in order for one solution to change into the other.

3.2.4 Multi-Start Hillclimbing (MSHC)

This is a variation on SAHC but has the advantage of being able to escape local maxima. The idea very simply is to restart the hill-climber once the current search has stopped finding improvements, but to retain the best 'current solution' across searches.

1. Take an initial possible solution $s \in S$ and evaluate its fitness by applying f. We make s the 'current solution'.

2. We then generate and evaluate all the solutions in the immediate 'neighbourhood' of s. The immediate neighbourhood of s is the set of all solutions one small change (a mutation, see Fig. 3.1) away from s. We choose the best solution in the neighbourhood of s and call it s'.

3. If $f(s') \geq f(s)$ then make s' the new current solution and discard s, else simply discard s'.

4. If the search has found no improvement for N iterations, randomly generate a new solution s' and evaluate it's fitness. If $f(s') \geq f(s)$ then make s' the new current solution and discard s, else simply discard s'. Now return to step 2.

Restarting the search randomly allows the hill-climber to 'jump' to a new location in the search space and climb the nearest hill. Once that has been climbed, it jumps somewhere else and repeats. This search is no more memory intensive than SAHC although obviously to be effective a larger number of iterations have to be performed to allow several hills to be climbed.

3.2.5 Genetic Algorithms (GAs)

Genetic algorithms (GAs) were devised and researched by Holland [66] in the 1960's and 70's. They were originally applied to optimization, search, and classifier (production-

rule) system problems. Generally, GAs work on binary strings of fixed length, with a fixed population size. The operators which followed naturally from this representation were a 'bit-flipping' mutation operator (Fig. 3.1), a random template crossover or *recombination* operator (Fig. 3.3), and a fitness proportional selection operator ('roulette wheel' selection). The simple GA, as devised by Holland is described in Goldberg [53] and is illustrated in Fig. 3.2.

```
1  begin
2     t := 0
3     initPopulation P ( t )
4     evaluate P ( t )
5     while ¬terminate do
6              t := t + 1
7              P' := selectParents( P )
8             recombine( P' )
9             mutate( P' )
10            evaluate( P' )
11            P := P'
12    end
13    printResults()
14 end
```

Figure 3.2: The Simple Genetic Algorithm

The simple genetic algorithm (SGA) (sometimes called 'canonical' GA) works in the following way: each iteration of the main loop is referred to as a generation, the first generation (generation 0) operates on a population of individuals initialised in some way (e.g. randomly, biased). Next, evaluate the fitness of all the individuals in the population, then select a proportion of the population to be the parents of the next generation (usually choose the fitter members of the population). Create the next generation by performing operations such as crossover and mutation to the selected population. Finally, the entire current population is replaced by the new population.

This loop is repeated until the termination criterion has been satisfied. An EA may terminate in at least three ways:

1. After a preset number of generations has been reached.

2. After a preset time-limit has expired.

3. After the population has 'converged' around some optimal point in the search space. Exactly how convergence is defined varies but usually it is some measure of increased similarity between members of the population.

In this way, the combination of applying genetic operators and evaluating the fitnesses of the offspring, work together to improve the fitness of the whole population. In other words, they evolve towards and ultimately converge upon a global solution to the problem.

Parent 1	0 0 0 0 0 0 0 0 0 0
Parent 2	1 1 1 1 1 1 1 1 1 1
Random Template =	0 1 0 1 0 1 1 1 0 1
Offspring 1	0 1 0 1 0 1 1 1 0 1
Offspring 2	1 0 1 0 1 0 0 0 1 0

Figure 3.3: Uniform Crossover Operation on Binary Strings

Parent 1	4 6 7 1 0 | 9 2 2 1 5
Parent 2	6 8 1 3 2 | 7 7 1 0 4

Random Crossover point = 5

Offspring 1	4 6 7 1 0 7 7 1 0 4
Offspring 2	6 8 1 3 2 9 2 2 1 5

Figure 3.4: One-Point Crossover Operation on Integer Strings

3.2.6 Evolution Strategies (ESs)

Evolution strategies (ESs) [118, 65, 5] were developed (independently of Hollands' work) also in the 1960's by Bienert, Rechenberg and Schwefel, at the Technical University of Berlin. ESs were devised as a simple means of optimizing the shape of equipment used in hydrodynamical engineering, such as the shape of a bended pipe and the drag minimization of a joint plate.

```
                          |         |
Parent 1        4 6 1 | 1 0 9 2 | 2 1 5
Parent 2        6 8 7 | 3 2 7 7 | 1 0 4
                          |         |

Random Crossover point 1 = 3
Random Crossover point 2 = 7

Offspring 1     4 6 1 3 2 7 7 2 1 5
Offspring 2     6 8 7 1 0 9 2 1 0 4
```

Figure 3.5: Two-Point Crossover Operation on Integer Strings

Instances of ESs are usually described by three parameters: μ is the size of the population, λ is the number of offspring produced each generation, and the third parameter, which represents the reinsertion strategy is either a plus sign '+' or a comma ','. When an ES utilises a *comma-strategy* it means that the members of the next generation will be selected *only* from the offspring generated, no members of the original population will survive into the next generation. When an ES utilises a *plus-strategy* the members of the next generation may be selected from either the parents *or* the offspring generated.

At present, the ES(μ,λ) represents the most advanced evolutionary strategy devised.

```
1  begin
2     t := 0
3     initPopulation P ( t )
4     evaluate P ( t )
5     while ¬terminate do
6           t := t + 1
7           P' := selectParents( P )
8           mutate( P' )
9           evaluate( P' )
10          P := P'
11    end
12    printResults()
13 end
```

Figure 3.6: The Evolution Strategy Algorithm

3.2.7 Comparison of GAs and ESs

There are several notable differences between the GAs and ESs. These differences have implications on the way each approach attempts to solve a problem, and also therefore on the kinds of problems each approach works best on.

The traditional GA works on binary strings, of fixed-lengths, with a population of a fixed size. The binary-string representation gives rise to two main operators; a bit-inversion mutation operator, and a uniform crossover operator with the mutation operator seen as a largely 'background' operator. The traditional selection operator is the fitness proportional 'roulette-wheel' method. The operator probabilities are fixed throughout the duration of the GA. A percentage of the population may be replaced by offspring after each generation (a steady-state GA) or the whole population may be replaced (a generational GA).

Evolution strategies by contrast were developed to work on strings of floating-point numbers, which may or may not vary in length (depending on the problem being tackled), and where the population size may fluctuate as the search progresses. The use of the floating-point representation gives rise to a more complex mutation operator, one which inflicts mutations with varying degrees of severity (e.g. a floating-point number may be mutated within a small or large range about its present value). As such, mutation is seen as a much more influential operator than crossover - indeed in a ES-(1+1) there is no crossover operator. Selection is determined by the strategy being employed (i.e. *comma* or *plus*). Some ESs also allow 'global' mutation and crossover operators, that is, operators which use many parents (or even the whole population) and recombine them in some way to produce one (or several) offspring.

3.3 Further Issues of EAs

There are numerous variations on the basic EA. One of the most important differences between GAs and ESs is the notion of *self-adaptation*. This idea originally took hold amongst ES researchers but has quickly been assimilated into GAs too.

Definition 3.3.1. Self-Adaptation. The notion of self-adaptation, as referred to by GA researchers is one where a search algorithm has the ability to change its own parameter values as the search progresses. These changes can be implemented in many ways but are usually triggered either by a predefined 'schedule' (e.g. change an operator probability every N generations) or else by the search-algorithm itself based on evaluation of conditions relating to how well the search is perceived to be progressing.

Researchers have experimented with adapting operator probabilities (crossover, mutation, etc) as well as other EA parameters (e.g. population size) and even chromosomal representation (by introducing a 'mutation gene') in investigating different ways to introduce self-adaptation into EAs.

The idea of changing parameter values as the EA progresses is based on an implicit realisation that different elements of the EA become important at different stages of the search. For example, the crossover operator is particularly important in the early stages of a search so as to achieve an initial wide 'spread' of the population across the fitness landscape - as the search draws to a close however, the mutation operator becomes more important since it may have the effect of 'fine tuning' some of the solutions found so far.

Why do GAs Work?

The most common explanation of why GAs work is due to Holland and is well documented in the literature [53, 66]. As such, it will only be briefly presented here.

The explanation offered applies to GAs which operate on binary-string encodings and relies on the notion of a *schema*. A template which can be used to match binary strings by use of a three letter 'alphabet' - 0, 1, and a don't care '*' symbol. So for example, if we consider strings and schemata of length 10, the schema

$$*\,1\,0\,1\,0\,0\,1\,1\,1\,0$$

matches the two strings

$$0\,1\,0\,1\,0\,0\,1\,1\,1\,0$$

$$1\,1\,0\,1\,0\,0\,1\,1\,1\,0$$

Logically, the schema 0101100010 matches only one string, and the schema **********, matches all strings of length 10. Conceptually, a schema represents all strings (a *hyperplane*, or subset of the search space) which match it on all positions other than '*'. The first property of schema to note is the idea of the *order* of a schema. This is defined as the number of 0 and 1 positions (i.e. fixed positions) within a schema. A further property of any schema is it's *fitness* at a particular time t. This is defined as the average fitness of all strings in a population matched by the schema at time t. This now allows us to express a theorem:

Schema Theorem "Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm".

The idea here is that the GA makes progress towards the global optima by processing the many thousands of schemata that match the members of the population. (This processing is called *implicit parallelism* by Holland). The implication of this theorem is that GAs progress across the search space in hyperplanes, represented by short, low-order schemata. The information they encode is exchanged through the crossover operation.

Building Block Hypothesis "a genetic algorithm seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata, called building-blocks".

This hypothesis is as yet unproven and is taken on faith by some GA researchers. Nonetheless, the hypothesis suggests that the problem of coding for GAs is crucial for its performance, and that such a coding should certainly satisfy the notion of short building-blocks.

Criticisms of the Schema Theorem

In recent years several significant criticisms have been made of the schema theorem, notably that the inexactness of the theorem makes it impossible to predict computational behaviour. A further criticism is that some schemata can mislead the GA to converge the a local optimal solution rather than the global. A simple example can illustrate the problem : suppose we wish to optimise a function f, where the GA implicitly processes

schemata of length 10. Suppose we identify the following subset of order 2 schemata which are being processed by the GA:

$$S_1 \ * \ * \ * \ 0 \ * \ * \ * \ * \ 0 \ * \qquad f00$$
$$S_2 \ * \ * \ * \ 0 \ * \ * \ * \ * \ 1 \ * \qquad f01$$
$$S_3 \ * \ * \ * \ 1 \ * \ * \ * \ * \ 0 \ * \qquad f10$$
$$S_4 \ * \ * \ * \ 1 \ * \ * \ * \ * \ 1 \ * \qquad f11$$

(we use $f00$, to represent the fitness of schema S_1, $f01$ for S_2, and so on). Now let us assume that the fitness values for the 4 schemata are ordered in the following way,

$$f11 > f00 > f01 > f10$$

and that furthermore, $f11$ is the global optimal solution. If a crossover operation is applied to two individuals, one who matches S_1 and one who matches S_4, the chances are very high that BOTH children potentially produced will be LESS FIT than BOTH parents - even though both parents are short, highly fit schemata (indeed, S_1 matches the global optimal solution). Hence these schemata are leading the search *away* from the global optimal solution.

For further criticisms of the schema theorem, see Dorigo [39]. One recent alternative explanation for the effectiveness of GAs has been suggested by Hirst [61] (i.e search-space neighbourhoods as an explanatory device).

Epistasis, GA Deceptive Problems, and Messy GAs

A further feature to take into account in the design of an EA is the potential existence of *epistasis*.

Definition 3.3.2. Epistasis. When GA researchers use the term epistasis they are referring to any kind of interaction between genes within a chromosome. It is used as an indication of how much the contribution of one gene to the chromosomes fitness is dependent on the values of the other genes in the chromosome.

High levels of interaction between genes can result in a GA converging to local optima. As such chromosome encodings which suffer from epistasis should be avoided by choosing a different representation wherever possible.

In [16] and [17], Beasley *et al*, distinguish between three levels of gene interaction:

- Level 0 - No interaction. A particular change in a gene always produces the same change in fitness.
- Level 1 - Mild interaction. A particular change in a gene always produces a change in fitness of the same direction (or of zero).
- Level 2 - Strong interaction. A particular change in a gene may produce a change in fitness which varies in sign and magnitude, depending on the values of other genes.

In terms of the building-block hypothesis the difficulty caused by strong gene interaction is that for a given problem, a high level of epistasis means that building blocks cannot form, resulting in the GA converging to a local optima - such a problem is said to be *GA-deceptive*.

Three approaches have been put forward to deal with epistasis, namely: (i) use heuristic information to find an alternative encoding, (ii) introduce an *inversion* operator, and (iii) re-encode the problem in the form of a *messy-GA* as described by Deb and Goldberg [54, 55].

3.4 Heuristic Information

In [37], Davis states that "traditional genetic algorithms, although robust, are generally not the most successful optimisation algorithm on any particular domain". Davis argues that incorporating the most successful optimization techniques for a problem into a genetic algorithm gives one the best of both worlds : indeed, when implemented correctly, these algorithms should do at least as good as (and usually better than) the method with which the hybridizing of the GA is done. (While introducing the additional computational cost of a population-based search).

Definition 3.4.1. No Free Lunch Theorem. In this research [145, 146], Wolpert and Macready show, with rigorous mathematics, that all algorithms that search for an extremum of a cost function perform exactly the same, according to any performance

measure, when averaged over all possible cost functions. In short, if search algorithm A outperforms search algorithm B on some cost functions, then loosely speaking, there must exist exactly as many other (probably inverse) functions where B outperforms A.

The significance of Wolpert and Macready's work is that it emphasises the importance of including heuristic information in a search through a solution space. Simply relying on a 'pure' general search technique will almost certainly be slower and produce poorer quality solutions. In order to get the best results one *must* incorporate *a priori* knowledge into the search - there's no such thing as a free lunch.

The relevance for tackling a problem of the magnitude of automatic parallelization is that in order for the approach to succeed, a large amount of heuristic knowledge about how to parallelize sequential programs must exist - and must be capable of being incorporated into the evolutionary algorithms we use. Based on the vast literature that exists on automatic parallelization, this knowledge would appear to be available.

Justification for New Operators

Many new operators have been created and studied in recent research. Some have proved effective for various problems, many have not. The justification for creating a new operator should not be seen as simply finding a new way of putting two chromosomes together to produce two new ones. The key to a successful operator is to imagine the effect it will have on the population of search strings as they move across the fitness landscape. If the operator will cause strings to move towards optima (preferably global) then it should be an effective one, otherwise it may not be worth further consideration.

Neighbourhood

Any point P in the search-space has a set of surrounding solution points called a neighbourhood. Each point in the neighbourhood is defined as being one 'operation' away from P (traditionally a mutation). Notably - different operators will define different neighbourhoods for P.

$$\text{Size of Neighbourhood} = \text{No. of points one mutation away.}$$

Some EAs employ localised searches to search the neighbourhood of a string in the population (usually a GA or ES in combination with a simple hill-climber). In [61] Hirst discusses how neighbourhoods may lead to a better understanding of how and why EAs work, (as opposed to the schema-theorem).

3.5 Applications of EAs

Most notably, although pure ESs and GAs have proven themselves robust and effective optimization techniques for general purpose use, they have also shown themselves to be suitable to customising for tackling a wide variety of problems in scheduling and order-based problems, constrained problems, and noisy, multimodal optimisation problems (e.g. traveling salesperson problem [53, 98], shape design [118], evolving circuit design [125], neural network optimisation [99]). The incorporation of problem specific heuristics to further aid the evolutionary process often involves devising new representations and subsequent genetic operators to suit the problem domain. The creation of these *hybrid-EAs* has proved particularly popular since hybrid techniques can guarantee that the best solution found by the EA will be at least as good as that found by the conventional techniques used to hybridise the EA.

Apart from strings of binary digits, integers or reals, other data representations (e.g. matrices, graphs, trees) are more suitable for a variety of problems, giving rise to other more appropriate versions of mutation and crossover operators.

Another reason for the popularity of evolutionary algorithms is that they are inherently parallelizable into one of three formats (i) massively parallel GAs [97] (ii) parallel island models [131], and parallel hybrid GAs [97, 51], and may be (at least) partially implemented in hardware [92].

Because the choice of problem representation (and hence, operator design) is so problem-specific it is difficult to generalize about any information found in the literature regarding other, similar problems being tackled by the evolutionary approach.

There are problems for which any feasible solution would be of value. In such problems we are not really concerned with optimization issues (i.e. finding *the best* feasible solution) but rather we try to find *any* point in the feasible search space. Such

problems are called *constraint satisfaction problems*. A classical example of this type of problem is the N-queens problem where the task is to position N queens on a chess board such that no two queens may attack each other. This problem (among others) was tackled with specialized operators in [42].

Some EAs temporarily allow infeasible solutions into their population since they may provide a 'short-cut' to a good feasible solution. However, such individuals must be identified and handicapped by penalty-functions (decrease fitness of invalid individuals) or ultimately a death-penalty - (delete invalid individuals). 'Repair' techniques - allow creation of invalid individuals, then correct them afterwards.

GAs with variable-length chromosomes (GAVaCLs) have been used before in robot path planning problems (where an upper bound was placed on how long the chromosomes could become [83, 84, 98].

Notably GAVaCLs were also used in optimization of rule-based systems. Smith [119] generalized many results previously valid only for GAs with fixed length strings to apply to GAVaCLs, again making extensive use of specialized operators.

Evolutionary Approaches to Automatic Parallelization

The increasing use of evolution represents a significant new trend in parallelizing compiler technology. Several approaches using genetic programming, genetic algorithms and neural networks have been developed in recent years - with some research still continuing.

PARAGEN and S.C.A.R.E

Walsh and Ryan describe using genetic programming techniques to parallelize sequential Occam programs in [128]. Their system (*PARAGEN*) converts sequential programs into functionally equivalent parallel code without using dependency-analysis by breeding parallel versions of the original sequential program. Genetic operators are applied at the statement level. Each parallel 'individual' is scored for *correctness* and *parallelism*. The final 'perfect' program must (and can be proved to be) be 100 percent correct, but at intermediate stages the system may contain programs that are only

partially correct, and if they contain enough parallelism.

The benefits cited include : no need for costly dependency-analysis; final Occam code can be formally verified for correctness using Communicating Sequential Process (CSP) techniques [63]. Negative points seem to be : evaluation of a programs 'fitness' will be time consuming ; results will be unpredictable (final program may not be very parallel - but this is also true of many parallelizing compilers); and knowing when to terminate the search. Ultimate aim of the approach is to be able to generate code direct from CSP specifications (i.e. no need to write an initial sequential program).

This line of research has evolved into ongoing work by the Soft-Computing And Re-Engineering (SCARE) group at the University of Limerick. A second system *PARAGEN II* is being developed which uses a combination of genetic-programming and genetic-algorithm techniques to parallelize sequential loops.

Genetic Compiling

An early genetic algorithm compiling technique for automatic parallelization (*genetic compiling*) is described by Williams in [135]. This paper describes the first genetic algorithm (as distinct from genetic program) for automatic parallelization of sequential programs. The paper (i) describes a GA with variable chromosomal lengths (a *GAVaCL*) as a possible representation of the automatic parallelization problem, (ii) describes the effects of three possible mutation operators as well as one and two-point crossover operators, and (iii) introduces a 'reduction transformation' as a means of evaluating the fitness of the code produced. The 'reduction transformation' has since been replaced by static performance estimation. No results were presented.

Genetic Algorithm Parallelization Systems (GAPS)

The GAPS system developed by Nisbet [101, 102] is an ongoing research project at University of Manchester, UK, investigating a genetic algorithm approach working with the `Petit` /UTF framework of Kelly (see section 2.5) and is partly inspired by Williams [135].

In GAPS, each chromosome represents a list of time-mappings for each statement (or block) in a program. Three operators (group-based mutation, group-based crossover, and iteration-based crossover) can be applied which may effect some combination of loop-fusion, statement-reordering, or loop-distribution. SPMD code for a shared-memory multiprocessor is generated. Two fitness functions have been investigated : naive overhead estimation, and actual execution times. Greatest improvements in fitness are when actual execution times are used.

Interestingly, application of genetic operators may create illegal time-mappings - which can be detected. These are retained in the population since they may provide a 'short-cut' towards some optimum. The fitness function is only applied to legal members of the population. Results indicate 21-25 % performance improvements against parallel code simply generated using `Petit` / UTF.

Hybrid-GA Task Allocation (HGATA)

In [91], Mansour and Fox describe a hybrid genetic algorithm for task allocation (HGATA) of workloads to a multicomputer keeping the workload balanced and communication down to a minimum as much as possible. They used a simple integer chromosomal representation to allocate task i to processor j. Operators used were mutation, inversion, and two-point crossover - their probability rates adapted to maintain diversity. Notably also employed a localised, greedy hill-climber with the GA to take advantage of problem-specific heuristics, this was found to improve the efficiency of the search significantly. Several variations of the GA were tested, all produced good results ($\approx 90\%$ optimal) except where local hill-climbing was not used.

Cellular Encoding

A novel approach utilising neural networks (in a technique called '*cellular encoding*') is used to parallelize Pascal programs as described in Gruau [59].

3.6 Summary

In this chapter we have given a brief introduction to evolutionary computation and defined a number of key concepts. We have described several evolutionary algorithms and indicated their variants. A brief comparison of two classes of EAs, namely GAs and ESs was made and some background theory (schema theorem, epistasis, No Free Lunch theorem, search-space neighbourhoods) was described. Some important applications of EAs were noted.

The chapter finished with an important section on evolutionary research into the automatic parallelization problem which sets the context in which the research presented in the following chapters can be placed.

4 Evolutionary Parallelization Models

4.1 Introduction

In this chapter we describe the current implementation of the REVOLVER system. We begin with the current features and limitations of the system, then describe two representations and associated evolutionary operators used to restructure code. An example parallelization session with REVOLVER is then presented and is used to illustrate the various features and functionality of the system. The hardware and software environments of the Meiko Computing Surface (CS-1) target message-passing architecture are then described, along with more detailed descriptions of the three REVOLVER processing modes, as well as the profiling, and static performance-estimation tools. Finally an extensive description of the code-generator is provided. The chapter concludes with considerations of interaction of these components of the overall system.

4.2 Implementation Details

The current implementation of the REVOLVER system uses the *Sage*$^{++}$ (version 1.7) restructuring compiler libraries and the `Omega` (version 3.0) dependency analysis libraries. The REVOLVER system parallelizes sequential Fortran-77 (F77) programs into a message-passing Master/Slave model of parallelism for execution on a Meiko CS-1 network of 12 $\times$ T800 Transputers connected in a 2-D mesh. It has been written in C/C^{++} on a SUN-4 SPARC Workstation (32Mb RAM) running the SunOS 4.1.3 operating system and compiles with the GNU g^{++} compiler (version 2.5.8). The current size of the system written by the author (i.e. excluding the library code) is approximately 21,000 lines, excluding comments. (This code is contained on the CD-ROM which accompanies this dissertation). In the best traditions of software engineering the various components of the system (e.g. profiler, performance predictor, transformation catalogue, evolutionary algorithm code, etc), have been implemented in a modular style

so as to allow flexibility and ease of future development.

The implementation is command-line driven and has no need of any windowing interface systems. The REVOLVER system consists of a profiler for gathering execution data of sequential F77 programs. This data is then written to a file and used by a performance estimation tool to statically analyze and estimate the execution time of a parallelized version of the program (parallelized for the CS-1). This is done automatically as part of the genetic algorithm, where the estimated execution time of the parallelized code represents the 'fitness' of the code produced by the restructuring performed so far. The objective function then is to minimise the execution time of the parallelized code produced by the restructuring process.

4.3 Assumptions and Features

The current REVOLVER implementation makes a number of assumptions about the F77 input code.

The first pre-requisite for automatic parallelization is that the original program can be compiled successfully with a standard Fortran-77 compiler. Other assumptions and features include:

- Automatically restructures and parallelizes Fortran-77 programs. Target architecture is a 12-node Meiko CS-1 computing surface using a message-passing model of parallelism.

- Only restructures and parallelizes Fortran-77 programs.

- Currently has small transformation catalogue (5 transformations), namely loop-fusion, loop-splitting, loop-interchange, loop-reversal and loop-normalisation. The loop-normalisation transformation is applied to loops immediately before attempting one of the other transformations.

- Naive parallelization strategy. After restructuring, all parallelizable loops are parallelized. This is a naive strategy because executing parallelized code introduces time-cost overheads (such as process start-up

times and communication costs) which may negate the benefit of executing the loop in parallel. Notably, communication costs *are* taken into account by the performance estimator.

- Only parallelizes loops (i.e. not blocks of code). Cannot parallelize loops that are nested within another loop, for two reasons (i) if the outer loop is also being parallelized then this would result in tree-parallelism, the performance predictor is currently only capable of analysing master/slave models of parallelism, and (ii) if the outer loop is *not* being parallelized then this would mean that multiple instantiations of the loop (and therefore the slave nodes) would be needed - this is not currently possible with the target architecture (CS-1), once a slave process has been terminated it cannot be reinstantiated.

- All programs are parallelized to a restricted master/slave model of parallelism. This was chosen because performance estimation is significantly easier with this model of parallelism. The REVOLVER system may be extended to use other models at a later date.

- No inter-procedural analysis is performed. As such no subroutines or functions can be handled in the current implementation. This may be overcome in future implementations where procedures are simply 'inlined' at their call sites, although this approach will run into problems if cycles exist in the program call-graph.

- Loops with non-constant bounds, and nested loops with coupled array access subscripts (such as skewed-loops), both cannot currently be analysed by the system. This is because of the extensive analysis that is required to restructure and parallelize loops of these kinds. Skewed loops are often found in 'wave-front' type applications. Improved analysis techniques to handle these types of loops may be added in the future.

- Values of all variables used in loop expressions (and their dependen-

cies) must be available (or capable of being determined) at compile-time.

- Cannot manipulate non-linear array subscript (*redirection*) expressions, e.g.

```
program redirection
integer i, N, a(100), b(100)
......
do  i = 1, N
   ... = b(a(i)) .....
enddo
......
```

- Implementation must be 'bullet proof' - i.e. transformations must be encoded to either succeed or fail but cannot be allowed to cause the program to crash. This is particularly difficult since each transformation has to be prepared for a wide range of possible inputs. No assumptions can be made about the input being 'reasonable' - since applying a sequence of transformations 'blindly' (as the EA will do), may well propagate highly unusual control structures (e.g. loop nests several levels deep, array subscripts containing highly complex expressions, complex combinations of if-statements and loop-structures, etc) which will be difficult to analyze and manipulate.

  This is also particularly true when attempting to debug a genetic algorithm, since the GA may execute perfectly well for say, 50 generations and then crash. In order to identify the problem you then have to examine the output to try to recreate the conditions which caused the GA to crash before you can attempt to fix the problem.

- Symbolic Analysis. We have imposed a condition that all values are known, or are capable of being known at compile-time. This is achieved by simplifying the programs we parallelize (i.e. no subroutines allowed, no dynamic structures, etc). Part of the program normalisation routine involves extensive propagation of numerical constants. The values

of any remaining symbolic values at any given point in the program can then be computed as and when needed. This is achieved by a recursive algorithm which searches lexicographically back through the program for the last *live* definition of the variable. If the right-hand side value of this definition is a constant expression, then the value of the variable is known. If the value is a variable expression, then the algorithm recurses to determine the values of the symbolic variables within the expression, since the values of all variable are known at compile time, the algorithm is guaranteed to terminate successfully. Hence the value of any variable or expression at any given point in the program is capable of being determined.

4.4 Problem Representations

It is well known that the choice of representation of a problem can often determine the success or failure of tackling the problem. In this section we present two representations of automatic parallelization (the *Gene-Transformation* representation, and the *Gene-Statement* representation) both of which are suitable for manipulation by evolutionary algorithms.

4.4.1 The 'Gene-Transformation' Representation

The first representation of the automatic parallelization problem developed (as presented in [135]) is called the 'gene-transformation' representation. As the name suggests, in this representation a single gene represents a single optimising transformation to be applied. Hence a whole chromosome (i.e. a sequence of genes) represents a sequence of transformations (called a *schedule*). Each transformation may be abbreviated to a TLA (three letter acronym), e.g.

LFU Loop Fusion.

LSP Loop Splitting.

LRV Loop Reversal.

LIC Loop Interchange.

... etc ....

Currently the REVOLVER transformation catalogue only consists of loop transformations, other transformations may be added and encoded in a similar style.

Our program model also requires some means of referring to individual loops within a program. Consequently, in a program containing n loops for example, we assign a number to each loop (from zero upwards) in lexicographic ordering. Hence all loops in the program will be assigned a unique number in the range [**0** ... *n-1*]).

All loop transformation functions currently take at least one parameter - the number of the loop they are to be applied to. For example:

LFU(6)	"*Fuse together loop 6 and any immediately following or immediately preceding loop in the program.*"
LIC(3)	"*Apply loop-interchanging to loop 3 in the program.*"
LSP(7)	"*Apply loop-splitting to loop 7 in the program.*"

Using the model we have defined so far we thus have a means of describing variable-length sequences of optimisations and transformations (applied, in order from left-to-right), e.g.:

[*LSP(3), LIC(5), LFU(7)*]

This chromosome consists of three genes which encode the following sequence of transformations:

"*Apply a loop-splitting transformation to loop 3 in the program followed by interchanging loop 5 and then finally loop-fusion to loop 7.*"

Given the representation thus far a number of issues now arise which need to be addressed, namely:

1. What to do if we try to apply a transformation which cannot be legally applied at the point where it is scheduled (i.e. it 'fails')?
2. How to further specify how some transformations are to be applied (in particular loop-fusion and loop-interchanging).
3. What evolutionary operators does the representation naturally give rise to?
4. What evolutionary algorithms are best suited to working on this representation?
5. What is the 'fitness function" (i.e. the function to be optimised) of the evolutionary algorithm?

These questions will be answered in the following paragraphs.

Decoders

It may happen in the course of restructuring that a particular transformation which is scheduled cannot be applied (due to dependencies, or the loop not being of the correct type, etc). This conflict may be resolved in one of three ways, respectively known as: *delete-and-continue*, *delete-and-stop*, and *repair*.

- *delete-and-continue*: if a transformation cannot be applied, delete the gene from the chromosome (i.e. the transformation from the schedule) and continue processing the schedule.
- *delete-and-stop*: if a transformation cannot be applied, delete the gene from the chromosome and stop processing the rest of the current schedule.
- *repair*: if a transformation cannot be applied, replace the gene with a randomly generated new gene. If this new gene cannot be applied then continue to generate and test new genes until a legal one is created - (essentially the approach is to 'repair' the chromosome so it contains a valid sequence of transformations.

The process of resolving these conflicts is called *decoding*. The current version of REVOLVER implements all three of these *decoders*. Which decoding strategy is to be employed may be specified by the user at run-time as a compiler-switch (the default value is *delete-and-continue*).

Transformation Specification

Using the 'gene-transformation' representation, some transformations may be implemented in more than one way (in particular loop-fusion and loop-interchanging).

In REVOLVER the loop-fusion transformation is implemented as: first, attempt to fuse the specified loop with an immediately following loop, if this is not possible then attempt fusion with an immediately preceding loop, if this is not possible then the transformation fails.

The loop-interchange transformation is implemented in the following manner: first, if the specified loop is not the outermost loop of a perfect loop-nest then the transformation fails. Otherwise, if the depth of the loop-nest is two, then simply interchange the two loops, else (i.e. depth more than two) then generate some random permutation of the loops and attempt interchange accordingly.

Operators and Algorithms

Using the 'gene-transformation' representation, what operators naturally arise, and what evolutionary algorithms work best with this representation?

The notion of mutation usually involves making small random changes to the chromosome (with the aim of improving the 'fitness' of the chromosome). Hence, three mutation operators immediately come to mind - (i) mutate the transformation specified by the gene, (ii) mutate the loop specified by the gene, and (iii) mutate the whole gene. These three mutation operators, (all implemented in REVOLVER) currently replace mutated information with random values, more purposeful replacement of information (perhaps using heuristic information) is considered in chapter 6. Mutation of an entire gene is illustrated in Fig. 4.1.

The notion of crossover usually involves *recombining* two chromosomes in some mean-

```
======
BEFORE
======
 Chromosome  LSP LFU LIC LFU LFU LIC LFU
               3   4   8   7   8   8   9

( Random Mutation point = rand(1, strlen(Chromosome)) = 4)

=====
AFTER
=====

 Chromosome  LSP LFU LIC LRV LFU LIC LFU
               3   4   8   2   8   8   9
```

Figure 4.1: Gene-Level Mutation Operation Randomly Changes both Transformation and Loop Number Parameters

ingful way so as to produce two new offspring. Using the 'gene-transformation' representation two factors had to be considered in the design of our new crossover operators: (i) we are working with variable-length chromosomes, and (ii) the type of problem we are tackling is a scheduling-type of problem where the preservation of good sub-sequences of transformations is a desirable property of any operators we design. The popular one and two-point crossover operators adapted to work with variable length chromosomes would be useful in preserving good sub-sequences of transformations. Such operators have been implemented in REVOLVER and are referred to as *VLX-1* (Variable-Length Xover, 1-point) and *VLX-2* (Variable-Length Xover, 2-point) respectively. General one and two point crossover operators are illustrated in Figs 3.4 and 3.5, VLX-1 is illustrated in Fig. 4.2.

A uniform crossover operator as described by Syswerda in [122] would have a disruptive effect on any potentially useful sub-sequences of transformations, yet some disruption is necessary for the search to make progress. Hence, a compromise was reached and a new operator created. Operator *VLX-3* protects developing sub-sequences of transformations by only applying uniform crossover to the final two-thirds (i.e. centre and right-hand two-thirds) of the two parent chromosomes. The first transformations to be applied (i.e the left-hand third of the chromosomes) are protected and not subjected to the uniform crossover operation. This operator is illustrated in Fig. 4.3.

```
======
BEFORE
======
                    |
 Parent 1 LSP LFU   | LIC LFU LFU LIC LFU
          3   4     |  8   7   8   8   9
                    |

 Random Crossover point = rand(1, strlen(Parent 1)) = 2


                                |
 Parent 2 LIC LSP LRV LIC       | LFU
          1   7   6   6         |  5
                                |

 Random Crossover point = rand(1, strlen(Parent 2)) = 4

=====
AFTER
=====

 Offspring 1      LSP LFU LFU
                   3   4   5

 Offspring 2      LIC LSP LRV LIC LIC LFU LFU LIC LFU
                   1   7   6   6   8   7   8   8   9
```

Figure 4.2: Crossover Operation on Variable Length Strings

Having applied our mutation and crossover operators, we then need to select which members are to constitute the new population. Holland [66] uses the roulette-wheel selection operator, however a number of theoretical concerns with the technique have arisen in recent years and which has led to increased popularity of the *tournament* selection technique. As such, tournament selection (with a default size of 2, unless otherwise specified) is the only selection operator currently used in REVOLVER with the 'gene-transformation' representation.

Most evolutionary algorithms can be made to work with this representation - some however, are more suitable than others. The current algorithms used in REVOLVER with this representation are random-mutation hill-climbing (RMHC), simulated annealing (SA), genetic algorithms (GAs), and self-adaptive genetic algorithms (SAGAs),

```
======
BEFORE
======
                              |
 Parent 1 LSP LFU LIC LFU | LFU LIC LFU
          3   4   8   7   |  8   8   9
                              |

 Random Crossover point
     = rand(strlen(Parent 1) DIV 3, strlen(Parent 1)) = 4

                              |
 Parent 2 LIC LSP LRV LIC | LFU
          1   7   6   6   |  5
                              |

 Random Crossover point
     = rand(strlen(Parent 2) DIV 3, strlen(Parent 2)) = 4

=====
AFTER
=====

 Offspring 1      LSP LFU LIC LFU LFU
                   3   4   8   7   5

 Offspring 2      LIC LSP LRV LIC LFU LIC LFU
                   1   7   6   6   8   8   9
```

Figure 4.3: VLX-3 Crossover Operation on Variable Length Strings with First Third of Both Strings Protected by Restricted Random Selection of Crossover Sites

(these algorithms are described in chapter 3). An idea of the implementation of this representation in REVOLVER can be seen in Fig. 4.5.

4.4.2 The 'Gene-Statement' Representation

In the gene-statement representation (see Fig. 4.6), as the name suggests one gene represents one statement in the program. Hence, a sequence of genes represents a sequence of statements. Each program statement has a type (e.g. a DO_STATEMENT, an IF_STATEMENT, an ASSIGNMENT_STATEMENT, etc). Transformations are mutation operators which are applied to statements. There is no 'meaningful' need for a crossover operator. Chromosome lengths may increase/decrease as mutations are

$$Gene = Transformation + Loop\ number$$

$$Chromosome = Sequence\ of\ Transformations$$

$$Mutation\ Operators = Change\ the\ sequence\ of\ Transformations$$

$$Crossover\ Operators = Recombine\ two\ sequences\ of\ Transformations$$

Figure 4.4: Summary of Gene-Transformation Representation

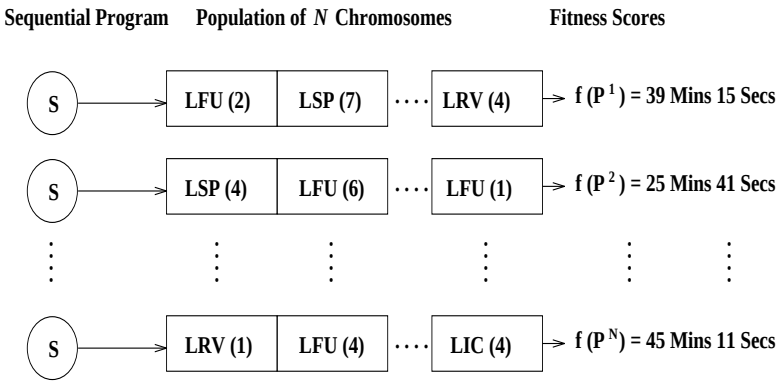


Figure 4.5: Application of a Population of Transformations using Gene-Transformation Representation

applied - application of a loop-fusion-mutation will reduce the number of statements in the program (i.e. the number of genes in the chromosome), application of a loop-splitting-mutation will increase the number of statements in the program (i.e. the number of genes in the chromosome).

Since transformations are now mutation-operators, each needs to be assigned a probability. (e.g.

```
pLRV = 0.02;
pLSP = 0.05;
pLFU = 0.15;
pLIC = 0.10;
.............
```

The mutation operators can now be applied probabilistically as they sweep across the genes of each chromosome in the population (Fig. 4.9). A meaningful order in

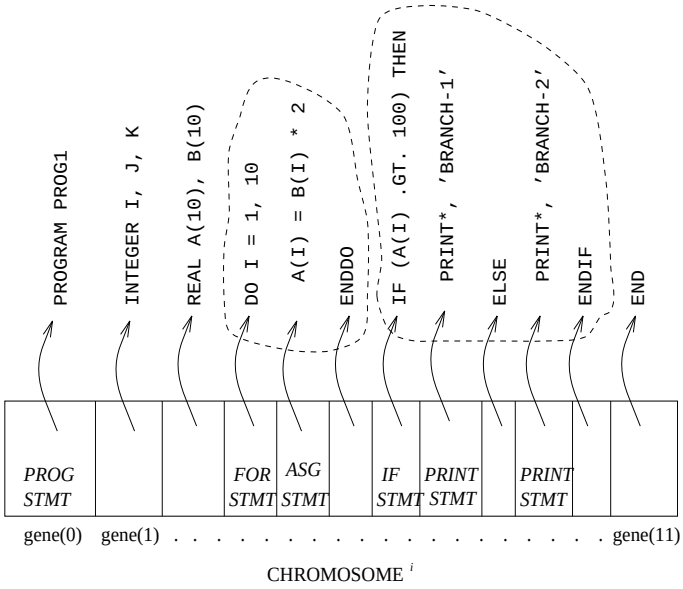


Figure 4.6: Gene-Statement Representation

which the operators may be applied then needs to be worked out. The diagram in Fig. 2.5 (page 45) which represents the optimising phases of a typical parallelizing compiler gives us a useful guide. (Essentially loops should initially be split/distributed so as to allow more potential for restructuring to take place, the final transformation to be applied should be fusion, so as to increase the granularity of slave processes).

It is noteworthy that the new evolutionary algorithm has the appearance of a population-based, iterative, probabilistic version of a standard parallelizing compiler

$$Gene = Statement$$

$$Chromosome = Program$$

$$A\ Mutation\ Operator = A\ type\ of\ Transformation$$

$$A\ Mutation\ Operation = An\ application\ of\ a\ Transformation$$

$$A\ Crossover\ Operator = Not\ used\ in\ this\ representation.$$

Figure 4.7: Summary of Gene-Statement Representation

```
1  begin
2      t := 0
3      initPopulation P ( t )
4      evaluate P ( t )
5      while ¬terminate do
6              t := t + 1
7              P' := selectParents( P )
8              LSPmutation( P' )
9              LRVmutation( P' )
10             LICmutation( P' )
11             ... (allow other mutations/transformations)
12             LFUmutation( P' )
13             evaluate( P' )
14             P := P'
15     end
16     printResults()
17 end
```

Figure 4.8: Gene-Statement Evolution Strategy

(i.e. it is essentially an extension of current parallelling compiler design). The new EA also naturally looks more like a traditional evolution strategy than a traditional GA.

4.4.3 EA Parameters and Fitness Function

It is a feature of all evolutionary algorithms that a number of parameters need to be set for each execution (such as population size, operator probabilities, number of generations, etc). It is a further feature of the gene-transformation representation which necessitates the setting of a further parameter (`decoder`) as described in section 4.4.1. The values for all parameter settings will be indicated where appropriate in the experimental results presented in chapter 5.

The use of a static performance estimation tool as a fitness function for our evolutionary approach is a natural choice since such tools (admittedly more advanced than that currently implemented in Revolver) are used in many existing state-of-the-art parallelizing compilers [19, 75, 8]. An important point to note however is that performance estimation is a software metric which is portable across many different architectures. Consequently, the use of such a metric as a fitness function holds out the possibility of architecture independent evolutionary parallelizing compilers.

```
1  for p := 1 to POPSIZE do
2      stmt := p- > firstStatement()
3      while (stmt) do
4              if (stmt- > type() == FOR - STMT)
5                 then
6                       x = rand()
7                       if (x < pLSP)
8                          then
9                                following = stmt- > followingStmt()
10                               success = ApplyLSP(stmt)
11                               if success
12                                  then
13                                       stmt = following
14                                   else
15                                       stmt = stmt- > followingStmt()
16                               fi
17                      fi
18             fi
19     end
20 end
```

Figure 4.9: Example Mutation-Transformation Algorithm

4.5 Automatic Parallelization using REVOLVER

4.5.1 Interactive, Batch or Fully Automatic Modes

REVOLVER may be used in one of three modes, interactive, batch or fully automatic (i.e. evolutionary) mode. These modes are described in the following sections.

4.5.2 Interactive Mode

Interactive mode currently consists of a simple scrolling screen which accepts user-input from a command-line prompt. The scrolling screen displays the current program listing - loops are annotated with unique identification numbers, i.e. loops are numbered 0, 1, 2, ... A transformation may be specified by entering its' three-letter acronym and a loop-number, as shown in Figures 4.11, 4.12.

