LABORATORY FOR CONCURRENT COMPUTING SYSTEMS

COMPUTER SYSTEMS ENGINEERING
School of Electrical Engineering
Swinburne Institute of Technology
John Street, Hawthorn 3122, Victoria, Australia.

Implementation of a Numerical Weather Prediction Model in SISAL

Technical Report 31-017

P.S. Chang
pau@stan.xx.swin.oz(.au)

G.K. Egan
gke@stan.xx.swin.oz(.au)

This technical report is a reformatted version of a thesis submitted by Pau S. Chang to RMIT Victoria University of Technology (Royal Melbourne Institute of Technology) in partial fulfilment of the requirements for the degree of Master of Engineering in the Faculty of Engineering.

Abstract

This thesis presents a research which explored the use of the implicit parallel programming language SISAL in the formulation of a spectral barotropic numerical weather prediction model originally formulated in FORTRAN. A process of direct transliteration and refinement using loop transformation heuristics was used resulting in a highly parallel implementation on a MIMD multiprocessor. The performance analysis used in this process identified limiting factors due to the formulation of the application, language expressive power and language implementation. Solutions to these limitations are proposed.

Implementation of a Numerical Weather Prediction Model in SISAL

by

Pau Sheong Chang
B.Eng.(Electrical) University of Melbourne

RMIT Victoria University of Technology
(Royal Melbourne Institute of Technology)

1990

A thesis submitted to RMIT Victoria University of Technology (Royal Melbourne Institute of Technology) in partial fulfilment of the requirements for the degree of Master of Engineering in the Faculty of Engineering.

Abstract

This thesis presents the research which explored the use of the implicit parallel programming language SISAL in the formulation of a spectral barotropic numerical weather prediction model originally formulated in FORTRAN. A process of direct transliteration and refinement using loop transformation heuristics was used resulting in a highly parallel implementation on a MIMD multiprocessor. The performance analysis used in this process identified limiting factors due to the formulation of the application, language expressive power and language implementation. Solutions to these limitations are proposed.

Declaration

No portion of the work referred to in this thesis has been submitted to this or other university, college of advanced education, or other institute of learning in support of any other degree or qualification.

Acknowledgements

I am indebted to my supervisor, Professor Gregory Egan, Director of Laboratory for Concurrent Computing Systems at Swinburne Institute of Technology, for his guidance in all aspects of this research and for his contribution to the publications arising from the research.

I would like to thank all members of the Joint Royal Melbourne Institute of Technology and Commonwealth Scientific and Industrial Research Organisation Parallel Systems Architecture Project for their contribution to discussions on issues encountered in the research. In particular, I would like to thank Warwick Heath specifically for developing a concurrency profiling tool primarily for the research and for the discussions he contributed to on multiprocessing issues. I would also like to thank Dr. David Abramson for his help in my applications for financial support, in particular the CSIRO Vacation Student Scholarship. Special thanks must also go to Dr. Ian Simmonds and Dr. Ian Smith of the Department of Meteorology at the University of Melbourne for providing access to, and helping with the interpretation of, the original FORTRAN version of the weather model.

Finally, my deepest thanks go to Angeline, my parents, and my eldest sister and brother in law, to whom I dedicate this thesis, for their patience, constant support and love.

This research was funded, in part, by RMIT Victoria University of Technology (Royal Melbourne Institute of Technology), Commonwealth Scientific and Industrial Research Organisation and in the last phase, Swinburne Institute of Technology.