The transformation is then applied - if it is legal, (the usual dependency checks are in operation). The loops are then renumbered, the screen is scrolled to display the restructured program and finally the '`Command >`' prompt is returned. Attributes may also be displayed. such as number of executions for each loop, a TF_RATIO for each

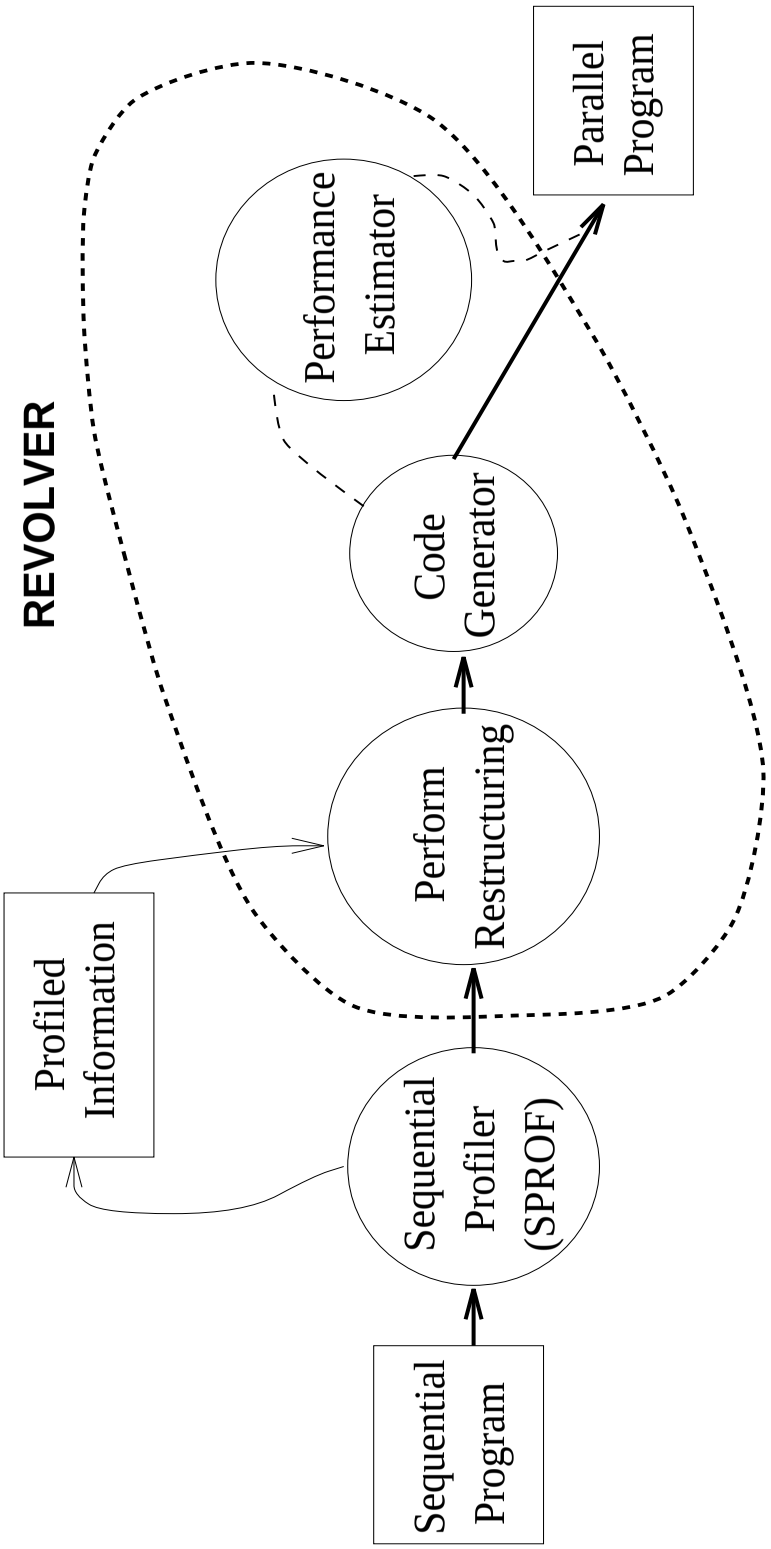


Figure 4.10: Phases of Automatic Parallelization in REVOLVER

conditional statement, as well as the current dependencies in the program. In this way, program files may be loaded, restructured, and saved. Currently an initial profiling run has to be performed so the more advanced REVOLVER facilities become available to the user, i.e. parallel code may be generated (which is written to disk rather than scrolled) and performance may be estimated, all from the command prompt.

The following series of figures (Figs. 4.11-4.19) demonstrate a typical interactive session with REVOLVER and illustrate some of its features. A brief commentary of each Figure now follows:

Fig.4.11 The initial program is loaded. The program is parsed, checked for errors and normalised (as described in 4.7). Performance data gathered during the initial profiling run is attached to each statement as an 'attribute'. This data is used to assist in retaining program correctness, estimating performance, and generating message-passing code. This example program is an Alternating Direction Implicit (ADI) method benchmark program as found in the Livermore Loop Kernels and also as distributed with the `Petit` dependency analysis libraries. This collection of loops represents the most computationally intensive section of the code. When using REVOLVER each loop is usually commented with an identification number but for this initial screenshot only I have disabled loop numbering so as to allow the whole program to be visible on the screen. Normally the user can simply scroll up to see any large program listings.

Our first action at the `Command>` prompt is to specify loop interchanging to loop 8 (`LIC 8`) in the program. The loop-interchange algorithm involves ensuring that the loop specified is the outermost loop of a perfect loop-nest (or else the transformation fails) and then identifying the depth of the loop nest. The loops in the nest are each assigned a number from 0...depth-1 (0 = outermost). The user is then prompted to enter a new permutation (we enter '`1 0`' to simply interchange the two loops). (In automatic mode a random permutation is currently

generated). Dependency analysis is then performed to see if the transformation can be legally applied.

Fig.4.12 A simple '`success`' message is returned to the user (offscreen) and the restructured program (with loop numbering enabled) is displayed. We now attempt to split loop number 9 (`LSP 9`).

Fig.4.13 This transformation succeeds and we now decide to split loop 8 (`LSP 8`), which contains two nested loops.

Fig.4.14 This also succeeds, so we now attempt to fuse loop 10 and the immediately following loop (at the same level) loop 12 together - they both iterate the same number of times, but dependency analysis reveals the transformation would be illegal and cannot be applied.

Fig.4.15 The program is scrolled again, and we decide now to look at the data dependency graph for the current program.

Fig.4.16 The data dependencies for each loop are displayed. We next decide to fuse loops 8 and 10 back together again.

Fig.4.17 The transformation succeeds and the listing is scrolled to display the new program. REVOLVER contains a number of environment variables (displayed with the '`ENV`' command) to view various features of the program (such as displaying the program attributed used in performance estimation, displaying data-dependencies, the symbol table, or also the expression table).

One environment variable `nprocs` specifies the number of processors in the target architecture for which message-passing code is to be generated. We use the '`SET`' command to change this value from the default value 4, to a new value 12. We now generate message passing code with the '`GEN`' command.

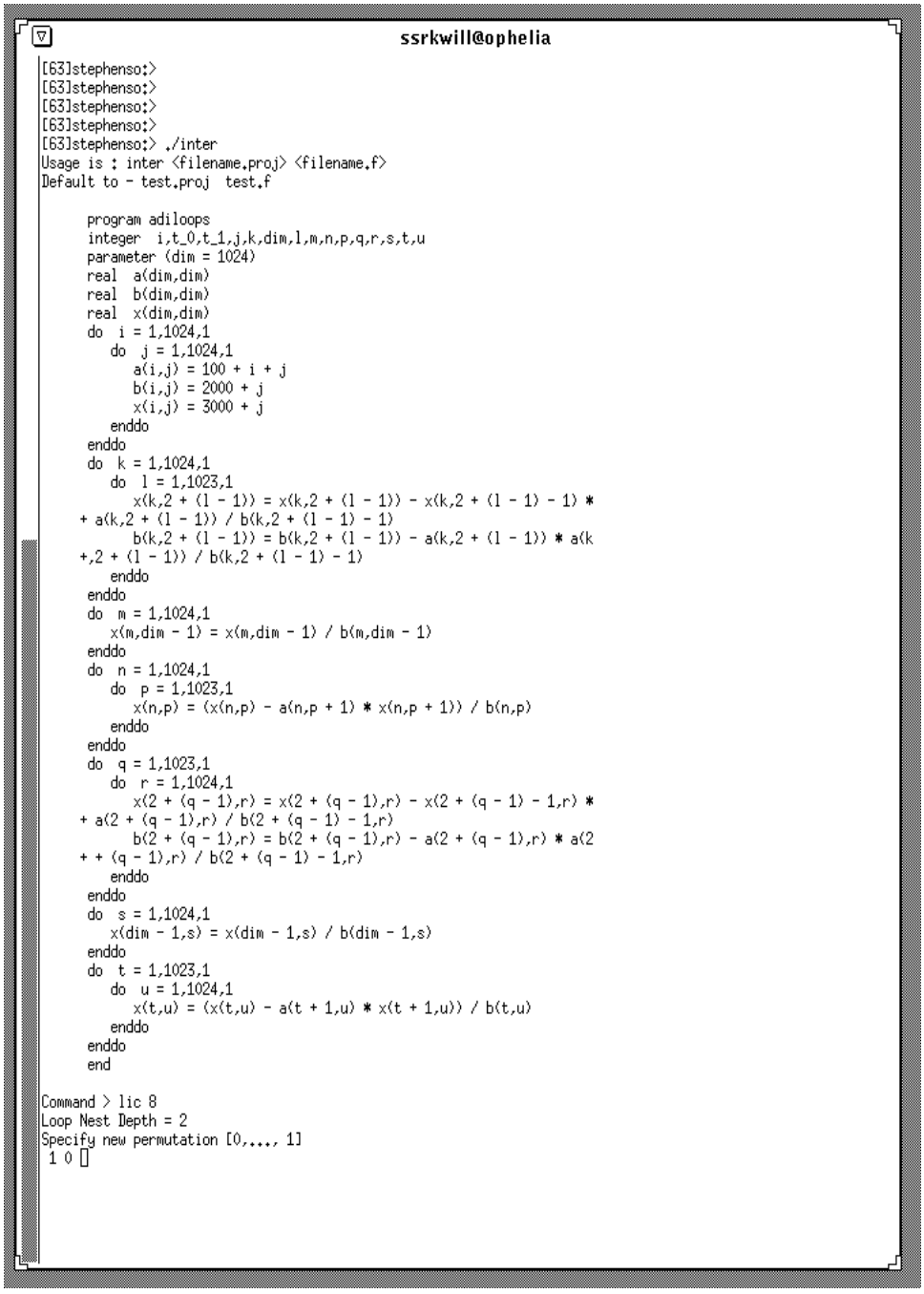


Figure 4.11: Restructuring code in REVOLVER in Interactive Mode 1

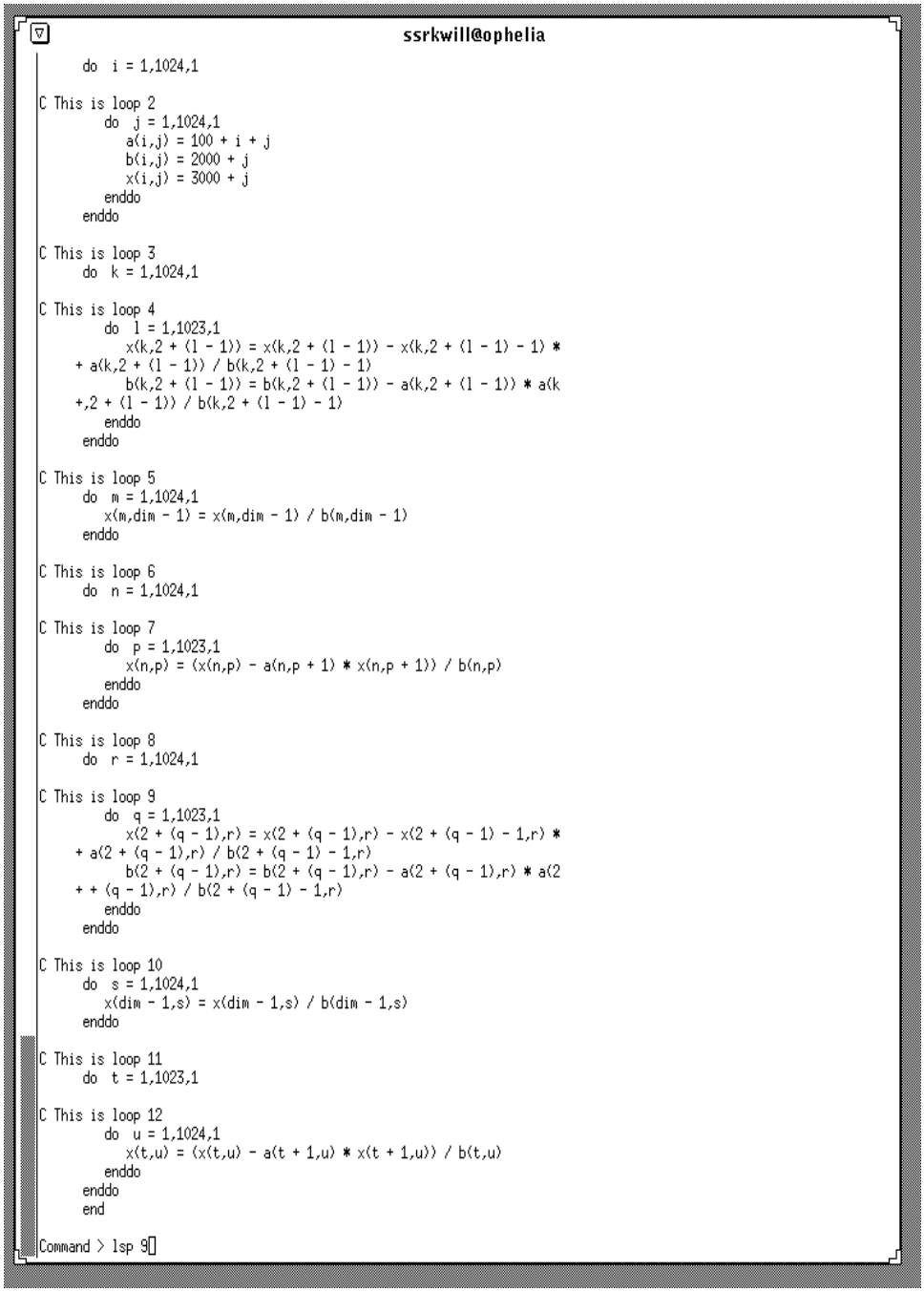


Figure 4.12: Restructuring code in REVOLVER in Interactive Mode 2

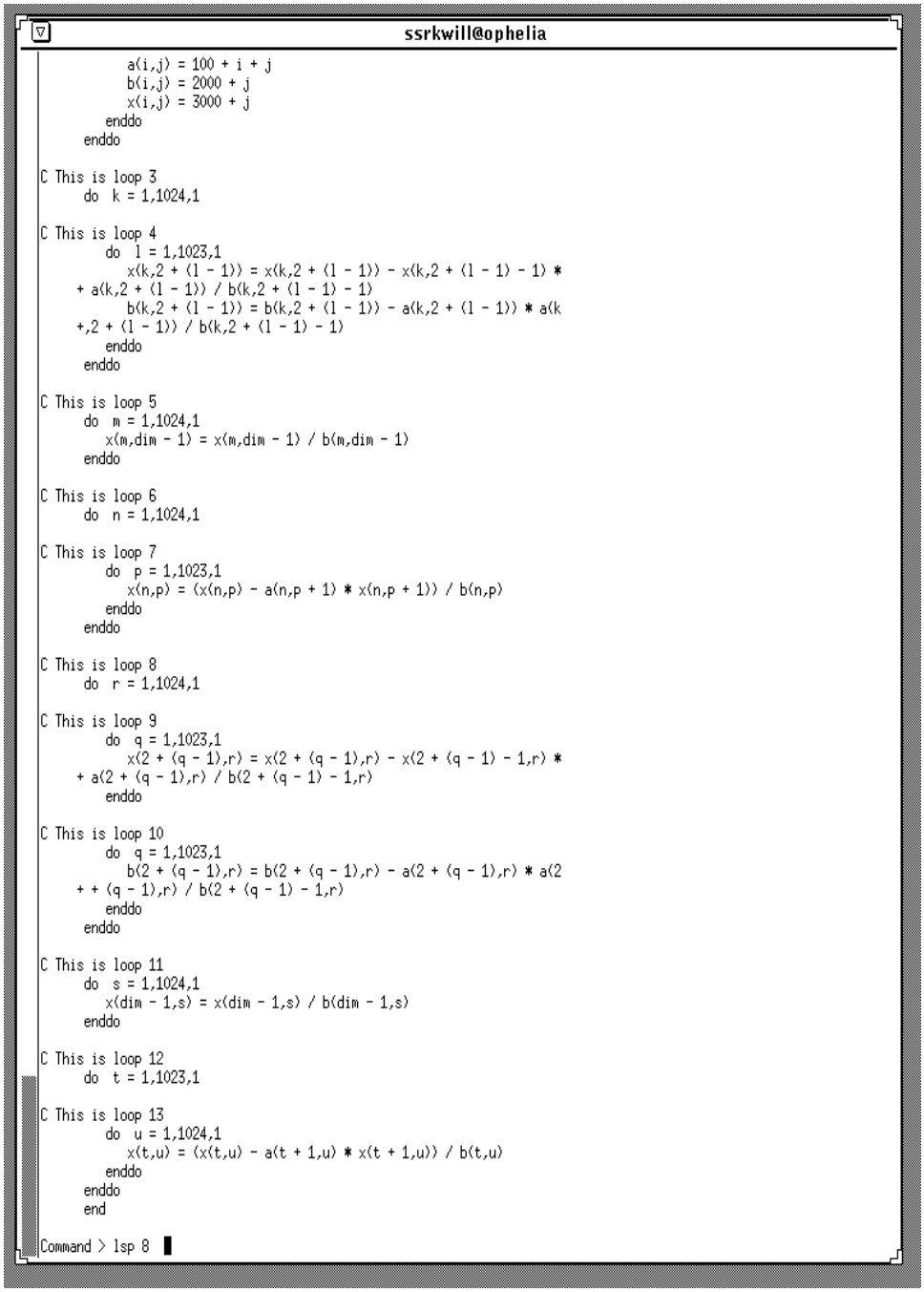


Figure 4.13: Restructuring code in REVOLVER in Interactive Mode 3

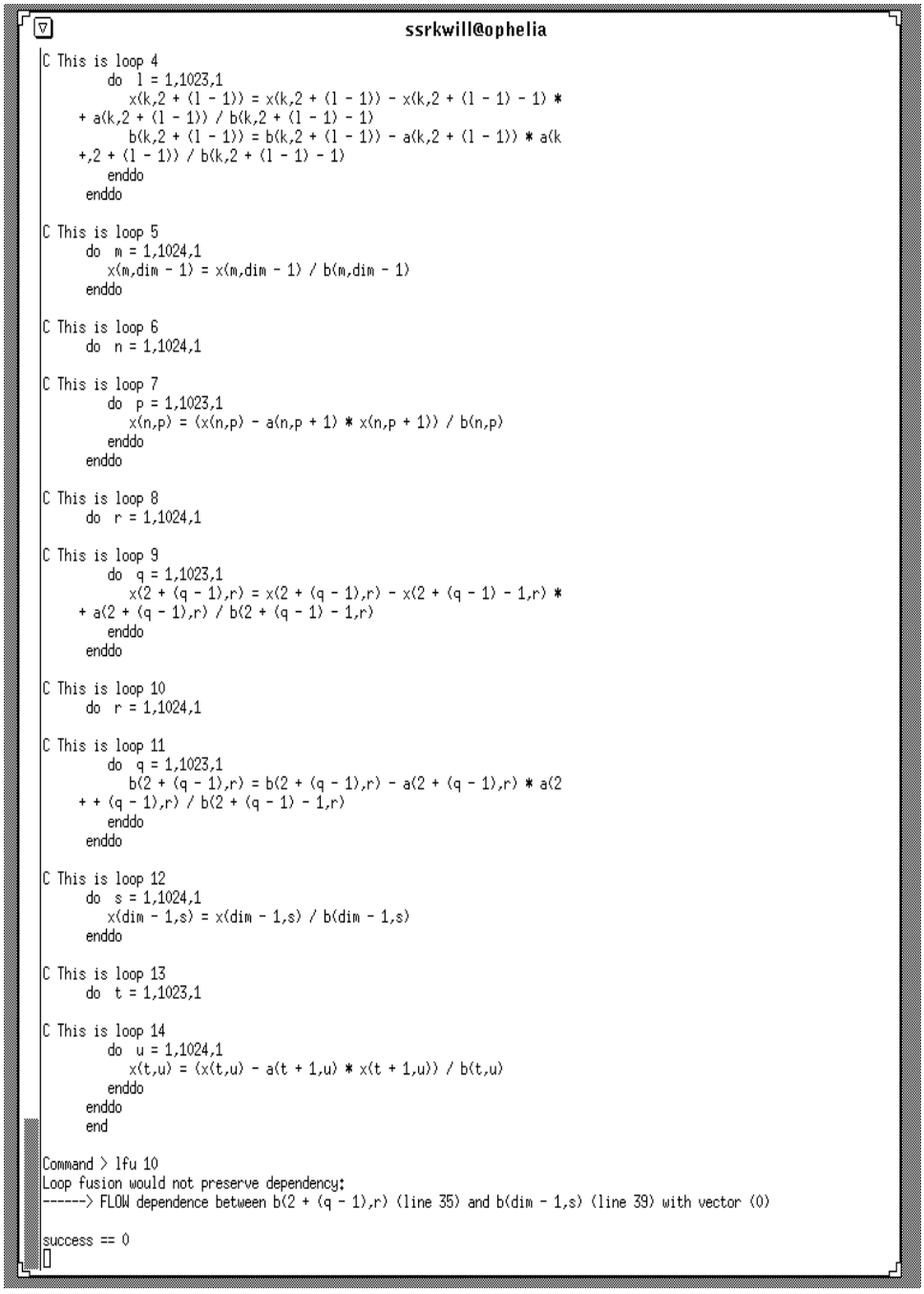


Figure 4.14: Restructuring code in Revolver in Interactive Mode 4

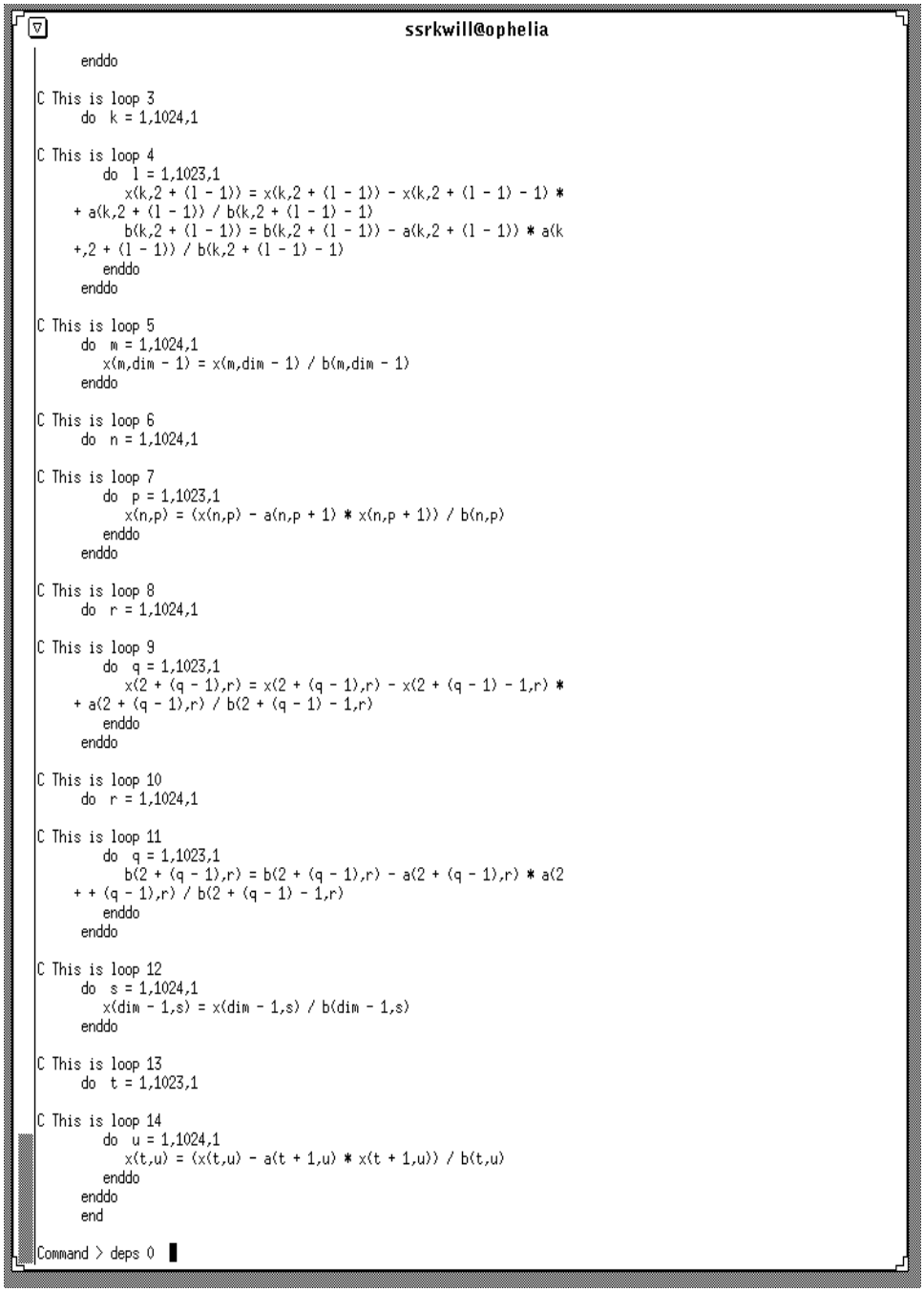


Figure 4.15: Restructuring code in REVOLVER in Interactive Mode 5

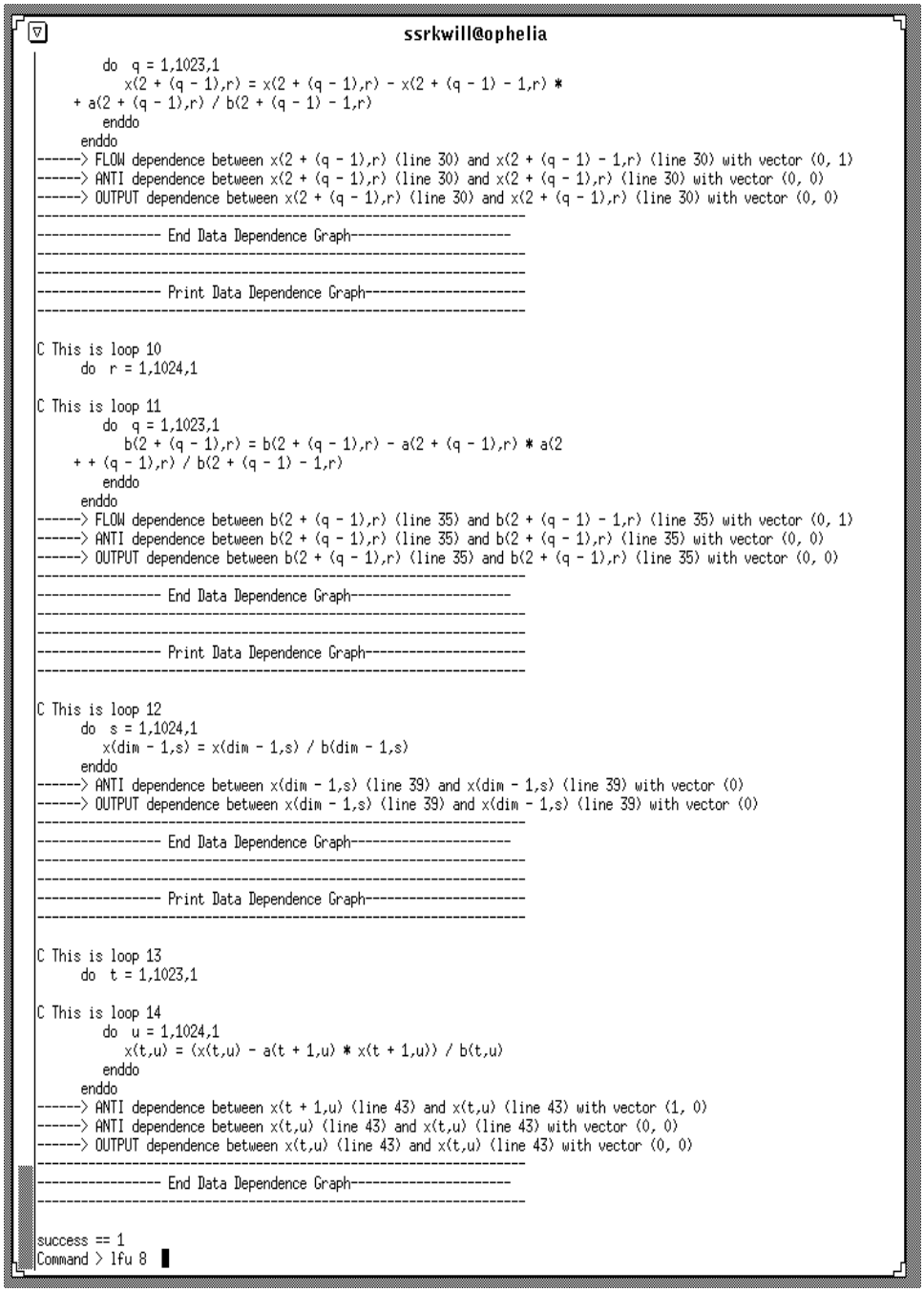


Figure 4.16: Restructuring code in REVOLVER in Interactive Mode 6

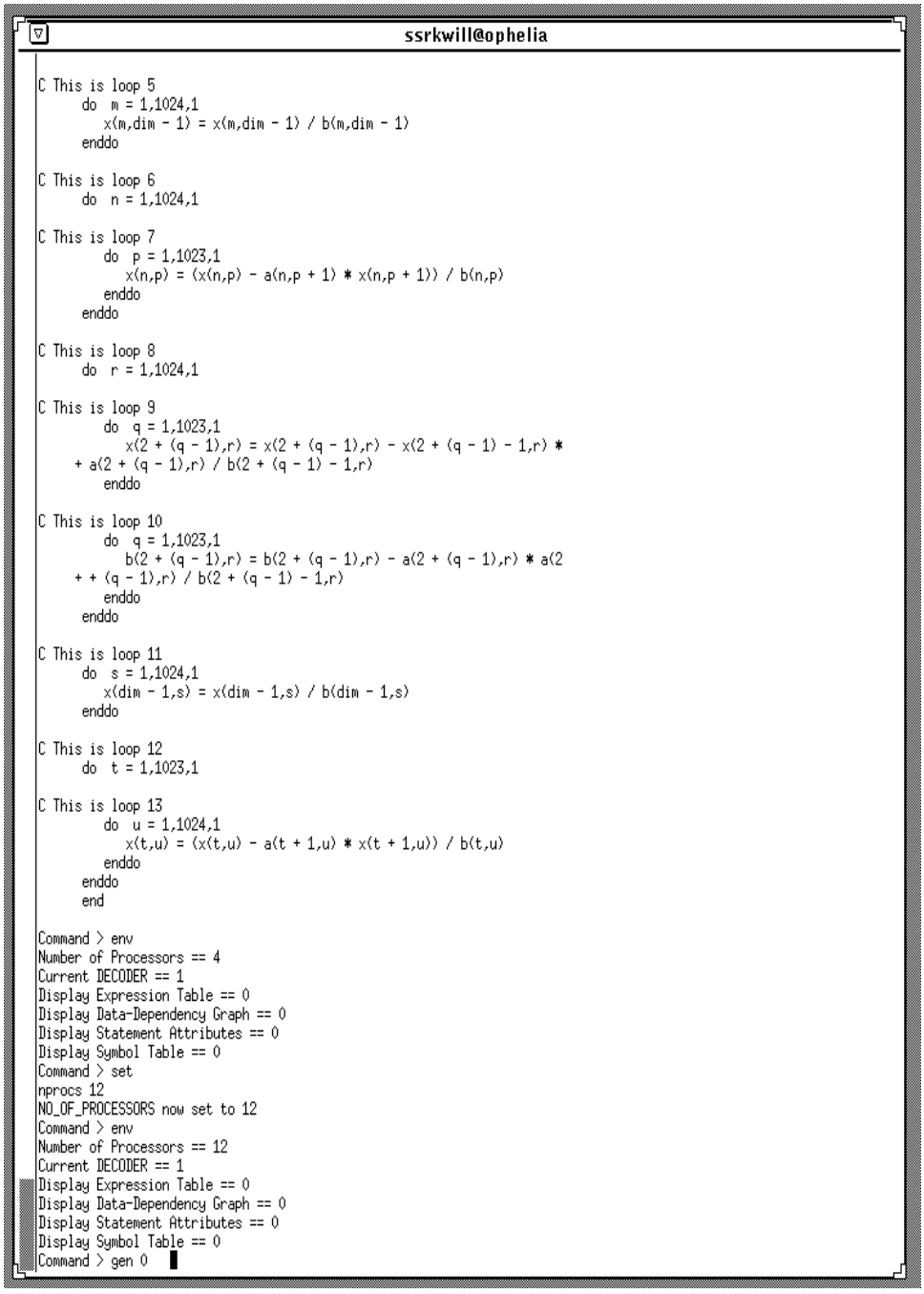


Figure 4.17: Restructuring code in REVOLVER in Interactive Mode 7

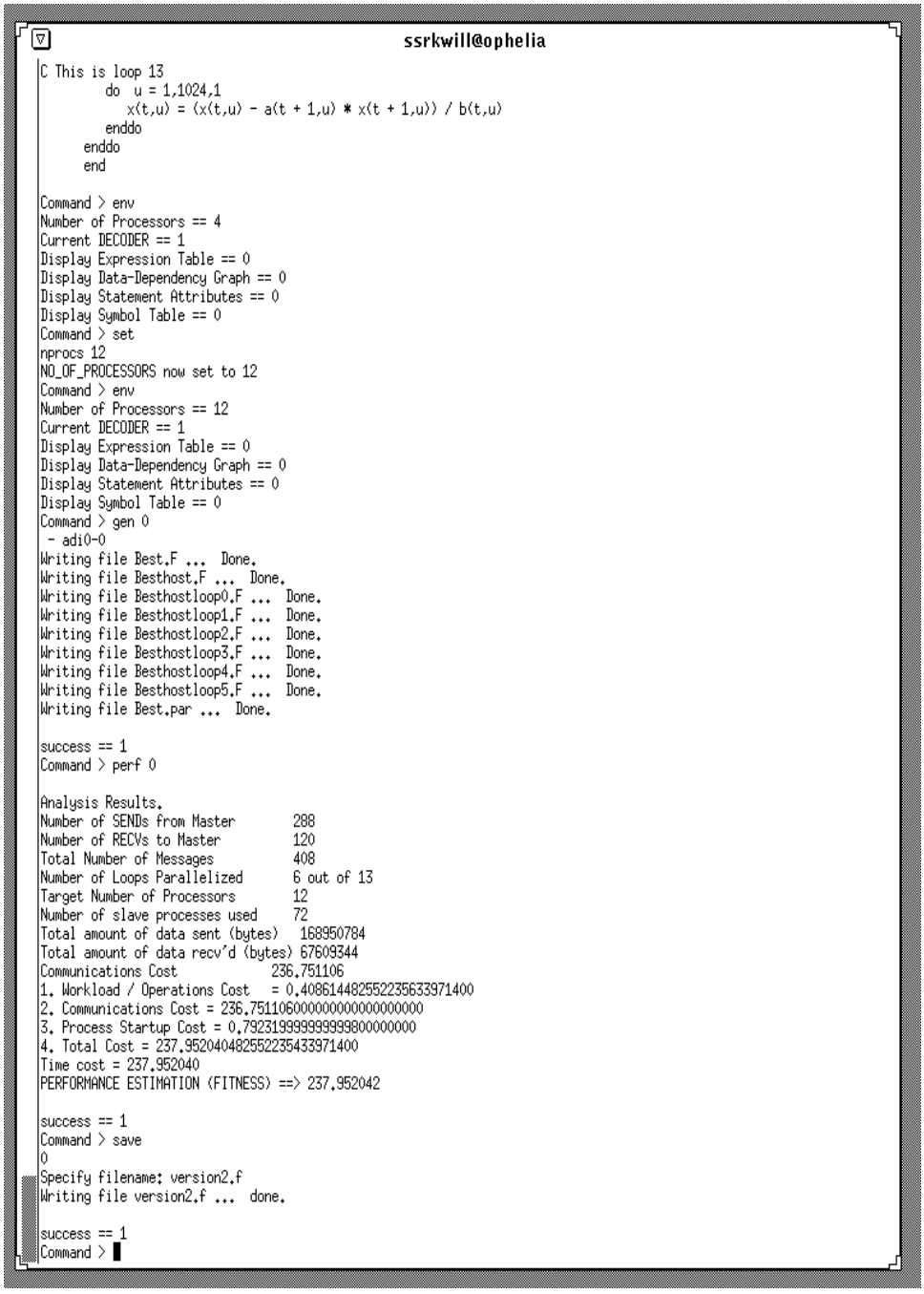


Figure 4.18: Restructuring code in REVOLVER in Interactive Mode 8

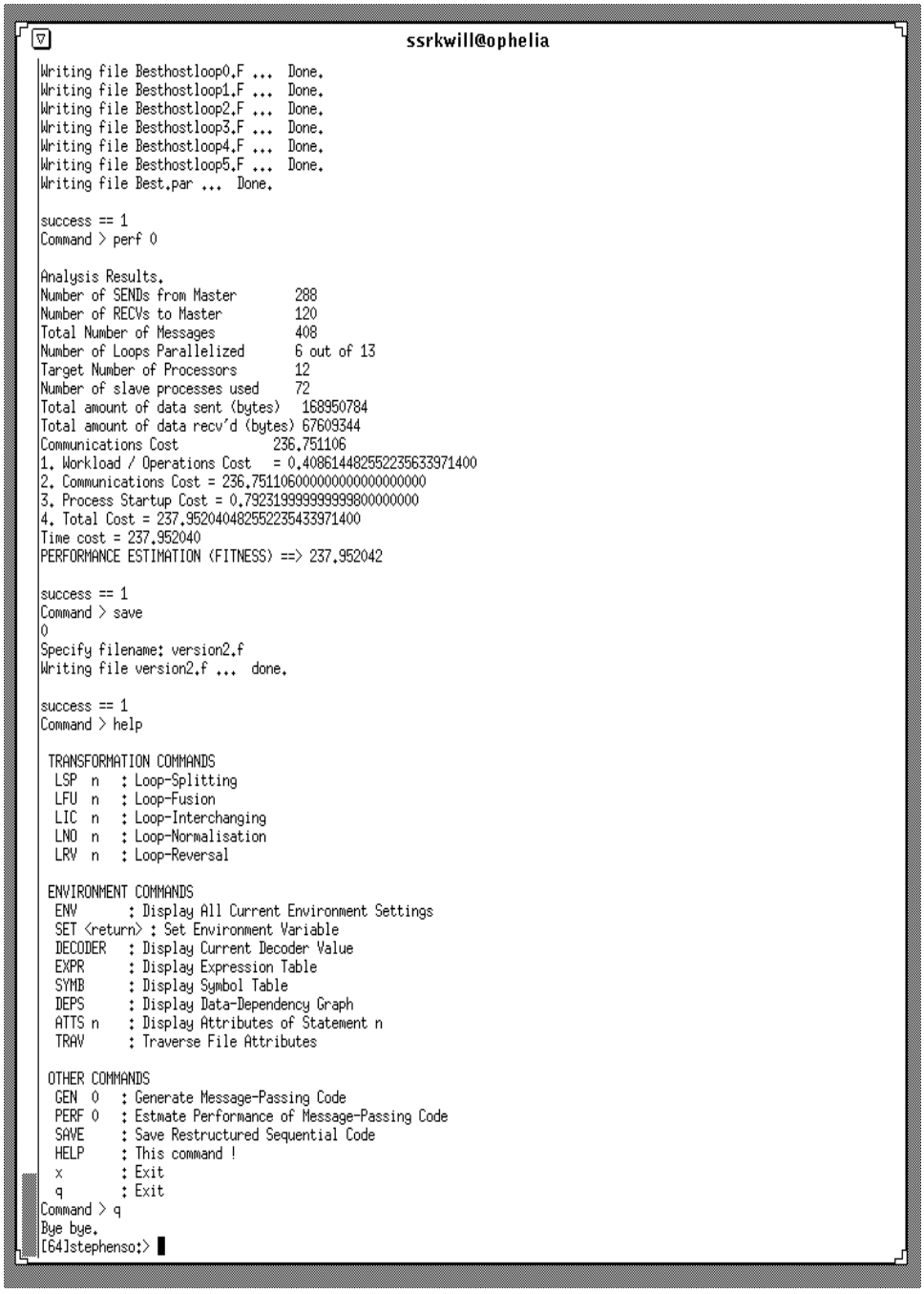


Figure 4.19: Restructuring code in REVOLVER in Interactive Mode 9

Fig.4.18 Message-passing code is generated and automatically written to disk (as indicated). Having generated parallel code we can now estimate the performance of the code on the Meiko CS-1. This is done with the '`PERF`' command. A set of analysis results are displayed including Workload costs, Communication costs, process start-up costs, and ultimately a total Time-Cost value, 237.952040 seconds. We decide to save the current restructured sequential code with the '`SAVE`' command. We are prompted to enter a file name (`version2.f`) and the file is saved.

Fig.4.19 A full set of most of the commands available in interactive-mode can be seen by entering '`HELP`'. We enter '`q`' to exit and terminate the session.

The user can now compile the message-passing code generated using the native CS-1 Fortran-77 compiler and execute the program in the usual way.

4.5.3 Batch Mode

REVOLVER can also accept restructuring commands from a simple ASCII text file and restructure a program without any direct user intervention. The commands entered in the example session could be stored in a file ('commands.bat', see Fig. 4.20) and used to restructure the program from the operating system shell prompt;

```
$ ./inter < commands.bat
```

Even the erroneous loop-fusion command ('`LFU 10`') would not cause REVOLVER to crash since the transformation simply returns a zero-fail and allows restructuring to continue. This is a consequence of the *decoding* strategy employed which specifies what REVOLVER should do when a transformation fails (section 31). The decoding strategy itself may be changed in both interactive or batch modes (in the same way `nprocs` is changed).

```
lic 8
1 0
lsp 9
lsp 8
lfu 10
deps 0
lfu 8
env
set
nprocs 12
env
gen 0
perf 0
save 0
version2.f
q
```

Figure 4.20: Example Text File of Input Commands (`commands.bat`) for Batch Mode Restructuring

4.5.4 Fully Automatic / Evolutionary Mode

The most powerful mode for using REVOLVER is the fully-automatic mode. This simply involves using an EA to perform restructuring and produce parallel code without any need for user intervention at all. It operates from the operating system shell prompt - just like a normal compiler - the only difference is that it defaults to an evolutionary mode of restructuring (current default is to use a GA/gene-transformation combination with a population size of 3, 3 mutation and 2 crossover operators, and set to run for 10 generations). However, all these parameters and more may be changed via a wide range of compiler switches which specify the initial values of the EA (such as choice of algorithm and population size, etc). Some of the switches currently available are shown in Fig. 4.21.

The starting length of chromosomes switch sets the length that all chromosomes will be at the start of processing. The lengths of the chromosomes will start to vary as crossover operations are performed (see Figs.4.2 and 4.1). The lengths of the chromosomes vary because we don't know how many transformations must be made in order to find the optimally parallelized code - it may be only one, (it may even be none) but it may also be a thousand. As such it was decided that the length of the chromosomes

```
-nprocs N    : Set number of processors.
-gens N      : Set max number of generations.
-pop N       : Set population size
-len N       : Set starting length of chromosomes.
-dec N       : Set decoder option (1, 2, or 3)
-alg s       : Set algorithm to use (s ::= saga1 | hc1 | sa1 | ga1 | etc ...)
-pVLX1 R     : Set probability of VLX-1 Crossover operator  0.0 <= R <= 1.0
-pMutGene R  : Set probability of Gene-Level Mutation operator
-pLSP R      : Set probability of LSP-Splitting Mutation operator
-tsize N     : Set size of Tournament Selection (default = 2)
-seed  N     : Set seed for random number generator.
```

Figure 4.21: Example REVOLVER Compiler Switches

must be allowed to vary at run time. Some upper bound on the length chromosomes can become may added as a feature of a future release.

An overview of the command-line execution of REVOLVER is presented in Fig. 4.22. To illustrate, an example REVOLVER execution plus command-line switches might be,

```
$ ./revolver -f test.f -nprocs 12 -alg ga1 -popsize 10 -gens 100
```

Given an example sequential f77 program (see Fig. 4.23) we must first profile the file with the sequential profiling tool (SPROF) command,

```
$  ./sprof test.f
```

This command produces two files, an instrumented version of the original program (see Fig.4.24) which will be compiled and executed and a data-file (see Fig.4.25) that consists of two columns of numbers, the first column is the line number of the program statement being referred to, (these correspond to the leader-statements of each basic block) the second column is the number of times that statement is executed (initialised to zero).

We next compile and execute the instrumented program to obtain basic-block execution figures. These are written to the profile data-file which, after execution, has been updated with the execution figures (see Fig. 4.25). This file will be read in at the start of the restructuring process. The figures will first be attached to each statement (so we know how many times each statement executed during the instrumented run) - and then, using this data, we are then able to compute how many times each conditional

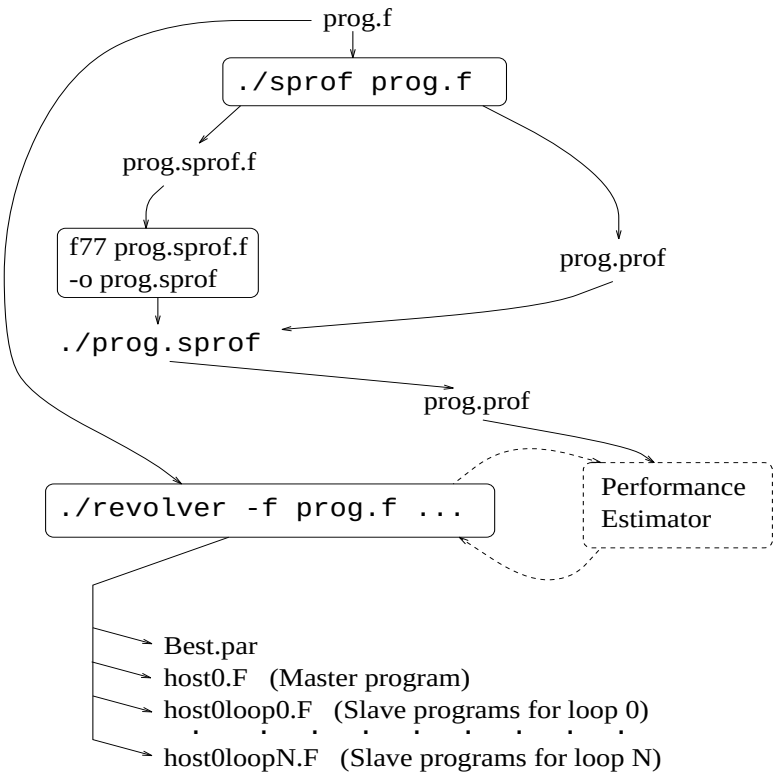


Figure 4.22: Schematic Diagram for Automated Parallelization with REVOLVER in Fully Automatic Mode

statement executed to $TRUE$ (this figure is called the TF_Ratio, see section 2.2). (Although there are no conditional statements in this example). All this information is attached to appropriate statements in the form of attributes (see Fig. 4.26) which are used to predict the performance of a restructured version of the program (namely, the fitness-function of the GA).

4.6 Profiling

The current sequential profiling tool (called `sprof`), incorporated into the REVOLVER system, utilises many of the profiling techniques described in section 2.7.1. It has also been programmed with the capability of timing sections of program code during the profiling run of the program. However, this timing information is not currently used by REVOLVER in the restructuring process, (as explained in 2.7.1).

```
      C simple test program for temperature conversions.

      C Uses 1-Dimensional arrays.

      C123456789

 1          PROGRAM weather1D
            INTEGER i, n
            PARAMETER (n = 10000)
            character *16 infile
            character *16 outfile
            integer a(n), b(n)

      C Open input file.

 2          infile = 'test1.dat'
 3          open (6, file=infile, status='UNKNOWN')

      C Initial file read loop

 4          do i = 1, n
 5            read(6, 199) a(i)
 6          enddo

      C Main temperature conversion loop.

 7          do i = 1, n
 8            b(i) = a(i) / 5 * 9 + 32
 9          enddo

      C Final file output loop.

10          outfile = 'test1.results'
11          open (7, file=outfile, status='UNKNOWN')
12          do i = 1, n
13            write(7, 199) b(i)
14          enddo

15    199   format (I)
16          end
```

Figure 4.23: Example Simple Fortran-77 Program (`prog.f`)

```
 1   C simple test program for temperature conversions.
 2   C Uses 1-Dimensional arrays.
 3   C123456789
 4         program weather1d
 5         character  *16 outfile
 6         character  *16 infile
 7         integer  value
 8         integer  lineNo
 9         integer  bb(7)
10         common bb
11         integer  i,j,n,p,q
12         parameter (n = 10000)
13         integer  a(n),b(n)
14         bb(1) = bb(1) + 1
15
16   C Initial file read loop
17         do  i = 1,n
18            bb(2) = bb(2) + 1
19            a(i) = i
20         enddo
21         bb(3) = bb(3) + 1
22
23   C Temperature conversion loop.
24         do  i = 1,n
25            bb(4) = bb(4) + 1
26            b(i) = a(i) / 5 * 9 + 32
27         enddo
28         bb(5) = bb(5) + 1
29
30   C Final file output loop.
31         do  i = 1,n
32            bb(6) = bb(6) + 1
33            print *,b(i)
34         enddo
35         bb(7) = bb(7) + 1
36         infile = 'test.b'
37         open (8,file=infile,status='UNKNOWN')
38         outfile = 'test.prof'
39         open (9,file=outfile,status='UNKNOWN')
40         write (9,200)7,0
41         do  i = 1,7
42            read (8,200)lineNo,value
43            write (9,200)lineNo,bb(i) + value
44         enddo
45   200      format (I5, I10)
46         end
```

Figure 4.24: Instrumented Simple Fortran-77 Program `prog.sprof.f`

```
                                  7       0
 4  0                             4       1
 5  0                             5   10000
 7  0                             7       1
 8  0                             8   10000
10  0                            10       1
13  0                            13   10000
15  0                            15       1
```

(Before) (After)

Figure 4.25: Profile Data-File Before (`prog.b`) and After (`prog.prof`) Execution of Instrumented Simple Fortran-77 Program. (Left column are line numbers for basic-block leader statements, right column is number of executions - first row of *After* says there are 7 basic blocks (zero ignored).

4.7 Program Normalisation

Both the profiling tool `sprof` and REVOLVER initially call a program normalisation function before starting their processing. This normalisation involves standardising the input program into a form which is easily manipulated - e.g. converting 'do...continue' loops into 'do..enddo', checking for the presence of any 'goto' statements, converting any `LOGICAL_IF` statements to 'if...endif' form, checking for any invalid input such as (`ARITHMETIC_IF` statements or skewed loops), numbering all executable statements, etc.

4.8 Dependency Analysis

The dependency analysis routines used in REVOLVER are the `Omega` libraries of Pugh [113, 114]. These libraries are able to analyse loop nests and construct a data dependency graph for array and scalar accesses within the nest. From this graph, they are able to determine if data dependencies exist across iterations of any loop in the nest. If dependencies exist, they are able to identify which references are the source and sink of the dependency and also, where possible, the type, direction and distance of each dependency. Dependency information has to be updated with library calls after each successful application of a transformation.

```
                                                        A T T R I B U T E S
                                                       --------------------
                                                         Number of     T/F
                                                         Executions   Ratio
     C simple test program for temperature conversions.

     C Uses 1-Dimensional arrays.

     C123456789

 1         PROGRAM weather1D                                 1          -
           INTEGER i, n                                      1          -
           PARAMETER (n = 10000)                             1          -
           character *16 infile                              1          -
           character *16 outfile                             1          -
           integer a(n), b(n)                                1          -

     C Open input file.

 2         infile = 'test1.dat'                              1          -
 3         open (6, file=infile, status='UNKNOWN')           1          -

     C Initial file read loop

 4         do i = 1, n                                       1          -
 5           read(6, 199) a(i)                           10000          -
 6         enddo                                         10000          -

     C Main temperature conversion loop.

 7         do i = 1, n                                       1          -
 8           b(i) = a(i) / 5 * 9 + 32                    10000          -
 9         enddo                                         10000          -

     C Final file output loop.

10         outfile = 'test1.results'                         1          -
11         open (7, file=outfile, status='UNKNOWN')          1          -
12         do i = 1, n                                       1          -
13           write(7, 199) b(i)                          10000          -
14         enddo                                         10000          -

15   199   format (I)                                        -          -
16         end                                               1          -
```

Figure 4.26: Attributed Example Simple Fortran-77 Program

4.9 Transformations and Optimisations

Currently the REVOLVER transformation catalogue consists of five transformations (all loop transformations), each with its own unique acronym, they are:

$$CAT = \text{LOOP\_FUSION (LFU), LOOP\_SPLITTING (LSP),}$$
$$\text{LOOP\_INTERCHANGE (LIC), LOOP\_REVERSAL (LRV),}$$
$$\text{LOOP\_NORMALISATION(LNO)}$$

Loop-normalisation is usually only applied to each loop simply to make it more amenable to dependency analysis. A full description of these loop transformations is presented in section 2.4.2. In REVOLVER , each transformation currently only takes one parameter - the reference number of the loop it is to be applied to. So a gene encoded `LRV 3`, would mean "apply the loop-reversal transformation to loop # 3 in the program". In loop-splitting all statements in the body of the loop are split into atomic SCC regions (see Fig. 2.8). The loop-interchange transformation is currently only applicable to perfectly-nested loops, hence all the loops in the perfect nesting with the loop referred to by the loop number are subject to interchanging in accordance with some cyclically randomly generated permutation of the loops. The loop-fusion transformation is implemented so that two loops can only be fused if (i) they execute the same number of iterations, and (ii) there are no intervening statements between the two loops. Hence if the loop referred to is not immediately followed by, or preceded by (*in that order*) another loop, then loop-fusion cannot occur, otherwise loop-fusion may occur.

Note that the encoding of a gene with a transformation is only a 'request' to apply the transformation *if legally possible*. There is no guarantee that any transformation can be applied until it is attempted.

No further transformations or optimisations (such as dataflow-optimisations) are currently implemented in REVOLVER .

4.9.1 Array Region Analysis / Triplet Notation

As part of analyzing the accessed regions of an array we wish to represent the array access pattern in a form that is flexible, accurate and easily manipulated. The way we achieve this is to use the *triplet notation*. A triplet can be used to describe an affine function access pattern for a single dimension of an array. Multi-dimensioned array regions are represented by a triplet for each dimension. Non-affine array access patterns are not considered in this thesis.

A triplet summarises the array accesses for a single dimension in the generic form:

$$[\ lb : ub : stride\ ]$$

where *lb* and *ub* represent the lower and upper bounds of the array region respectively, and *stride* represents the step-size of accesses between the two bounds. Typical array access patterns are summarised by triplets in the examples in Fig. 4.27.

```
                                        Array Access Triplets
DO i = 1, 100
   ... A(i, i+1) ...                    A ([1:100:1], [2:101:1])
ENDDO

DO j = 3, N, 5
   ... B(j, 7) ...                      B ([3:N:5], [7:7:1])

   DO k = 100, 1, -1
     DO l = 2, M, 2
        ... C(j, k*2, l) ...            C ([3:N:5], [200:2:-2], [2:M:2])
        ... D(2*l+1) ...                D ([5:2*M+1:4])
     ENDDO
   ENDDO
ENDDO
```

Figure 4.27: Example Array Access Triplets

Where possible, it is desirable to replace symbolic entries in a triplet with constant values (these can often be determined from the bounds of the index variable). This makes later array region analysis exact and much easier. In the REVOLVER system, since we impose the condition that the values of all variables used in loop expressions can be determined at compile-time, this will always be possible.

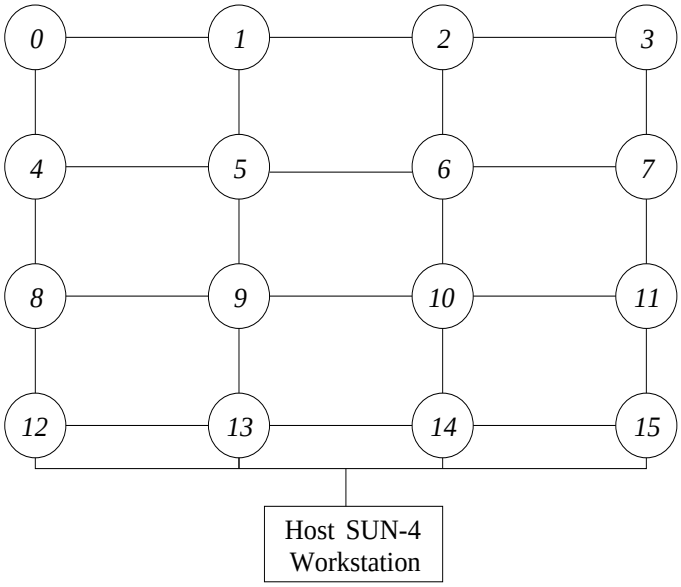


Figure 4.28: Meiko CS-1 Processor Interconnection Architecture

4.10 Code-Generation for Message-Passing CS-1

4.10.1 Introduction

In any machine translation process, the quality of the code generation has an enormous impact of the performance of the output code. This is particularly true of a code-generator for a message passing architecture since communication is the greatest overhead of any parallel computation and where so many communications optimisations are possible which may produce significant increases in program performance.

4.10.2 Meiko Computing Surface-1 Architecture

The target machine of the REVOLVER system in this research, is a Computing Surface 1 (CS-1) from Meiko Computers Ltd. This is a distributed memory, MIMD multiprocessor, with 12 processing elements connected in a simple two-dimensional mesh topology by a 32-bit bus (as detailed in [94]). The surface is accessed via a host terminal, in this case a SUN-4 workstation running the SunOS 4.1.3 operating system (see Fig. 4.28).

Each processing element is an Inmos T800 transputer with an operating speed of 20 MHz, 4 Kb of on-chip memory, a 64-bit floating point unit (FPU), and a 32-bit central

processing unit (see Fig. 4.29). Each element also has 4 bi-directional communications links which are used to send/receive data to/from other processing elements. The T800 can directly address a memory space upto 4 Gbytes in size. Memory above the on-chip 4 Kb is accessed via the external memory interface. The on-chip memory is used as each processor's local memory. Any communication between local memory devices on different processors will be handled by the communications links under the supervision of the central processor. As such, the parallel execution of processes running on the processors will be affected. The CS-1 is of interest to us since it represents a typical example of an MIMD, distributed memory architecture.

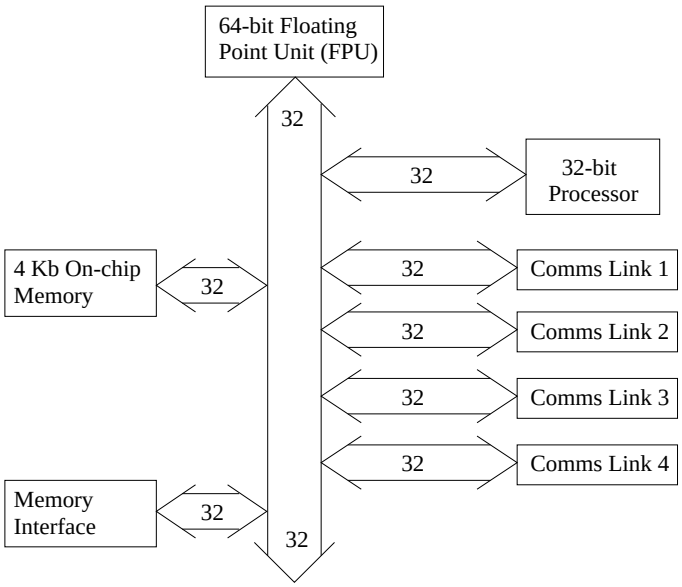


Figure 4.29: Inmos IMS T800 Microprocessor Architecture

4.10.3 Software Environment of CS-1

The CS-1 provides a software environment (called *CSTools* [95]) allowing sequential and message-passing parallel programming of the CS-1 in Meiko-C (`mcc`) and Meiko Fortran-77 (`mf77`). Individual processes may be defined statically or else spawned dynamically. Communication between any two processes may be via blocking (i.e. each process suspends its execution until the communication has been successfully completed), or non-blocking (i.e. processors suspend their execution only to place/retrieve

data onto/from the interconnection network) modes. Processes may be placed onto particular processors by specifying a parameter value (if spawned dynamically) or through specification within a '.par' file, (if to be created statically).

The current implementation of the REVOLVER system automatically parallelizes sequential Fortran-77 programs into a static Meiko Fortran-77 parallel program, executing a restricted version of Master/Slave parallelism, with individual processes placed (as specified by the creation of a '.par' file for each parallel program created), using only synchronous, blocking, inter-process communications.

Using this model, both processes must synchronise before data may be transferred between them, with the result that one of the processes may waste time waiting for the other to become ready. Synchronous-blocking communication is however, the most reliable way to avoid a failed data transfer and as such, is the message passing mechanism used in code generated by the REVOLVER system.

Further consideration to dynamic spawning of processes and to non-blocking communication mechanisms is not given in this discussion.

4.10.4 Linearization of Arrays

CSTools provides two primitives for synchronous-blocking communication of messages. To send data,

```
call csntx(transport, 0, slaveid, arrayVar(1), messageSize)
```

and to receive data,

```
call csnrx(transport, null, arrayVar(1), messageSize)
```

Both of these functions work by direct memory-block transfer where a contiguous block of memory of size `messageSize` starting at address `arrayVar(1)` is transferred. The transfer of contiguous blocks of memory is unfortunate since Fortran stores array data in column-major format. Furthermore, no 'stride' parameter exists for the `csntx/csnrx` function calls. Hence, the only generic way to send/receive N-dimensional array data is by the explicit declaration of buffer arrays and loops to pack (before send-

ing) and unpack (after receiving) data. This imposes a considerable overhead on already costly communications.

For example, to transfer a 1-d array region specified by the triplet [1:1000:2], if a stride parameter was available, would require only a simple function call to send

```
call csntx(transport, 0, slaveid, arrayVar(1), 500, 2)
```

and to receive data,

```
call csnrx(transport, null, arrayVar(1), 500, 2)
```

However, since a block-memory data transfer method is being used, combined with the facts that Fortran stores data in column-major order, and that there is no stride parameter in the CS1 send/receive functions, we must generate instead the following pack and send code (send from master to 4 slaves, assumes arrayVar is a 1-d array of 16-bit integers),

```
C   Linearize array accesses and send data.
      do  i = 1,4,1
         count_1 = 1
         lb = 125 * (i - 1) + 1
         ub = 125 - 1 + lb
         if (lb. ge. 1 .and. ub .le. 500) then
             do  t_0 = lb,ub,1
               BufferArray1(count_1) = arrayVar(t_0)
               count_1 = count_1 + 1
            enddo
          endif
         call csntx(transport,0,slaveid(i),BufferArray1(1),500)
      enddo
```

and the corresponding receive and unpack code,

```
C   Receive and unpack array data.
```

```
call csnrx(transport,masterid,BufferArray1(1),500)
count_1 = 1
lb = 125 * procNo + 1
ub = 125 - 1 + lb
if (lb. ge. 1 .and. ub .le. 500) then
    do  t_0 = lb,ub,1
      arrayVar(t_0) = BufferArray1(count_1)
      count_1 = count_1 + 1
   enddo
 endif
```

for each array reference in the loops being parallelized.

Code to compute `lb` and `ub` is also necessary to pack/unpack array segments such that only segments of the correct size are sent to the slaves in accordance with the partitioning strategy which divides iterations (and hence array regions) across the processors available. The conditional code is necessary in cases where the partition size does not evenly divide the number of iterations.

It is clear therefore that communications costs (already the most expensive component of parallel computation) are now almost prohibitively expensive. It should also be noted that this has to be reflected in our performance estimator.

4.10.5 Horizontal Parallelism

The code generated by REVOLVER is a form of parallelism called Horizontal parallelism (Fig. 4.30). It is a form of master/slave parallelism in which the master process suspends itself while all cooperating slaves all spawn, work and terminate together (asynchronously), afterwhich the master resumes processing - and that this may happen several times throughout the program execution.

The reasons for this choice of parallelism were (i) a master process is needed to distribute the data to the slaves (which is more memory efficient on the CS-1), and (ii) it is a model of parallelism where explicit message-passing combined with regular computations and communication patterns should make performance estimation easier

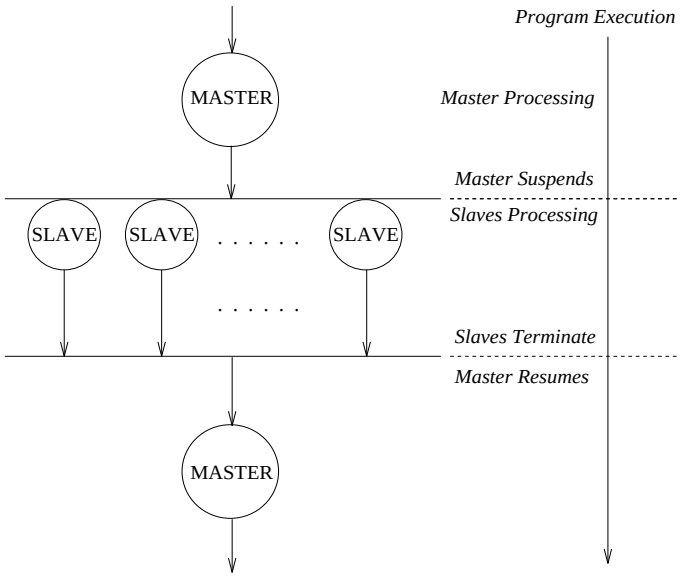


Figure 4.30: Horizontal Parallelism

and more accurate.

Slaves do not spawn their own sub-tasks since this would create *tree-parallelism* which is harder to estimate the performance of programs written in this style. This is why we cannot parallelize loops which are nested inside other loops which are being parallelized.

Network contention is not taken into account in our performance estimator. Since all communications are sent via the master process, the network will only ever be carrying at most, one message. In other words, we have restricted parallelism for the sake of eliminating network contention completely - thereby making estimation of program performance easier.

Although accurate performance prediction is an essential part of the evolutionary parallelization process it does not form a major part of this research. The performance predictor currently implemented in REVOLVER is considered rudimentary but sufficient for our investigations.

```
par
   processor 0 master
   processor 1 for 4 slave1
   processor 1 for 4 slave2
endpar
```

Figure 4.31: Example `par file` of A Master with 2 * 4 Slaves

Slave	Iterations	Sum
0	1-80	80
1	81-160	160
2	161-240	240
3	241-320	320
4	321-400	400

(5 Slaves)

Slave	Iterations	Sum
0	1-67	67
1	68-134	134
2	135-201	201
3	202-268	268
4	269-335	335
5	334-400	402

(6 Slaves)

Figure 4.32: Regular and Irregular Partitioning of 400 Iterations

4.10.6 Process Scheduling and Load-Balancing

Processes are statically mapped onto processors by CS-1 automatically. After compilation a 'par' file is produced (Fig. 4.31) which contains details onto which processor each process will be placed.

To make performance estimation easier, REVOLVER tries to produce code which has regular communication patterns. This code follows a Master/Slave model of parallelism, where the Master process is always placed on processor number 0, and sends data to and receives data from a fixed number N of slaves. The workload of each parallelized loop is always spread evenly across the N slaves using the algorithm in Fig. 4.33. For example, if a loop being parallelized iterates 400 times and its workload is to be spread across 5 slave processes then each slave will execute 80 iterations (see Fig. 4.32).

If the workload does not partition evenly among the N slaves, then parallel redundant computation is introduced into the $N-1$th slave to make the processing even. So if a loop being parallelized iterates 400 times and its workload is to be spread across 6 slave processes then each slave will execute 67 iterations.

```
int partition_iterations( int NIterations, int NSlaves )
{
  int PartitionSize = (int) NIterations / NSlaves;
  if ((NIterations MOD NSlaves) > 0) PartitionSize += 1;
  return PartitionSize;
}
```

Figure 4.33: Algorithm to evenly partition loop iterations across slave processes

4.10.7 Code Generation Algorithm

Generating parallel loop code for a program in REVOLVER involves creating one message-passing master program and N corresponding message-passing slave programs (where N is the number of loops being parallelized). If P is the number of processors available, then P instances of each slave program will be instantiated during the program run. Since we want to make full use of all the processors available, we always spawn $NSlaves$ number of slave processes with each parallel phase of the program, where $NSlaves$ is equal to the value of P.

To help illustrate the following algorithm, the basic data-structure of the population is shown below.

```
typedef struct {
  FILE *seq_file;     /* Pointer to copy (restructured) of sequential code */
  FILE *master;       /* Pointer to Master Process Code */
  FILE_REC *slaves;   /* Pointer to linked list of Slave Process Codes */
} POP_REC;


POP_REC population [POP_SIZE];
```

After restructuring, the sequential code is analysed to determine which loops can be parallelized. To mark the loops for parallelization, the induction variable of each loop is changed to "`i_N`" where 'N' is the number of the loop, all parallelizable loops are numbered from 0, 1, ... onwards. Loops are always parallelized if:

1. they carry no dependencies across iterations

2. they are not nested within another loops being parallelized (this would lead to tree-parallelism)

3. they are not nested within another loop (i.e they cannot be instantiated more than once since this is incompatible with the static CS-1 model)

Any loop that meets these three criteria is parallelized, regardless of the amount of work it does. The commented algorithm is described in Figs 4.34 to 4.38.

```
1  void funct parallelize (POP_REC*p, int NSlaves)
2        var FILE*master_file,* slave_file,* seq_file;
3          var int par_loops,* loop_ids;
4            seq_file := p- > seq_file;
5
6            / * Do dependency analysis * /
7            / * Input : Restructured sequential file (seq_file) * /
8            / * Outputs: Number of parallelizable loops (par_loops) * /
9            / * and integer array of loop ids (loop_ids). * /
10           par_loops := doDependencyAnalysis(seq_file, loop_ids)
11           for (i := 0; i < par_loops; i++)
12               if (is_dominated(loop_ids[i])) ∨
13                  (is_nested(loop_ids[i]))
14                  then
15                        / * Cannot parallelize it * /
16                        par_loops := par_loops − 1;
17                        loop_ids[i] = 0;
18               fi
19           end
20
21           / * Mark remaining loops for parallelization * /
22           / * by changing induction symbol to i_N * /
23           for (i := 0; i < par_loops; i++)
24               symb = create_new_symbol("i_%d", i);
25               declare_new_variable(seq_file, symb);
26               loop_ids[i]- > replaceInductionSymbol(symb);
27           end
28           p- > master_file = create_master(p);
29           for (i := 0; i < par_loops; i++)
30               p- > slave_files[i] = create_slave(p, loop_ids[i], i);
31               InsertParallelLoopCode(p, loop_ids[i], i);
32           end
33           p- > par_loops = par_loops;
34         .
```

Figure 4.34: High-Level Code Generation Driver

```
1  FILE*funct  create_master(POP_REC*p)
2            var FILE*seq_file;
3              /* Create Master Process File */
4              seq_file := p-> seq_file;
5              p-> master_file := copyAcrossCode(seq_file);
6              copyAcrossAttributes(seq_file, p-> master_file);
7              declare_master_variables(p-> master_file);
8              insert_master_CS1setup_code(p-> master_file);
9              return master_file;
10           .
```

Figure 4.35: Create_Master File Algorithm

```
1  FILE*funct  create_slave(POP_REC * p, int loop_id, int loop_no)
2            var FILE*seq_file,* slave_file;
3              /* Create a Slave Process File */
4              seq_file := p-> seq_file;
5              slave_file := copyAcrossCode(seq_file);
6              copyAcrossAttributes(seq_file, slave_file);
7              declare_slave_variables(slave_file);
8              insert_slave_CS1setup_code(slave_file);
9              /* remove all executable statements from slave */
10             /* except the loop to be parallelized */
11             RemoveAllExecutableStatements(slave_file, loop_id, loop_no);
12             return slave_file;
13           .
```

Figure 4.36: Create_Slave File Algorithm

```
1  void funct InsertParallelLoopCode (POP_REC*p, int loop_id, int loop_no)
2        /* Analyse Loop Attributes */
3        loopTriplet := computeLoopTriplet(loop_id);
4        NIters := determine_number_of_iterations(seq_file, loop_id);
5        partition_size := partition_iterations(NIters, NSlaves);
6        setNewLoopBounds(slave_file, NSlaves, partition_size);
7
8        /* Now start generating and inserting code. */
9        /* 1. Scalar variables USE'd within the loop. */
10       UseScalarSymbIds := computeScalarUseIds(loop_id);
11       for (∀ symb ∈ UseScalarSymbIds)
12           /* Insert code into MASTER file, BEFORE loop */
13           val := computeCorrectValue(master_file, loop_id, symb);
14           ass1 := makeAssignmentStatement(symb, val);
15           master_file-> loop_id-> addStatement(ass1, BEFORE);
16           ass1-> addComment("C   Scalar insertion : %s\n",
17                                        symb-> identifier);
18           addAttributes(ass1);
19
20           /* As above, Lines 13 to 18, except */
21           /* Insert code into SLAVE file, BEFORE loop */
22           .........
23       end
24
25       /* 2. Scalar variables DEF'd within the loop. */
26       DefScalarSymbIds := computeScalarDefIds(loop_id);
27       for (∀ symb ∈ DefScalarSymbIds)
28           /* As above, Lines 13 to 18, except */
29           /* insert code into MASTER file, AFTER loop */
30           .........
31           symb-> identifier);
32
33           /* As above, Lines 13 to 18, except */
34           /* Insert code into SLAVE file, AFTER loop */
35           .........
36           symb-> identifier);
37       end
38 .
```

Figure 4.37: Generate Parallel Code Algorithm (Fig. 1 of 2)

```
39 cont...
40        /* For ARRAY Access Regions within the loop */
41        /* need to create and insert csn_tx(SEND) and */
42        /* csn_rx(RECEIVE) message-passing code. */
43
44        /* 3. Array Regions USE'd within the loop. */
45        UseArraySymbolTriplets :=
46                          computeArrayUseTriplets(loop_id, loopTriplet);
47        for (∀ symb ∈ UseArraySymbolTriplets)
48            /* Insert PACK_LOOP and SEND code into */
49            /* MASTER file, BEFORE loop */
50            insert_master_pack_loop(master_file, loop_id, symb);
51            /* Insert UNPACK_LOOP and RECEIVE code into */
52            /* SLAVE file, BEFORE loop */
53            insert_slave_unpack_loop(slave_file, loop_id, symb);
54        end
55
56        /* 4. Array Regions DEF'd within the loop. */
57        DefArraySymbolTriplets :=
58                          computeArrayDefTriplets(loop_id, loopTriplet);
59        for (∀ symb ∈ DefArraySymbolTriplets)
60            /* Insert PACK_LOOP and SEND code into */
61            /* SLAVE file, AFTER loop */
62            insert_slave_pack_loop(slave_file, loop_id, symb);
63            /* Insert UNPACK_LOOP and RECEIVE code into */
64            /* MASTER file, AFTER loop */
65            insert_master_unpack_loop(master_file, loop_id, symb);
66        end
67
68        unpdateAttributes(slave, loop_is);
69        insert_UB_LB_calculation_code(slave_file, loop_id);
70        deleteLoopCode(master_file, loop_id);
71 .
```

Figure 4.38: Generate Parallel Code Algorithm (Fig. 2 of 2)

4.10.8 Communication Optimisation

An important part of compilation for distributed-memory multiprocessors is the analysis of the applications' communication needs and the introduction of explicit message-passing operations into it. Before a statement is executed on a given processor, any data it relies on that is not available locally must be obtained. Since REVOLVER is an experimental research tool whose development is ongoing, no communication optimisations are currently performed. A communications message is generated for each nonlocal data item referenced by a statement with the resulting high communications overhead that this approach entails. However, the same code-generator is used by REVOLVER at all times thus allowing fair comparison to be made of the evolutionary parallelization techniques we are interested in.

Once parallel code has been generated its performance needs to be estimated.

4.11 Static Performance Estimation

The performance estimation techniques implemented in REVOLVER are presented between pages 70, and 80.

The profiling techniques and attribute system used are described in section 2.7.1. Algorithms to accumulate workload and communications time-costs as part of cost-model analysis are described in section 2.7.3. Methods for gathering benchmark times of operations and communications are described in section 2.7.2. The actual time figures for operations gathered and stored in lookup tables can be found Appendix A (page 222) along with the derived characteristic equations for communications-costs. Operations evaluation time costs are described in section 2.7.2. Performance estimations also include process and message start-up times.

4.11.1 Data Gathering

Statistics on the performance of a REVOLVER run are gathered and written to a text file to allow post-parallelization analysis of the run. The format of typical data gathered during a run can be seen in Fig. 4.39.

```
===========
Parameters.
===========
PROGRAM : livermore18
ALGORITHM: genetic_algorithm1()
MAXGENS = 10
POPSIZE = 5
preNormalisation = 0
SEED = 7919
DECODER = 1
START_LENGTH = 5
NPROCS = 12
TSIZE = 2
pMutGene = 0.200000
pMutLoop = 0.200000
pMutTF   = 0.200000
pVLX1 = 0.500000
pVLX2 = 0.500000
pVLX3 = 0.500000
==============
==============

 generation   best      average   standard
   number     value    fitness   deviation

    1       17.595    24.255      2.987
    2       17.542    17.831      2.587
    3       17.502    17.547      0.047
    4       17.323    19.724      3.926
    5        6.267    15.948      8.972
    6        6.261    13.695      6.439
    7        0.879    18.197     10.177
    8        0.878    16.037      9.098
    9        0.878    13.777      6.142
   10        0.878    12.400     10.269

  Simulation completed

  Best Member:
  1. TransCode = LRV param1 2 param2 0 param3 0
  2. TransCode = LRV param1 4 param2 0 param3 0
  3. TransCode = LSP param1 4 param2 0 param3 0
  4. TransCode = LRV param1 6 param2 0 param3 0
  5. TransCode = LRV param1 8 param2 0 param3 0

  Best fitness = 0.878
```

Figure 4.39: Performance Statistics Gathered During a Typical REVOLVER Run

4.11.2 Arbitrary Precision Library

As part of the performance estimation system in REVOLVER a small arbitrary precision library was implemented. This was needed to prevent accumulated 'rounding' errors as part of performance estimation. This was occurring because of the clock resolution (1 clock-tick = 10 × 64E-6 secs) of the CS-1. Performing arithmetic in C with numbers of this degree of precision was accumulating rounding-off errors. The library simply treats fixed-floating point numbers as character-strings. It only implements addition and multiplication operations (since they are all that is needed in accumulating time-costs). Accurate to 20 decimal places but may be varied.

4.12 Factors Affecting Results

Along with the code-generation problem described in 4.10.3 another factor affecting the use of REVOLVER was caused by the Sage^{++} libraries.

4.12.1 Sage^{++} Implementation Problems

One problem affecting the results of REVOLVER was with the Sage^{++} libraries which are used for restructuring code (lexing, parsing, moving statements around, etc). It became clear that the libraries (written over several years in K&R C, ANSI-C and C^{++}) suffer from memory leaks which are particularly complex and difficult to fix. This has the effect of restricting the number of generations an EA can run for. The libraries offer much functionality and savings in development time and effort but were not written with use as part of an EA in mind.

4.13 Summary

In this chapter we described the two original representations of the automatic parallelization problem (the 'gene-transformation' and 'gene-statement' representations) both of which are suitable for manipulation by a variety of evolutionary algorithms and whose performance and effectiveness we shall compare in the next two chapters. The concept of decoding an invalid transformation into some alternative action and

the difference it will have on the population as they move through the solution space is introduced. Six genetic operators are described - 3 mutation and 3 crossover operators.

An interactive mode session with REVOLVER is then described, with screenshots, and also batch and fully-automatic (i.e. evolutionary) processing modes are described. The various profiling, dependency analysis, transformation, code-generation and performance estimation phases of REVOLVER are then described using detailed algorithms and diagrams where necessary.

Finally, difficulties with code-generation for the CS-1 and and with use of the Sage^{++} libraries are noted.

5 Experiments and Results

5.1 Introduction

The REVOLVER system analyses, restructures and automatically parallelizes a subset of Fortran-77 programs. It exploits a restricted form of master/slave parallelism. In this chapter we describe the experiments we performed with REVOLVER , and present the raw-results as well as some statistical analysis. We begin by describing the set of 5 Fortran 77 benchmark programs we use in our experiments. We then describe the 6 evolutionary algorithms (EAs) currently implemented in REVOLVER . We then present performance figures using the 6 EAs to restructure and parallelize the 5 benchmark programs.

5.2 Benchmark Test Programs

1. `LFK-18.` The Livermore-Loops Fortran Kernel program number 18 (LFK-18). This fragment of code is used in a 2-D Explicit Hydrodynamics computation. The version used is the double-precision, dense array version (actual code is in the case study presented in section A.4, pgs 240 to 251).

2. `ADI.` - This code represents the most computationally intensive loops of the Alternating Direction Implicit (ADI) integration code. This benchmark program is also found in the Livermore Loop Kernels although the implementation used here is the version distributed with the `Petit` dependency analysis libraries. The essential code is presented in Fig. 5.2.

3. `EFLUX.` Subroutine EFLUX is from the FLO52 program of the Perfect Benchmarks which analyzes the transonic flow past an airfoil by finding a solution to the set of unsteady Euler equations. It is one of the

most interesting and time-consuming benchmark subroutines because of its patterned computation and manipulation of both two and three dimensional arrays. (For code, see section A.2, from page 235.)

4. `M44`. Stencil kernel M44.f features a generic fragment of code which is found in many scientific program such as matrix multiplication, Gauss-Jordan and LU decomposition. The benchmark code is in Fig. 5.2.

5. `TEST-1`. This is a specially constructed example program. It contains several interesting loops and has been developed used to test the various features of REVOLVER . Program code is in the Appendices from page 238.

5.3 Description of Evolutionary Algorithms (EAs)

In the following experiments we use six evolutionary algorithms. Each algorithm uses one of the two representations described in Chapter 4 (gene-transformation or gene-statement) and, if using gene-transformation, will be using one of the three decoding strategies described in section 4.4.1. Furthermore, six genetic operators (also described in 4.4.1) will also be available to the EAs, although not all can be used with the various algorithm/representation combinations we experiment with - none of the crossover operators can be used with the hill-climbing or simulated-annealing algorithms, for example. To clarify, a number of abbreviations are defined to describe the EAs, representations, decoding strategies, and operators we use (see Fig. 5.3).

```
do t=0, iters
  do j=0, dim-1
    do k=1, dim-1
      X(j,k) = X(j,k) - X(j,k-1) * A(j,k) / B(j,k-1)
      B(j,k) = B(j,k) - A(j,k) * A(j,k) / B(j,k-1)
    enddo
  enddo

  do j=0, dim-1
    X(j,dim-1) = X(j,dim-1) / B(j,dim-1)
  enddo

  do j=0, dim-1
    do k=dim-2, 0, -1
      X(j,k) = (X(j,k) - A(j,k+1) * X(j,k+1)) / B(j,k)
    enddo
  enddo

  do j=1, dim-1
    do k=0, dim-1
      X(j,k) = X(j,k) - X(j-1,k) * A(j,k) / B(j-1,k)
      B(j,k) = B(j,k) - A(j,k) * A(j,k) / B(j-1,k)
    enddo
  enddo

  do k=0, dim
    X(dim-1,k) = X(dim-1,k) / B(dim-1,k)
  enddo

  do j=dim-2, 0, -1
    do k=0, -1+dim
      X(j,k) = (X(j,k) - A(j+1,k) * X(j+1,k)) / B(j,k)
    enddo
  enddo
enddo
```

Figure 5.1: Program Fragment of ADI Example Code

```
C     FILE : m44.f

      PROGRAM MAIN
      INTEGER N
      INTEGER I1D, J2D, I1D00, J2D00, I1D01, J2D01
      INTEGER A(1000,1000), B(1000,1000), C(1000)
      INTEGER I, J

      N = 1000

C     READ ARRAYS
      READ*,((A(I,J), I=1, N), J=1, N)
      READ*,((B(I,J), I=1, N), J=1, N)
      READ*,(C(I), I=1, N)

      DO 31 I1D = 1, N
        DO 30 J2D = 1, N
          A(I1D, J2D) = (-N) + N * I1D + J2D
          B(I1D, J2D) = N * I1D - N * J2D
30      CONTINUE
31    CONTINUE

      DO 11 I1D00 = 1, N
        C(I1D00) = 0
        DO 10 J2D00 = 1, N
          C(I1D00) = C(I1D00) + B(I1D00, J2D00)
10      CONTINUE
11    CONTINUE

      DO 41 I1D01 = 2, N-1
        DO 40 J2D01 = 1, N
          A(I1D01, J2D01) = B((-1) + I1D01, J2D01) -
     &    B(I1D01, J2D01) + B(1 + I1D01, J2D01)
40      CONTINUE
41    CONTINUE

      END
```

Figure 5.2: Code for M44 Test Program

```
*** REPRESENTATIONS ***

  GT = Gene-Transformation representation.
  GS = Gene-Statement representation.

*** ALGORITHMS ***

  6 Evolutionary Algorithm/Representation combinations tested.
  HC-1 = hill-climber + GT representation.
  HC-2 = hill-climber + GS representation.
  SA-1 = simulated-annealer + GT representation.
  ES-1 = evolution-strategy + GS representation.
  GA-1 = genetic-algorithm + GT representation.
  SAGA-1 = self-adaptive genetic-algorithm + GT representation.

  3 population-based algorithms (ES-1, GA-1, SAGA-1)
  3 solo-based algorithms (HC-1, HC-2, SA-1)

*** OPERATORS ***

  3 crossover operators: VLX-1, VLX-2, VLX-3  are suitable for use in
  population-based EAs using GT representation only (GA-1, SAGA-1)

  7 mutation operators:
     MutGene, MutTF, MutLoop are suitable for use
     in EAs using GT representation only (GA-1, SAGA-1).
     (MutGene is the only operator used in HC-1 and SA-1).

     MutLSP, MutLFU, MutLIC, and MutLRV are suitable
     in EAs using GS representation only (ES-1, HC-2).

*** DECODERS ***

  When using the GT representation, 3 decoding strategies
  are available:

  DELETE-AND-CONTINUE = DAC
  DELETE-AND-STOP = DAS
  REPAIR = REPAIR  (i.e. no difference)
```

Figure 5.3: Abbreviations of the EAs, Representations, Decoding Strategies, and Operators

EAs Using Gene-Transformation (GT) Representation							
(Default Decoding Strategy = `DELETE-AND-CONTINUE` (DAC)							
EA Params		Operator Probabilities					
EA	POPSIZE	pMutTF	pMutL	pMutGene	pVLX-1	pVLX-2	pVLX-3
HC-1	1	0.2	0.2	0.2	Nil	Nil	Nil
SA-1	1	0.2	0.2	0.2	Nil	Nil	Nil
GA-1	5	0.2	0.2	0.2	0.75	0.75	0.75
SAGA-1	5	0.2	0.2	0.2	0.75	0.75	0.75

Table 5.1: Default Parameters Settings for EAs Using GT Representation

EAs Using Gene-Statement (GS) Representation					
Algorithm		Operator Probabilities			
Algorithm	POPSIZE	pLSP	pLRV	pLIC	pLFU
HC-2	1	0.2	0.2	0.2	0.2
ES-1	5	0.2	0.2	0.2	0.2

Table 5.2: Default Parameters Settings for EAs Using GS Representation

For further parameters such as population size (`POPSIZE`) and number of generations (`MAXGENS`) we simply use the compiler switch abbreviation (see section A.6).

Because of the memory management problems with the Sage^{++} libraries noted in section 4.12.1 runs are limited to 50 generations each.

The default settings for each EA are shown in Tables 5.1 and 5.2.

All EAs used elitism, (the best (one) solution found so far was guaranteed to be kept for the next generation) - except for the simulated annealing algorithm (SA-1) where the change in the best solution found so far is made according to a 'cooling' schedule (i.e. initially, less-fit solutions may be retained to avoid entrapment in local optima, as the run progresses however the chance of retaining a less fit solution diminishes/'cools'). The particular cooling timetable we use in SA-1 is illustrated in Fig. 5.4.