CONTENTS

Chapter 1 Introduction 1.1

1.1 Language Issues 1.2

1.2 Application Studies 1.2

1.3 Research Formulation 1.3

1.4 Research Aims 1.3

1.5 Thesis Outline 1.3

Chapter 2 Descriptions of Language, Compiler, Multiprocessor and Model for Application Study 2.1

2.1 Parallel Programming Language SISAL 2.1

2.2 Optimising SISAL Compiler (OSC) 2.3

2.3 Encore Multimax Multiprocessor 2.5

2.4 Application Study: Spectral Barotropic Numerical Weather Prediction Model 2.6

2.4.1 Mathematical Description of the Model 2.7

2.4.2 Original Sequential Implementation: Fortran Realisation 2.11

Chapter 3 Implementations 3.1

3.1 Direct Transliteration Approach 3.1

3.1.1 Encountered Problems 3.1

3.1.1.1 Late Availability of OSC 3.2

3.1.1.2 Debugging SISAL Programs 3.2

3.1.1.3 Accessing Out of Bound Array Elements 3.2

3.1.1.4 Keeping Track with New Variable Names 3.3

3.1.1.5 OLD Statements of SISAL: an Easy Mistake 3.3

3.1.2 Separate Parallel and Sequential Loops of SISAL 3.3

3.1.3 Typical Mapping from FORTRAN to SISAL 3.3

3.1.4 Results of Implementation 3.5

3.2 Parallel Implementations 3.6

3.2.1 Heuristic of Transforming a Sequential Loop to a Parallel Loop in SISAL 3.6

3.2.2 Transformation of Individual Child Functions to Product Form Loops 3.7

3.2.2.1 Parallelisation of Functions Outside Subroutine *Nonlinear* 3.8

3.2.2.2 Parallelisation of Common Functions Nested in *Nonlinear* 3.8

3.2.2.3 Parallelisation of Function *FreqToSpec* Nested in *Nonlinear* 3.10

3.2.3 *Globalisation* of Major Functions: Further Parallelisation 3.10

3.2.3.1 Relocation of *SymAsym* to Model Initialisation Section 3.11

3.2.3.2 *Globalisation* of Common Functions Nested in *Nonlinear* 3.11

3.2.3.3 *Globalisation* of Function *FreqToSpec* Nested in *Nonlinear* 3.12

3.2.4 Resultant Computational Algorithm from Parallel Realisation 3.18

3.2.5 Monitoring Bottlenecks with Concurrency Profiles 3.18

3.2.5.1 Sequential Code Sections in Timeloop 3.18

3.2.5.1.1 Singly Nested Parallel Loop with Small Loop Body 3.19

3.2.5.1.2 *Quasi Doubly Nested* Technique 3.20

3.2.5.2 Sequential Code Section in Initialisation Section 3.21

3.2.5.2.1 Serial Implementation 3.21

3.2.5.2.2 Parallel Implementation 3.22

3.2.6 Results from Final Implementation 3.23

Chapter 4: Performance Analysis 4.1

4.1 General Execution Time Curves for Varying Model Sizes 4.1

4.2 Analysis of Timeloop Performance 4.2

4.2.1 General Speedup Profiles for Varying Model Sizes 4.2

4.2.2 Analysis in terms of Model Sizes 4.4

4.2.3 Speedup Analysis from Benchmark Ratios 4.5

4.3 Effect of Memory Deallocation Overhead 4.6

4.3.1 Investigations for Dynamic Memory Management Scheme of OSC 4.7

4.3.1.1 Direct Transliteration from FFT in C 4.7

4.3.1.2 ABD Synthesis for Effect of Memory Management Scheme of OSC Runtime 4.9

4.3.2 Proposals for Better Dynamic Memory Management Schemes 4.13
4.3.2.1 Fixed Array Sizes Through Loop Iterations 4.13
4.3.2.2 Varying Array Sizes Through Loop Iterations 4.14