The selection of population members to go into the next generation is performed

```
1  program   Simulated − Annealing
2  begin
3     /* Define 'e' - the base of natural logarithms */
4     #define E 2.718282
5     ...
6     Temperature = 100.0; /* Temperature goes from hot to cold */
7     step_size = (int) (GENS / 10); /* cooling step-size */
8    if (step_size == 0)
9      then
10           step_size = GENS;
11   fi
12    ...
13    CurrentStr = ...
14    ...
15   while ¬terminate do
16           ...
17           NewStr = ...
18           ...
19          if (f(NewStr) < f(CurrentStr))
20            then
21                  CurrentStr = NewStr
22                  /* Keep the best so far with probability
23                  proportional to Temperature */
24          elsif (randreal(0.0, 1.0) < pow(E, (NewStr − CurrentStr)/Temperature))
25               then
26                    CurrentStr = NewStr
27          fi
28
29           /* Temperature is cooling ...*/
30          if (((generation mod step_size) == 0) ∧ (Temperature >= 10.0))
31            then
32                  Temperature -= 10.0;
33          fi
34   end
35 end
```

Figure 5.4: Simulated Annealing **SA-1** Template Algorithm

by a *selection operator*. A selection operator is only necessary in population-based EAs. We use a tournament selection operator with a size of 2. Say we need to select 10 members to go into the next generation, we randomly select 2 members from the current population and allow the fittest one to go through. At no stage was any other selection method used. Tournament selection was used for EAs GA-1 and SAGA-1. The ES-1 algorithm is an evolution strategy and not a genetic algorithm, as such it uses a *selection strategy* rather than a specific operator (see section 3.2.6). Algorithm ES-1 uses a comma-selection strategy (where members of the next generation are selected only from the offspring) and with a POPSIZE of 5 may be described as ES1 = (5,5).

For EAs using the GT representation an initial sequence (chromosome) of transformations is randomly generated when processing begins. For HC-1 and SA-1 the length of this chromosome will always be 1. For GA-1 and SAGA-1 the default starting length is 5 and the length of chromosomes may change as the run commences (due to crossovers, etc).

All operator probabilities were kept at a fixed value throughout the course of a run except for in SAGA-1 where they could adapt according to a schedule. Operator rates may adapt in accordance with some pre-defined schedule (as here) or in accordance with some criteria related to how the GA is performing (e.g. has the population converged, how many generations have been performed, how long since the last improvement in fittest solution found, etc). An outline of the self-adaptive GA algorithm SAGA-1 as implemented in REVOLVER is presented in Fig. 5.5. It illustrates how the operator probabilities are changed as a function of time (i.e. generations). The mutations increase over time while the crossovers decrease. The reasoning behind this is that an early number of crossovers allow the population to sample a wide area of the search space, as the run progresses an increase in mutations enable more 'fine-tuning' of solutions to take place.

The number of processors parallel code was produced for was fixed at 12 (although REVOLVER itself may generate code for varying numbers of processors).

Throughout the following experiments, all parameters kept as above unless stated where appropriate.

```
1  program   SAGA − 1
2  begin
3     // Operator rates are adapted as GA-run progresses
4     inc_size = (float) ((float) 1.0) / ((float) GENERATIONS);
5     pMut-Gene = 0.00; /* probability of MutGene mutation */
6     PMutLoop = 0.00; /* probability of MutLoop mutation */
7     PMutTF = 0.00; /* probability of MutTF mutation */
8     PVLX1 = 1.00 ; /* probability of VLX1 crossover */
9     ...
10    ...
11    while ¬terminate do
12            ...
13            ...
14            /* Probability of Mutation(s)
15             increases as a function of time */
16            PMutGene += inc_size;
17            PMutLoop += inc_size;
18            PMutTF += inc_size;
19
20            /* Probability of Crossover
21             decreases as a function of time */
22            PVLX1 -= inc_size;
23    end
24    ...
25 end
```

Figure 5.5: Self-Adaptive GA Template Algorithm

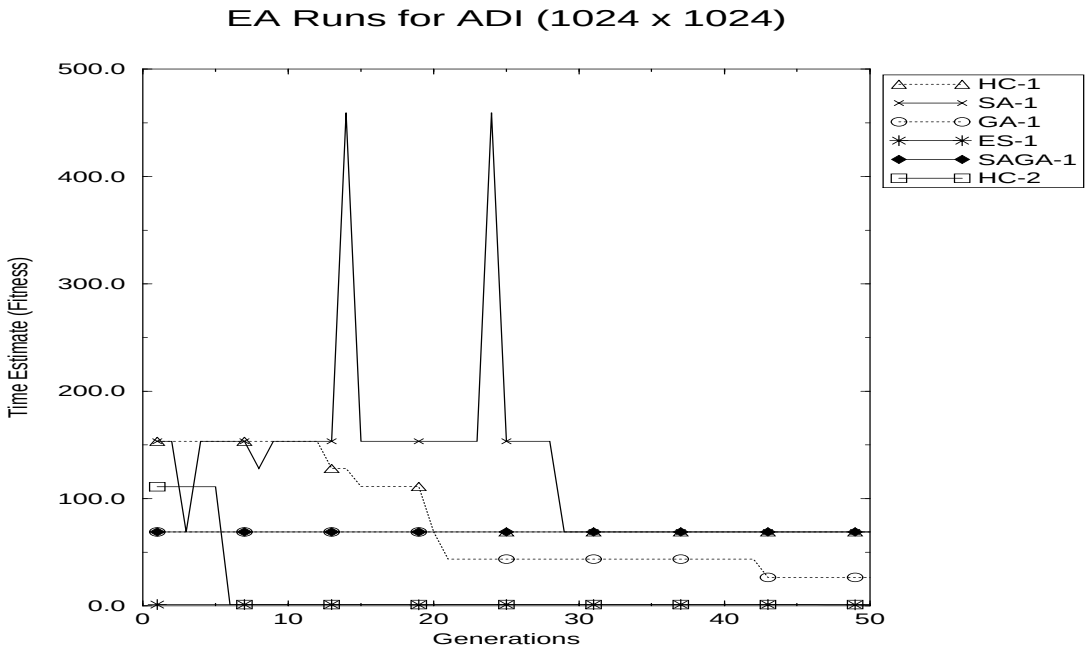


Figure 5.6: EA Runs on ADI (1024 x 1024) Results

5.4 Comparison of Evolutionary Algorithms (EAs)

A simple comparison of each EA restructuring, parallelizing and estimating the fitness of each of the 5 test programs can now be made. Initially, each EA was run once on each test program and its progress plotted onto a graph. The results of these runs are visible in Figs. 5.6 to 5.10. Time estimates are given in seconds (the aim obviously is to produce parallel code with minimal time-costs). With the ADI code algorithms ES-1 and HC-2 produced particularly efficient code. An example of parallelized LFK-18 code can be found in section A.4.

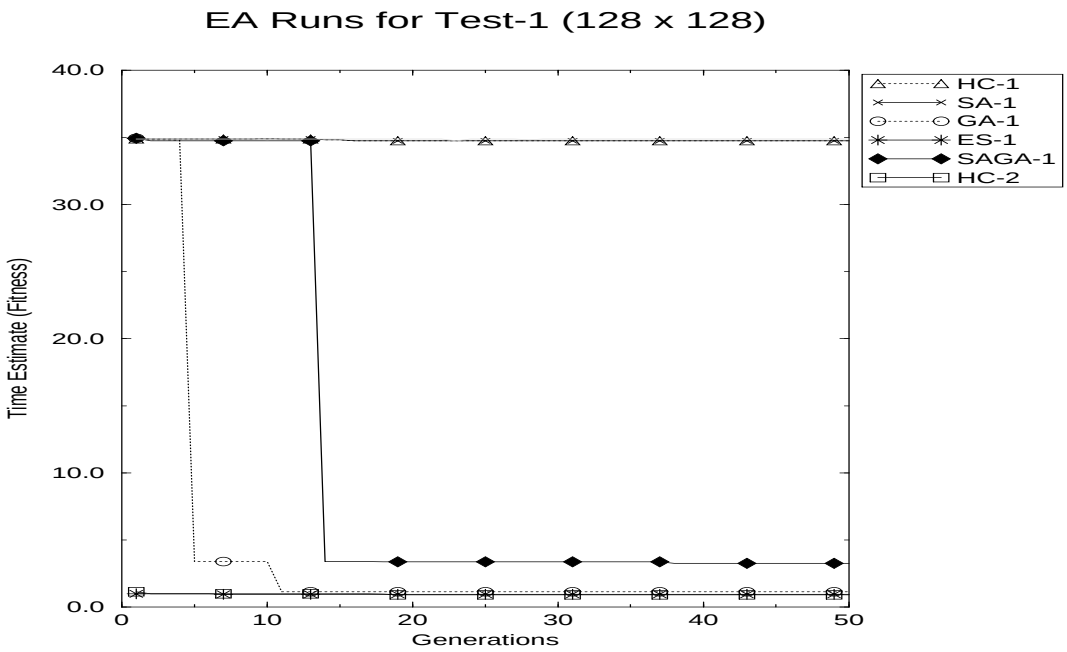


Figure 5.7: EA Runs on Test-1 (128 x 128) Results

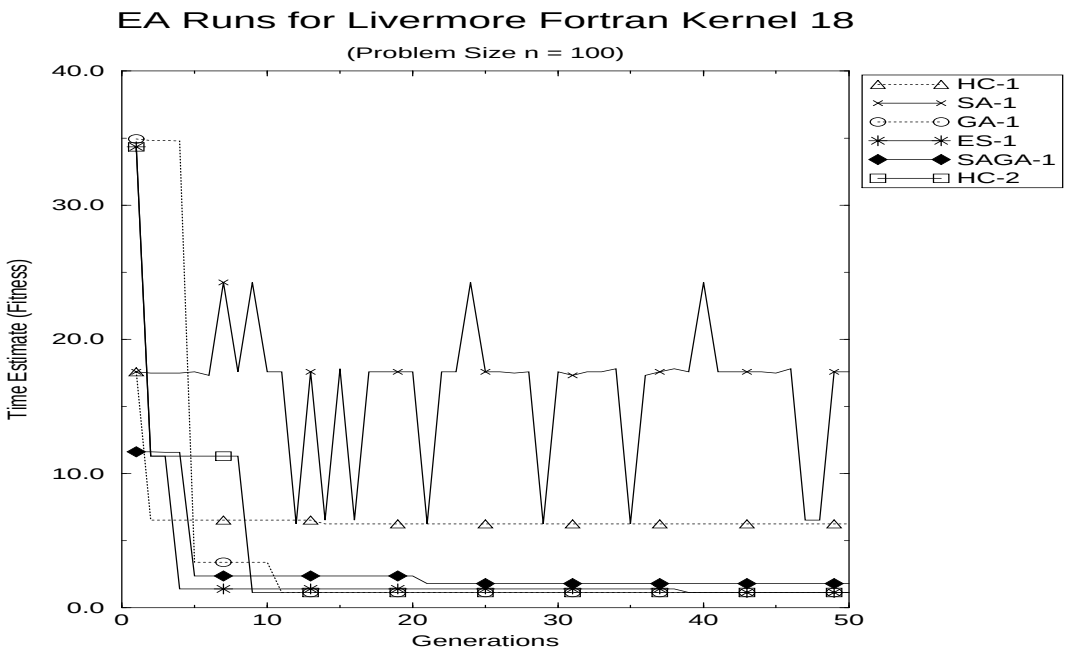


Figure 5.8: EA Runs on Livermore Fortran Kernel 18 (Problem size n = 100) Results

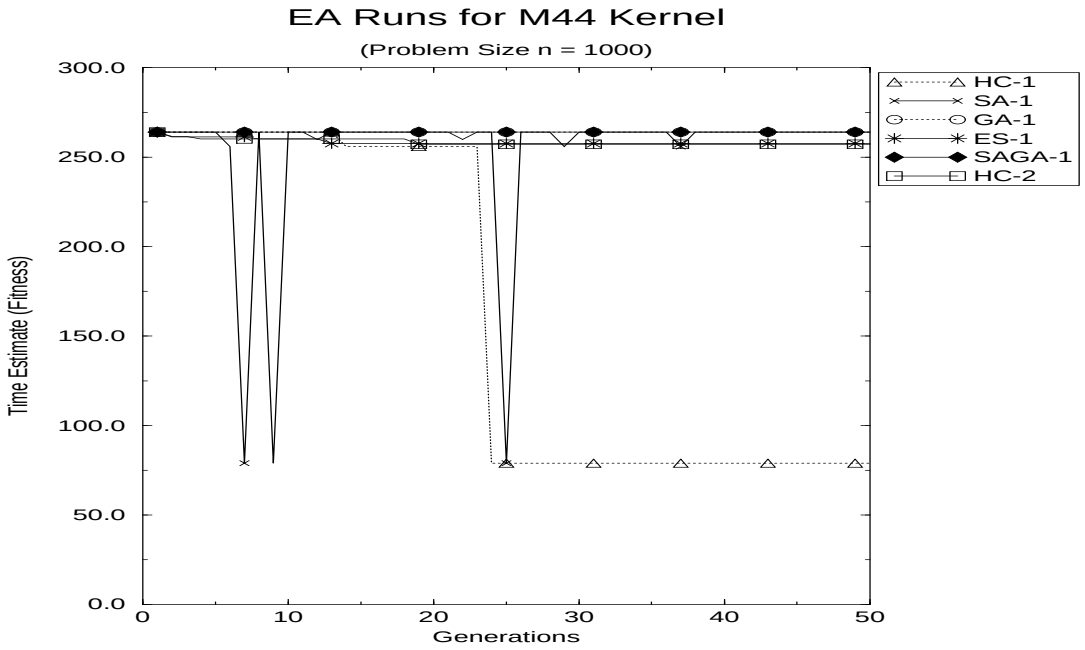


Figure 5.9: EA Runs on M44 Kernel

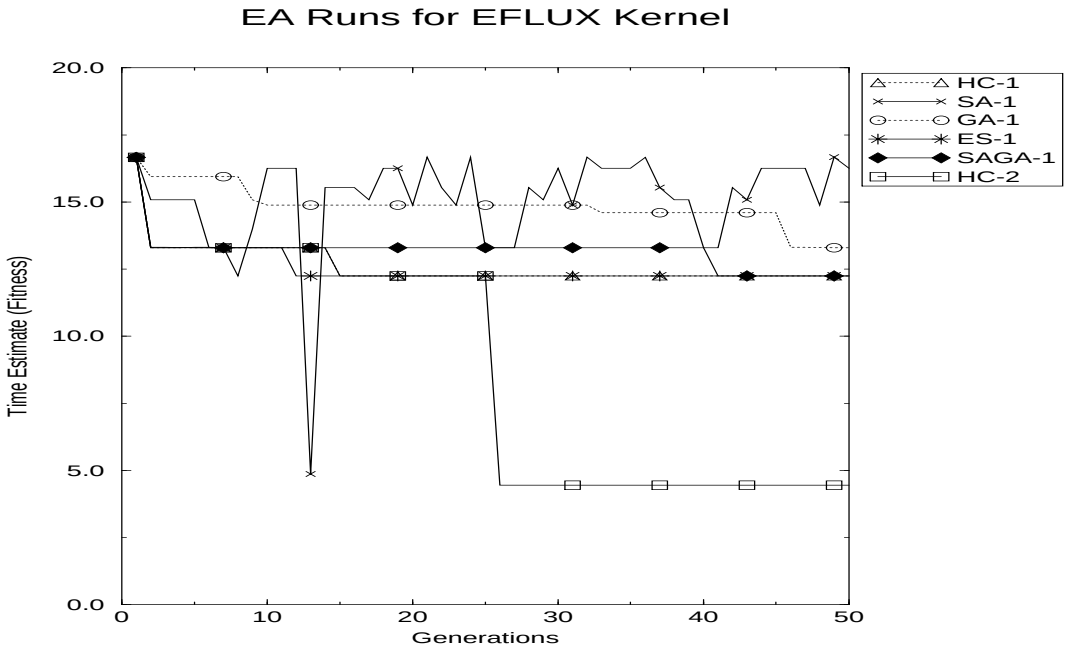


Figure 5.10: EA Runs on EFLUX Kernel

5.5 Comparison of EAs as Stochastic Processes

It is in the nature of stochastic search techniques such as evolutionary algorithms that they incorporate some form of randomness into their processing. This often takes the form of employing a random number generator to influence the decision-making at crucial moments in the algorithm. This is the case in REVOLVER . The generator used is the standard UNIX *rand()* multiplicative congruential generator with period 2^{32} which returns successive pseudo-random integers in the range 0 to $2^{31} - 1$ inclusive. Initially this is seeded with a call to *srand()* with the initial *seed* value set to a default of 1, unless set by the compiler `-seed` switch. The spectral properties of `rand()` leave much to be desired yet even using this simple generator is enough to highlight the variable and probabilistic nature of evolutionary parallelization. In other words, two identical EAs can produce completely different results simply by seeding the random number generators differently.

The average performance of the EAs across the set of test programs can be illustrated by recording the fittest member found by each algorithm for each of the test programs*. The results of 10 runs on each test program are presented in the following tables. All parameters are set as described in section 5.3 and Tables 5.1 and 5.2. The only difference in the set up of each run was the seeding of the random number generator. The seed values for each of the 10 runs are as indicated in the *Seed* column. The choice of seeds is an arbitrary mix of prime and non-prime integers. The bottom row of each table presents the arithmetic mean-average ($\overline{x}$) of each set of 10 runs.

From these figures, we can see two things:

- The solution space is clearly uneven and noisy with sharp peaks, troughs and discontinuities. This is indicated by by the wide variety of values deemed 'best' by different EAs in different runs. If the solution space was smooth with one clear optima then the EAs would invariably find it regardless of the random number generator seed.

- As such, the choice of seed for the random number generator used can

*The value recorded for simulated-annealing algorithm **SA-1** is the best value found in the whole run, rather than the fittest value at the end of the run.

Test Program: `ADI`						
	Evolutionary Algorithms					
Seed	HC-1	SA-1	HC-2	ES-1	GA-1	SAGA-1
1	34.745	68.922	1.243	1.095	26.664	68.880
4711	68.922	68.922	1.243	1.243	153.437	26.664
7919	68.922	153.437	1.243	1.095	26.664	68.922
48611	128.057	128.057	1.095	1.095	43.500	68.922
104729	68.922	68.922	26.664	1.243	26.664	26.664
3571	153.437	68.922	1.243	1.243	68.922	43.542
1299709	68.922	68.922	1.284	1.284	43.542	68.922
1021	68.922	68.922	1.284	1.095	68.922	43.542
611953	153.437	68.922	1.243	1.284	26.664	68.922
2	68.922	111.179	26.664	1.284	26.664	43.542
$\overline{x}$	88.321	87.513	6.321	1.196	51.164	52.852

Table 5.3: Fittest values produced over 10 runs by all EAs on ADI using different seeds for the random number generator.

Test Program: `M44`						
	Evolutionary Algorithms					
Seed	HC-1	SA-1	HC-2	ES-1	GA-1	SAGA-1
1	79.039	79.039	257.277	257.571	263.960	263.960
4711	79.039	79.039	70.963	9.017	9.017	70.963
7919	79.039	79.039	70.963	9.017	70.963	70.963
48611	79.039	79.039	70.963	9.017	79.039	259.893
104729	79.039	79.039	70.906	0.942	70.963	70.963
3571	79.039	79.039	9.017	0.942	70.963	70.963
1299709	79.039	79.039	70.906	70.963	70.906	70.963
1021	79.039	79.039	70.963	9.017	70.963	79.039
611953	79.039	79.039	9.017	9.017	70.963	70.963
2	79.039	79.039	0.942	70.963	255.884	78.982
$\overline{x}$	79.039	79.039	70.191	44.647	103.362	110.766

Table 5.4: Fittest values produced over 10 runs by all EAs on M44 using different seeds for the random number generator.

Test Program: `LFK-18`						
	Evolutionary Algorithms					
Seed	HC-1	SA-1	HC-2	ES-1	GA-1	SAGA-1
1	6.261	6.261	1.136	1.136	1.137	1.806
4711	6.540	6.261	0.878	0.879	1.661	1.806
7919	6.261	6.261	0.879	0.878	0.878	1.806
48611	6.261	6.261	0.894	0.878	0.894	0.878
104729	6.261	6.261	0.878	0.878	1.661	1.806
3571	6.261	6.261	0.878	0.878	0.894	1.136
1299709	6.540	6.261	0.878	0.878	0.894	0.878
1021	6.261	6.261	1.806	0.894	1.137	1.806
611953	6.540	6.261	0.878	1.136	1.137	0.878
2	6.261	6.261	0.879	0.878	0.894	1.136
$\overline{x}$	6.3447	6.261	0.998	0.931	1.119	1.394

Table 5.5: Fittest values produced over 10 runs by all EAs on LFK-18 using different seeds for the random number generator.

Test Program: `EFLUX`						
	Evolutionary Algorithms					
Seed	HC-1	SA-1	HC-2	ES-1	GA-1	SAGA-1
1	12.247	4.867	4.455	12.247	13.298	12.247
4711	13.298	12.247	4.455	12.247	16.991	12.247
7919	12.247	13.314	4.455	4.455	13.298	12.247
48611	12.247	4.867	4.455	4.455	13.298	13.298
104729	16.669	12.247	12.247	4.455	12.247	13.298
3571	12.247	12.247	4.455	12.591	13.298	17.220
1299709	13.298	12.247	4.455	4.455	13.314	13.314
1021	12.247	14.882	12.247	4.455	13.298	13.298
611953	12.247	12.247	4.455	4.455	13.298	12.247
2	16.669	12.247	4.455	12.591	13.314	13.298
$\overline{x}$	13.342	11.141	6.013	7.641	13.565	13.271

Table 5.6: Fittest values produced over 10 runs by all EAs on EFLUX using different seeds for the random number generator.

Test Program: `TEST-1`						
	Evolutionary Algorithms					
Seed	HC-1	SA-1	HC-2	ES-1	GA-1	SAGA-1
1	34.745	34.745	0.924	0.960	1.137	3.258
4711	3.331	3.331	1.039	0.984	3.331	3.331
7919	3.331	34.745	0.966	0.940	3.331	3.331
48611	3.258	3.331	0.978	1.064	3.331	3.331
104729	34.745	3.331	0.984	0.960	34.671	3.302
3571	3.331	3.258	0.984	0.940	3.307	3.331
1299709	34.891	34.866	1.022	0.984	3.258	3.258
1021	3.331	3.331	0.960	0.960	1.064	1.064
611953	3.331	34.866	1.066	0.960	3.258	3.258
2	3.331	3.331	1.064	0.940	3.307	3.331
$\overline{x}$	12.762	15.914	0.999	0.969	6.000	3.080

Table 5.7: Fittest values produced over 10 runs by all EAs on TEST-1 using different seeds for the random number generator.

clearly be seen to have some impact on the performance of the EA. Statistically these irregularities should be evenly distributed across the set of all possible runs. For a small, realistic number of runs however, it shows that random, 'one-off' results can and do occur.

5.5.1 t-Test Comparisons

At this stage we wish to compare and measure the effectiveness of the various evolutionary algorithms we have described against each other. Since the automatic parallelization problem is a function minimization problem we may compare algorithms by the optimal objective values of the function they achieve - for the automatic parallelization problem, we may use the smallest estimated execution times produced by a series of runs of an EA on a sequential program, as some indication of the general performance of the EA. To do this we use a statistical method called a *t-test* as fully described in section A.5. Here, we can say briefly that the test compares two data-sets and returns a *confidence value* (from 0% to 100%) which indicates the likelihood of one set of data being better than the other set purely *by chance*. Hence we can take 10 outputs of one EA (dataset 1) and compare them with 10 outputs of another EA (dataset 2), compute the arithmetic mean-average $\overline{x}$ of each of the two datasets (which would allow us to hypothesise that one EA was better than another) and then use the t-Test to give us a confidence value in our hypothesis. Although clearly *not* providing absolute proof of the superiority of one algorithm over another, the indication may be useful and of interest to us.

Example 17. As an example, let us compare the performances of the two hill-climbing algorithms HC-1 and HC-2.

Let us call dataset-1 the 5×10 values taken from the HC-1 column of each of the five tables.

Let us call dataset-2 the 5×10 values taken from the HC-2 column of each of the five tables.

We can now compute,

$$dataset1(\overline{x}) = \frac{1998.09}{50} = 39.96$$

HYPOTHESIS	*CONFIDENCE VALUE*
GA-1 has performed better than HC-1	38%
SAGA-1 has performed better than HC-1	30%
GA-1 has performed better than SAGA-1	9%
ES-1 has performed better than GA-1	99%
ES-1 has performed better than SAGA-1	99%
ES-1 has performed better than HC-2	53%
HC-1 has performed better than SA-1	8%
ES-1 has performed better than HC-1	100%
ES-1 has performed better than SA-1	100%
HC-2 has performed better than SA-1	100%
GA-1 has performed better than SA-1	44%
SAGA-1 has performed better than SA-1	36%
HC-2 has performed better than GA-1	93%
HC-2 has performed better than SAGA-1	95%

Table 5.8: t-Test Comparisons of EAs.

$$dataset2(\overline{x}) = \frac{845.23}{50} = 19.90$$

Since we are minimising a function we hypothesise that the EA that produced dataset-2 has performed better than the EA that produced dataset-1.

HYPOTHESIS : HC-2 has performed better than HC-1.

We now apply the t-Test and produce a confidence value in our hypothesis.

CONFIDENCE = 99%

which means that there is very little chance (1%) that these two sets differed this much purely by chance - strong evidence that HC-2 has out-performed HC-1 on the data presented.

Using similar data from the five tables for the other algorithms we are now able to make further measured comparisons between EAs. These are summarised in Table 5.8.

5.6 Comparison of Representations

From the graphs in Figs. 5.6 to 5.10 and the five Tables plus Table 5.8 it is also interesting to compare the performance of different EAs using the two representations. Indeed, this is perhaps the most striking feature of these results. In each of the five tables, the two EAs using the GS-representation (ES-1, HC-2) both averaged the lowest values for each test program across all 10 runs. This means that EAs using the GT-representation were out-performed by EAs using GS for every program. Notably, this includes the population-based GA-1 and SAGA-1 (which use GT), being regularly out-performed by the solo-based HC-2 (which uses GS).

5.7 Comparison of Solo and Population-Based EAs

One of the first questions researchers into evolutionary algorithms ask when faced with a new application is - can we get as good if not better results using a simple hill-climber rather than use a genetic algorithm? In simple terms, can we tackle the problem to an acceptable level using a 'solo' (i.e. population-size = 1) based EA rather than having to use the computational time and resources required by a population-based EA.

Looking at the results in Table 5.8 we see that when the population-based EAs (GA-1, SAGA-1, ES-1) were compared directly with solo-based EAs (SA-1, HC-1, HC-2) the population-based EAs always won *except* when a population-based EA using GT representation was compared with a solo-based EA using GS representation (namely HC-2) then HC-2 was deemed to have performed better. This suggests that choice of representation had more impact than size of population or choice of algorithm. However within the GT representation the population based EAs (GA-1, SAGA-1) performed better than solo-based EAs (SA-1, HC-1), and likewise within the GS representation ES-1 out-performing HC-2. (Despite HC-2 achieving many respectable results). This suggests that automatic parallelization is not a problem where solo-based EAs may seriously compete with population-based approaches, however further investigation would clearly be needed to verify this result.

5.8 Comparison of Decoders

All the EA results presented so far were produced by EAs using (where appropriate) the `DELETE-AND-CONTINUE` decoding strategy. When using the gene-transformation representation it sometimes arises that a transformation is illegal and cannot be applied. In such a case the decoding strategy is used to tell REVOLVER what to do next. (It is called the decoding strategy because it determines what REVOLVER should do in the event of a transformation failing - hence it translates (or *decodes*) a failed transformation into some alternative action.

It will be noted that for EAs HC-1 and SA-1 the use of `DELETE-AND-STOP` decoding strategy reduces each to the same algorithm as when using the `DELETE-AND-CONTINUE` strategy, since if a transformation cannot be applied it is deleted and because both algorithms do not work on chromosomes of length > 1 then it makes no difference whether processing stops or continues. Hence EA HC-1 employing the `DELETE-AND-CONTINUE` strategy is equivalent to HC-1 employing the `DELETE-AND-STOP` strategy. And SA-1 using the `DELETE-AND-CONTINUE` strategy is equivalent to SA-1 using the `DELETE-AND-STOP` strategy. This equivalence is borne out in the results of graphs 5.12, 5.13, 5.15, 5.17 (i.e all DELETE-AND-STOP graphs) which may be compared with graphs 5.6, 5.7, 5.8, 5.9, and 5.10 respectively. The following graphs, which are illustrations of EA runs using the `DELETE-AND-STOP` and `REPAIR` decoders, may be compared with Figs. 5.6 to 5.10, which were using the `DELETE-AND-CONTINUE` strategies. In all other respects the algorithms were the same, using the parameters defined in Tables 5.1 and 5.2.

As before, we obtain multiple runs of each EA/Decoding-Strategy combination by seeding the random number generator differently. The results are summarised in Tables 5.9 and 5.10.

As previously, we are able to perform t-Test comparison of all the results produced by each strategy. These results are presented in Table 5.11. Use of the `REPAIR` strategy is the clear winner. This result is particularly interesting because the reasons for the `REPAIR` strategy performing significantly better than the other two are not clear. It might be expected that the `DELETE-AND-STOP` (D-A-S) strategy may not perform so well since the approach may prevent the population from sampling a large proportion

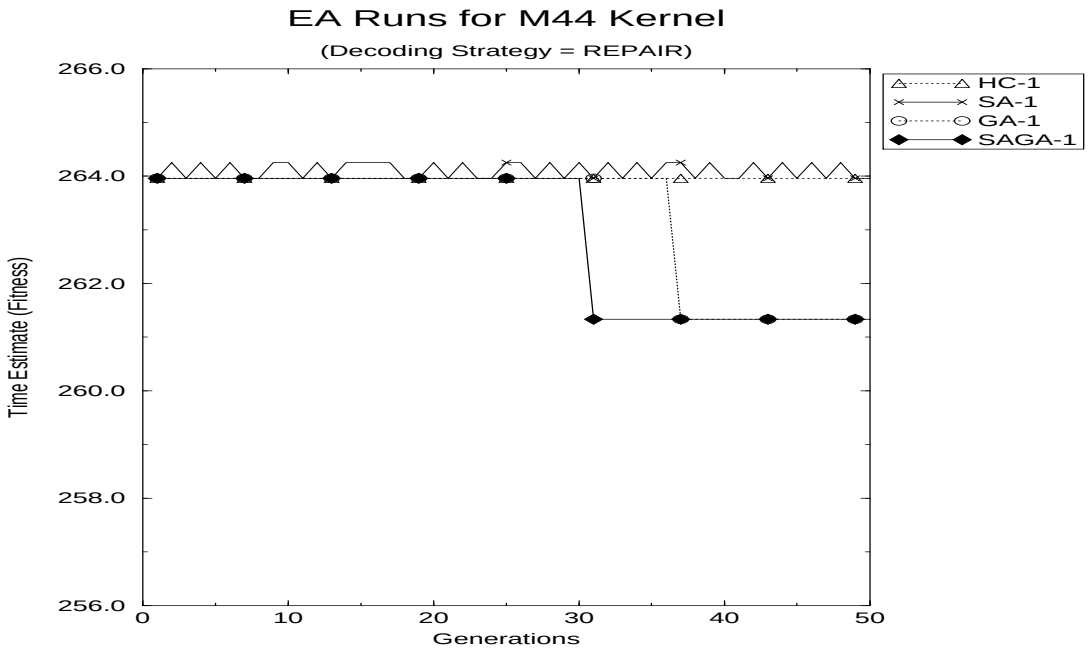


Figure 5.11: EA Runs on M44 Kernel (Decoder = REPAIR)

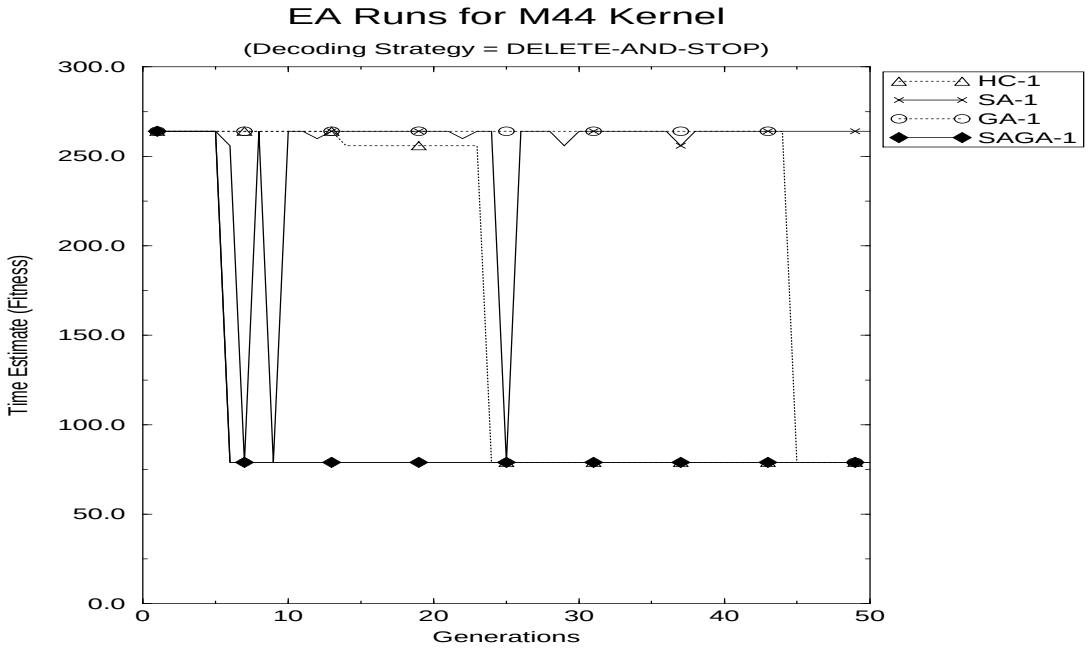


Figure 5.12: EA Runs on M44 Kernel (Decoder = DELETE-AND-STOP)

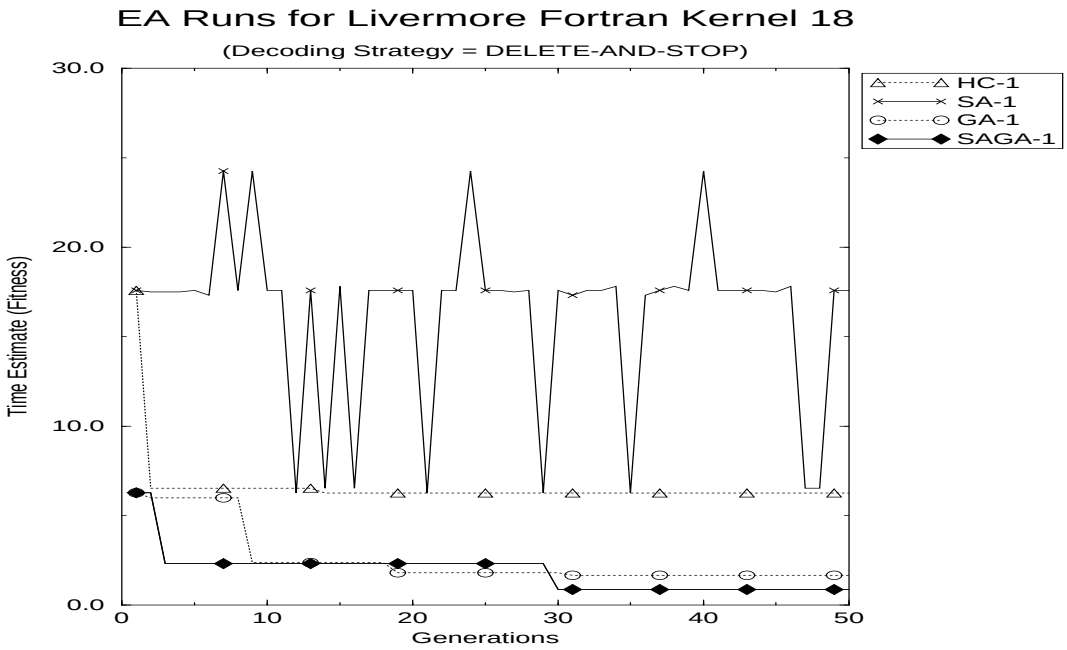


Figure 5.13: EA Runs on Livermore Fortran Kernel 18 (Decoder = DELETE-AND-STOP)

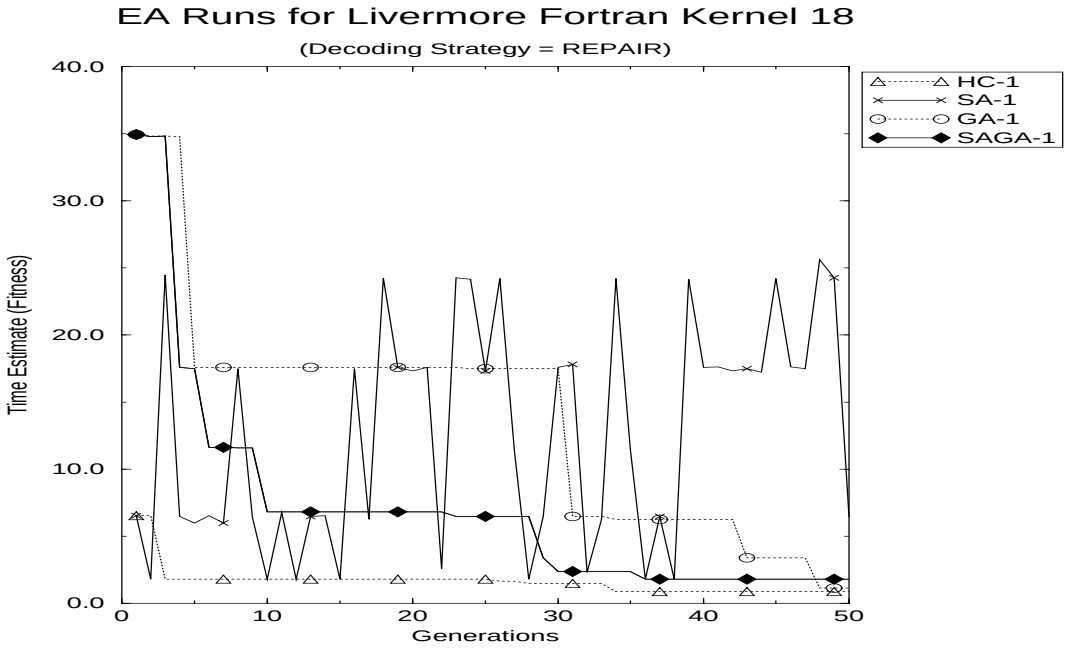


Figure 5.14: EA Runs on Livermore Fortran Kernel 18 (Decoder = REPAIR)

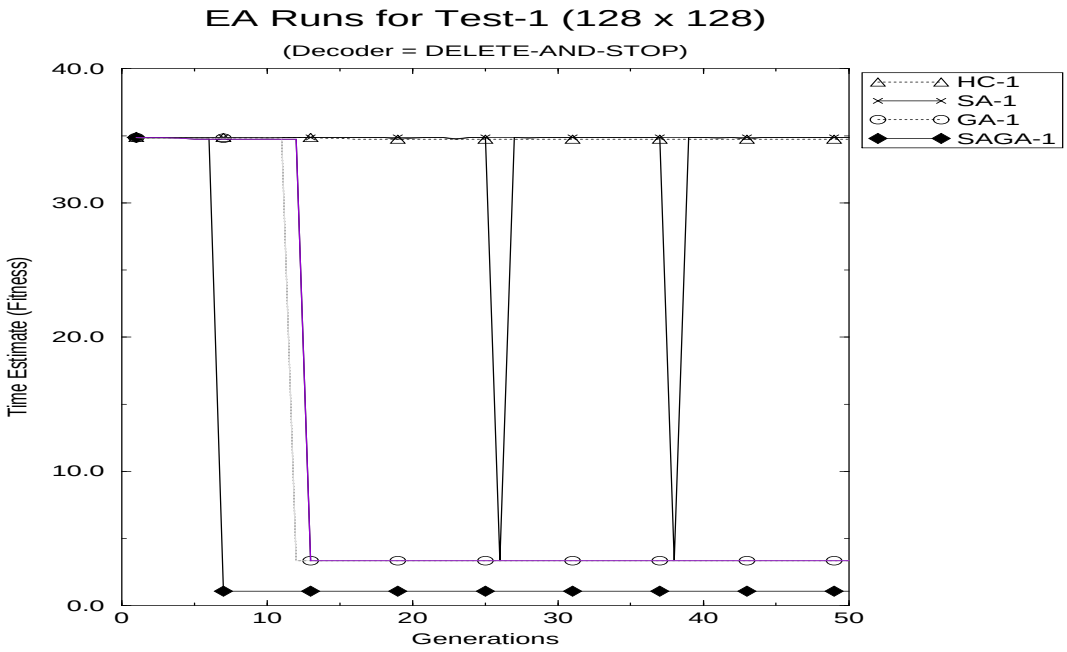


Figure 5.15: EA Runs on Test-1 (Decoder = DELETE-AND-STOP)

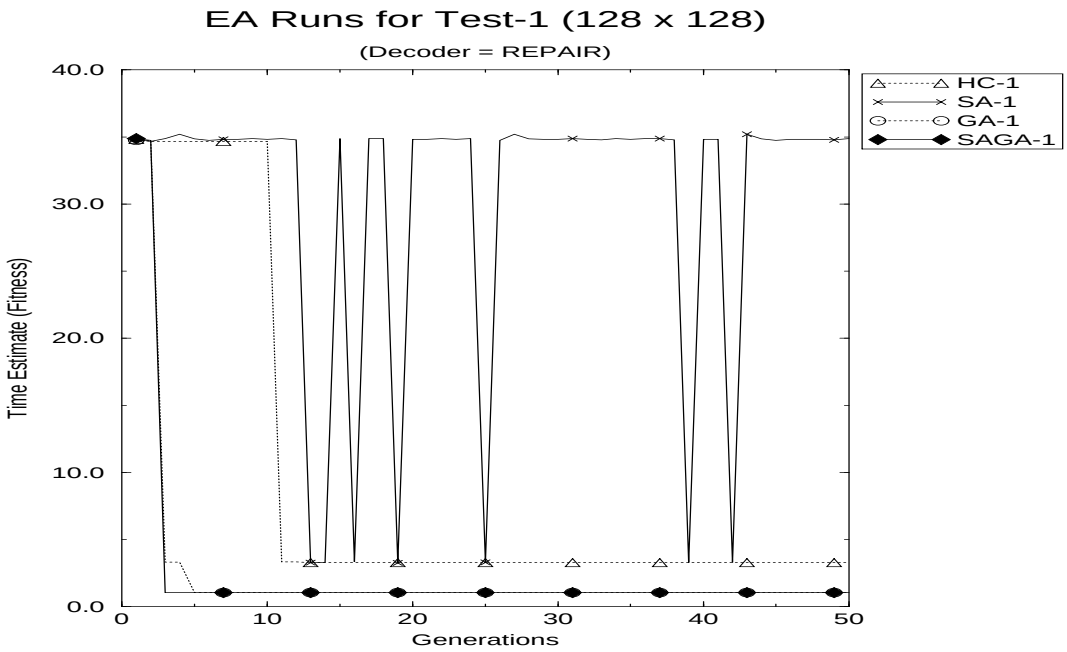


Figure 5.16: EA Runs on Test-1 (Decoder = REPAIR)

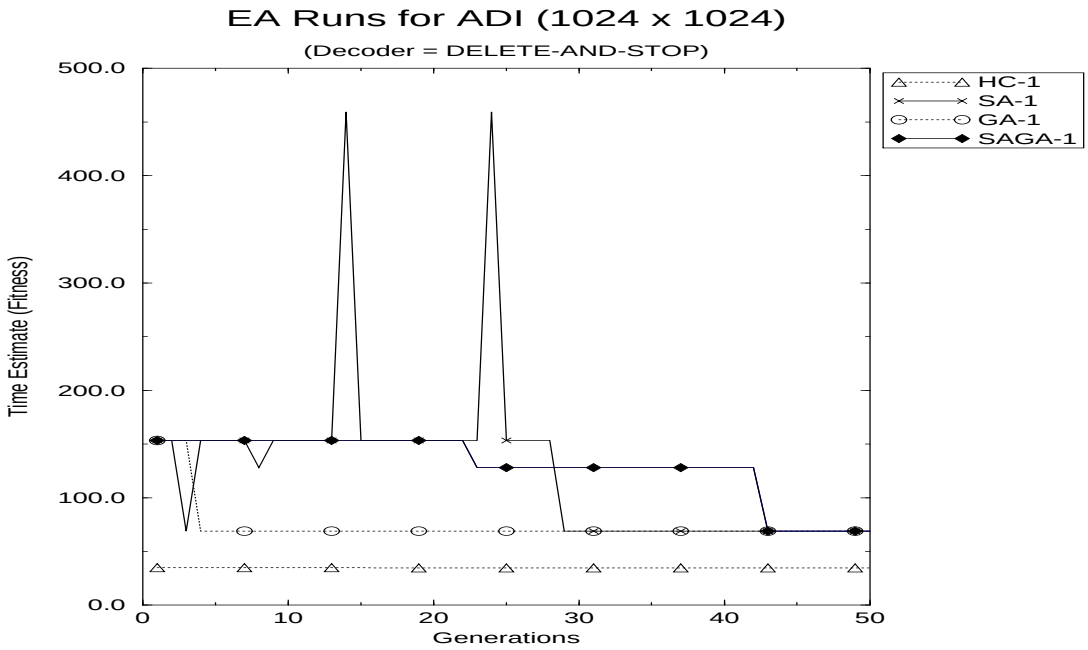


Figure 5.17: EA Runs on ADI (1024 x 1024) (Decoder = DELETE-AND-STOP)

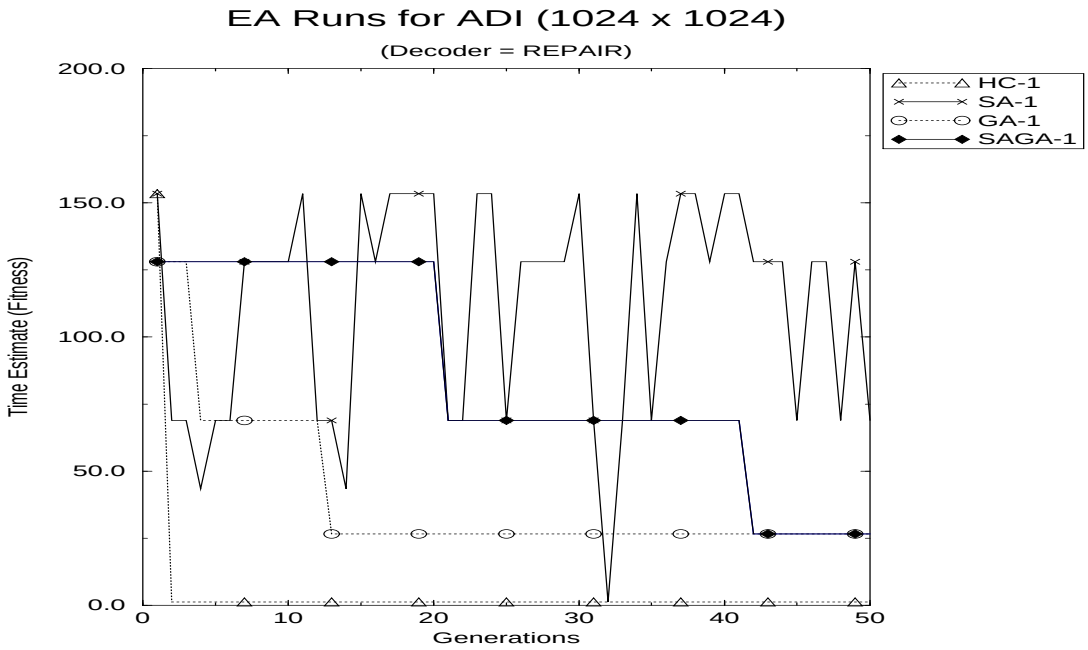


Figure 5.18: EA Runs on ADI (1024 x 1024) (Decoder = REPAIR)

EA Runs on Test Programs using Decoding Strategy = DELETE-AND-STOP												
	ADI				LFK-18				TEST-1			
Seed	SA-1	HC-1	GA-1	SAGA-1	SA-1	HC-1	GA-1	SAGA-1	SA-1	HC-1	GA-1	SAGA-1
1	68.922	34.745	68.922	68.922	6.261	6.261	1.661	0.879	34.745	34.745	3.331	3.331
4711	68.922	68.922	26.664	26.664	6.261	6.540	5.989	6.446	3.331	3.331	3.331	34.818
7919	153.437	68.922	26.664	43.542	6.261	6.261	1.137	1.136	34.745	3.331	3.331	34.866
48611	128.057	128.057	68.922	128.057	6.261	6.261	0.894	0.879	3.331	3.258	34.671	34.745
104729	68.922	68.922	68.880	128.057	6.261	6.261	17.595	17.595	3.331	34.745	34.745	3.302
$\overline{x}$	97.652	73.914	52.010	79.094	6.261	31.584	5.455	5.387	15.897	15.822	15.882	22.212

Table 5.9: Fittest values produced over 5 runs by EAs on 3 Test Programs using different seeds for the random number generator and Decoding Strategy = DELETE-AND-STOP

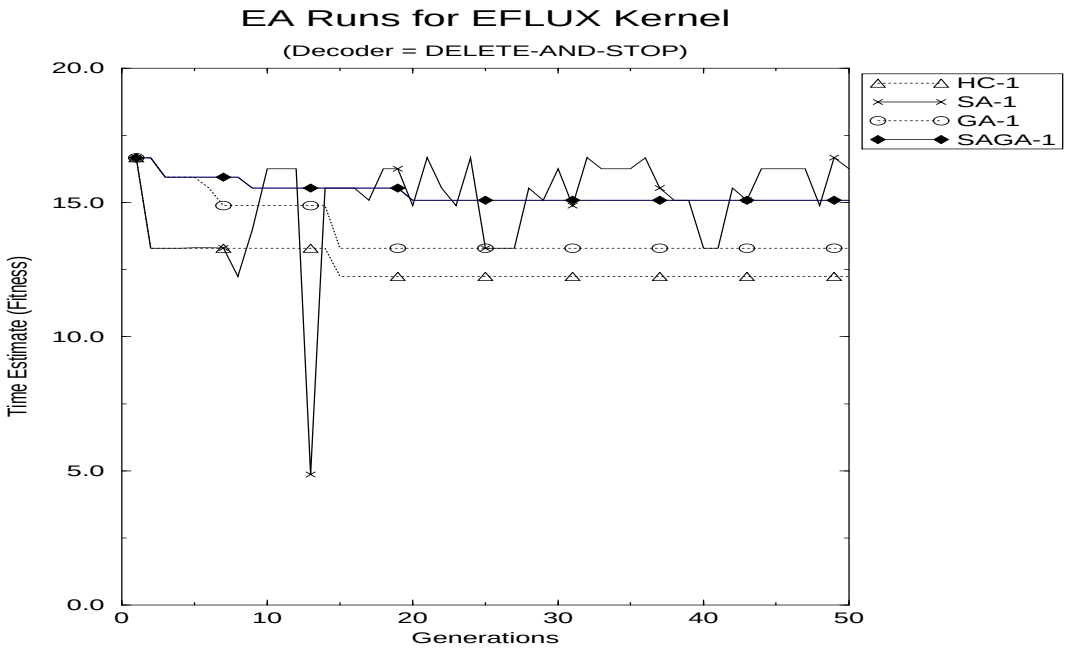


Figure 5.19: EA Runs on EFLUX Kernel (Decoder = DELETE-AND-STOP)

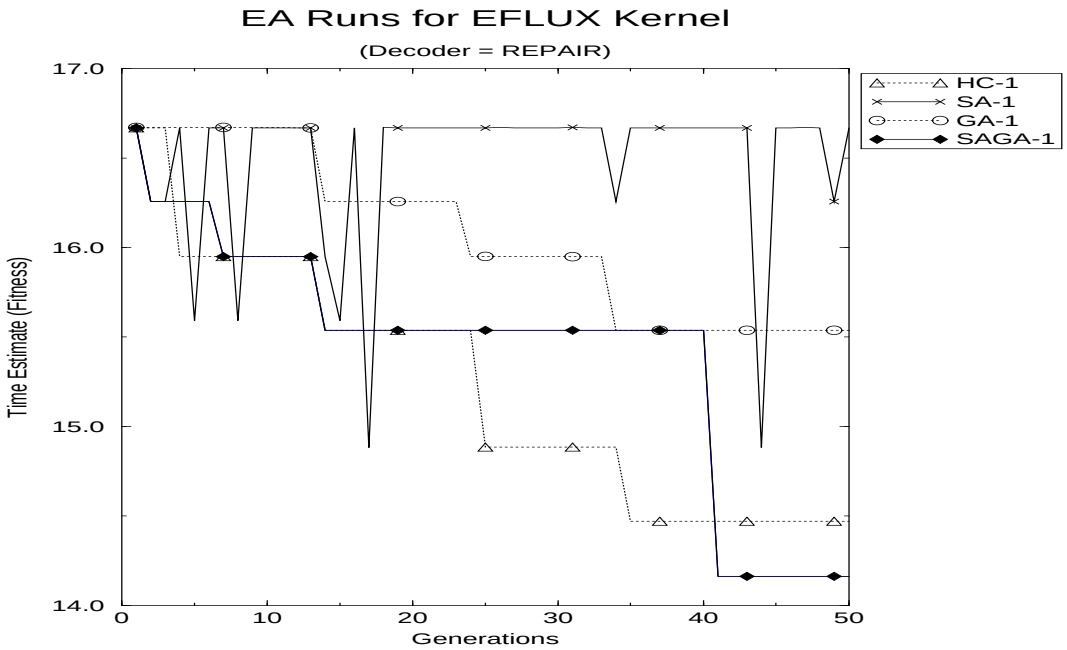


Figure 5.20: EA Runs on EFLUX Kernel (Decoder = REPAIR)

EA Runs on Test Programs using Decoding Strategy = REPAIR												
	ADI				LFK-18				TEST-1			
Seed	SA-1	HC-1	GA-1	SAGA-1	SA-1	HC-1	GA-1	SAGA-1	SA-1	HC-1	GA-1	SAGA-1
1	1.284	1.284	26.664	26.664	1.806	0.878	1.137	1.806	3.258	3.288	1.052	1.045
4711	68.922	68.922	68.922	68.922	6.261	6.261	6.208	1.661	3.258	3.331	3.258	34.873
7919	34.745	68.922	26.664	26.664	1.806	6.261	0.857	1.136	3.331	34.745	34.714	0.984
48611	68.922	68.922	26.664	1.243	6.261	1.806	1.661	0.921	3.331	3.331	0.966	3.258
104729	68.922	68.922	26.664	68.922	6.261	6.261	1.086	1.806	34.745	3.331	1.064	3.301
$\overline{x}$	48.559	55.394	35.116	192.415	4.479	4.293	2.334	1.466	9.585	9.605	8.211	8.692

Table 5.10: Fittest values produced over 5 runs by EAs on 3 Test Programs using different seeds for the random number generator and Decoding Strategy = REPAIR

HYPOTHESIS	*CONFIDENCE VALUE*
REPAIR has performed better than D-A-C	88%
REPAIR has performed better than D-A-S	98%
D-A-C has performed better than D-A-S	50%

Table 5.11: t-Test Comparisons of Decoding Strategies.

of the search space. However this is not true for `DELETE-AND-CONTINUE` (D-A-C). It may well be that forcing *some* transformation to be applied (as the *REPAIR* strategy does and D-A-C does not) may result in the population searching a larger proportion of the solution space and hence finding better solutions. An interesting result.

5.9 Evaluation of Operators

As described in Fig. 5.3, ten genetic operators are defined as part of REVOLVER . However, not all operators are useful with all the algorithms or compatible with the representations available.

The three crossover operators (VLX-1, VLX-2, and VLX-3) are are suitable for use in population-based EAs using GT representation only (GA-1, SAGA-1). Of the seven mutation operators, three (MutGene, MutTF and MutLoop) are suitable for use in EAs using GT representation only (GA-1, SAGA-1) - MutGene is the only operator used in HC-1 and SA-1, and the remaining four (MutLSP, MutLFU, MutLIC, and MutLRV) are loop transformations suitable in EAs using the GS representation only (ES-1, HC-2). For this discussion we do not include tournament selection as an operator, although it is noted that other selection operators and strategies do exist.

In the results presented in sections 5.4 and 5.5 the default EA parameters were used. This meant that the population based EAs which use the GT representation (GA-1, SAGA-1) were using 3 mutation operators (MutGene, MutTF, MutLoop) and 3 crossover operators VLX-1, VLX-2, VLX-3). Our strategy then to test the effectiveness of a particular operator, is to re-run the EA with the probability of the operator set to zero - and then compare the output of the EA with the original run. In this way

Test Programs without Operator VLX-1						
	ADI		LFK-18		TEST-1	
Seed	GA-1	SAGA-1	GA-1	SAGA-1	GA-1	SAGA-1
1	26.664	43.542	1.806	0.878	3.301	34.745
4711	26.664	26.664	1.661	1.661	1.064	1.064
7919	26.664	26.664	0.878	0.878	1.064	3.258
48611	153.395	153.395	0.879	1.661	3.331	3.331
104729	68.922	43.542	2.371	1.806	34.671	34.745
$\overline{x}$	60.462	58.761	1.519	1.379	8.687	15.429

Table 5.12: Fittest values produced over 5 runs by EAs on 3 Test Programs using different seeds for the random number generator and without use of Operator VLX-1.

we will be able to determine if the the operator had a generally beneficial effect on the EAs performance or not.

Only one operator was eliminated at any one time, so the other 5 operators were always in use. Note, these experiments are all performed on EAs using the GT representation (and therefore the GT operators). It did not make sense to test loop-transformation/mutation operators (used with the GS representation) by removing them from the algorithm since this results in effectively removing them from the transformation catalogue too. The results for the experiments performed are detailed in Tables 5.12 to 5.17.

Operator `MutGene` is used with EAs `HC-1`, `SA-1`. These EAs use only this operator and as such it cannot be removed from their processing for comparison.

We perform a t-Test comparison in the following way (essentially we compare the results of the EA *before* the operator was removed and *after* to see if there is any difference - better or worse). To illustrate we give an example below.

Example 18. As an example, let us evaluate the effectiveness of operator VLX-1.

Let us call dataset-1 the 2×5 values taken from the GA-1 and SAGA-1 columns

Test Programs without Operator VLX-2						
	ADI		LFK-18		TEST-1	
Seed	GA-1	SAGA-1	GA-1	SAGA-1	GA-1	SAGA-1
1	26.664	26.474	0.879	0.878	1.064	3.258
4711	26.664	26.664	0.878	1.806	3.331	3.307
7919	26.664	26.664	1.806	1.806	3.331	3.258
48611	153.395	43.542	1.137	1.806	1.064	3.331
104729	68.922	43.542	1.806	1.136	3.331	34.866
$\overline{x}$	60.462	33.377	1.301	1.487	2.424	9.604

Table 5.13: Fittest values produced over 5 runs by EAs on 3 Test Programs using different seeds for the random number generator and without use of Operator VLX-2.

Test Programs without Operator VLX-3						
	ADI		LFK-18		TEST-1	
Seed	GA-1	SAGA-1	GA-1	SAGA-1	GA-1	SAGA-1
1	68.922	26.664	1.806	0.879	3.331	34.671
4711	26.664	26.474	0.878	0.879	3.331	3.331
7919	68.922	26.664	1.661	1.806	3.284	3.301
48611	153.395	68.922	0.894	1.806	1.064	3.331
104729	43.542	1.284	2.099	0.879	3.302	3.302
$\overline{x}$	72.289	30.002	1.468	1.250	2.862	9.587

Table 5.14: Fittest values produced over 5 runs by EAs on 3 Test Programs using different seeds for the random number generator and without use of Operator VLX-3.

Test Programs without Operator MutGene						
	ADI		LFK-18		TEST-1	
Seed	GA-1	SAGA-1	GA-1	SAGA-1	GA-1	SAGA-1
1	26.664	43.542	1.661	0.878	3.302	1.045
4711	26.474	26.664	1.806	1.806	3.331	3.331
7919	43.542	68.922	1.661	1.136	3.258	3.331
48611	68.922	43.542	1.137	1.661	3.331	3.331
104729	43.542	43.542	1.664	1.812	3.307	3.302
$\overline{x}$	41.829	45.242	1.586	1.459	3.306	2.868

Table 5.15: Fittest values produced over 5 runs by EAs on 3 Test Programs using different seeds for the random number generator and without use of Operator MutGene.

Test Programs without Operator MutTF						
	ADI		LFK-18		TEST-1	
Seed	GA-1	SAGA-1	GA-1	SAGA-1	GA-1	SAGA-1
1	68.922	43.542	1.137	1.661	3.302	1.045
4711	26.664	26.664	1.806	2.278	3.313	3.302
7919	43.542	68.922	0.878	1.661	3.313	3.258
48611	153.395	43.542	1.806	1.806	0.991	3.331
104729	68.922	43.542	1.812	1.806	34.671	34.726
$\overline{x}$	72.289	45.242	1.488	1.842	9.118	9.132

Table 5.16: Fittest values produced over 5 runs by EAs on 3 Test Programs using different seeds for the random number generator and without use of Operator MutTF.

Test Programs without Operator MutLoop						
	ADI		LFK-18		TEST-1	
Seed	GA-1	SAGA-1	GA-1	SAGA-1	GA-1	SAGA-1
1	68.922	43.542	0.878	0.879	3.302	1.045
4711	26.664	26.664	1.806	2.278	3.331	3.313
7919	26.664	68.922	1.661	2.371	3.313	3.258
48611	153.395	43.542	0.879	1.661	0.991	3.331
104729	68.922	43.542	1.812	1.806	34.671	34.726
$\overline{x}$	68.913	45.242	1.407	1.799	9.127	9.137

Table 5.17: Fittest values produced over 5 runs by EAs on 3 Test Programs using different seeds for the random number generator and without use of Operator MutLoop.

of Tables 5.3 (ADI), 5.5 (LFK-18), and 5.7 (TEST-1), the rows corresponding to *Seed* values 1, 4711, 7919, 48611, and 104729 (i.e those in Table 5.10). This gives us a dataset of 30 items - all produced by GA-1 and SAGA-1 with operator VLX-1 active.

Let us call dataset-2 the 5×6 values taken from Table 5.10 (data produced without operator VLX-1).

We can now compute,

$$dataset1(\overline{x}) = \frac{613.67}{30} = 20.46$$
$$dataset2(\overline{x}) = \frac{731.17}{30} = 24.37$$

Since we are minimising a function we hypothesise that the EAs that produced dataset-1 have performed better than the EAs that produced dataset-2.

HYPOTHESIS : EAs for dataset-1 performed better than EAs of dataset-2.

We now apply the t-Test and produce a confidence value in our hypothesis.

CONFIDENCE = 32%

which means that although dataset-1 is better than dataset-2 the evidence is not strong for the hypothesis. Although the datasets are drawn from EAs applied to 3 different test programs (ADI, LFK-18, and TEST-1), the comparison is legal because we

HYPOTHESIS	*CONFIDENCE VALUE*
GA-1 and SAGA-1 performed better without VLX-1	32%
GA-1 and SAGA-1 performed better without VLX-2	22%
GA-1 and SAGA-1 performed better without VLX-3	8%
GA-1 and SAGA-1 performed better without MutGene	46%
GA-1 and SAGA-1 performed better with MutTF	25%
GA-1 and SAGA-1 performed better with MutLoop	20%

Table 5.18: t-Test Evaluation of Effectiveness of Genetic Operators.

are comparing the results of applying the EAs to the *same* 3 different test programs. If we were to replace one dataset with results from applying the EAs to the test programs ADI, LFK-18 and say, EFLUX, then the comparison would become invalid because we would not be comparing the application of the EAs to the same test programs (essentially the EAs would be working on different data).

We are now able to accumulate comparisons as before (see Table 5.18).

Several features can be noted about these results. First, in no instance did removing a crossover operator diminish the performance of the EAs, in fact, in each case the removal of a crossover operator resulted in improved performance. This suggests that perhaps the crossover operators are too disruptive and that smaller mutation operators are more suitable to our problem. This may be true, inclusion of MutTF and MutLoop both helped the EAs, despite the sizeable improvement (46%) when MutGene was removed. Secondly, of the three crossover operators, the two 'blind' operators VLX-1 and VLX-2 were both more of a hinderence when they were used than the 'heuristic' operator VLX-3. Although the removal of VLX-3 still resulted in an improvement in the EAs performance, this was only small (8%) and suggests that some further work on this operator might result in some new significant crossover operator. Third, it may be the case that we have too many operators working on too small a population, or that 50 generations is not enough to conduct a full evaluation of the effectiveness of a genetic operator - these are both valid criticisms and as such these results should only

be taken as indications rather than positive evidence of usefulness. Lastly, it may also be argued that none of the confidence values are particularly high and that as such the results are generally inconclusive. This is an argument I would be inclined to agree with.

Another final point is that genetic operators do not work in 'isolation' of each other - the effects of one may help or hinder the work of another. This combined effect of multiple operators is difficult to measure in any experiment.

5.10 Comparison of Auto-Parallelization Strategies

In all the results presented so far the code-generator has worked by trying to generate code for all loops which contain no cross-iteration dependencies regardless of whether the workload makes it worthwhile or not (this is the naive parallelization strategy described in section4.10.7). One important point to note here is that parallel code is not generated for loops which are not in normal form. This is significant for us since in our transformation catalogue we only have one transformation which takes a loop 'out' of normal form, namely Loop-Reversal. Given the extensive communications costs incurred by parallelizing a loop (section 4.10.3) application of the Loop-Reversal transformation is the only way an EA can 'prevent' a loop from being parallelized. It therefore became an interesting idea to compare the results of EAs working with two auto-parallelization strategies (i) normalise all loops prior to code-generation, and (ii) do not normalise all loops prior to code-generation (the default strategy). The compiler switch used for this option is `-preNorm`. If no pre-normalisation of loops is performed, then `preNorm` = 0 (default), if pre-normalisation of loops is required then switch `-preNorm 1` should be set. The idea is that by not normalising loops before code-generation we are giving the EA the option of preventing parallelization of code by applying Loop-Reversal and hence taking a loop out of normal form. An EA might want to do this to avoid incurring communications costs associated with parallel code. By normalising loops before code-generation we are taking away this option. The experiment then is to see if preventing parallelization of loops is a worthwhile strategy - and if so, is this strategy found by any of our EAs. If this is the case, then by

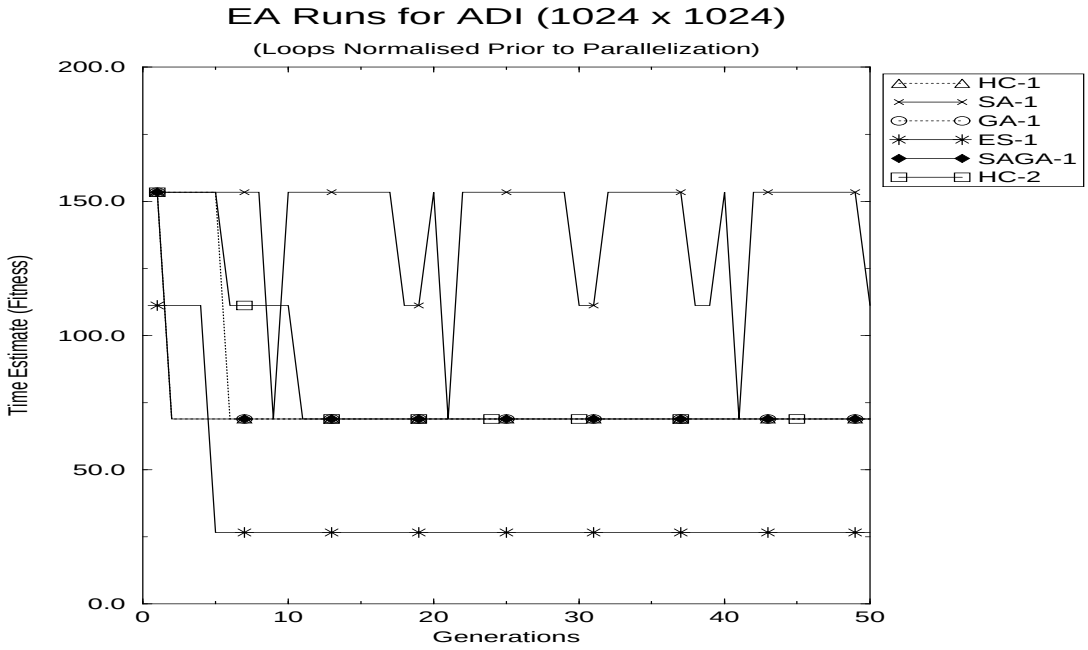


Figure 5.21: EA Runs on ADI (1024 x 1024) (All loops normalised prior to parallelization)

normalising loops prior to code-generation we would expect the code to perform less well than that produced when no pre-Normalisation is performed. A direct comparison of the of the results generated with preNorm = 0 (Figs. 5.6 to 5.10) can be made with results of EAs using `-preNorm 1` in the following Figs. 5.21 to 5.25.

A comparison of the graphs clearly shows that at no time did EAs using `-preNorm 1` generate better code than EAs using `-preNorm 0`. Furthermore, an examination of code after restructuring but before parallelization shows that many of the fittest programs produced actually have very few loops parallelized because extensive Loop-Reversal has been performed so as to deliberately prevent them from being parallelized.

Paradoxical as it may seem, this is an immensely significant result. By *not* normalising loops prior to code-generation we have inadvertently allowed the EAs to find a new and unexpected strategy for automatically parallelizing code. This strategy is to apply Loop-Reversal to prevent them from being parallelized - which makes sense given the extensive costs of communication incurred when a loop is parallelized. It follows that at some point the benefits of parallelism will outweigh the costs of any communication overhead - this is true, but given the prohibitive costs of communication in our

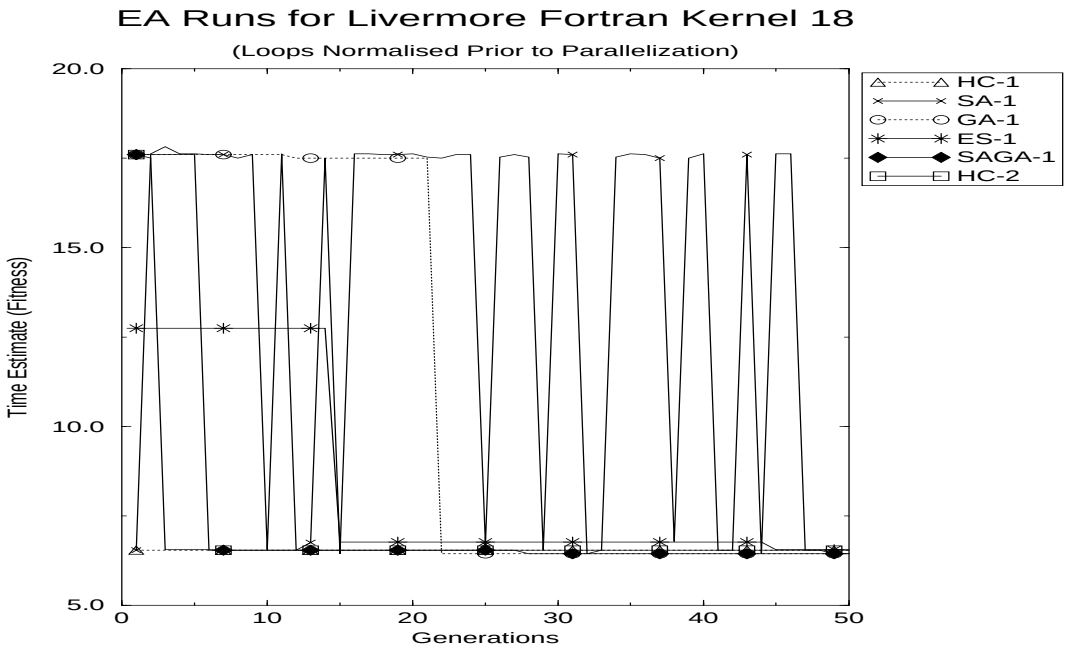


Figure 5.22: EA Runs on Livermore Fortran Kernel 18 (All loops normalised prior to parallelization)

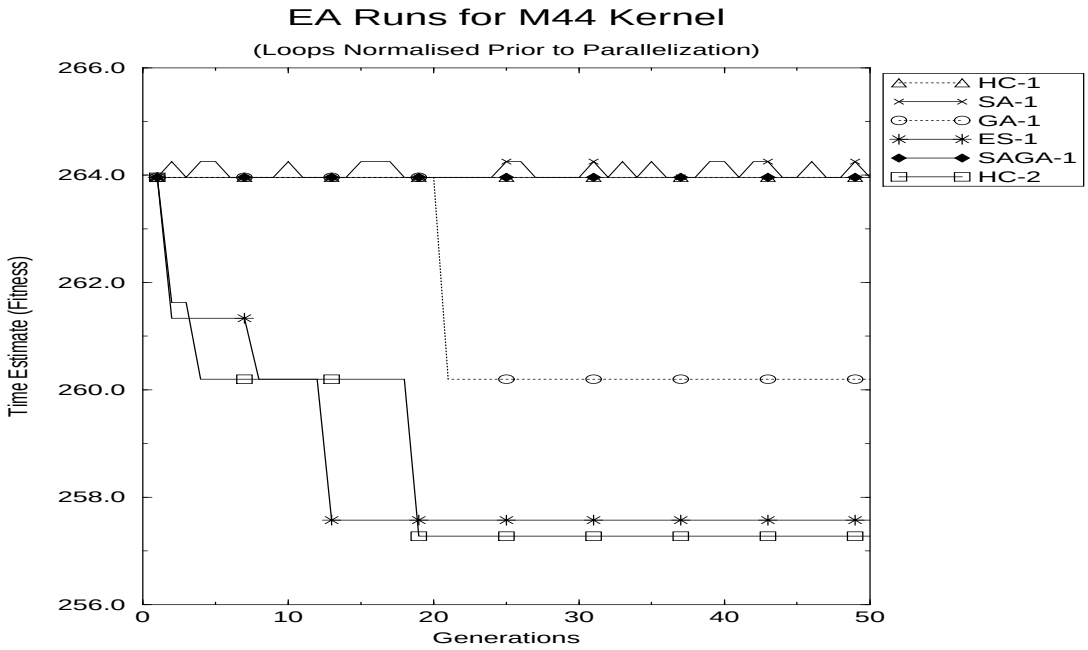


Figure 5.23: EA Runs on M44 Kernel (All loops normalised prior to parallelization)

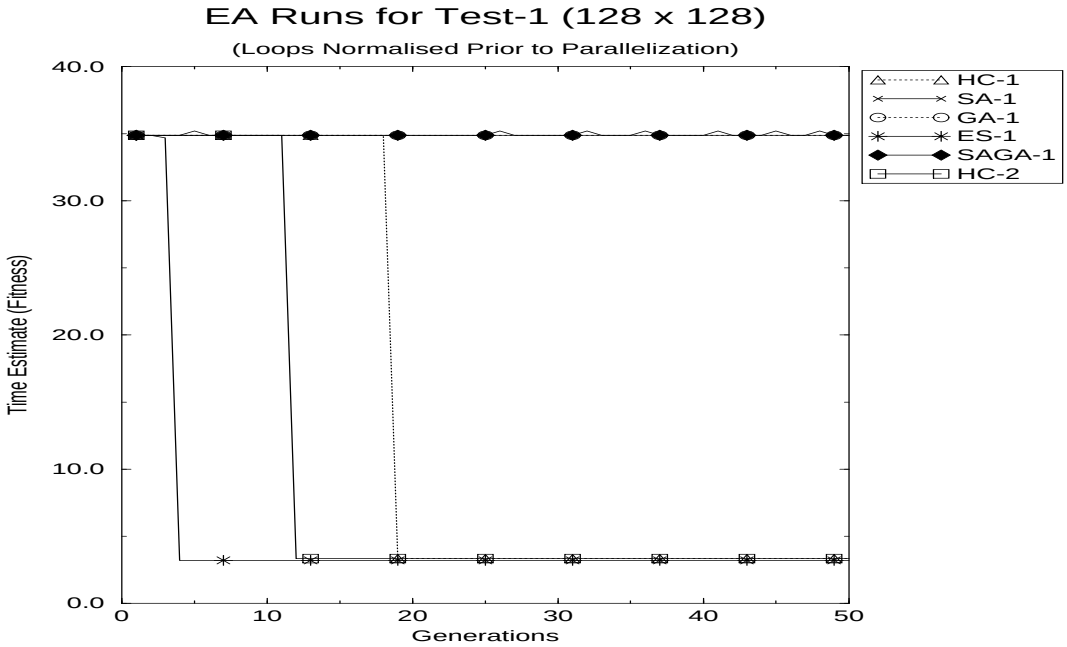


Figure 5.24: EA Runs on Test-1 (128 x 128) Results (All loops normalised prior to parallelization)

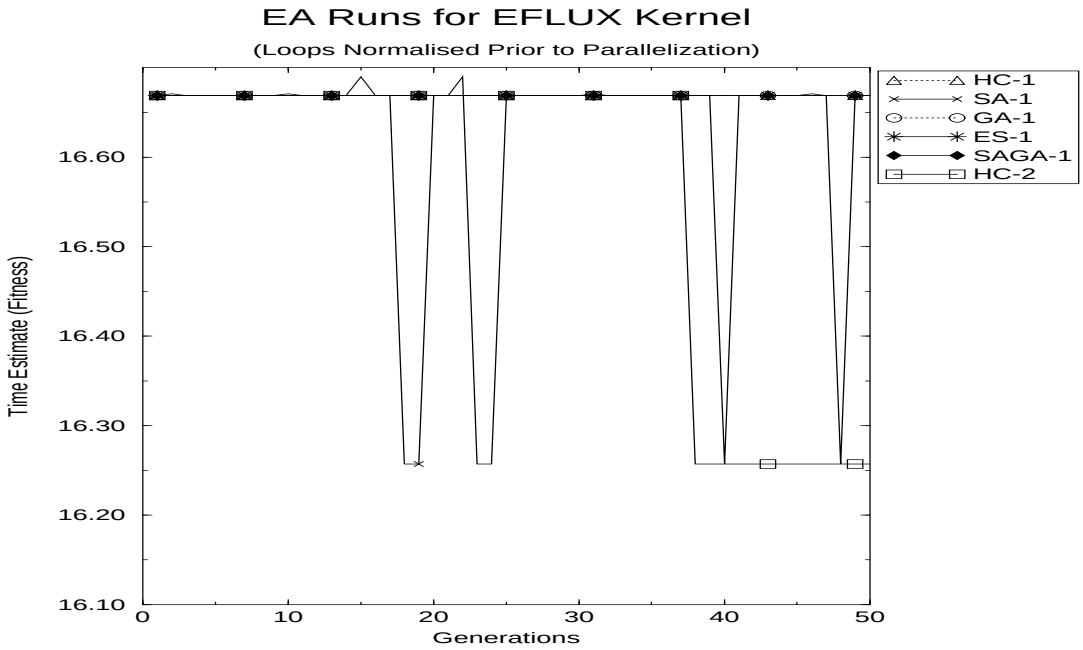


Figure 5.25: EA Runs on EFLUX Kernel (All loops normalised prior to parallelization)

target machine the amount of parallelism required is probably larger than the machine is capable of, given its memory size and speed of processors.

Whatever the accuracy of the target machine as presented by the performance estimator the essential point remains - that the EAs (most notably the ones using the GS representation) *were able to find an automatic parallelization strategy that would not have been obvious to a human programmer.* Given a larger target machine and a more accurate performance estimator - *EAs would still be capable of finding ways of optimising code that a programmer using an interactive tool or a traditional optimising compiler - may have missed.* Given the uneven landscape of the solution-space (as indicated in earlier results) this offers potentially major improvements in the quality of optimised code generated - and significant speed-ups over code produced using current techniques.

5.11 Accuracy of Performance Estimator

Performance estimation must be accurate in an any realistic automatic parallelization tool if it is to guide the restructuring process towards optimal (or near-optimal) parallelized versions of the sequential code. However, for distributed memory MIMD machines (like the CS-1) such information is extremely difficult to encapsulate at compile-time, even with the aid of information gathered from a profiled run (as is the case with Revolver). Indeed, development of a retargetable, accurate performance estimator is itself the subject of ongoing separate research projects (see [44, 8, 75], and sections 2.7 and 2.8).

The complexity of the problem is indicated by the number of variable factors that have to be taken into account - the code-generation algorithm, the model of parallelism being exploited (or imposed), optimisations performed by the native compiler, as well as the features of the target architectures' operating system, interconnection topology and technology, memory hierarchy, speed of processors, communication bandwidths and latencies, message-routing strategies, the input data to be worked on, etc.

In short, the problem is an important and complex one - but one in which advances are being made. Furthermore, it has at no stage formed a major component of this

Actual and Estimated Performance Figures for ADI							
ACTUAL EXECUTION				PERFORMANCE ESTIMATION			
NProcs	Actual MAX	Actual MIN	Actual Average	Workload	Comms	Process Start-up	ESTIMATE
1	9.70	8.44	8.677	0.757	150.933	0.792	152.483
2	37.52	35.89	36.224	0.580	150.946	0.792	152.319
4	41.05	38.71	39.787	0.492	150.968	0.792	152.486
8	47.10	45.98	46.120	0.448	151.012	0.792	152.252
12	48.96	47.62	48.352	0.409	152.235	0.792	153.437

Table 5.19: Actual and Estimated Performance Figures for ADI

Actual and Estimated Performance Figures for LFK-18							
ACTUAL EXECUTION				PERFORMANCE ESTIMATION			
NProcs	Actual MAX	Actual MIN	Actual Average	Workload	Comms	Process Start-up	ESTIMATE
1	9.75	9.42	9.522	31.751	13.371	0.792	45.914
2	13.14	12.79	13.003	15.947	13.439	0.792	30.179
4	16.53	15.27	15.538	8.013	13.515	0.792	22.321
8	21.19	19.32	20.333	4.047	13.667	0.792	18.506
12	24.06	22.20	22.959	2.768	14.035	0.792	17.596

Table 5.20: Actual and Estimated Performance Figures for LFK-18

research, the main aim of which has been to investigate the use of evolutionary algorithms for automatic parallelization tools. The further development of Revolver will inevitably require the improved sophistication of the performance estimator currently used, but this was only ever worthwhile once the basic premise of using EAs had been proved worthwhile (as is the case). Bearing these thoughts in mind then, these figures, which measure the accuracy of the performance estimator, are presented to indicate the effectiveness of the main techniques employed (namely; accumulation of benchmarked communication costs and operations-costs, utilising information gathered during an initial profiling run).

Actual and Estimated Performance Figures for TEST-1							
ACTUAL EXECUTION				PERFORMANCE ESTIMATION			
NProcs	Actual MAX	Actual MIN	Actual Average	Workload	Comms	Process Start-up	ESTIMATE
1	11.67	10.29	10.664	224.562	0.478	0.792	225.833
2	12.75	11.87	12.211	115.736	0.556	0.792	117.085
4	13.51	13.07	13.201	62.662	0.709	0.792	64.164
8	16.76	15.40	15.767	38.801	1.015	0.792	40.609
12	20.64	18.96	19.462	32.764	1.334	0.792	34.891

Table 5.21: Actual and Estimated Performance Figures for TEST-1

It can seen that that the estimates differ significantly from the actual recorded execution times (which themselves varied noticeably). All processes are mapped statically, spawn at start-up time and execute either to completion or until some blocking-communication statement has been reached. Hence process start-up time is a measured constant value, 0.792. All actual-time values are 'wall-times' recorded using the UNIX *time* command. Communications cost is clearly related to the number of array access references and size of the access regions in a loop, and to the number of loops parallelized. In the ADI program 5 out of 12 loops were parallelized, in LFK-18 3 out of 6, and in TEST-1 3 out of 7 loops were parallelized. Where number of loops parallelized N is greater than 1 and NProcs is 1, this means that N loops were parallelized and that all slave processes were placed along with the master process on processor number 0 (see section 4.10.6) for model of parallelism details. Some degree of accuracy was more noticeable when estimates were made of code generated for 12 processors as was intended for the target architecture all along. The increase in execution time despite the increase in NProcs is again explained by the generated code not being optimal or optimised. Hence, the prohibitive costs of communication which the EAs attempt to avoid.

5.12 Summary

In this chapter we presented the raw results of the experiments. The 6 evolutionary algorithms (SA-1, HC-1, ES-1, HC-2, GA-1, and SAGA-1) are described in detail as are their parameter settings. An extensive comparison was made on the performance each algorithm with graphs indicating typical runs of the EAs across a test suite of 5 benchmark Fortran-77 programs. The importance of the random number generator is assessed and the performance of the EAs subject to statistical (t-Test) analysis. Representations are compared, solo and population-based EAs are compared, performance of decoders is extensively compared, and the effectiveness of 6 genetic operators is assessed. An important result relating to the automatic parallelization strategy employed is reported and we finish by indicating the accuracy of the static performance estimation tool.

6 Discussion

6.1 Introduction

In this chapter we summarise the results presented in chapter 5, attempt to explain some of the reasons for the results observed and discuss some of the implications of the findings. A number of critical general observations are made on the work and its findings. Problems caused by epistasis, and the size and terrain of the fitness landscape and how they might be overcome are addressed. Future research directions of evolutionary parallelization are then indicated with several potentially fruitful ideas noted. Future work specific to the REVOLVER tool is then outlined. The chapter concludes with a brief summary.

6.2 Summary of Results

The results found in the experiments of Chapter 5 can be summarised as follows:

1. The EAs found a strategy for automatic parallelization which would not have been obvious to a human programmer. The programmer would have initially (probably) tried to apply exhaustive loop-fusion (to get large granularity) before parallelizing code - this possibility *was* available to the EAs used in REVOLVER (through the Loop-Fusion transformation) yet, based on feedback provided by the performance estimator, it realised that a strategy to produce more effective code was to keep loops sequential (by applying Loop-Reversal and hence preventing the code-generator from parallelizing the loop) and therefore avoid incurring the prohibitive communications overheads outlined in section 4.10.3. This strategy would not have been immediately obvious to a human programmer. The fact that the performance estimator is not particularly accurate does not detract from this result - if it was replaced in the future by a more accurate performance estimator then

the EAs would still be capable of finding parallelization strategies that are not obvious to a human programmer.

2. The use of the direct Gene-Statement representation significantly out-performed the use of the indirect Gene-Transformation representation. Any EA using the gene-statement representation consistently performed significantly better than one using the gene-transformation representation. By 'better' we mean in terms of quality of solution (code with shorted estimated times) and usually also in terms of rapidity in finding a solution - many of the EAs using the GS representation usually found good quality solutions in under 10 generations (i.e. usually much quicker than their GS counter-parts).

3. The `REPAIR` decoding strategy for use with the GT representation out-performed the `DELETE-AND-CONTINUE` and `DELETE-AND-STOP` strategies.

4. Results on the effectiveness of genetic operators were generally inconclusive.

5. Results show that population-based EAs usually out-performed solo-based EAs. Despite this, the hill-climbing algorithm using the Gene-Statement representation (HC-2) achieved a number of respectable results in short spaces of time.

6. The EA using adaptive operator probabilities (SAGA-1) was seen to perform slightly worse than the standard fixed-probability genetic algorithm (GA-1). However, I think the use of adaptive operator probabilities is worth persisting with in evolutionary parallelization for three reasons :

   (a) Theoretically, it makes sense to allow different operators to dominate an EA at different stages of the run (e.g. allow the mutations to increase over time while the crossovers decrease because an

early number of crossovers allow the population to sample a wide area of the search space and as the run progresses an increase in mutations enable more 'fine-tuning' of solutions to take place). There is also good deal of experimental evidence to support this approach.

(b) It is quite probable that 50 generations is not long enough for the subtle effects of changing operator probabilities to have much effect. (This limit was imposed by implementation problems described in section 4.12.1). Even if an ES-2 algorithm had been implemented (using Gene-Statement representation and transformation/mutation operators with adaptive probabilities) it is unlikely that significantly better results than that achieved with ES-1 would have been obtained.