Chapter 5: Final Discussions and Conclusions 5.1
5.1 Programming in SISAL 5.1
5.1.1 Parallelism in SISAL 5.1
5.1.2 Parallelising at SISAL Level 5.2
5.1.2.1 Loop Transformation Heuristics for SISAL 5.3
5.1.3 Debugging SISAL Programs 5.3
5.1.4 Problems of SISAL 5.3
5.1.4.1 Language Support for Complex Numbers 5.3
5.1.4.2 Matrices in SISAL 5.5
5.1.4.2.1 Dope Vector Scheme: Declaration of Matrices as Arrays of Arrays 5.5
5.1.4.2.2 Matrix Scheme: Declaration of Matrices as Matrices 5.6
5.2 Problems of OSC 5.7
5.2.1 Exploitation of Coarser Grain Parallelism for Conventional Multiprocessors 5.7
5.2.2 Effective Loop Slicing 5.8
5.2.2.1 Programmer's Effort: QDN 5.8
5.2.2.2 Better Cost Estimation Routine for OSC's Runtime System 5.9
5.2.2.3 Modular Compilation with Local Cost Pragmas 5.9
5.2.3 Dynamic Memory Management Scheme 5.9
5.3 Numerical Weather Prediction Model 5.10
5.4 Conclusions 5.11

Appendix A Bugs in OSC A.1

A.1 Starting Index of "**FOR** array **RETURNS VALUE OF CATENATE**" A.1

A.2 "**FOR** array **RETURNS VALUE OF CATENATE** of concatenations of vectors" A.3

A.3 Incomplete Graph Normalisation A.5

A.3.1 Joint Routines Which Produces *Normalisation Error* A.7

Appendix B SISAL Program of Spectral Barotropic Numerical Weather Prediction Model B.1

B.1 Parallel SISAL Version B.1

Appendix C Fast Fourier Transformation (FFT) Codes C.1

C.1 Original Code in C C.1

C.2 SISAL Version C.5

References R.1

Chapter 1

INTRODUCTION

With the advances of digital circuit technologies, the speed of computing has improved by many orders of magnitude in the past decades. Successive generations of supercomputers including most recently those produced by Cray Research, ETA, Fujitsu, NEC have employed architectures derived from the sequential computational model attributed to John von Neumann [AA82]. These machines have used a variety of schemes including vectorisation in the effort to overcome the fundamental limitations imposed by the speed of light. As these limits are approached the cost performance ratio resulting from these schemes has become progressively less attractive. In the pursuit of higher performance, super scalar RISC architectures are employing schemes pioneered in the CDC 6600 architecture in an effort to extract the last piece of gain from a single instruction stream. It is now widely recognised that none of these approaches by themselves will satisfy the needs for higher and higher computational rates and that the only avenue left is to exploit parallelism. In theory at least the technologies developed for high performance uniprocessors may be used to implement parallel processors although more conservative technologies may be used to achieve the same level of performance at significantly less cost.

Many forms of parallel computing models have emerged, all claiming to take advantage of the concurrency in algorithms. The validation of these claims is the subject of substantial research in parallel computers and parallel computing including further hardware design, parallel algorithm design and computational experiments. Although many issues are yet to be resolved, manufacturers are now committed to multiprocessor machine designs of one form or another. These machines include Cray, Connection Machine, NCUBE, Denelcor HEP, TERA, Intel iPSC, Ametek, Ultracomputer, RP3, SUPERNUM, GF-11, Transputer multiprocessor systems, Sequent Balance and Encore Multimax [McB88, EMC] as well as the unconventional dataflow design of Sigma-1, Manchester Dataflow Computer [GB88], MIT Monsoon and CSIRAC II [EWB90]. Some have been commercialised for research and real applications while others are design exercises for research purposes. While there is a strong commitment to parallel machines, issues relating to programming languages are largely unresolved. Not surprisingly, most research by manufacturers to date has been directed at machine specific languages. Given the volatility of machine development there is a strong awareness by users that languages, as far as possible, should be machine independent; in other words, there is a perceived need to decouple software investment from what may be transient architectures and manufacturers [CA88].

1.1 Language Issues

The common programming languages for parallel machines available today evolved from imperative programming languages such as FORTRAN and C. They explicitly express parallelism, and rely on source level annotation to guide their compilers.

However, these languages can limit the expression of concurrency due to their reliance on the sequential control constructs originally designed for a sequential model. Moreover, more complicated and explicit control over the runtime environment is required in both the expression and execution of an algorithm on a parallel machine, making these machines harder to program correctly, particularly in imperative languages. Additionally, parallel annotators for imperative languages, which conceal the parallelisation process from the user, are still at the stage of infancy since their syntax has not yet been established and properly standardised. This immaturity suggested at the beginning of this project that explicit parallelism was not a suitable path for this research. The product of these problems is the need to look at parallel programming languages designed primarily to explore parallelism on parallel machines.

Parallel languages, and functional languages such as MIRANDA [Turn86] and HASKELL [Has90, HP89], rely on their functional constructs to ease the definition of parallel processes. For most of these languages, their compilers identify parallelism not from the source programs but rather in the lower dataflow graph or combinator graph levels. This is particularly true for the functional applicative languages ID [Whi88, NPA86, Nik88] and SISAL [MS85].

ID is limited in its prospects of being *widely* used and researched because it has been designed exclusively for the Massachusetts Institute of Technology TTDA and Monsoon dataflow computers. SISAL, developed in a joint collaboration between Digital Equipment Corporation, the University of Manchester, Lawrence Livermore National Laboratory and Colorado State University, is the product of significant efforts between a major computer manufacturer, researchers and users. SISAL is claimed to be a general-purpose functional language that can run efficiently on sequential and parallel architectures including Sun, VAX, Sequent Balance, Cray-XMP, Alliant and the Encore Multimax [CO89, LSF88]. It has been demonstrated that SISAL programs can achieve sequential and parallel execution performance competitive with programs written in conventional languages. These claims have been supported by benchmark results including the Livermore Loops and other kernels which are widely recognised as representative benchmark codes [Feo87]. For the claims to be convincing there is a need to compare SISAL's favourable results on relatively small benchmarks with those obtained from a large application benchmark; this comparison has not been performed before.

1.2 Application Studies

The applications suitable for this purpose are large scientific computational models consisting of codes which have large amounts of inherent parallelism and are recognised to be computationally demanding. Numerous researches have indicated that computational fluid dynamics problems, such as weather simulation models, fit in this class of applications [AG, TB88]; numerical weather prediction is acknowledged to be of significant social and economic importance.

The availability of a weather code written in FORTRAN and access to the originating group in the Department of Meteorology at the University of Melbourne made the code a logical choice for this study.

1.3 Research Formulation

The definition of the model underlying the weather code was incomplete, only providing in broad outline the mathematical formulation for mathematical phases. The detailed inner workings of these phases, in particular the discrete formulation, and their inter-relationship were not available. To obtain this information it was necessary to achieve a detailed understanding of the computational model from a detailed analysis of the FORTRAN realisation of the model. As expected the actual structure of the numerical model had been obscured by the sequential formulation in FORTRAN. It was believed that the code may be representative of existing large scientific application FORTRAN codes; it is mature but was developed and refined by many people resulting in varying programming styles throughout the program. As there was, at the time of original development, no apparent gain in preserving any parallelism in the original model it was viewed as likely that the available parallelism in the formulation may be low; a parallelising FORTRAN compiler was not available to verify this conjecture directly.

Given that the primary documentation of the model was the FORTRAN code, it was decided to turn adversity into gain by translating the FORTRAN directly to SISAL. This process could at once allow the examination of the likely results to be obtained from direct transliteration while permitting the details of the mathematical model to be uncovered. The initial results of this approach may not be better, but it would be useful to know how much worse it could be in real applications. Having uncovered the model details, the prospect of a re-formulation in SISAL using its features to best effect would be possible.

Existing large application programs have had an irreplaceable amount of time invested in their development, maintenance and further development in imperative languages, particularly in FORTRAN. To promote the rewriting of these applications in SISAL will be difficult, especially as many computational algorithms will have to be redesigned from their original mathematics and physics in a manner which leaves the parallelism of the application intact while conforming to the expressive requirements of SISAL [JF87]. If the models are large or complicated, the truth that recoding the algorithms will be even more difficult will dampen the willingness of many scientists and researchers to adopt SISAL. As a consequence, there is a need to ease and encourage the adoption of SISAL by exploring other paths. One such path is to attempt to identify heuristics for effectively parallelising SISAL codes transliterated from FORTRAN, without the need to completely re-formulate the codes (computational algorithms) from the original mathematics and physics of the model.