(c) In a more realistic automatic parallelization environment the search space will be much larger (more transformations will be available) and will continue for longer (more generations). This environment is more suited to the gradual change of operator probabilities over time - and would be particularly suited to adapting probabilities of the transformation/mutation operators used with the Gene-Statement representation. This is one significantly promising direction of future research.

7. The effects of seeding the random number generator with different values were noted and the possibility of producing 'one-off' results (sometimes better, sometimes worse) was acknowledged. Notably a population-based algorithm can capitalise on these 'sometimes better' breakthroughs.

8. The impact of testing EAs using the GT representation and initialising the population with longer initial strings (i.e longer initial sequences of transformations) was not investigated. The default value is of length

5. The value of making this initial value bigger or smaller is unclear.

Most notably of all perhaps, the wide spread of the fitness values recorded indicates the irregularity of the solution space and how important the ordering of transformations is when optimising code.

6.3 Critical Evaluation of Results

In view of the results and subsequent analysis, we are now in a position to make a number of criticisms and observations with respect to the results presented in this research.

The first criticism of the results might be that the poor accuracy of the performance estimator means the results are not realistic. This is not the case. The performance estimator could easily be replaced by a more accurate estimator - the EAs used would be unchanged and they would still be capable of finding original parallelization strategies. They would simply adapt to whatever information the performance estimator was feeding-back to them.

This leads to the second criticism - that the use of EAs to parallelize sequential code is dependent on the development of static accurate performance estimation techniques, which is a notoriously difficult enterprise to undertake. This is true. However a number of existing parallelizing tools use feedback provided by an estimation tool (e.g. Parafrase at Rice University, and VFCS at the University of Vienna) and although to achieve best results these tools are used in 'interactive mode' further research is continuing in this direction and it is not unreasonable to expect that further improvements will be made.

A third criticism of the results is that because no comparison is made with other parallelizing compilers it is not possible to tell how good the EAs really were. Comparison with another parallelizing compiler(s) was one of the original objectives of the research, however due to implementation problems encountered with code-generation for the CS-1 (see section 4.10.3) and memory management problems with the Sage^{++} libraries (4.12.1) such a comparison would have been clearly unfair to the EAs. The difficulties encountered are however only implementation difficulties and given an im-

plementation free from memory and code-generator problems the EAs would still have the ability to find an effective parallelization strategy and perform in the ways we have seen in the results presented.

A further criticism might be that the solution space for parallelizing a program will be enormous, too large to expect an EA to find reasonable results in an acceptable period of time. As noted in chapter 1 - a minimal lower-bound can be put on the quality of the solution produced by the evolutionary parallelizing compiler by hybridizing the EA. Like any search technique, its performance can be improved by the use of heuristic information and the incorporation of existing heuristics into the evolutionary parallelizing compiler can guarantee that the solution found will be at least as good as that found by existing techniques.

Other observations and criticisms are as follows;

- The Gene-Transformation (GT) under-specifies the automatic parallelization problem - the net effect of this is that more parameters are needed such as the decoders and the need for each instance of a transformation to specify which loop it is to be applied to.

- Crossover/Mutation operators used with the GT representation are not 'meaningful'. The operators are essentially 'blind' and perform no function which is semantically meaningful in the context of restructuring code. This problem *may* be addressed by creating *heuristic-operators* such as VLX-3 which was seen to perform slightly better than the other two crossover operators and suggests that heuristic operators may have a big part to play in the future of evolutionary parallelization.

- It must be remembered however, that blind-search is often cited as one of the most important features of evolutionary algorithms and can sometimes produce new and unexpectedly good results.

- Epistasis. The theoretical problem of epistasis is that the strong-interaction between the genes can deceive the GA into converging

around local-optima. In practice, it also makes it difficult to define meaningful genetic operators and hence to steer the population towards the global (or near-global) optima areas that really interest us.

The epistasis problem most clearly manifests itself in the design of the mutation operators for the gene-transformation representation. Mutating one gene (say half way along a chromosome of transformations) effectively changes the impact of all the following transformations - they are not working on the semantically same code they were before the mutation. The mutation operation changes the gene it is applied to - it also has a "knock on" effect on the genes which follow. A similar criticism can also be made of the crossover operation. Crossing over two substrings is hard to make 'meaningful' in the context of applying a sequence of optimisations/transformations.

This is a potentially serious theoretical problem, although it must be stated that at no time in the experiments were population-based EAs observed to converge and become trapped prematurely. In a larger system, over a longer period of generations perhaps this may prove to be a significant problem.

- Use of a messy-GA representation may be one way to overcome the epistasis problem although this approach also implicitly assumes that EAs will be of fixed length strings.

- Notably, the Gene-Statement representation does not suffer from the epistasis problem and this may be one reason to account for its superior performance over the Gene-Transformation representation.

- The 'gene-transformation' representation by attempting to apply multiple transformations at a time has the effect of 'propelling' solution strings across the fitness landscape potentially searching a much wider area than is covered by the evolution strategy using the 'gene-statement' representation. There may be a problem in that a solution string may actually be propelled 'over' (and hence miss) a significant optima in the search-space. As such, a GA using the gene-transformation' representation may perform better in conjunction with

a localised hill-climbing search. It should be noted however that this phenomenon may also happen when using the GS representation since it is probable that several mutations will be applied to one program in the same generation (thereby producing a similar 'propelling' effect). However, since the mutations/transformations are applied probabilistically, (rather than being 'compulsory' as with the GT representation), it follows that the propulsions will usually contain less momentum and be over shorter distances. Hence a finer degree of sampling of the solution space will be obtained. Interestingly then, this effect may also be tackled in the Gene-Statement representation by use of a localised hill-climber. Perhaps we might say then that a *gentle* propelling effect is probably beneficial to the overall performance of the EA.

- Other representations of the automatic parallelization problem certainly do exist and efforts should be made to discover and investigate them. In the traditions of lateral thinking, many problems can be solved simply by changing one's perception of the problem, in computing terms this translates as 'changing the problem representation'.

- Size of the Solution Space. The size of the solution space to be searched by the EA is determined by two things (i) the number of transformations available in the catalogue, and (ii) the number of opportunities to apply them (e,g, the number of compatible loops, etc). It may be that the space is infinitely large (is it always possible to apply one more transformation?). It should also be noted that the solution space contains cycles - it is possible to continually split and fuse the same loop(s) over and over again. These cycles may be direct split/fuse transformations, or may be longer and indirect involving sequences of several transformations.

- The terrain of the fitness landscape can be described in two ways. Firstly, it can be 'imagined' that given a program containing several

loops, application of one transformation to a point in the program may easily result in a non-linear change in performance of the program. This suggests that the solution space is discontinuous and irregular. If we now look at our results presented in chapter 5 we can see that the *variety* of solutions offered as being optimal by different EAs under different conditions supports this view. If the solution space were smooth and with a single optimal point then the EAs would find it every time. The fact that they very rarely agreed on a single solution implies the solution space is not like this at all. Furthermore, it follows that the solution space will be different for each program we try to parallelize. All we can describe here are general features.

It is also worth noting that sharp edges in the solution space could be made smoother and more near-optimal points would be uncovered with addition of more restructuring transformations. The irregular nature of the solution landscape highlights the fact that the order in which restructuring transformations are applied (particularly high-level transformations such as loop-transformations) can have a significant impact on the performance of the final generated code. This is the *phase ordering* problem found in much parallelizing compiler technology. The results show how using an evolutionary algorithm with the gene-statement representation can overcome this problem.

6.4 Future Research in Evolutionary Parallelization

The future of prospects of using evolutionary algorithms to parallelize sequential code are now bright. The enormous potential benefits of make attractive to substantial further research and investigation. The possibility of evolving new parallel algorithms with evolutionary parallelization (and also possibly with genetic programming) are particularly exciting.

It is clear however that the effectiveness of the EA approach in automatic parallelization is limited by the abilities of static performance estimation and it is in this

direction that more immediate work needs to be done.

Another potential benefit is the use of an evolutionary parallelizing compiler (EPC) as the front end of a knowledge acquisition tool. Knowledge gained through evolutionary techniques can be used to improve design of existing, traditional compilers and restructurers.

The largest differences in performance in the results presented in chapter 5 were gained by using a different representation rather than a different algorithm. It follows therefore that research into finding other representations may be worthwhile. A few possibly useful ideas and considerations for future research are presented next.

6.4.1 Selection Strategies and Operators

Instances of ESs are usually described by three parameters as detailed in section 3.2.6. Most of the experiments conducted with REVOLVER were made with an ES(5,5). The point here is that only a comma-reinsertion strategy was used, due to time and system constraints I was not able to implement a plus-strategy and make a comparison. This is unfortunate since the choice of reinsertion-strategy is known to have a potentially significant impact on the performance of a ES.

The GA equivalent of the ES selection-strategy is the selection operator. Traditionally, this has been Holland's roulette-wheel technique. In REVOLVER I have only used binary tournament selection with EAs **GA-1**, and **SAGA-1**. (No explicit selection operator is needed for **HC-1** and **SA-1**.) The **GA-1** and **SAGA-1** EAs both use the gene-transformation representation. This is significant since it means that although the population of transformations change from iteration to iteration the actual code does not. With each iteration the new population of transformations is applied to the original sequential program - in effect, it is returned to the starting point in the solution space each time.

The alternative is to retain restructured sequential code across generations. This will have the effect of sending population members out, deeper into the search-space without the need for them to return to the starting point with each iteration. This would have a profound effect on the performance of the GAs used, the search space

would immediately become much larger and take correspondingly longer to search, but it would allow the possibility of finding better quality solutions. The computational price for such a search currently may be too high, but in the future this may change.

6.4.2 Relative vs. Absolute Addressing

An alternative problem representation is to use Relative rather than Absolute addressing. It will be recalled that loops in the GT representation are referred to using *absolute addressing*, e.g. chromosomal representation consists of 4 fields

TRANSFORMATION-NO LOOP-NO PARAMS-1 PARAMS-2

essentially each gene encodes WHAT transformation is to be applied, and WHERE it is to be applied (i.e. what loop number) in the program. The alternative is is to use an encoding of *relative addressing*. This would require the introduction of extra genes to instruct the 'pointer' to move within the program (e.g. NEXT-LOOP, PREVIOUS-LOOP, STEP-INTO-LOOP-NEST, STEP-OUT-OF-LOOP-NEST, GO-FIRST-LOOP, GO-LAST-LOOP, etc). A sequence of genes in a chromosome now may look like

INSTRUCTION	PARAMS-1 PARAMS-2
NEXT-LOOP	- -
LOOP-FUSION	- -
PREVIOUS-LOOP	- -
PREVIOUS-LOOP	- -
LOOP-INTERCHANGE	- -
GO-LAST-LOOP	- -
....	... ...

The interesting point about this representation is that it defines another completely different approach to automatic parallelization and as such is one that may be worth investigation in the future.

6.4.3 Messy-GA Representation

Another alternative representation as noted previously, is to use the 'messy-GA' approach. Due to the problem of epistasis (section 3.3) it may well prove to be the case

that encoding the problem of automatic parallelization into the form of a messy-GA as described by Deb and Goldberg [54, 55] may produce improvements in results. However, although this method overcomes the epistasis problem, it is still essentially a blind search technique unless a way could be found to introduce heuristics, and still implicitly assumes chromosomes of a fixed length.

6.4.4 'Additional Gene' Representations

A number of further features of restructuring can be incorporated into the representation simply by adding an 'additional' gene to the chromosome. Immediate possibilities are

1. 'Model-of-Parallelism' Gene. Each program being restructured would carry an extra gene to instruct the code-generator to what 'model' of parallelism code should be generated for. Not all problems map best onto SPMD parallelism, some map best into pipelines, or master/slave models, depending on software algorithm and hardware architecture. The extra gene may take one of three possible values for example : SPMD, Pipelined, Master/Slave, and would be subject to a mutation operator which would change the value of the gene in some fashion.

2. 'Number of Processors' Gene. Specifies how many processors the loop is to be parallelized for. Ideally the optimal value could be computed during code-generation but if not (dynamic code, perhaps) then this might become necessary.

6.4.5 Upper Bound on Chromosome Length / NULL Transformation

This again applies to the Gene-Transformation representation. The idea here is that in a larger, more realistic system the length of chromosomes may become exceptionally (hundreds) long. It may then become necessary to investigate the benefits/disadvantages of imposing some upper-bound on the length which chromosomes may become. This would then effectively mean that we are back to working with fixed length chromosomes which suffer from epistasis. However, development of more heuristic crossover

operators (along the lines of VLX-3) may make this representation more attractive in the future.

The other alternative for working with fixed-length chromosomes is to instead of trying to parallelize the code optimally (which may possibly take some time), the user could specify that the EA try to find the best N transformations for restructuring the program (where N is the length of the chromosomes). This would limit the search-space but may shorten the time to find a reasonable solution (if one is wanted quickly). It may however prove a burden to 'force' the GA to apply N transformations each time (where only a few may be necessary) so it should be possible to pad the rest of a chromosome with NULL transformations, where a chromosome contains M transformations and M $<$ N. This idea may require further investigation.

From the points noted above it is clear that further investigation into the selection methods used by EAs using both representations is a significant area for further research.

6.4.6 Combined Code-Restructuring / Data-Layout Transformations

Intriguingly, the use of the evolutionary techniques can be extended to optimise data-layout of parallelized programs. This would involve using automated data-layout transformations such as described in section 2.4.6 and could be used in a stand-alone data-layout distribution tool. More importantly this offers the enticing prospect of combining code-restructuring and data-layout transformations 'in tandem'. Data-layout transformations would simply form mutation-operators in exactly the way as loop-transformations do when using the gene-statement representation (as described in section 4.4.2).

6.5 Future of REVOLVER

The work to improve the REVOLVER system is ongoing, immediate improvements include:

- In order to overcome the memory-leak problems described in section 4.12.1 it will be necessary to reimplement some of the underlying com-

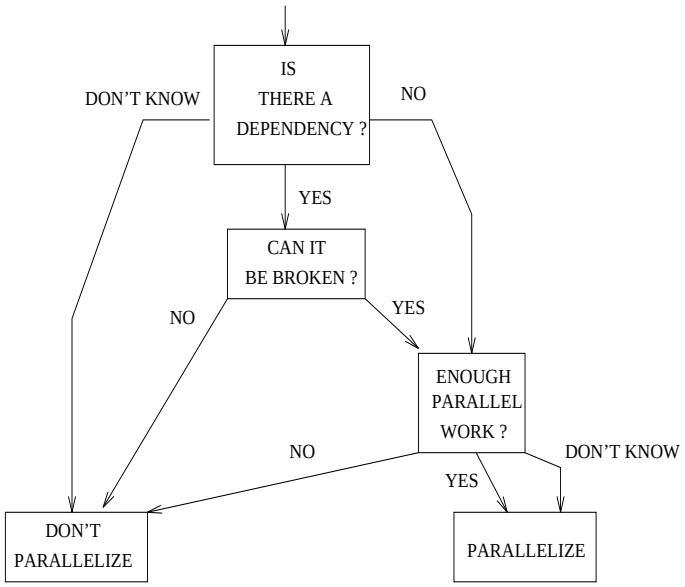


Figure 6.1: Application of Dependency Breaking Transformations

piler structures. This may be along the lines of restructuring compiler libraries using string objects described by Crooks' [34]. What is particularly attractive about this approach is the flexibility (functionality is easily added and changed) and in particular reliability - if your restructuring compiler contains a bug or error, then an EA is sure to find it.

- Incorporate improved dependency analysis techniques. In particular this involves replacing the `Omega` libraries with their more recent incarnation, the `Petit` libraries [71, 72] but also includes development of interprocedural analysis functions.

- Target a shared-memory multiprocessor machine, and investigate the potential for auto-parallelizing code for networks of workstations (NOWs) and in particular Message-Passing Interface (MPI) [96, 69, 105].

- Add more transformations to the current transformation catalogue. In particular the current catalogue (apart from only containing four transformations) contains only loop-transformations. A greater vari-

ety of transformations (such as statement/block reordering, dataflow optimisations) is required to realise the full power of the loop transformations. A large number of other classes of transformations need to be added (see section 2.4), these classes include:

  - Dataflow Optimisations / Partial Evaluation Optimisations
  - Further Loop Transformations
    * Loop Reordering Transformations
    * Loop Restructuring Transformations
    * Loop Replacement Transformations
  - Memory Access Optimisations
  - Procedure/Function Call Transformations
  - Data Layout and Decomposition Transformations
  - INSPECTOR/EXECUTOR Run-time Optimisations
  - Dependency-breaking (dependency 'zapping') transformations (see Fig. 6.1.

- Make improvements to the performance estimator. This currently relies heavily on a number of assumptions (such as a master/slave model of parallelism, and a constant number of slaves being created) which are imposing otherwise unnecessary restrictions on the CS-1 code-generator.

- Implement an expression simplifier: the application of many transformations causes many expressions to be changed substantially from their original form. The expressions often become increasingly more complex (such as below)

```
        do  k = 1,5,1
           x(3 + (j - 1),0 + (k - 1)) = (x(3 + (j - 1),0 + (k - 1))
+- a(3 + (j - 1) + 1,0 + (k - 1)) * x(3 + (j - 1) + 1,0 + (k - 1)))
+ / b(3 + (j - 1),0 + (k - 1))
        enddo
```

with the repeated application of further transformations. This has the effects of (i) increasing the overheads in performance prediction and dependency analysis, (ii) making the output code larger and slower to execute, and (iii) outputting code which is incomprehensible to humans.

6.6 Summary

This chapter began with summarising the results of Chapter 5 and noting a number of significant findings. In particular the generally superior performance of EAs using the gene-statement representation rather than gene-transformation and discussed reasons for this. We then described a number of critical observations about the EAs (epistasis, 'propelling' solution strings), and indicated areas in evolutionary parallelization ripe for future research. The chapter finished with future work specific to the Revolver tool being outlined.

7 Conclusions

7.1 Introduction

The hypothesis presented in this dissertation is that in order to achieve *optimal* automatic parallelization (i.e. minimisation of execution time) of sequential code the best results are likely to be achieved using some form of evolutionary optimisation, rather than relying on purely analytical techniques.

As is the case with all optimisation problems, the use of purely analytic techniques (e.g. linear programming) is to be preferred above the use of stochastic techniques and should be used wherever possible, especially if they can compute optimal (or near-optimal) parallelizations in 'reasonable' time. However, given the continuing trends towards (i) the increasing complexity of computer code, (ii) the increasing size of applications people wish to auto-parallelize, (iii) the widening diversity of target architectures available, which results in (iv) the increasing number of compiler optimisations possible - it would seem that the use of purely analytical techniques can only be applied in a limited number of cases such as where the code is static, relatively small or simply written.

This thesis does not argue that a purely evolutionary approach should replace analytic methods. The ideal proposed is a marriage of the two resulting in a hybrid technique which allows us to enjoy the benefits of both approaches. The use of evolutionary techniques does not exclude the use of the analytical techniques. On the contrary, many (if not all) of these current techniques form a valuable body of heuristic knowledge which can be incorporated into an evolutionary algorithm.

This thesis also does not argue that evolutionary algorithms (EAs) are suitable for all machine translation tasks. The quality of code produced is an important feature of *any* compiler and most of them do not necessitate the use of EAs. It is argued that automatic parallelization is a unique case given that (i) the quality of code produced can have a dramatic impact of the performance of code produced (inefficient code runs inefficiently - even on a fast machine), and that (ii) the precise reason most people want

to parallelize their sequential code is so as to take advantage of the potential speed-ups that only parallel computing can offer them.

It must also be stated that the benefits of this hybrid approach come at a cost of increased computational time and effort - which intriguingly may be overcome by parallelizing the EA itself. The result however is still essentially a trade-off between quality of the parallelized code offset against the expenditure of the computing effort required to produce it. This thesis takes the view that the long-term benefits of producing good quality, high performance, parallelized code are worth the one-off computational effort required.

7.2 Contributions

In this wide-ranging work we have touched on four main areas of research, namely (i) all aspects of automatic parallelization, including optimisations and transformations, analysis, and code-generation (ii) dependency analysis (iii) static performance estimation, and (iv) evolutionary computation. At times this work has felt almost inter-disciplinary. The original contributions of this thesis are:

- Two representations of the automatic parallelization problem that are suitable for manipulation by an evolutionary algorithm. This is fundamentally important. Evaluating 6 different evolutionary algorithms on their own would not have been interesting or resulted in anything useful. They were only useful in the context of evaluating the representations they were working on. (Perhaps this thesis would have been better entitled 'Evolutionary Representations for Automatic Parallelization'...).

- Detailed descriptions of the first implementation of several evolutionary algorithms (GA, ES, hill-climbers and simulated-annealer) to translate sequential program code into parallel form, based on an original idea as outlined and presented by the author in [135].

- Statistical evaluation of the effectiveness of 6 genetic operators developed as part of the research. This evaluation resulted in the calculation of confidence values which indicate which operators were beneficial to the EAs in the restructuring experiments performed. A particular contribution in this respect is the development of a new hybrid operator, VLX-3, which has proved promising within the domain of automatic parallelization.

- The development of 3 'decoding strategies' - techniques which instruct the EA what to do in the event of a scheduled transformation being inapplicable - namely `DELETE-AND-STOP`, `DELETE-AND-CONTINUE` and `REPAIR`. Likewise, statistical comparison of the decoders was made and the `REPAIR` strategy found to be the most effective in the experiments performed.

- Presentation of many of the practical implementation difficulties involved (profiling, performance estimation, code-generation, restructuring transformations) and descriptions, usually with algorithms, of how they were overcome.

- Consideration of problem representation issues, and discussion of a number of alternative representations and possible future research directions in evolutionary parallelization.

- Progress into the field of automatic parallelization is a particularly important new application of evolutionary algorithms since it opens up an entirely new line of attack into the 'grand challenge' problem of automatic parallelization.

We can now compare our results with the initial aims and objectives of this research (section 1.4.3).

From the outset the central objective was to determine whether an evolutionary approach could be used to effectively translate sequential Fortran programs into parallel form. The main value of this research I believe has been to show that this can clearly be

done. This research has in fact demonstrated 6 different evolutionary algorithms, using 10 different operators, restructuring and parallelizing a set of 5 benchmark Fortran-77 kernels for a message-passing architecture.

Another objective was the investigation into which ways can the forces of evolution be brought to bear on the problem of automatic parallelization - and determine which ways are the most effective. In essence it turned out (according to experimental results) that the fundamentally most important aspect of applying an EA to automatic parallelization was to find a good representation which could be manipulated by the EAs. The discovery of a heuristic operator, the setting of various parameters (population-size, operator probabilities, number of generations, etc) were all of secondary importance to the choice representation.

Another aim was to characterise the shape of the solution-space for typical automatic parallelization problems. This is important since it may help in the design of future EAs for the problem. As noted in the discussion on page 208 and based on the results presented in Chapter 5, given the variety of solutions which were found to be optimal by the EAs at various stages, running under different conditions, it is clear that the terrain of the solution-space is 'blocky', discontinuous and uneven with many sharp peaks and valleys. It is intuitively obvious that the solution space for each program to be parallelized will be different - some programs are easier to parallelize than others. Factors which influence the terrain of the solution space include the number of transformations available in the catalogue - and also the number of opportunities for them to be applied in the restructuring process.

Furthermore the ultimate quality of the parallel code produced (not just the restructured code) will also be dependent on the quality and algorithm of the code-generator (and hence the target architecture) and also the accuracy of the performance estimator.

Another aim was to experiment with new hybrid-operators and evaluate each for their effectiveness. This aim was achieved. Six genetic operators were devised for the Gene-Transformation representation and statistically evaluated for their effectiveness. (4 loop-transformations were also perceived as actually being 'mutation-operators' for use with the Gene-Statement representation - their functioning was clearly critical to

the EA so did not need to be evaluated for their effectiveness - see section 5.9). The results were largely inconclusive, although the heuristic operator VLX-3 performed encouragingly.

A further aim was to identify points in the evolutionary process where restructuring heuristics can be included to the best effect. This is really dependent on the choice of representation to be used. If working with the Gene-Transformation representation then the development of further heuristic operators incorporating current knowledge and restructuring techniques may provide a useful starting point. With the Gene-Statement representation it is less clear how heuristic information could be incorporated into the EA - perhaps upon recognition of a particular loop type (e.g. a summation) the loop could simply be parallelized in a 'known optimal way' rather than persisting with the evolutionary process, in that case.

The last (but by no means least) objective was to identify what conditions make it easy / difficult for the EA to parallelize sequential code. After all the experimentation performed it has become clear that because the evolutionary approach builds so much upon current knowledge, that essentially programs which are hard to parallelize using current techniques will still be hard to parallelize using evolutionary techniques - the difference is that the evolutionary approach introduces a degree of randomness into the process that will occasionally result in some improvement that purely analytical techniques would not have found (at the cost of additional computing time and resources).

A number of performance metrics were also specified in Chapter 1. The main performance metric was the 'quality' of the parallel code produced and how well the EA worked in comparison to traditional parallelizing techniques. Unfortunately this research has been unable to allow this comparison to be made. Their are two reasons for this - both implementation related. Firstly, the lack of a 'stride' parameter for sending/receiving data on the CS-1 target machine forced generated code to also pack and unpack array regions (see section4.10.3). This introduced a substantial overhead into any communications and made realistic comparison with any other parallelizing compiler impossible. Secondly, the memory management problems encountered with

using the Sage^{++} libraries (see section4.12.1) imposed an upper limit of 50 generations on the execution of the EAs. This is not as long as I would have liked on a problem of this kind, however, in many cases I believe the EAs found a good solution fairly quickly (10 to 20 generations) and that 50 was still enough for us to gauge the effectiveness of each EA.

Another performance metric was the 'robustness' of the EA produced, that is, the ability of the algorithm to consistently parallelize efficiently a large number and wide variety of Fortran programs - in particular, the size and type of the Fortran application should not prevent it being parallelized using evolutionary techniques (any more than for conventional techniques). The 5 Fortran-77 programs selected are typical (and in some places quite complex) pieces of F-77 code used in many scientific and engineering applications. The codes were chosen for their variety of sizes (35 (`M44`) to 90 (`EFLUX` lines long), the combinations of loops they had (perfect/imperfectly nested 2/3 deep) and their array access patterns (2 and 3 dimensional arrays). The EAs were able to parallelize all these quite effectively and as such have shown themselves to essentially be as robust as the techniques they employ.

The next performance metric was the 'efficiency' of the EA - that is the time to produce quality code should not be prohibitive. The time taken to run 50 generations varied between 45 and 90 mins depending on the algorithm and the size of the population. I should point out that the current implementation of Revolver is known to have plenty of opportunities for performance improvements having been written with development and experimentation as priority and speed of processing and efficiency secondary.

The final performance metric was that the EA should not get trapped in local optima when searching the solution-space. This problem is really related to the type of algorithm used. It occurred several times with the HC-1 hill climbing algorithm which, as a simple steepest ascent hill-climber was perhaps to be expected. The population-based equivalent would be to see if the population converged prematurely and despite the possible effects of epistasis this was not seen in practice. Whenever the population seemed to converge it seemed able to diversify itself again quite easily.

A Appendix

A.1 Benchmark Timings for Meiko CS-1

A.1.1 Introduction

This appendix contains performance data for the 16-node Meiko CS-1 computing surface used as the target architecture for the REVOLVER parallelizing compiler. The data was accumulated using the techniques described in section 2.7.2 and is presented here as additional information.

A.1.2 Arithmetic, Relational and Logical Operation Benchmarks

Tables A.1, A.2, A.3 and A.4 are used as look-up tables to determine the time-cost value (in $\mu secs$*) for various combinations of operations on different types of data on the CS-1.

Each binary operator is deemed to have a datum of a particular type on each side of its appearance in the program. The operator, on its left-hand side (noted as *lhs* in the tables) will have datum of one type, and a datum on its right hand side (noted as *rhs* in the tables) also of a particular type.

Data may be of several types : integer, real (4-byte reals only are considered here), logical, or sometimes character strings. Data of these four types may also be either 'constant', that is - 'hard-coded' into the program code, or else a 'variable' - where the value will have to be retrieved from memory before use. Combinations of these types make up the data types referred to in the tables (namely: `int const`, `int var`, `real const`, `real var`, `logical const`, `logical var`).

To determine the time-cost of an operation:

1. Determine the type of the datum on the left-hand side *lhs* of the operator.

*That is *micro*-seconds, 1 μsecond = 10^{-6} seconds.

2. Determine the type of the datum on the right-hand side *rhs* of the operator.

3. Find the correct table for the operator you have (arithmetic, relational, or logical). Now, look along the left-column (marked *lhs-Type*) and find the type that matches the *lhs* of your operator. Do the same for the *rhs*. Now find the correct operator from the column headings. The point where the correct row and column meet, gives the mean average time cost for the operation you are examining.

Example 19. Given a statement, as below, where we know variable `i1` to be declared of type `integer`:

```
i1 = 25
```

We determine that the *lhs* of the assignment operator is an `int var`, and the *rhs* is an `int const`. We now go to table A.1 where we find the intersection of *lhs* `int var`, *rhs* `int const`, and the asignment operator `=` returns a time value of 23.272704 μ seconds. Hence, we can say, on average, this assignment statement will take 23.272704 μ seconds to execute.

Expression Evaluation Order

Note also that the *order* in which expressions are evaluated has an important impact on the time-cost estimate for the expression (see section 2.7.2 for details).

A.1.3 Communications Benchmarks

One of the most expensive overheads of any parallel computation is the cost of communication between processes. Processes communicate to send data to and from each other that they need in order to complete their computations. The communications protocol used in REVOLVER is the blocking-synchronous protocol, as described in section 4.10.3). It is essential therefore that the time-cost of any communications made must be taken into account in estimating the performance of a parallel program.

Arithmetic Operations (Timings in $\mu secs$ - i.e. $1\mu = 10^{-6}$ secs)

lhs Type	=	+	-	×	/	**	MOD	*rhs Type*
int var	23.536896	0.5734400	0.3581440	0.0666112	0.2169088	1.0084608	0.2384640	int var
int var	23.272704	0.5281280	0.3276800	0.0668672	0.2414080	1.7355520	0.1884416	int const
int var	23.994368	0.1960960	0.6492160	0.0647168	0.0789504	14.865663	-	real var
int var	23.285760	0.1968640	0.1425920	0.0646656	0.1155840	14.848614	-	real const
int const	-	0.1103360	0.0224000	0.0562688	0.2234880	1.4411008	0.2242560	int var
int const	-	0.1039360	0.0642559	0.0028159	0.0417279	0.0179199	0.0098816	int const
int const	-	0.3714560	0.0615168	0.0578816	0.0765696	14.805350	-	real var
int const	-	0.0161280	0.0515072	0.0016896	0.0166400	14.815155	-	real const
real var	24.098040	0.7091200	0.6364160	0.0792320	0.1496832	8.2997247	-	int var
real var	23.984896	0.3630080	0.6113280	0.0538880	0.0737792	1.8845952	-	int const
real var	24.198656	0.3289600	0.1060096	0.0490240	0.0581376	14.808217	-	real var
real var	23.805696	0.3550720	0.6161920	0.0541952	0.0744192	14.797107	0.4591616	real const
real const	-	0.5409280	0.0801280	0.0433664	0.0041728	5.7233151	-	int var
real const	-	0.1510400	0.0018688	0.0016896	0.0514816	0.0033024	-	int const
real const	-	0.3322880	0.0150272	0.1025024	0.0868096	14.738687	0.3936256	real var
real const	-	0.0092160	2.3823359	0.0010240	0.0017920	14.740991	0.4310272	real const
logical var	23.303680	-	-	-	-	-	-	logical var
logical var	22.912640	-	-	-	-	-	-	logical const

Table A.1: Arithmetic Operation Timings for Meiko CS-1

Logical Operations (Timings in $\mu secs$ - i.e. $1\mu = 10^{-6}$ secs)

lhs Type	.and.	.or.	.eqv.	.neqv.	*rhs Type*
`logical var`	0.2426496	0.2433024	0.2334400	0.2433280	`logical var`
`logical var`	0.2399168	0.2363200	0.2354112	0.2349824	`logical const`
`logical const`	0.2399168	0.2363200	0.2354112	0.2349824	`logical var`
`logical const`	0.2356928	0.2276608	0.2354112	0.2349824	`logical const`

Table A.2: Logical Operation Timings for Meiko CS-1

Unary Operations (Timings in $\mu secs$ - i.e. $1\mu = 10^{-6}$ secs)

UNARY −	UNARY +	.not.	*rhs Type*
0.0204799	0.2353689	-	`int var`
-	-	-	`int const`
0.1023999	1.1592711	-	`real var`
-	-	-	`real const`
-	-	0.2336768	`logical var`
-	-	0.2335500	`logical const`

Table A.3: Unary Operation Timings for Meiko CS-1

Relational Operations (Timings in $\mu secs$ - i.e. $1\mu = 10^{-6}$ secs)

lhs Type	.eq.	.ne.	.lt.	.gt.	.le.	.ge.	*rhs Type*
`int var`	0.2397183	0.2415103	0.2383103	0.2466047	0.2456831	0.2427391	`int var`
`int var`	0.2360319	0.2377471	0.2341631	0.2375679	0.2336511	0.2439935	`int const`
`int var`	0.2482431	0.2450175	0.2451967	0.2456063	0.2468351	0.2499327	`real var`
`int var`	0.2425343	0.2449919	0.2500351	0.2409471	0.2441215	0.2456831	`real const`
`int const`	0.2322943	0.2339327	0.2323711	0.2343935	0.2360319	0.2334975	`int var`
`int const`	0.2309375	0.2328831	0.2311423	0.2265599	0.2328575	0.2256383	`int const`
`int const`	0.2466559	0.2434303	0.2451711	0.2477311	0.2445823	0.2439935	`real var`
`int const`	0.2328319	0.2266639	0.2328575	0.2571519	0.2264831	0.2308095	`real const`
`real var`	0.2494463	0.2489855	0.2442527	0.2428415	0.2526975	0.2461439	`int var`
`real var`	0.2467327	0.2435327	0.2450431	0.2476287	0.2443007	0.2438399	`int const`
`real var`	0.2448127	0.2449407	0.2419199	0.2455295	0.2419199	0.2403327	`real var`
`real var`	0.2455807	0.2421503	0.2468863	0.2452479	0.2487295	0.2445823	`real const`
`real const`	0.2377727	0.2490623	0.2417663	0.2427135	0.2525695	0.2459903	`int var`
`real const`	0.2266367	0.2254591	0.2266623	0.2310911	0.2254591	0.2330367	`int const`
`real const`	0.2453759	0.2419711	0.2469119	0.2448895	0.2485759	0.2450431	`real var`
`real const`	0.2255871	0.2308863	0.2252799	0.2328319	0.2310399	0.2267903	`real const`

Table A.4: Relational Operation Timings for Meiko CS-1

Implementing a general performance estimator for parallel programs (on *any* architecture) is a non-trivial problem. Our performance estimator then, takes advantage of the form of parallelism we exploit in REVOLVER . Indeed, the restricted form of master/slave parallelism we use (section 4.10.5) was partly chosen because the regular communications patterns allow accurate estimations of performance to be made. Our approach to the estimation of communications costs then, is based on the following two observations on the parallel code we generate:

1. The master process is *always* placed on processor #0.
2. Slave processes are *always* allocated to processors from 0 increasing up to N-1, within each phase of parallel execution the program enters.

As such, it can be seen that all communications will *only* be between processor #0 and each of the other processors (i.e. *not* between these other processors). Therefore the only communications we need to benchmark are messages of various sizes being sent/received between processor #0 and each of the other processors.

As detailed in section 2.7.2 this was done for a range of message sizes (in bytes), namely in the range: 256 bytes, 512, 768, ... , 5120 bytes. Each message was sent 5000 times and a time-cost recorded for each. An average time-cost was then calculated. To prevent the occurrence and propagation of truncation and round-off errors, an arbitrary precision library was implemented for use in these calculations.

These average time-costs were then plotted onto the graphs in Figs. A.1 to A.6, and a chi-squared line-fitting algorithm [111, 8] used to compute an accurate representative function for each set of data produced. (Each of the graphs also displays its representative line-fit function). Fortunately, when the data points were plotted, reasonably straight lines were formed, so the line-fit functions achieved a good degree of 'fit' to their data.

Example 20. Given a statement, as below, appearing in the master process:

```
call csntx(transport, 0, slaveid(1), arrayVar(0), 2048)
```

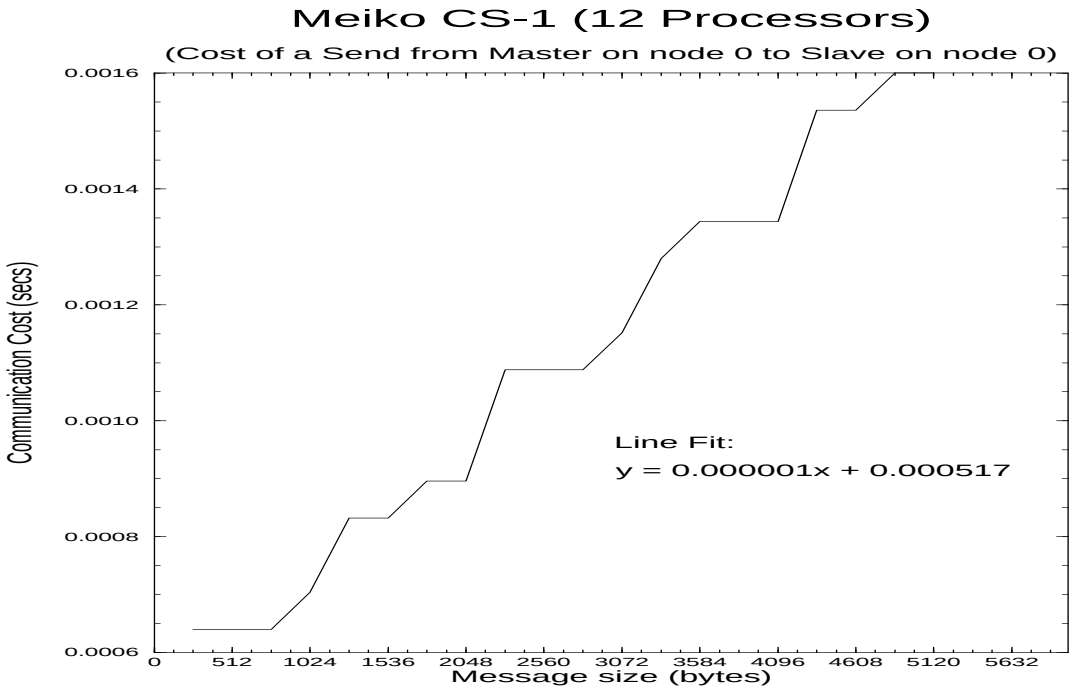
We know also (from the '.par file' generated) that the process with `slaveid(1)` is executing on processor node #1. Hence, we need to determine the time-cost of a

message communication from the master process (on processor #0) to a slave process (on processor #1) of size 2048 bytes. In Fig. A.1 we can see the cost of such a communication can be determined visually (as indicated by the dashed arrow). The REVOLVER system will use the line-fit function and compute the time cost for this communication as:

$$\begin{aligned} y &= 0.000001x + 0.000528 \\ &= 0.002048x + 0.000528 \\ &= 0.002570 \end{aligned}$$

Total Estimated Execution Time = 0.002570 Seconds

Communications from slaves back to the master use the same line-fit equations in their time cost analysis. This is reasonable since, on average, a communication between two processors (for example from processor A to B) should take the same time as a communication from processor B to A (given the same message sizes and network traffic (network contention is also discussed in section 4.10.5).



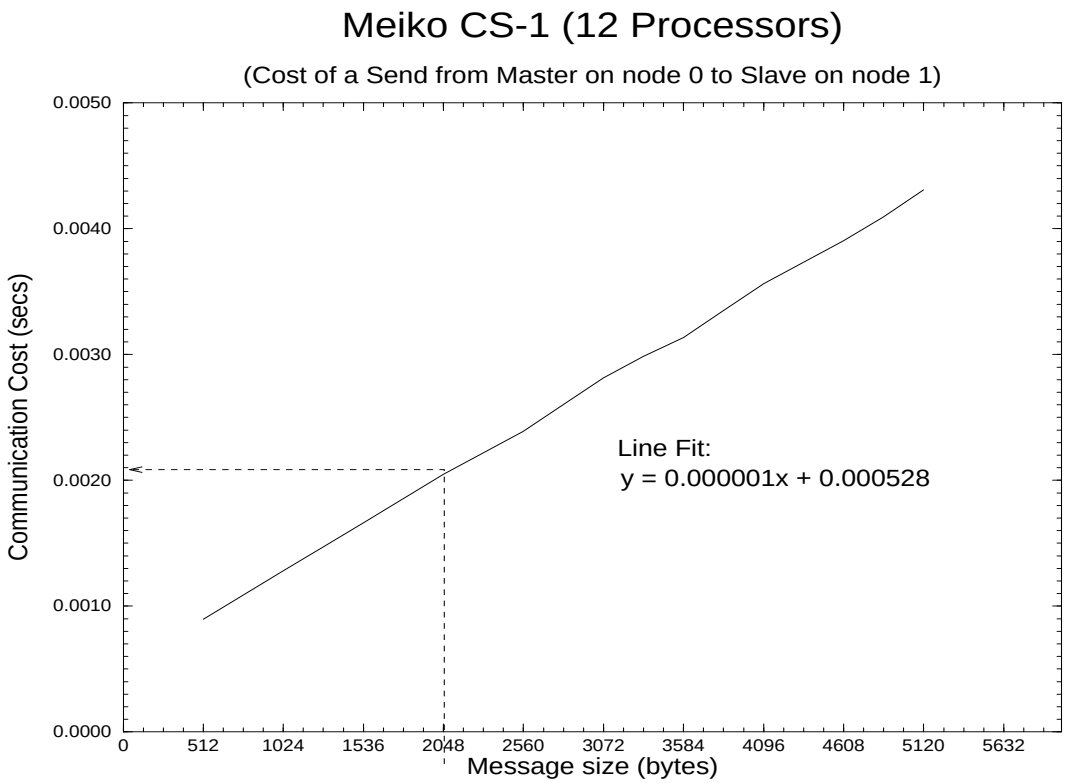


Figure A.1: Communication Graphs for CS-1

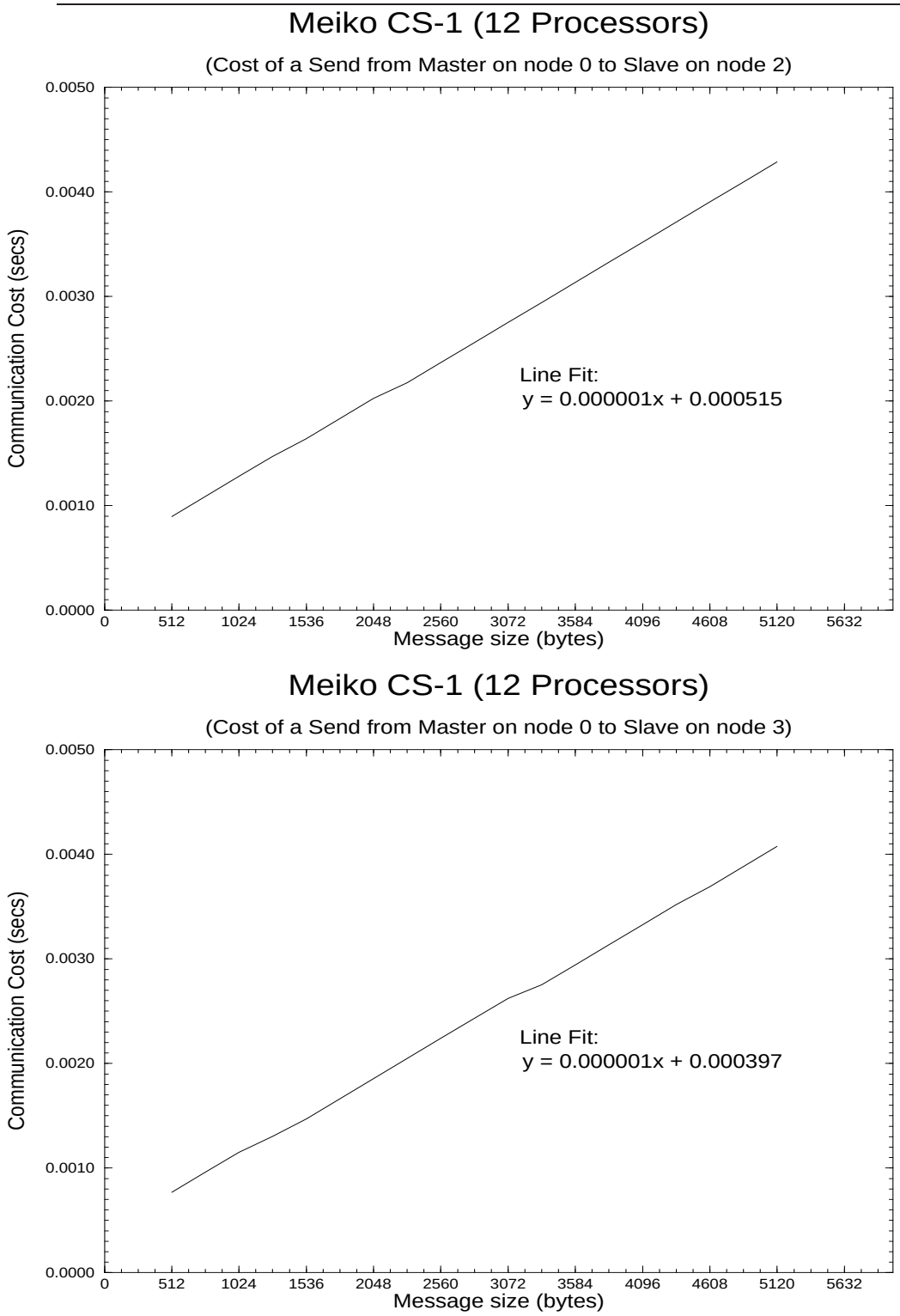


Figure A.2: Communication Graphs for CS-1

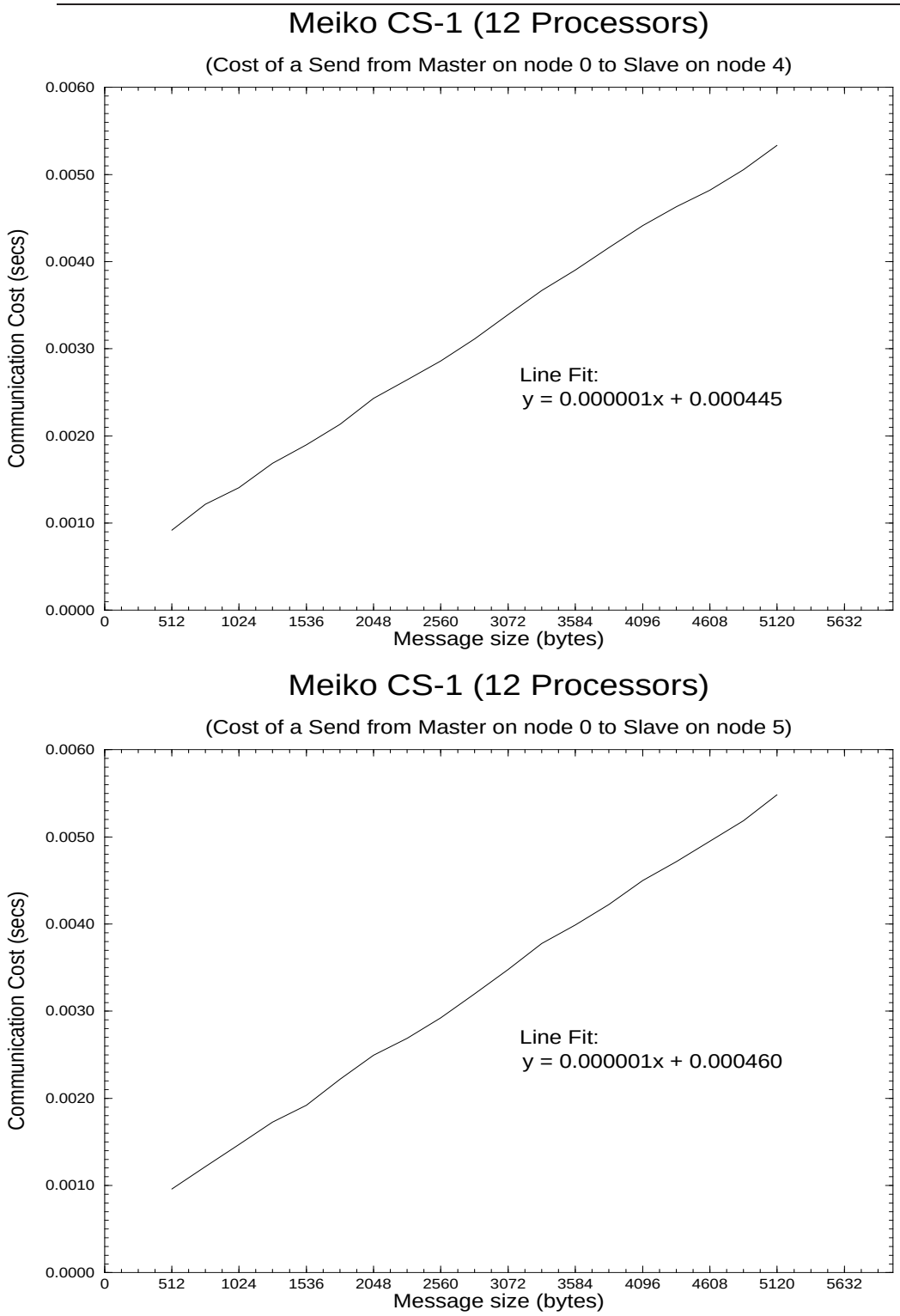


Figure A.3: Communication Graphs for CS-1

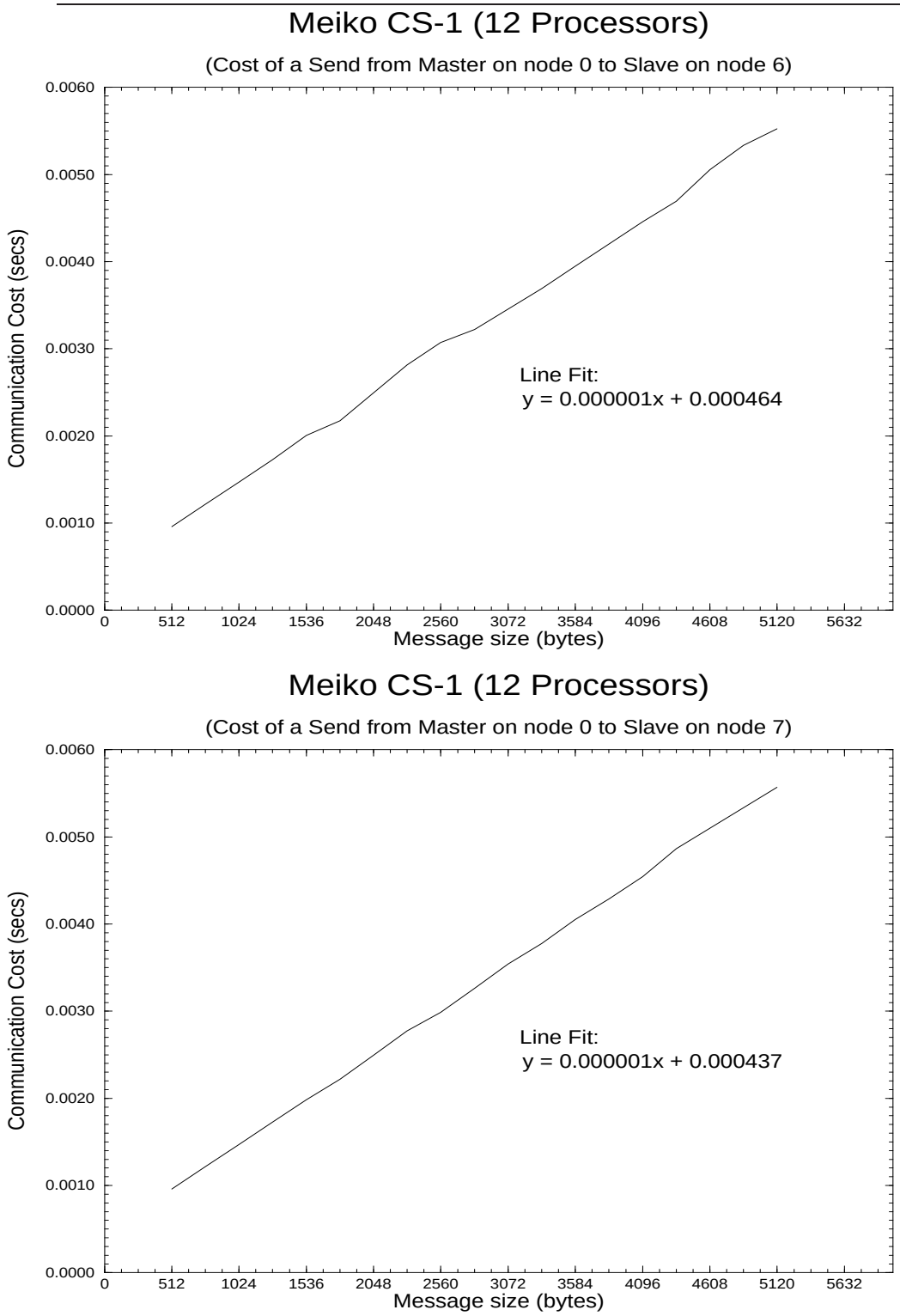


Figure A.4: Communication Graphs for CS-1

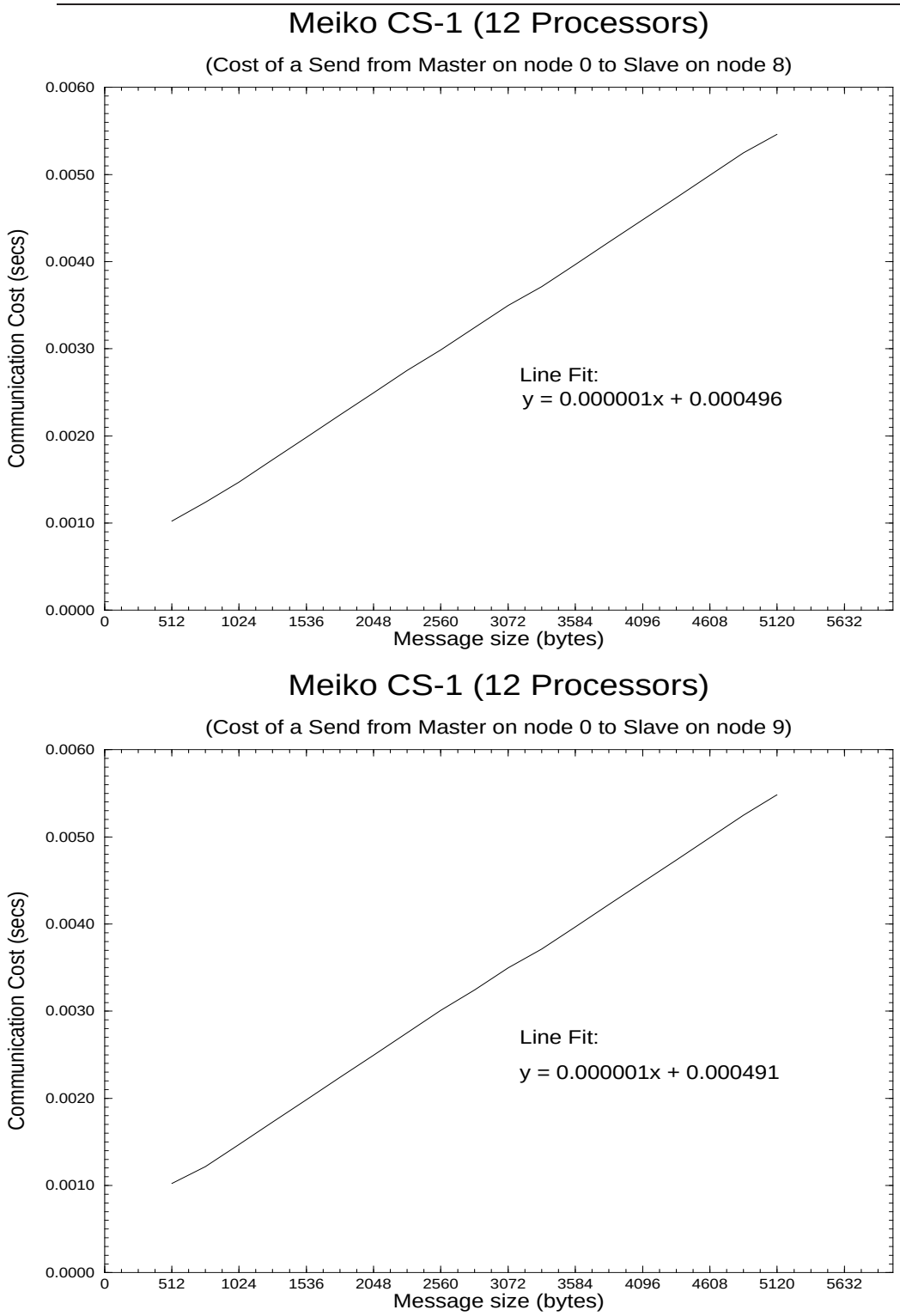


Figure A.5: Communication Graphs for CS-1

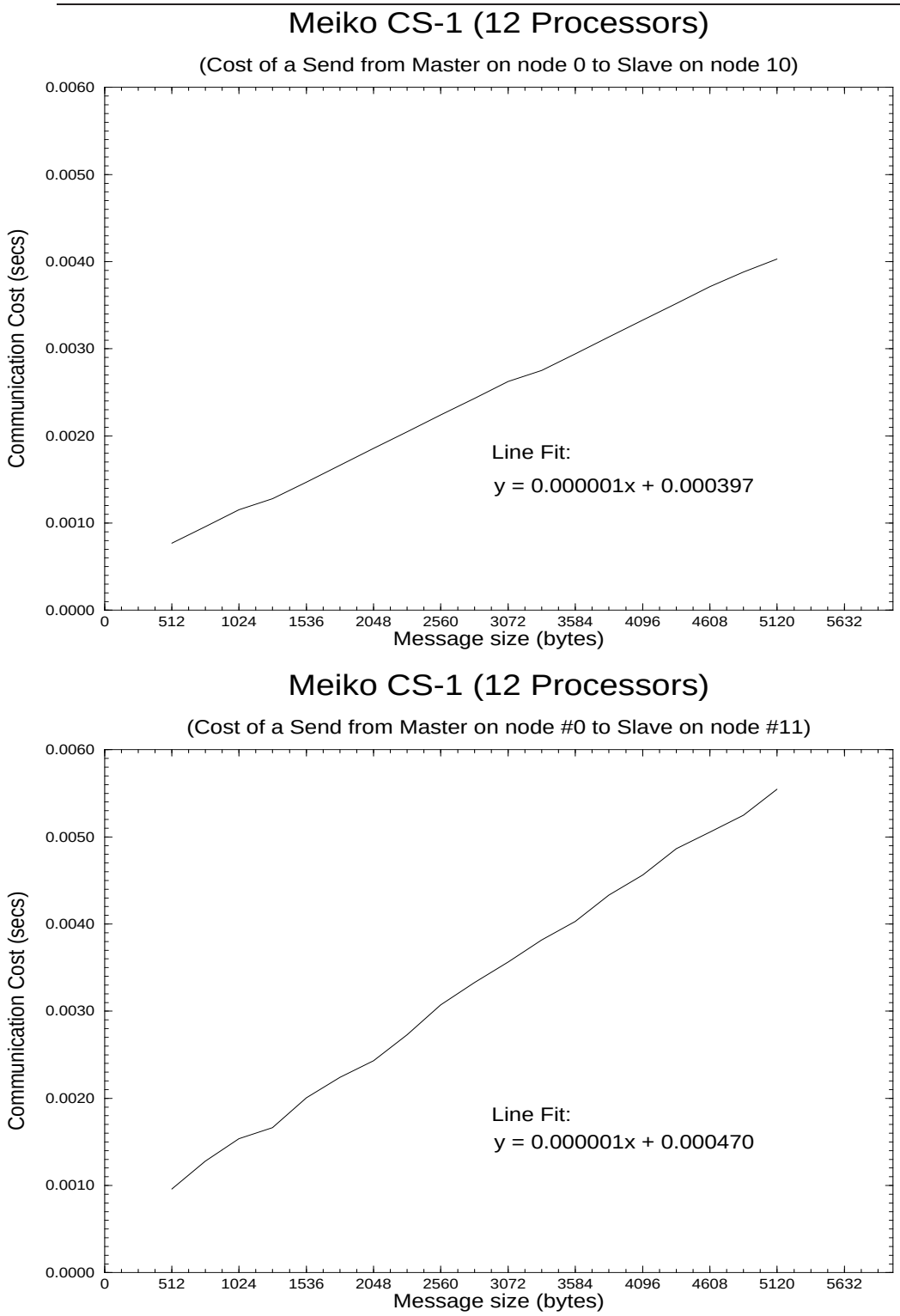


Figure A.6: Communication Graphs for CS-1

A.2 EFLUX Example Program Code

```
C     FILE : eflux.f
C
C  Subroutine EFLUX from the FLO52 program of the
C  Perfect Benchmarks which analyzes the transonic
C  flow past an airfoil by finding a solution to the
C  set of unsteady Euler equations.
C
C  One of the most interesting and time-consuming
C  subroutines because of its manipulation of both
C  two and three dimensional arrays.
C

      PROGRAM EFLUX
      implicit none
      integer i2, j2, il, jl
      real*8 DW(194,34,4)
      real*8 FS(193,34,4)
      real*8 W(194,34,4), P(194,34), X(194, 34, 2)
      real*8 sum
      integer a,b,c,d,i,j,k,n,xx,xy,yx,yy,pa,qsp,qsm

      jl = 33
      il = 193

C Initialise arrays W, P and X to reasonable values.
      do a = 1, 194
        do b = 1, 34
          P(a,b) = jl * il
          do c = 1, 2
            W(a,b,c) = jl * il * c
            X(a,b,c) = jl * il * c
          enddo
          do d = 3, 4
            X(a,b,d) = jl * il * d
          enddo
        enddo
      enddo


      DO 10 j = 2, jl
        DO 11 i = 1, il
          xy = x(i, j, 1) - x(i, j-1, 1)
          yy = x(i, j, 2) - x(i, j-1, 2)
          pa = p(i+1, j) + p(i,j)
          qsp = (yy * w(i+1,j,2) - xy*w(i+1,j,3))/w(i+1,j,1)
          qsm = (yy * w(i,j,2) - xy*w(i,j,3))/w(i,j,1)
          fs(i,j,1) = qsp*w(i+1,j,1) + qsm*w(i,j,1)
```

```
          fs(i,j,2) = qsp*w(i+1,j,2) + qsm*w(i,j,2) + yy*pa
          fs(i,j,3) = qsp*w(i+1,j,3) + qsm*w(i,j,3) - xy*pa
          fs(i,j,4) = qsp*(w(i+1,j,4)+p(i+1,j))+qsm*(w(i,j,4)+p(i,j))
11      CONTINUE
10    CONTINUE

      DO 20 N=1, 4
        DO 21 J=2, JL
          DO 22 I=2, IL
            DW(i,j,n) = fs(i,j,n) - fs(i-1,j,n)
22        CONTINUE
21      CONTINUE
20    CONTINUE

      DO 25 i=2, il
        xx = x(i,1,1) - x(i-1,1,1)
        yx = x(i,1,2) - x(i-1,1,2)
        pa = p(i,2) + p(i,1)
        fs(I,1,1) = 0
        fs(I,1,2) = -yx*pa
        fs(I,1,3) = xx*pa
        fs(I,1,4) = 0
25    CONTINUE


      DO 30 j=2, jl
        DO 31 i=2, il
          xx = x(i, j, 1) - x(i-1, j, 1)
          yx = x(i, j, 2) - x(i-1, j, 2)
          pa = p(i, j+1) + p(i,j)
          qsp = (xx * w(i,j+1,3) - yx*w(i,j+1,2))/w(i,j+1,1)
          qsm = (xx * w(i,j,3) - yx*w(i,j,2))/w(i,j,1)
          fs(i,j,1) = qsp*w(i,j+1,1) + qsm*w(i,j,1)
          fs(i,j,2) = qsp*w(i,j+1,2) + qsm*w(i,j,2) - yx*pa
          fs(i,j,3) = qsp*w(i,j+1,3) + qsm*w(i,j,3) + xx*pa
          fs(i,j,4) = qsp*(w(i,j+1,4)+p(i,j+1))+qsm*(w(i,j,4)+p(i,j))
31      CONTINUE
30    CONTINUE


      DO 40 n=1, 4
        DO 41 j=2, jl
          DO 42 i=2, il
            dw(i,j,n) = dw(i,j,n) + fs(i,j,n) - fs(i,j-1,n)
42        CONTINUE
41      CONTINUE
40    CONTINUE

C      WRITE(*,*) C
```

```
END
```

A.3 TEST-1 Example Program Code

```
 1  C Loop transformation testing program.
 2
 3        PROGRAM testA
 4        INTEGER i, j, k, l, m, n, o, p, q
 5        integer SIZE
 6        integer a(128,128), b(128,128), mat(128,128)
 7        integer c(128), d(128)
 8
 9        size = 128
10
11  C Initialisation loop.
12        do i = 1, SIZE
13          do j = 1, SIZE
14            a(i,j) = i + j
15            b(i,j) = i * j
16          enddo
17        enddo
18
19        do k = 1, SIZE
20          c(k) = b(k, 1) * a(k, 1)
21        enddo
22
23        do l = 1, SIZE
24          d(l) = a(1, l)
25          c(l) = b(1, l)
26        enddo
27
28        do m = 1, SIZE
29          c(m) = b(1, m) * a(1, m)
30          d(m) = b(1, m)
31        enddo
32
33        do m = SIZE / 2, SIZE, 2
34          d(m) = b(1, m)
35          c(m) = b(1, m) * a(1, m)
36        enddo
37
38        do n = 1, SIZE
39          do o = 1, SIZE
40            b(n,o) = a(n,o) / 5 * 9 + 32
41          enddo
42        enddo
43
44  C Matrix multiplication-type loop.
45        do i = 1, SIZE
46          do j = 1, SIZE
47            do k = 1, SIZE
```

```
48                mat(i, j) = mat(i, j) + a(i, k) * b(k, j)
49              enddo
50            enddo
51          enddo
52
53
54          do p = 1, SIZE
55            do q = 1, SIZE
56              print*, 'mat(p,q) = ', mat(p,q)
57            enddo
58          enddo
59
60          end
61
```

A.4 Case Study: Livermore Kernel 18 Example Program Code

A.4.1 Original Sequential Program

```
      PROGRAM LIVERMORE18
c
c*****************************************************************
c***  KERNEL 18     2-D EXPLICIT HYDRODYNAMICS FRAGMENT
c*****************************************************************
c
c
      IMPLICIT NONE
      REAL*8 S, T
      INTEGER I, J, K, N, JN, KN
      PARAMETER( JN = 100)
      PARAMETER( KN = 6)
      REAL*8 ZA(JN, KN)
      REAL*8 ZB(JN, KN+1)
      REAL*8 ZM(JN, KN+1)
      REAL*8 ZP(JN, KN+1)
      REAL*8 ZQ(JN, KN+1)
      REAL*8 ZR(JN+1, KN+1)
      REAL*8 ZU(JN, KN)
      REAL*8 ZV(JN, KN)
      REAL*8 ZZ(JN+1, KN+1)

C     Read arrays.
      read*,((ZM(i,j), i=1, JN), j=1, KN+1)
      read*,((ZP(i,j), i=1, JN), j=1, KN+1)
      read*,((ZQ(i,j), i=1, JN), j=1, KN+1)
      read*,((ZR(i,j), i=1, JN+1), j=1, KN+1)
      read*,((ZU(i,j), i=1, JN), j=1, KN)
      read*,((ZZ(i,j), i=1, JN+1), j=1, KN+1)

1018  T= 0.003700d0
      S= 0.004100d0

      DO 70  k= 2,KN
      DO 70  j= 2,JN
          ZA(j,k)= (ZP(j-1,k+1)+ZQ(j-1,k+1)-ZP(j-1,k)-ZQ(j-1,k))
     1            *(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))
     1            *(ZR(j,k)+ZR(j,k-1))/(ZM(j,k)+ZM(j-1,k))
70    CONTINUE
c
      DO 72  k= 2,KN
      DO 72  j= 2,JN
```

```
44              ZU(j,k)= ZU(j,k)+S*(ZA(j,k)*(ZZ(j,k)-ZZ(j+1,k))
45         1                       -ZA(j-1,k) *(ZZ(j,k)-ZZ(j-1,k))
46         2                       -ZB(j,k)   *(ZZ(j,k)-ZZ(j,k-1))
47         3                       +ZB(j,k+1) *(ZZ(j,k)-ZZ(j,k+1)))
48              ZV(j,k)= ZV(j,k)+S*(ZA(j,k)*(ZR(j,k)-ZR(j+1,k))
49         1                       -ZA(j-1,k) *(ZR(j,k)-ZR(j-1,k))
50         2                       -ZB(j,k)   *(ZR(j,k)-ZR(j,k-1))
51         3                       +ZB(j,k+1) *(ZR(j,k)-ZR(j,k+1)))
52    72    CONTINUE
53    c
54          DO 75  k= 2,KN
55          DO 75  j= 2,JN
56            ZR(j,k)= ZR(j,k)+T*ZU(j,k)
57            ZZ(j,k)= ZZ(j,k)+T*ZV(j,k)
58    75    CONTINUE
59    c
60    c     Print arrays.
61          print*,((ZR(i,j), i=1, JN+1), j=1, KN+1)
62          print*,((ZZ(i,j), i=1, JN+1), j=1, KN+1)
63          end
```

A.4.2 Restructured but Pre-Parallelization Code

Example sequential code (Livermore Kernel 18 program) produced by REVOLVER *after* restructuring but *before* parallelization. NOTE loops indexed by variables `i_N` (where `N` is an integer ¿= 0, indicate that the loop has been marked for parallelization by code-generator. All loops are initially normalised at the start of any run. The code was restructured by the fittest chromosome produced by algorithm **GA-1** after 50 generations and consisted of the following transformations which were applied in order (1 to 4);

```
1) LSP 4

2) LRV 4

3) LIC 6  [1 0]

4) LRV 2
```

From the code it can be seen that 1 of 7 loops has been marked for parallelization. Remember that this definition of 'fitness' is based on the information returned to the GA by the performance estimator and is not based on actual execution time. Parallel code is generated next (see next 2 sections) and it's performance estimated - in this

case a fitness value of 1.113288 secs was returned.

```
 1         program Best
 2         integer  i_0
 3         implicit none
 4         real*8    s,t
 5         integer  a,b,c,d,e,f,i,j,k,n,jn,kn,i_0,i_1,i_2
 6         parameter (t = 0.0037)
 7         parameter (s = 0.0041)
 8         real*8    za(100,6)
 9         real*8    zb(100,6 + 1)
10         real*8    zm(100,6 + 1)
11         real*8    zp(100,6 + 1)
12         real*8    zq(100,6 + 1)
13         real*8    zr(100 + 1,6 + 1)
14         real*8    zu(100,6)
15         real*8    zv(100,6)
16         real*8    zz(100 + 1,6 + 1)
17         read (unit = *, fmt = *)((zm(i,j), i = 1,100), j = 1,6 + 1)
18         read (unit = *, fmt = *)((zp(i,j), i = 1,100), j = 1,6 + 1)
19         read (unit = *, fmt = *)((zq(i,j), i = 1,100), j = 1,6 + 1)
20         read (unit = *, fmt = *)((zr(i,j), i = 1,100 + 1), j = 1,6 + 1)
21         read (unit = *, fmt = *)((zu(i,j), i = 1,100), j = 1,6)
22         read (unit = *, fmt = *)((zz(i,j), i = 1,100 + 1), j = 1,6 + 1)
23
24   C* Analysis Report: Not Parallel because data dependence on ARRAY
25         do  a = 1,5,1
26
27   C* Analysis Report: Not Parallel because data dependence on ARRAY
28            do  b = 99,1, (-(1))
29               za(2 + (b - 1),2 + (a - 1)) = (zp(2 + (b - 1) - 1,2 + (a - 1
30        +) + 1) + zq(2 + (b - 1) - 1,2 + (a - 1) + 1) - zp(2 + (b - 1) - 1,
31        +2 + (a - 1)) - zq(2 + (b - 1) - 1,2 + (a - 1))) * (zr(2 + (b - 1),
32        +2 + (a - 1)) + zr(2 + (b - 1) - 1,2 + (a - 1))) / (zm(2 + (b - 1)
33        +- 1,2 + (a - 1)) + zm(2 + (b - 1) - 1,2 + (a - 1) + 1))
34               zb(2 + (b - 1),2 + (a - 1)) = (zp(2 + (b - 1) - 1,2 + (a - 1
35        +)) + zq(2 + (b - 1) - 1,2 + (a - 1)) - zp(2 + (b - 1),2 + (a - 1))
36        + - zq(2 + (b - 1),2 + (a - 1))) * (zr(2 + (b - 1),2 + (a - 1)) + z
37        +r(2 + (b - 1),2 + (a - 1) - 1)) / (zm(2 + (b - 1),2 + (a - 1)) + z
38        +m(2 + (b - 1) - 1,2 + (a - 1)))
39            enddo
40         enddo
41
42   C* Analysis Report: Not Parallel because data dependence on ARRAY
43         do  c = 1,5,1
44
45   C* Analysis Report: Not Parallel because data dependence on ARRAY
46            do  d = 99,1, (-(1))
47               zu(2 + (d - 1),2 + (c - 1)) = zu(2 + (d - 1),2 + (c - 1)) +
```

```
48        +s * (za(2 + (d - 1),2 + (c - 1)) * (zz(2 + (d - 1),2 + (c - 1)) -
49        +zz(2 + (d - 1) + 1,2 + (c - 1))) - za(2 + (d - 1) - 1,2 + (c - 1))
50        + * (zz(2 + (d - 1),2 + (c - 1)) - zz(2 + (d - 1) - 1,2 + (c - 1)))
51        + - zb(2 + (d - 1),2 + (c - 1)) * (zz(2 + (d - 1),2 + (c - 1)) - zz
52        +(2 + (d - 1),2 + (c - 1) - 1)) + zb(2 + (d - 1),2 + (c - 1) + 1) *
53        + (zz(2 + (d - 1),2 + (c - 1)) - zz(2 + (d - 1),2 + (c - 1) + 1)))
54              enddo
55
56    C* Analysis Report: Not Parallel because data dependence on ARRAY
57              do  d = 1,99,1
58                 zv(2 + (d - 1),2 + (c - 1)) = zv(2 + (d - 1),2 + (c - 1)) +
59        +s * (za(2 + (d - 1),2 + (c - 1)) * (zr(2 + (d - 1),2 + (c - 1)) -
60        +zr(2 + (d - 1) + 1,2 + (c - 1))) - za(2 + (d - 1) - 1,2 + (c - 1))
61        + * (zr(2 + (d - 1),2 + (c - 1)) - zr(2 + (d - 1) - 1,2 + (c - 1)))
62        + - zb(2 + (d - 1),2 + (c - 1)) * (zr(2 + (d - 1),2 + (c - 1)) - zr
63        +(2 + (d - 1),2 + (c - 1) - 1)) + zb(2 + (d - 1),2 + (c - 1) + 1) *
64        + (zr(2 + (d - 1),2 + (c - 1)) - zr(2 + (d - 1),2 + (c - 1) + 1)))
65              enddo
66           enddo
67
68    C* PARALLELIZE Loop i_0
69          do  i_0 = 1,99,1
70              do  e = 1,5,1
71                 zr(2 + (i_0 - 1),2 + (e - 1)) = zr(2 + (i_0 - 1),2 + (e - 1)
72        +) + t * zu(2 + (i_0 - 1),2 + (e - 1))
73                 zz(2 + (i_0 - 1),2 + (e - 1)) = zz(2 + (i_0 - 1),2 + (e - 1)
74        +) + t * zv(2 + (i_0 - 1),2 + (e - 1))
75              enddo
76           enddo
77
78           print *,((zr(i,j), i = 1,100 + 1), j = 1,6 + 1)
79           print *,((zz(i,j), i = 1,100 + 1), j = 1,6 + 1)
80           end
```

A.4.3 Parallelized Code

Full listing of Master/Slave code generated for Meiko CS-1 as part of example of Livermore Kernel 18 parallelization.

A.4.4 Master Program

```
1         program Besthost
2
3     #include <csn/names.inc>
4     #include <cs.inc>
5     #include <csn/csn.inc>
```

```
 6
 7         real*8    IntArray1(46)
 8         integer  count_1
 9         implicit none
10         integer  loop0node(12)
11         character  *30 buff
12         integer  ub
13         integer  lb
14         integer  status
15         integer  transport
16         integer  i_0
17         real*8    s,t
18         integer  a,b,c,d,e,f,i,j,k,n,jn,kn,i_0,i_1,i_2
19         parameter (t = 0.0037)
20         parameter (s = 0.0041)
21         real*8    za(100,6)
22         real*8    zb(100,6 + 1)
23         real*8    zm(100,6 + 1)
24         real*8    zp(100,6 + 1)
25         real*8    zq(100,6 + 1)
26         real*8    zr(100 + 1,6 + 1)
27         real*8    zu(100,6)
28         real*8    zv(100,6)
29         real*8    zz(100 + 1,6 + 1)
30
31   C   Initialise CS-1 Network.
32         call csninit()
33
34   C   Open the host transport.
35         status = csnopen (csnnullid,transport)
36         if (status .ne. csnok) then
37             call csabort('host cannot open transport',-1)
38          endif
39
40   C   Register the host transport.
41         status = csnregname (transport,'HostTransport')
42         if (status .ne. csnok) then
43             call csabort('host cannot register transport',-1)
44          endif
45
46
47   C   Store transport ids for loop 0.
48         do  i = 1,12,1
49            write (buff,20)'loop0node',i-1
50            status = csnlookupname (loop0node(i),buff,.true.)
51            if (status .ne. csnok) then
52                call csabort('host cannot lookup loop0',-1)
53             endif
54         enddo
```

```
 55         read (unit = *, fmt = *)((zm(i,j), i = 1,100), j = 1,6 + 1)
 56         read (unit = *, fmt = *)((zp(i,j), i = 1,100), j = 1,6 + 1)
 57         read (unit = *, fmt = *)((zq(i,j), i = 1,100), j = 1,6 + 1)
 58         read (unit = *, fmt = *)((zr(i,j), i = 1,100 + 1), j = 1,6 + 1)
 59         read (unit = *, fmt = *)((zu(i,j), i = 1,100), j = 1,6)
 60         read (unit = *, fmt = *)((zz(i,j), i = 1,100 + 1), j = 1,6 + 1)
 61
 62         do  a = 1,5,1
 63            do  b = 99,1, (-(1))
 64               za(2 + (b - 1),2 + (a - 1)) = (zp(2 + (b - 1) - 1,2 + (a - 1
 65        +) + 1) + zq(2 + (b - 1) - 1,2 + (a - 1) + 1) - zp(2 + (b - 1) - 1,
 66        +2 + (a - 1)) - zq(2 + (b - 1) - 1,2 + (a - 1))) * (zr(2 + (b - 1),
 67        +2 + (a - 1)) + zr(2 + (b - 1) - 1,2 + (a - 1))) / (zm(2 + (b - 1)
 68        +- 1,2 + (a - 1)) + zm(2 + (b - 1) - 1,2 + (a - 1) + 1))
 69
 70               zb(2 + (b - 1),2 + (a - 1)) = (zp(2 + (b - 1) - 1,2 + (a - 1
 71        +)) + zq(2 + (b - 1) - 1,2 + (a - 1)) - zp(2 + (b - 1),2 + (a - 1))
 72        + - zq(2 + (b - 1),2 + (a - 1))) * (zr(2 + (b - 1),2 + (a - 1)) + z
 73        +r(2 + (b - 1),2 + (a - 1) - 1)) / (zm(2 + (b - 1),2 + (a - 1)) + z
 74        +m(2 + (b - 1) - 1,2 + (a - 1)))
 75            enddo
 76         enddo
 77
 78         do  c = 1,5,1
 79            do  d = 99,1, (-(1))
 80               zu(2 + (d - 1),2 + (c - 1)) = zu(2 + (d - 1),2 + (c - 1)) +
 81        +s * (za(2 + (d - 1),2 + (c - 1)) * (zz(2 + (d - 1),2 + (c - 1)) -
 82        +zz(2 + (d - 1) + 1,2 + (c - 1))) - za(2 + (d - 1) - 1,2 + (c - 1))
 83        + * (zz(2 + (d - 1),2 + (c - 1)) - zz(2 + (d - 1) - 1,2 + (c - 1)))
 84        + - zb(2 + (d - 1),2 + (c - 1)) * (zz(2 + (d - 1),2 + (c - 1)) - zz
 85        +(2 + (d - 1),2 + (c - 1) - 1)) + zb(2 + (d - 1),2 + (c - 1) + 1) *
 86        + (zz(2 + (d - 1),2 + (c - 1)) - zz(2 + (d - 1),2 + (c - 1) + 1)))
 87            enddo
 88            do  d = 1,99,1
 89               zv(2 + (d - 1),2 + (c - 1)) = zv(2 + (d - 1),2 + (c - 1)) +
 90        +s * (za(2 + (d - 1),2 + (c - 1)) * (zr(2 + (d - 1),2 + (c - 1)) -
 91        +zr(2 + (d - 1) + 1,2 + (c - 1))) - za(2 + (d - 1) - 1,2 + (c - 1))
 92        + * (zr(2 + (d - 1),2 + (c - 1)) - zr(2 + (d - 1) - 1,2 + (c - 1)))
 93        + - zb(2 + (d - 1),2 + (c - 1)) * (zr(2 + (d - 1),2 + (c - 1)) - zr
 94        +(2 + (d - 1),2 + (c - 1) - 1)) + zb(2 + (d - 1),2 + (c - 1) + 1) *
 95        + (zr(2 + (d - 1),2 + (c - 1)) - zr(2 + (d - 1),2 + (c - 1) + 1)))
 96            enddo
 97         enddo
 98
 99   C   Linearize array accesses and send data.
100         do  i = 1,12,1
101            count_1 = 1
102            lb = 9 * (i - 1) + 1
103            ub = 9 - 1 + lb
```

```
104            if (lb. ge. 1 .and. ub .le. 99) then
105                do  t_1 = 2,6,1
106                  do  t_0 = lb,ub,1
107                     IntArray1(count_1) = zr(t_0,t_1)
108                     count_1 = count_1 + 1
109                  enddo
110                enddo
111             endif
112            call csntx(transport,0,loop0node(i),IntArray1(1),360)
113          enddo
114
115     C    Linearize array accesses and send data.
116          do  i = 1,12,1
117            count_1 = 1
118            lb = 9 * (i - 1) + 1
119            ub = 9 - 1 + lb
120            if (lb. ge. 1 .and. ub .le. 99) then
121                do  t_1 = 2,6,1
122                  do  t_0 = lb,ub,1
123                     IntArray1(count_1) = zu(t_0,t_1)
124                     count_1 = count_1 + 1
125                  enddo
126                enddo
127             endif
128            call csntx(transport,0,loop0node(i),IntArray1(1),360)
129          enddo
130
131     C    Linearize array accesses and send data.
132          do  i = 1,12,1
133            count_1 = 1
134            lb = 9 * (i - 1) + 1
135            ub = 9 - 1 + lb
136            if (lb. ge. 1 .and. ub .le. 99) then
137                do  t_1 = 2,6,1
138                  do  t_0 = lb,ub,1
139                     IntArray1(count_1) = zz(t_0,t_1)
140                     count_1 = count_1 + 1
141                  enddo
142                enddo
143             endif
144            call csntx(transport,0,loop0node(i),IntArray1(1),360)
145          enddo
146
147     C    Linearize array accesses and send data.
148          do  i = 1,12,1
149            count_1 = 1
150            lb = 9 * (i - 1) + 1
151            ub = 9 - 1 + lb
152            if (lb. ge. 1 .and. ub .le. 99) then
```

```
153                do  t_1 = 2,6,1
154                  do  t_0 = lb,ub,1
155                     IntArray1(count_1) = zv(t_0,t_1)
156                     count_1 = count_1 + 1
157                  enddo
158                enddo
159              endif
160              call csntx(transport,0,loop0node(i),IntArray1(1),360)
161          enddo
162
163    C     Receive and unpack array data.
164          do  i = 1,12,1
165             call csnrx(transport,0,IntArray1(1),368)
166             count_1 = 1
167             lb = 9 * IntArray1(46) + 1
168             ub = 9 - 1 + lb
169             if (lb. ge. 1 .and. ub .le. 99) then
170                 do  t_1 = 2,6,1
171                   do  t_0 = lb,ub,1
172                      zz(t_0,t_1) = IntArray1(count_1)
173                      count_1 = count_1 + 1
174                   enddo
175                enddo
176              endif
177          enddo
178
179    C     Receive and unpack array data.
180          do  i = 1,12,1
181             call csnrx(transport,0,IntArray1(1),368)
182             count_1 = 1
183             lb = 9 * IntArray1(46) + 1
184             ub = 9 - 1 + lb
185             if (lb. ge. 1 .and. ub .le. 99) then
186                 do  t_1 = 2,6,1
187                   do  t_0 = lb,ub,1
188                      zr(t_0,t_1) = IntArray1(count_1)
189                      count_1 = count_1 + 1
190                   enddo
191                enddo
192              endif
193          enddo
194
195    C     Scalar insertion : e
196          e = 0
197
198    C     Scalar insertion : i_0
199          i_0 = 99
200          print *,((zr(i,j), i = 1,100 + 1), j = 1,6 + 1)
201          print *,((zz(i,j), i = 1,100 + 1), j = 1,6 + 1)
```

```
202   20      format (A, I2)
203         end
204
```

A.4.5 Slave Program

```
 1          program Besthostloop0
 2
 3   #include <csn/names.inc>
 4   #include <cs.inc>
 5   #include <csn/csn.inc>
 6
 7          real*8    IntArray1(46)
 8          integer  count_1
 9          implicit none
10          character  *30 buff
11          integer  ub
12          integer  lb
13          integer  localId
14          integer  hostId
15          integer  procNo
16          integer  nProcs
17          integer  returnCode
18          integer  status
19          integer  transport
20          integer  i_0
21          real*8    s,t
22          integer  a,b,c,d,e,f,i,j,k,n,jn,kn,i_0,i_1,i_2
23          parameter (t = 0.0037)
24          parameter (s = 0.0041)
25          real*8    za(100,6)
26          real*8    zb(100,6 + 1)
27          real*8    zm(100,6 + 1)
28          real*8    zp(100,6 + 1)
29          real*8    zq(100,6 + 1)
30          real*8    zr(100 + 1,6 + 1)
31          real*8    zu(100,6)
32          real*8    zv(100,6)
33          real*8    zz(100 + 1,6 + 1)
34
35   C   Initialise CS-1 Network.
36          call csninit()
37
38   C   Find out which number processor we are on (0...n-1)
39          returnCode = csgetinfo (nProcs,procNo,localId)
40
41   C   Open the transport network.
42          status = csnopen (csnnullid,transport)
43          if (status .ne. csnok) then
```

```
44              call csabort('loop0 cannot open transport',-1)
45            endif
46
47    C   Register the local process id.
48          write (buff,20)'loop0node',procNo
49          status = csnregname (transport,buff)
50
51
52          if (status .ne. csnok) then
53              call csabort('loop0 cannot register loop0',-1)
54            endif
55
56    C   Lookup the host process id.
57          status = csnlookupname (hostId,'HostTransport',.true.)
58
59
60          if (status .ne. csnok) then
61              call csabort('loop0 cannot lookup HostTransport',-1)
62            endif
63
64    C   Scalar insertion : e
65          e = 0
66
67    C   Scalar insertion : e
68          e = 0
69
70    C   Receive and unpack array data.
71          call csnrx(transport,hostid,IntArray1(1),360)
72          count_1 = 1
73          lb = 9 * procNo + 1
74          ub = 9 - 1 + lb
75          if (lb. ge. 1 .and. ub .le. 99) then
76              do  t_1 = 2,6,1
77                do  t_0 = lb,ub,1
78                   zr(t_0,t_1) = IntArray1(count_1)
79                   count_1 = count_1 + 1
80                enddo
81              enddo
82            endif
83
84    C   Receive and unpack array data.
85          call csnrx(transport,hostid,IntArray1(1),360)
86          count_1 = 1
87          lb = 9 * procNo + 1
88          ub = 9 - 1 + lb
89          if (lb. ge. 1 .and. ub .le. 99) then
90              do  t_1 = 2,6,1
91                do  t_0 = lb,ub,1
92                   zu(t_0,t_1) = IntArray1(count_1)
```

```
 93                 count_1 = count_1 + 1
 94               enddo
 95             enddo
 96           endif
 97
 98   C   Receive and unpack array data.
 99         call csnrx(transport,hostid,IntArray1(1),360)
100         count_1 = 1
101         lb = 9 * procNo + 1
102         ub = 9 - 1 + lb
103         if (lb. ge. 1 .and. ub .le. 99) then
104             do  t_1 = 2,6,1
105               do  t_0 = lb,ub,1
106                  zz(t_0,t_1) = IntArray1(count_1)
107                  count_1 = count_1 + 1
108               enddo
109             enddo
110           endif
111
112   C   Receive and unpack array data.
113         call csnrx(transport,hostid,IntArray1(1),360)
114         count_1 = 1
115         lb = 9 * procNo + 1
116         ub = 9 - 1 + lb
117         if (lb. ge. 1 .and. ub .le. 99) then
118             do  t_1 = 2,6,1
119               do  t_0 = lb,ub,1
120                  zv(t_0,t_1) = IntArray1(count_1)
121                  count_1 = count_1 + 1
122               enddo
123             enddo
124           endif
125
126
127   C   Slave work loop.
128         if (lb. ge. 1 .and. ub .le. 99) then
129             do  i_0 = lb,ub,1
130               do  e = 1,5,1
131                  zr(2 + (i_0 - 1),2 + (e - 1)) = zr(2 + (i_0 - 1),2 + (e -
132        + 1)) + t * zu(2 + (i_0 - 1),2 + (e - 1))
133                  zz(2 + (i_0 - 1),2 + (e - 1)) = zz(2 + (i_0 - 1),2 + (e -
134        + 1)) + t * zv(2 + (i_0 - 1),2 + (e - 1))
135               enddo
136             enddo
137           endif
138
139   C   Linearize array accesses and send data.
140         count_1 = 1
141         lb = 9 * procNo + 1
```

```
142         ub = 9 - 1 + lb
143         if (lb. ge. 1 .and. ub .le. 99) then
144             do  t_1 = 2,6,1
145               do  t_0 = lb,ub,1
146                  IntArray1(count_1) = zz(t_0,t_1)
147                  count_1 = count_1 + 1
148               enddo
149            enddo
150          endif
151         IntArray1(46) = procNo
152         call csntx(transport,0,hostid,IntArray1(1),368)
153
154   C   Linearize array accesses and send data.
155         count_1 = 1
156         lb = 9 * procNo + 1
157         ub = 9 - 1 + lb
158         if (lb. ge. 1 .and. ub .le. 99) then
159             do  t_1 = 2,6,1
160               do  t_0 = lb,ub,1
161                  IntArray1(count_1) = zr(t_0,t_1)
162                  count_1 = count_1 + 1
163               enddo
164            enddo
165          endif
166         IntArray1(46) = procNo
167         call csntx(transport,0,hostid,IntArray1(1),368)
168   20      format (A, I2)
169         end
170
```

A.4.6 Process Placement File

```
par
  processor 0 Besthost
  processor 0 for 12 Besthostloop0
endpar
```

A.5 Student's t-Test

At some stage stage we would wish to compare and measure the effectiveness of the various evolutionary algorithms we have described against each other. Since the automatic parallelization problem is a function minimization problem we may compare algorithms by the optimal objective values of the function they achieve - for the automatic parallelization problem, we may use the smallest estimated execution times produced by

a series of runs of an EA on a sequential program, as some indication of the general performance of the EA. For example, say we parallelize a particular sequential program p by running a GA on it eight times, and the following estimated execution times (the best of each run) are produced (call this *data set A*):

270.11 257.5 280.71 260.0 264.77 280.0 275.9 272.17

Now say we run a hill-climbing algorithm ten times on program p and the following estimated execution times are produced, *data set B*:

263.11 278.92 293.11 280.55 274.80 294.12 274.65 278.55 283.55 277.72

A naive approach would be to simply calculate the arithmetic means ($\overline{x}$) of each data set and (since we are trying to minimize our objective value) say that whichever algorithm produced the data set with the smaller arithmetic mean would be the better algorithm, 'on average'. Namely:

$$\begin{array}{llll} \sum_A & = 2161.16 & \sum_B & = 2799.08 \\ N_A & = 8 & N_B & = 10 \\ \overline{x_A} & = 270.145 & \overline{x_B} & = 279.908 \end{array}$$

So here we would say the GA performed 'on average' better that the hill-climber.