1.4 Research Aims

This research aims to explore the use of a representative implicit parallel programming language, SISAL, in a large scientific application, being the spectral barotropic numerical weather prediction model, and in doing so, to explore the development of heuristics for refining SISAL codes produced by direct transliteration, and to identify limiting factors to performance due to the formulation of the application, language expressive power and language implementation.

1.5 Thesis Outline

This chapter aimed to illustrate the background factors leading to the definition of the research area. In Chapter 2, the parallel programming language SISAL and its compiler OSC for general purpose multiprocessors, the multiprocessor Encore Multimax and the weather simulation model adopted for the research will be described.

This is followed by illustrations of various implementations of the model in SISAL in Chapter 3. The first approach is a direct transliteration of the code from FORTRAN to

SISAL. The problems encountered in the approach, how the mapping was performed and the results of the implementation will be described. The discussions will lead to the next section which elaborates a parallel implementation of the model in SISAL from the resulting code of the direct transliteration, the heuristics for loop transformation and the issues relating to parallelisation of codes, parallel algorithms and the overheads of the OSC runtime system.

The results of the second approach will lead to detailed performance analysis of the final implementation in Chapter 4. The discussions will include performance of the code and the impact that the new parallel formulation of the spectral barotropic weather model could have on larger model sizes. In doing so various performance analysis techniques will be examined.

Finally in Chapter 5, the results of the research will be discussed in the context of the research aims.

Chapter 2

DESCRIPTION OF LANGUAGE, COMPILER, MACHINE AND APPLICATION STUDY

This section describes SISAL and its compiler OSC [Cann89] for shared memory multiprocessors, as well as the multiprocessor Encore Multimax [ECC] and the numerical weather model adopted for the research.

2.1 Parallel Programming Language SISAL [MS85]

If intended for expressing parallelism, imperative programming languages, like FORTRAN and C, have a serious deficiency. They inherently reflect the storage structure of the von Neumann concept of computer organisation in that each language has some method of effecting a change in memory state that can create side effects in the entire program. Allowing the specification of global state changes, these languages lead to programs that are very difficult to analyse for concurrency. Without a complete analysis of the entire program, it is generally impossible to trace the flow of data. Only with such analysis is it possible to find and eliminate inessential constraints on the sequencing of parallel program parts. As a result, there is a need for a language which makes such analysis easy and effective. As discussed in Chapter 1, SISAL has been chosen for this investigation. The descriptive material below is drawn in part from [MS85, GB88].

The definition of SISAL was a cooperative effort of Lawrence Livermore National Laboratory, Digital Equipment Corporation, the University of Manchester and Colorado State University. A derivative of VAL [AD79], SISAL is a strongly typed, single assignment language intended for research in parallel scientific computation. It differs from VAL in possessing simpler error types, general recursion, a stream data type and improved iteration forms. SISAL syntax is Pascal-like (block structured), a familiar paradigm which claims to facilitate SISAL program readability [CO89] and ease the learning of the language.

SISAL stands for *Streams and Iterations in a Single Assignment Language*. Programs expressed in SISAL obey the semantic single assignment rule, where a name can be defined once only in a given scope. This feature exposes data dependency and helps control synchronization mechanism for SISAL programs. The language has no control constructs, such as *goto* statements and traditional loops. It is *functional* or *side effect free*,

the main advantage of which is the *locality* of variable names. In other words, there is no programming variables which depend upon the state of a computation. The concept of variable name is much closer to the idea of a mathematical variable [Sharp]. This and the structured feature of SISAL claim to make modifications to a SISAL program at a later time easy and safe by restricting the modifications to a limited number of modules.