However, the naivity of this approach lies in that the arithmetic mean can be a deceptive representation of a data set, where one extreme value (a 'one off' value) can seriously distort the value of $\overline{x}$. For instance, say we change one of the values in data set B from 263.11 to 163.11 (the hill-climber worked exceptionally well on this one occasion), and call this new data set C, our calculations now become:

$$\begin{array}{llll} \sum_A & = 2161.16 & \sum_C & = 2699.08 \\ N_A & = 8 & N_C & = 10 \\ \overline{x_A} & = 270.145 & \overline{x_C} & = 269.908 \end{array}$$

Our hill-climber now appears to perform better than our GA! The fragility of making an assessment purely on the basis of comparing arithmetic means is clear - a single 'one-off' value can change the whole picture. A clearly more robust method is required.

Student's t-test is a test of two data sets for significance of difference of means. The significance it returns can be interpreted as a *confidence value* of the results we have obtained (based on arithmetic means). The confidence value represented by the t-value itself takes into account the *variances* of the two data sets (denoted $Var(x_A)$ and $Var(x_B)$). (The version presented here is called the *two-tailed t-test*).

Say we have two data sets which we may allow to differ in their means and their variances, and we wish to obtain some measure of significance of their difference of means. The conventional statistic for measuring the significance of a difference of means is termed *Student's t-test*[†]

[111, 120] the relevant equation for data sets with potentially unequal variances is:

$$t = \frac{\overline{x_A} - \overline{x_B}}{\left[Var(x_A)/N_A + Var(x_B)/N_B \right]^{\frac{1}{2}}} \tag{A.1}$$

This statistic is distributed approximately as Student's t with the number of degrees of freedom calculated as:

$$\frac{\left[\frac{Var(x_A)}{N_A} + \frac{Var(x_B)}{N_B} \right]^2}{\frac{\left[Var(x_A)/N_A \right]^2}{N_A - 1} + \frac{\left[Var(x_B)/N_B \right]^2}{N_B - 1}} \tag{A.2}$$

Importantly, what the t value gives us is a measure of *confidence* that the first set of values are in some way 'better' (i.e. higher, if we are maximizing a function, or lower if we are minimizing) than the other. The value t and the combined size of the two distributions $(N_A + N_B)$ can be used to produce (via a function or look-up table) a percentage confidence value that one set of values is more optimal for our purpose than the other.

[†]This test is called *Student's t-test* after its discoverer, W.S Gossett, who published his works under the pseudonym "Student" during the early part of the twentieth century.

Example 21. Using data sets A and B in accordance with equations A.1 and A.2 we compute:

$\overline{x_A}$ = 270.145 $\overline{x_B}$ = 279.908

Degrees of freedom = 15.33

t-value = -2.32

Confidence value = 0.97

Hence we can say that, since ours is a minimization problem, that algorithm A, (the GA), performed better than algorithm B, (the hill-climber), as indicated by the arithmetic means - with confidence of 97%. (The t-value is computed in accordance with equation A.1 which then gives us a confidence value (via a lookup table, or function).

Using data sets A and C in accordance with equations A.1 and A.2 we compute:

$\overline{x_A}$ = 270.145 $\overline{x_C}$ = 269.908

Degrees of freedom = 10.17

t-value = 0.02

Confidence value = 0.01

Here we can say that the algorithm that produced data set C, (the hill-climber), performed better than the GA, as indicated by the arithmetic means - with confidence of just 1%.

A.6 Summary of Compiler Switches

This section gives a brief description of all compiler switches available in the current implememtation of REVOLVER . Note **R** represents a real number in the range $0.0 \leq 1.0$ which should be entered immediately after the switch, and **N** represents an integer ≥ 1.

-popsize N Specify size of population to be used (size for hill-climbing and simulated-annealing EAs is set to 1 only).

-alg s Specify which EA is to be used - acceptable string values for **s** are one of : **hc1** hill-climber/GT combination,

-gens N Specify how many generations EA is to run for.

-len N Specify starting-length of chromosomes (only for use with gene-transformation representation).

-dec N Specify which decoding strategy is to be used (current options are: 1 = `DELETE-AND-CONTINUE`; 2 = `DELETE-AND-STOP`; and 3 = `REPAIR`. This option is only for use with gene-transformation representation. **sa1** simulated-annealing/GT combination, **hc1** hill-climber/GS combination, **es1** evolution etrategy/GS combination, **ga1** genetic algorithm/GT combination, **saga1** self-adaptive genetic algorithm/GT combination.

-pVLX1 R Set probability for Variable-Length Crossover (1-point) operator (default is 0.6)

-pVLX2 R Set probability for Variable-Length Crossover (2-point) operator (default is 0.6)

-pVLX3 R Set probability for VLX-3 Crossover operator (default is 0.6)

-pMutGene R Set probability for Gene-level mutation operator (default is 0.2)

-pMutLoop R Set probability for Gene-Loop mutation operator (default is 0.2)

-pMutTF R Set probability for Gene-transformation only mutation operator

-pLSP R Set probability for Loop-Splitting mutation operator.

-pLFU R Set probability for Loop-Fusion mutation operator.

-pLIC R Set probability for Loop Interchange mutation operator.

-pLRV R Set probability for Loop Reversal mutation operator.

-tsize N Set value for size of Tournament selection operator.

-nprocs N Specify number of processors in target-machine architecture.

-seed N Set seed value for random number generators used in REVOLVER .

-preNorm N Boolean flag indicating whether to Normalise all loops before parallel code-generation or not (default is 0). (default is 0.2)

Various other debugging options can be found in thhe code in `ga.C`.

A.7 CD-ROM Details

This section gives a brief description of some of the REVOLVER files and the reusable useful code on the disk.

The files on the CD-ROM are a straight copy of the directories related to REVOLVER . The REVOLVER code is located in `/sage++-1.7/Sage++/demos/hill_climbing/` (refered to as `REV_HOME`from now on). The system consists of a Makefile and the following 19 files:

```
-rw-------  1 ssrkwill    19346 Jul 28 08:23 analyze.C
-rw-------  1 ssrkwill    78199 Sep  2 02:41 codegen1.C
-rw-------  1 ssrkwill    49725 Aug 27 17:12 codegen2.C
-rw-------  1 ssrkwill     8826 Aug 27 17:29 decoders.C
-rw-------  1 ssrkwill    25217 Aug  6 19:02 deps.C
-rw-------  1 ssrkwill    24717 Jul 28 06:34 dev.C
-rw-------  1 ssrkwill    79452 Aug 30 22:50 exp.C
-rw-------  1 ssrkwill    36116 Aug 27 23:00 ga.C
-rw-------  1 ssrkwill    17174 Aug 27 18:19 ga2.C
-rw-------  1 ssrkwill    20418 Aug 30 20:13 globals.h
-rw-------  1 ssrkwill    12289 Aug 27 23:49 inter.C
-rw-------  1 ssrkwill    10796 Aug 30 23:02 normalise.C
-rw-------  1 ssrkwill    25050 Aug 30 20:12 perf.C
-rw-------  1 ssrkwill    24216 Sep  2 05:12 revolver.C
-rw-------  1 ssrkwill    20109 Jul 28 04:58 subscript.C
-rw-------  1 ssrkwill    39926 Aug 30 20:21 transformations.C
-rw-------  1 ssrkwill    60807 Aug 30 20:26 transformations2.C
-rw-------  1 ssrkwill    26524 Aug 27 22:56 utils.C
-rw-------  1 ssrkwill    28040 Aug 30 20:00 utils2.C
```

As well a profiler code in `REV_HOME\..\profiler`.

Of these the most important is the header file `globals.h` which contains all the major typedefs, structs, and constants, as well as prototypes for all the functions. It also indicates in which file each function can be found. All the source files in Revolver are well commented.

Some generally useful code can also be found in the following files:

deps.C contains code to perform a topological sort; an implementation of Tarjan's algorithm to find the strongly connected components (SCC's) of a graph, and a function to compute the acyclic condensation of a graph - all used in loop distribution.

`REV_HOME/`**Fourier** contains code to determine if two equations are linearly independent or not (`lin_dep.c`); reduce an $m \times n$ integer matrix to echelon form (`ech_red.c`); determine solutions (if any) to systems of linear, integer diophantine equations `dio_1.c`, `dio_2.c`; reduces a given integer matrix to diagonal form `diag.c`; perform piece-wise rational arithmetic `two_var.h`, `rational.c`; and ultimately Fourier-Motzkin variable Elimination (`fourier.c`). Code is from Banerjee's books [9, 12].

stand_alone3.c contains code to purely tokenise and parse expressions which has been adapted for a number of purposes for use at several points in Revolver . It can tokenise and parse expressions containing C and FORTRAN arithmentic operators (+,-,*,/, operators (>, >=, <, <=, !, !=, ==, &&, , .gt., .ge., .lt., .le., .not., .neq., .eqv., .neqv., .and., .or., .xor.), variables (the code has it's own typed-symbol table), assignment expressions (including nested assignments), and N-dimensional array expressions (including nested N-dimensional array expressions). It's uses one-token lookahead recursive-descent parsing (except where array expressions occur) and needs no other files, hence it's stand_alone code . . . .

transformations2.C contains code to compare if one directed graph is a sub-graph of another, as well as code to cyclically generate sequences of N-long permutations of numbers (2 ¡= N ¡= 7) - used to generate random loop permutations in LIC.

atts and todos are C-shell scripts to automate the whole restructuring process using REVOLVER . `atts` is used to profile, compile, execute and hence gather attribute information on a source file for use with REVOLVER . `todos` is a list of to-do's which does everything `atts` does *and* goes on to invoke REVOLVER and perform restructuring in fully-automatic mode.

xtutest.c is a stand-alone program to perform Student's t-Test as described in the thesis.

Demo programs that came with *Sage*$^{++}$ are in the `demos` directory above `REV_HOME`.

A log-in C-shell file exists under `sage++-1.7` called `sage.login` which sets up all the path names needed to run *Sage*$^{++}$ (on a SUN Workstation anyway).

All the other code is pretty self-explanatory.

Bibliography

[1] Aho, A.V, Sethi R., and Ullman, J.D, *Compilers: Principles, Techniques and Tools*, Addison-Wesley (2nd Edition), (1986).

[2] Allen, R., and Kennedy, K., *Automatic Translation of Fortran Programs to Vector Form*, ACM Trans. on Programming Languages and Systems (ACM-TOPLAS), Vol 9, 4, pgs 491-542, Oct (1987).

[3] Almasi, G., and Gottlieb, A., *Highly Parallel Computing*, (2nd Edition), Benjamin/Cummins Publishing, (1994).

[4] Amdahl, G., *The Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, in Proceedings of AFIPS, Vol 30, pgs 483-485, (1967).

[5] Bäck, T., *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*, Oxford University Press, (1996).

[6] Bacon, D.F, Graham, S.L., and Sharp, O.J., *Compiler Transformations for High-Performance Computing*, ACM Computing Surveys, Vol 26, No. 4, pgs 345-420, Dec (1994).

[7] Balasundaram, V., and Kennedy, K., *A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations*, in "Proceedings of ACM SIGPLAN PLDI '89", Portland, Oregon, June 21-23, 1989, SIGPLAN Notices Vol 24, 7, pgs 41-53, July (1989).

[8] Balasundaram, V., Fox, G., Kennedy, K., and Kremer, U., *A Static Performance Estimator to Guide Data Partitioning Decisions*, Proceedings of 3rd ACM SIGPLAN Symp. ACM-PPOPP, April 21-24, pgs 213-223, (1991).

[9] Banerjee, U., *Dependence Analysis for Supercomputing*, Kluwer Academic Press, (1988).

[10] Banerjee, U., *A Theory of Loop Permutations*, in Proceedings of 2nd Workshop on Advances in Languages and Compilers for Parallel Computing, Urbana (IL), Aug., 1989, pgs 54-74, Research Monographs in Parallel and Distributed Computing, Cambridge : MA, MIT Press, (1989).

[11] Banerjee, U., *Unimodular Transformations of Double Loops*, in Proceedings of 3rd Workshop on Advances in Languages and Compilers for Parallel Computing, Irvine (CA), Aug., 1990, pgs 192-219, Research Monographs in Parallel and Distributed Computing, Cambridge : MA, MIT Press, (1990).

[12] Banerjee, U., *Loop Transformations for Restructuring Compilers: The Foundations*, Norwell, Mass: Kluwer Academic Publishers, (1993).

[13] Banerjee, U., *Loop Parallelization*, Norwell, Mass: Kluwer Academic Publishers, (1994).

[14] Banerjee, U., Eigenmann, R., Nicolau, A., and Padua, D.A., *Automatic Parallelization*, Center for Supercomputing Research and Development (CSRD), Tech. Report CSRD-1250, Urbana-Champaign, IL, Feb., (1993).

[15] Banerjee, U., Kuck, D.J., Chen, S.C., & Towle, R.A., *Time and Parallel Processor Bounds for Fortran-like Loops*, IEEE Transactions on Computers, C-28, 9, pgs 660-670, Sept. (1979).

[16] Beasley, D., Bull, D.R., and Martin, R.R., *An Overview of Genetic Algorithms: Part 1 - Fundamentals*, University Computing, 15 (2), pgs 58-69, (1993).

[17] Beasley, D., Bull, D.R., and Martin, R.R., *An Overview of Genetic Algorithms: Part 2 - Research Topics*, University Computing, 15 (4), pgs 170-181, (1993).

[18] Beguelin, A., Dongarra, J., Geist, G.A., Manchek, R., Jiang, W., and Sunderam, V., *A User's Guide to PVM (Parallel Virtual Machine) Version 3.0*, Tech. Report ORNL/TM-12187, May, (1994).

[19] Benkner, S., Andel, S., Blasko, R., Brezany, P., Celic, A., Chapman, B.M., Egg, M., Fahringer, T., Hulman, J., Kelc, E., Mehofer, E., Moritsch, H., Paul, M., Sanjari, K., Sipkova, V., Velkov, B., Wender, B., and Zima, H.P., *Vienna Fortran Compilation System Version 1.2 - User's Guide*, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, Feb (1996).

[20] Bernstein, A.J., *Analysis of Programs for Parallel Processing*, IEEE Trans. on Computers, EC-15, 5, pgs 757-763, Oct (1966).

[21] Bik, A.J.C, and Wijshoff, H.A.G., *Implementation of Fourier-Motzkin Elimination*, Tech. Report TR94-42, Department of Computer Science, Leiden University (NL), (1994).

[22] Burke, M., and Cytron, R., *Interprocedural Dependence Analysis and Parallelization*, "Proceedings of ACM SIGPLAN '86 Symposium on Compiler Construction", SIGPLAN Notices Vol 21, 7, pgs 162-175 July (1986).

[23] Blume, W., Eigenmann, R., Hoeflinger, J., Padua, D.A., Petersen, P., Rauchwerger, L., and Tu, P., *Automatic Detection of Parallelism : A Grand Challenge for High Performance Computing*, Center for Supercomputing Research and Development (CSRD), TR-1348, (1994).

[24] Blume, W., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, W., Rauchwerger, L., Tu, P.,

and Weatherford, S., *Effective Automatic Parallelization with 'POLARIS'*, Center for Supercomputing Research and Development (CSRD), Tech. Report, TR-1442, (1995).

[25] Brandes, T., *Automatic Vectorisation for High-Level Languages Based on an Expert System*, in "Lecture Notes in Computer Science no. 237, CONPAR86", pgs 303-310, (1986).

[26] Callahan, D., *The Program Summary Graph and Flow-Sensitive Interprocedural Dataflow Analysis*, in "Proceedings of ACM SIGPLAN PLDI '88", Atlanta, Georgia, June 22-24, 1988, SIGPLAN Notices Vol 23, 7, pgs 47-57, July (1988).

[27] Chang, W-L., and Chu, C-P., *The Generalized Lambda Test*, in Proceedings of 12th International Symposium and 9th Symposium on Parallel Distributed Processing (IPPS/SPDP-98), March 30th to April 3rd, Orlando, FL, (pgs 181-187), IEEE Computer Science Press, (1998).

[28] Chen, Ding-Kai, and Yew, Pen-Chung, *An Empirical Study on DoAcross Loops*, Supercomputing '91, Albuquerque, NM, pgs 620-632, Nov., (1991).

[29] Choi, Jong-Deok, Cytron, R., Ferrante, J., *On the Efficient Engineering of Ambitious Program Analysis*, IEEE Transactions on Software Engineering, 20, 2, (1994).

[30] Clement, Mark J., *Analytical Performance Prediction of Data Parallel Programs*, Ph.D Thesis. Oregon State University, June, (1995).

[31] Clement, Mark J., and Quinn, Michael J., *Analytical Performance Prediction on Multicomputers*, in Proceedings of Supercomputing '93, Jan., (1994).

[32] Cohen, J., *Computer-Assisted Microanalysis of Programs*, Communications of the ACM, 25, 10, pgs 724-733, Oct, (1982).

[33] Collard, Jean-François, *Code Generation in Auomatic Parallelization*, Laboratoire de l'Informatique du Parallélisme (LIP), Ecole National et Supérieure de Lyon, FR, Research Report 93-21, July (1993).

[34] Crooks, Philip, *An Automatic Program Translator for Distributed Memory MIMD Machines*, Ph.D Thesis, Dept. of Computer Science, Queen's University of Belfast, Northern Ireland, Nov. (1994).

[35] Cytron, R., *Efficiently Computing Static Single Assignment Form and The Control Dependence Graph*, IEEE Transactions on Programming Languages and Systems, pgs 451-490, October, (13), 4, (1991).

[36] Darema, F., George, D.A., Norton, A., and Pfister, G., *A Single-Program Multiple Data Computational Model for EPEX Fortran*, Parallel Computing, 7, pgs 11-24, (1988).

[37] Davis, L. (Ed.), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, NY, (1991).

[38] Delves, M, and Zima, H., *High Performance Fortran: A Status Report - or - Are We Ready to Give Up MPI Yet?*, in Proceedings of 5th European PVM/MPI Users' Group Meeting, Liverpool, UK, Sept. 7-9, 1998, (Eds. V. Alexandrov and J.J.Dongarra), Springer, LNCS 1497, pgs 161-171, Sept., (1998).

[39] Dorigo, M., and Bertoni, A., *Implicit Parallelism in Genetic Algorithms*, Artificial Intelligence, 61, (2), pgs 307-314, (1993).

[40] Dowd, K., *High Performance Computing*, O'Reilly and Associates Inc., (1993).

[41] Dowling, M., *Optimal Code Parallelization Using Unimodular Transformations*, Parallel Computing, pgs 157-171, 16, (1990).

[42] Eiben, A.E., Raue, P-E., and Ruttkay, Zs., *Solving Constraint Satisfaction Problems Using Genetic Algorithms*, in Proceedings of 1st IEEE International Conference on Evolutionary Computation (IEEE-ICEC 1), IEEE Service Center, Piscataway, NJ, Vol 2., pgs 542-547, Orlando 27-29, June, (1994).

[43] Executive Office of the President of the United States of America: Office of Science and Technology Policy, *A Research and Development Strategy for High Performance Computing*, Nov., (1987).

[44] Fahringer, T., *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers*, Ph.D Thesis, University of Vienna, Austria, Sept., (1993).

[45] Fahringer, T., *The Weight Finder - An Advanced Profiler for Fortran Programs*, in *Automatic Parallelization*, (Ed. C. Kessler), Vieweg-Verlag, Wiesbaden, (1994).

[46] Fahringer, T., *Estimating and Optimizing Performance for Parallel Programs*, IEEE Computer, 28(11), Nov., (1995).

[47] Ferrante, J., Ottenstein, K.J., & Warren, J.D., *The Program Dependence Graph and its Use in Optimization*, ACM Transactions on Programming Languages and Systems (ACM-TOPLAS), 9, 3, July (1987).

[48] Feautrier, P., *Compilng for Massively Parallel Architectures: A Perspective*, Microprocessing and Microprogramming, 41, pgs 425-439, (1995).

[49] Flynn, M.J., *Very High-Speed Computing Systems*, Proceedings of IEEE, 54, 12, pgs 1901-1909, Dec., (1966).

[50] Graham, Susan L., Lucco, S., and Sharp, Oliver J., *Orchestrating Interactions Among Parallel Computations*, in Proceedings of the

SIGPLAN Conference on Programming Languages Design and Implementation (Albuquerque, New Mexico, June), SIGPLAN Not. 28, 6, pgs 100-111, (1993).

[51] Gorges-Schleuter, M., *ASPARAGOS: An Asynchronous Parallel Genetic Optimization Strategy*, in Proceedings of 3rd International Conference on Genetic Algorithms (ICGA-3), (Ed. J. Schaffer), pgs 422-427, Morgan Kaufman Publishers, San Mateo, CA, (1989).

[52] Goff, G., Kennedy, K., & Tseng, C-W., *Practical Dependence Testing*, in Proceedings of the SIGPLAN Conf. on Programming Language Design and Implementation (ACM-PLDI '91), (Toronto, Ontario, June), SIGPLAN Not. 26, 6, pgs 15-29, (1991).

[53] Goldberg, D. E., *Genetic Algorithms in Search, Optimization and Machine-Learning*, Addison-Wesley, (1989).

[54] Goldberg, D. E., and Deb, Kalyanmoy, and Korb, B., *Do Not Worry, Be Messy*, in Proceedings of the 4th International Conference on Genetic Algorithms (ICGA-4), pgs 24-30, Morgan Kaufman Publishers, San Mateo, CA, (1991).

[55] Goldberg, D. E., and Deb, Kalyanmoy, *mGA in C: A Messy Genetic Algorithm in C*, Illinois Genetic Algorithms Laboratory (IlliGAL), Report No. 91008, Department of General Engineering, University of Illinois at Urbana-Champaign, Sept., (1991).

[56] Griebl, M., and Lengauer, C., *The Loop Parallelizer: LooPo*, in Proceedings of 6th Workshop on Compilers for Parallel Computers (Ed. M. Gerndt), Forschungzentrum Jülich, pgs 311-320, 21, (1996).

[57] Griebl, M., and Lengauer, C., *On Scanning Space-Time Mapped WHILE Loops*, in *Parallel Processing: CONPAR 94 - VAPP VI*, (Eds. B. Buchberger and J. Volkert), Lecture Notes in Computer Science 854, pgs 677-688, Springer-Verlag, (1994).

[58] Griebl, M., Lengauer, C., *On the Space-Time Mapping of WHILE Loops*, Parallel Processing Letters, 4, 3, pgs 221-232, Sept., (1994).

[59] Gruau, Frédéric, Ratajszczak, J-Y., and Wiber, G., *A Neural Compiler*, Theoretical Computer Science, 1834, pgs 1-52, Elsevier, (1994).

[60] Gustafson, John L., *Reevaluating Amdahl's Law*, Communications of the ACM, 31, 5, pgs 532-533, May (1988).

[61] Hirst, Anthony J., *The Structure and Transformation of Landscapes* in Proceedings of Artificial Intelligence and Simulation of Behaviour (AISB) International Workshop on Evolutionary Computation, 7-8th April, Manchester, UK, (Eds. D. Corne and J. Shapiro), pgs 19-24, Springer LNCS-1305, (1997).

[62] Hendren, L. J., Donowa, C., Emami, M., Gao, G. R., Justiani, and Sridharan, B., *Designing the McCat Compiler Based on a Family of Structured Intermediate Representations*, in Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, (Eds. Utpal Banerjee, David Gelertner, Alex Nicolau, and David Padua), No. 757 in lecture Notes in Computer Science, New Haven, Connecticut, pgs 406-420, Springer-Verlag, August 3-5, 1992, published (1993).

[63] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, (1985).

[64] Hockney, R.W., and Jesshope, C., *Parallel Computers 2*, (2nd Edition), Adam Hilger, UK, (1988).

[65] Hoffmeister, F., and Bäck,, T., *Genetic Algorithms and Evolutionary Strategies: Similarities and Differences*, Tech. Report SYSD-1/92, Department of Computer Science, University of Dortmund, Feb. (1992).

[66] Holland, J. H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbour, MI, (1975).

[67] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 1.0*, Tech. Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston: TX, (1993).

[68] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*, Tech. Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston: TX, Jan. 31st, (1997).

[69] *International Journal of Supercomputing Applications and High-Performance Computing, Special Issue - MPI: A Message Passing Interface Standard*, Vol 8, No. 3/4, (ISSN 0890-2720), Fall/Winter, MIT Press, (1994).

[70] Karp, A., Miler, R., and Winograd, S., *The Organization of Computations for Unification Recurrence Equations*, Journal of the ACM, pgs 563-590, 14, 3, (1967).

[71] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., and Wonnacott, D., `Petit` *Version 1.00 User Guide*, Omega Project, Dept. of Computer Science, University of Maryland, April (1996).

[72] Kelly, W.A., *Optimization within a Unified Transformation Framework*, Ph.D Thesis, Dept. of Computer Science, (CS-TR-3725), University of Maryland, (1996).

[73] Kennedy, K., Hiranandani, S., and Tseng, C.W., *Compiling Fortran-D for MIMD Distributed Memory Machines*, Communications of ACM, 35, 8, pgs 1166-1180, Aug (1992).

[74] Kennedy, K., McKinley, K.S., and Tseng, C-W., *Analysis and Transformation in an Interactive Programming Tool*, Concurrency Practice and Experience, Vol 5, 7, pgs 575-602, Oct (1993).

[75] Kennedy, K., McIntosh, N., and McKinley, K.S., *Static Performance Estimation in a Parallelizing Compiler*, Tech. Report CRPC-TR92204-S (2nd Revision), Center for Research on Parallel Computation, Rice University, Oct., (1993).

[76] Knuth, Donald E., *The Art of Computer Programming - Volume 3: Sorting and Searching*, Addison-Wesley, (Reading, MA), (1973).

[77] Kremer, Ulrich, and Kennedy, Ken, *Automatic Data Layout for High Performance Fortran*, in Proceedings of the Workshop on Automatic Data Layout and Performance Prediction - Rice University, April 19-21, (1995).

[78] Kuck, D.J., *A Survey of Parallel Machine Organisation and Programming*, ACM Computing Surveys, Vol 9, No. 1, pgs 29-59, Mar. (1977).

[79] Kuck, D.J., Davidson, E.S., Lawrie, D.H., and Sameh, A.H., *Parallel Supercomputing Today and the Cedar Approach*, in "Experimental Computing Architectures" - (Ed. Jack Dongarra), Elsevier (North-Holland), (1987).

[80] Kuhn, R.H., Leasure, B., and Shah, S.M., *The KAP Parallelizer for DEC Fortran and DEC C Programs*, Digital Technical Journal, Vol 6, 3, Summer 1994, pgs 57-70, (1994).

[81] Lam, M., notification posted to `suif-talk` mailing-list, 29th May, (1997).

[82] Lamport, L., *The Parallel Execution of DO-loops*, Communications of the ACM, Vol 17, 2, pgs 83-93, Feb (1974).

[83] Lin, H-S., Xiao, J., and Michalewicz, Z., *Evolutionary Algorithms for Path Planning in a Robot Environment*, in Proceedings of 1st IEEE International Conference on Evolutionary Computation (IEEE-ICEC 1), IEEE Service Center, Piscataway, NJ, Vol 1., pgs 211-216, Orlando 27-29, June, (1994).

[84] Lin, H-S., Xiao, J., and Michalewicz, Z., *Evolutionary Navigator for A Mobile Robot*, in Proceedings of IEEE Conference on Robotics and Automation, pgs 2199-2204, IEEE Computer Society Press, New York, (1994).

[85] Le Verge, H., Mauras, C., and Quinton, P., *The* ALPHA *Language and it's Use for the Design of Systolic Arrays*, Journal of VLSI Signal Processing, pgs 173-182, 3, 3, Sept., (1991).

[86] Li, Z., Yew, P. & Zhu, C., *Data Dependence Analysis on Multi-Dimensional Array References*, IEEE Transactions on Parallel and Distributed Systems, 1, 1, pgs 26-34, (1990).

[87] Lu, B., and Mellor-Crummey, J., *Compiler Optimization of Implicit Reductions for Distributed Memory Multiprocessors*, in Proceedings of 12th International Symposium and 9th Symposium on Parallel Distributed Processing (IPPS/SPDP-98), March 30th to April 3rd, Orlando, FL, (pgs 42-51), IEEE Computer Science Press, (1998).

[88] Lu, L-C., *A Unified Framework for Systematic Loop Transformations*, in Proceedings of 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Williamsburg, VA, April, pgs 28-38, (1991).

[89] Mace, M.E., & Wagner, R.A., *Globally Optimum Selection of Memory Storage Patterns*, in Proceedings of the International Conference on Parallel Processing, (Ed. D. DeGroot), IEEE Computer Society, Washington D.C., pgs 264-271, (1985).

[90] Mace, M.E., *Memory Storage Patterns in Parallel Processing*, Kluwer Academic Press, Norwell: MASS, (1987).

[91] Mansour, N., and Fox, G.C., *A Hybrid Genetic Algorithm for Task Allocation in Multicomputers*, in Proceedings of the 4th International Conference on Genetic Algorithms (ICGA-4), pgs 466-473, Morgan Kaufman Publishers, San Mateo, CA, (1991).

[92] Megson, G.M, and Bland, I.M., *Generic Systolic Array for Genetic Algorithms*, IEE Proc. Computers and Digital Techniques, 144, 2, pgs 107-121, Mar., (1997).

[93] Megson, G.M., and Chen, X., *Automatic Parallelization for a Class of Regular Computations*, World Scientific, (1997).

[94] *Computing Surface 1: Hardware Reference Manual*, Meiko Ltd, (1989).

[95] *CSTools for SunOS*, Vols I and II, Meiko Ltd, (1989).

[96] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Esprit Project P6643(PPPE) (1994).

[97] Mühlenbein, H., *Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization*, in Proceedings of 3rd International Conference on Genetic Algorithms (ICGA-3), (Ed. J. Schaffer), pgs 416-421, Morgan Kaufman Publishers, San Mateo, CA, (1989).

[98] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, New York, 3rd Edition, (1996).

[99] Montana, D.J., and Davis, L., *Training Feedforward Neural Networks Using Genetic Algorithms*, in Proceedings of the 1989 International Joint Conference on Artificial Intelligence (IJCAI-89), Morgan Kaufmann Publishers, San Mateo, CA, (1989).

[100] Muchnik, Steven S., and Jones, Neil D., (Eds.), *Program Flow Analysis: Theory and Applications*, Englewood Cliffs: N.J: Prentice-Hall, (1981).

[101] Nisbet, A., *GAPS: A Compiler Framework for Genetic Algorithm Optimised Parallelisation*, published poster in High-Performance Computing and Networking in Europe 1998 (HPCNE-98), April 21-23, Amsterdam, NL, (1998).

[102] Nisbet, A., *GAPS: Genetic Algorithm Optimised Parallelisation*, invited presentation - 7th Workshop on Compilers for Parallel Computing, Linkoeping, Sweden, June, (1998).

[103] Novak, S. and Nicolau, A., *Mutation Scheduling : A Unified Approach to Compiling for Fine-Grain Parallelism*, Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science No. 1033, Springer, pgs 289-303, (1995).

[104] Nudd, G.R., Papaefstathiou, E., Kerbyson, D.J., Atherton, T.J., and Harper, J.S., *An Introduction to the Layered Characterisation of High Performance Systems*, Tech. Report CS-RR-335, Computer Science Dept., University of Warwick, December, (1997).

[105] Pacheco, P.S., *Parallel Programming with MPI*, Morgan Kauffman Publishers, (1997).

[106] Padua, D.A., Kuck, D.J., and Lawrie, D.H., *High-Speed Multiprocessors and Compilation Techniques*, IEEE Trans. on Computers, Vol C-29, 9, pgs 763-776, Sept (1980).

[107] Padua, D.A., and Wolfe, M.J., *Advanced Compiler Optimizations for Supercomputers*, Communications of the ACM, 29, 12, pgs 1184-1200, Dec (1986).

[108] Perrott, R.H., *Parallel Programming*, Addison-Wesley, (1987).

[109] Perrott, R.H. (Ed.), *Software for Parallel Computers*, Chapman and Hall, (1992).

[110] Pfister, G.F., *In Search of Clusters : The Coming Battle in Lowly Parallel Computing*, Prentice-Hall, (1995).

[111] Press, William H., Teukolsky, Saul A., Vetterling, William T., and Flannery, Brian P., *Numerical Recipes in C : The Art of Scientific Computing*, (2nd. Edition), Cambridge University Press, (1992).

[112] Press, William H., Teukolsky, Saul A., Vetterling, William T., and Flannery, Brian P., *Numerical Recipes in Fortran 77 : The Art of Scientific Computing*, (2nd. Edition), Cambridge University Press, (1992).

[113] Pugh, W.H., *A Practical Algorithm for Exact Array Dependence Analysis*, Communications of the ACM, Vol 35, 8, pgs 102-114, Aug (1992).

[114] Pugh, W. H., *Uniform Techniques for Loop Optimization*, Proceedings of the ACM Conference on Supercomputing, (Cologne, Germany), June, 1991, pgs 341-352, ACM Press, New York, (1991).

[115] Pugh, W., and Wonnacott, D., *Going Beyond Integer Programming with the Omega Test to Eliminate False Data Dependencies*, ACM-SIGPLAN PLDI '92, Dec (1992).

[116] Quinn. M.J., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, (1987).

[117] Quinn, M.J., *Parallel Computing: Theory and Practice*, McGraw-Hill, (1994).

[118] Rechenberg, R., *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*, Frommann-Holzboog, Stuttgart, (1973).

[119] Smith, S.F., *A Learning System Based on Genetic Algorithms*, Ph.D Dissertation, University of Pittsburgh, (1980).

[120] Spiegel, M.R., *Theory and Problems of Statistics*, (2nd Edition), Schaum's Outline Series, McGraw-Hill, (1994).

[121] Steed, Mark R., and Clement, Mark J., *Performance Prediction of PVM Programs*, in Proceedings of 10th International Parallel Processing Symposium, April, (1996).

[122] Syswerda, G., *Uniform Crossover in Genetic Algorithms*, in Proceedings of 3rd International Conference on Genetic Algorithms (ICGA-3), (Ed. J. Schaffer), pgs 2-9, Morgan Kaufman Publishers, San Mateo, CA, (1989).

[123] Tarjan, R.J., *Depth-First Search and Linear Graph Algorithms*, SIAM Journal of Computing, Vol 1, No. 2, June (1972).

[124] Tenny, L.J., *Rule-based Program Restructuring for High-Performance Parallel Processor Systems*, Ph.D Thesis, Indiana University (1992).

[125] Thompson, A., *Creatures From Primordial Silicon*, pgs 30-34, New Scientist, No. 2108, Nov., (1997).

[126] Towle, R.A., *Control and Data Dependence For Program Transformations*, PhD. Thesis, Tech. Report 76-788, Computer Science Dept., University of Illinois at Urbana-Champaign, (1976).

[127] van Gemund, Arjan J.C., *Compile-Time Performance Prediction of Parallel Systems*, in Proceedings of 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Heidelberg, (DE), Sept. 20-22, (1995).

[128] Walsh, P., and Ryan, C., *Automatic Conversion of Programs from Serial to Parallel Using Genetic Programming - The PARAGEN*

System, in Proceedings of Parallel Computing, PARCO-95, Universitëit Gent, Belgium, 19-22 September, (1995).

[129] Wang, K-Y., and Gannon, D.P., *Applying AI Techniques to Program Optimization for Parallel Computers*, in "Parallel Processing for Supercomputers and Artificial Intelligence", (Kai Hwang and Dennis DeGroot, Eds.), Chapter 12, pgs 441-485, McGraw-Hill, (1989).

[130] Whitfield, D., Soffa, M.L., *An Approach to Ordering Optimizing Transformations*, Proceedings of 2nd ACM Symposium on Principles and Practice of Parallel Programming (ACM-PPOPP), Seattle, Washington, March 14-16, pgs 137-147, Mar., (1990).

[131] Whitley, D., *GENITOR II: A Distributed Genetic Algorithm*, Journal of Experimental and Theoretical Artificial Intelligence, Vol. 2, pgs 189-214, (1991).

[132] Wilde, Dorian K., *A Library for Doing Polyhedral Operations*, IRISA, Rennes (FR), Projet API, Publication Interne 785, Dec., (1993).

[133] Williams, H.P., *Fourier-Motzkin Elimination Extension to Integer Programming Problems*, Journal of Combinatorial Theory (A), 21, pgs 118-123, (1976).

[134] Williams, K.P., *Implementation of an Efficient and Powerful Parallel Pseudo-random Number Generator*, 2nd Euro-PVM User Group Meeting, Lyons, FRA, (Sept. 13-15), (pgs 197-202), Hermes Press, Paris, France (1995).

[135] Williams, K.P., and Williams, S.A., *Genetic Compilers : A New Technique for Automatic Parallelisation*, in "Parallel Programming Environments for High Performance Computing", Proceedings of the 2nd European School of Parallel Processing Environments

(ESPPE-96), IMAG-INRIA, L'Alpe d'Huez, France, April 1-5, pgs 27-30, (1996).

[136] Williams, K.P., Williams, S.A., and Mitchell, P.C.H, *A Cluster Computing Implementation of a Particle Growth Simulator*, Parallel Computing Conference (PARCO-95), Universitëit Gent, (BEL), Sept. 19-22, (1995).

[137] Williams, S.A., Mitchell, P.C.H., Fagg, G.E., and Williams, K.P., *A Distributed Parallel Computing Workbench for Modelling of the Oxygenation Process*, PVM User-Group Meeting, Pittsburgh: PE, May 7-9, (1995).

[138] Williams, S.A., *Programming Models for Parallel Systems*, John Wiley and Sons, (1990).

[139] Wilson, R., French,, R., Wilson, C., Amarasinghe, S., Anderson, J., Tjiang, S., Liao, S-W., Tseng, C-W., Hall, M., Lam, M., and Hennessey, J.L., *An Overview of the SUIF Compiler System*, Computer Systems Lab, Stanford University, (1993).

[140] Wolf, M.E., and Lam, M.S., *A Loop Transformation Theory and an Algorithm to Maximize Potential Parallelism*, IEEE Trans. on Parallel and Distributed Systems (IEEE-PADS), pgs 1-38, Oct., (1991).

[141] Wolfe, M.J., *Optimizing Supercompilers for Supercomputers*, Pitman Publishing, (1989).

[142] Wolfe, M.J., *The* Tiny *Loop Restructuring Research Tool*, in Proceedings of the 1991 International Conference on Parallel Processing, Chicago: IL, (1991).

[143] Wolfe, M.J., & Tseng, C-W., *The Power Test for Data Dependence*, IEEE Transactions on Parallel and Distributed Systems, 3, 5, pgs 591-601, Sept. (1992).

[144] Wolfe, M.J., *High Performance Compilers for Parallel Computing*, Addison-Wesley, (1996).

[145] Wolpert, David H., and Macready, William G., *No Free Lunch Theorems for Search*, Santa Fe Institute - Working Report 95-02-010, (1996).

[146] Wolpert, David H., and Macready, William G., *No Free Lunch Theorems for Optimization*, IEEE Trans. on Evolutionary Computation (IEEE-EC), Vol 1, 1, pgs 67-82, April, (1997).

[147] Zima, H., and Chapman, B., *Supercompilers for Parallel and Vector Computers*, ACM Press, Frontier Series, (1991).