SISAL is designed as a parallel language to express algorithms for execution on computers capable of highly concurrent operation. The application area it supports is specifically numerical computations that strain the limits of high performance machines. It is designed to enable parallelism to be implicitly expressed so that programmers are enlightened from the burden of job scheduling of multiple processors and synchronization.

Theoretically, on a dataflow machine for which SISAL was originally intended, the maximum parallelism of a SISAL program is only limited by data dependencies. In a multiprocessor environment, however, parallelism is currently exploited only from parallel loop constructs, and pipelines from stream constructs [DO89]; Figure 2.1 illustrates the definition of these two forms of concurrency. During the finalisation of this research project definition and in most parts of this research, the operations of streams were not yet completely implemented in OSC. As a result, the pipeline form of concurrency is not explored here.

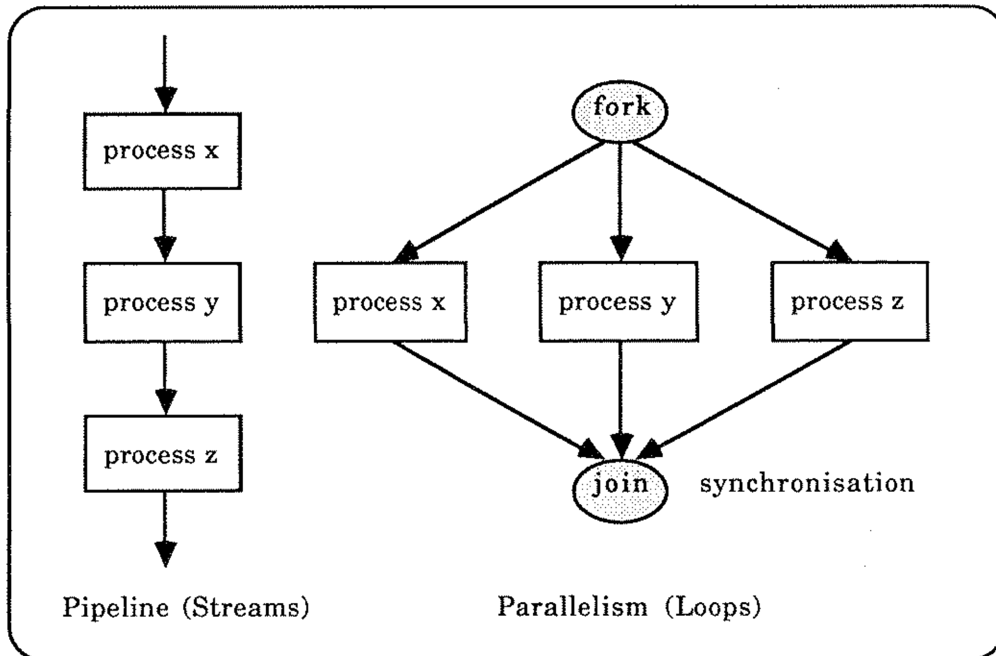

Figure 2.1: Two forms of concurrency

To date, SISAL programs can be executed on a variety of vector and parallel architectures such as Cray, HEP, multiprocessor VAX, Alliant, Sequent Balance and Encore Multimax as well as on uniprocessor machines such as SUN and VAX. It is claimed that with the availability of a SISAL compiler which compiles from SISAL source to IF1 graphs, IF1 to C, and from C to the object code of a variety of shared (and distributed) memory multiprocessors, SISAL will slowly evolve into a common high-level programming language appropriate for writing programmes to run on future general purpose parallel computers. This compiler will be described in the next section.

2.2 Optimising SISAL Compiler (OSC) [Cann89]

Both at the hardware and software levels, there are various grains of parallelism that can be exploited, such as very coarse-grain parallelism which is distributed processing across network nodes to form a single computing environment, coarse-grain parallelism which is multiprocessing of parallel processes in a multiprogramming environment, medium-grain parallelism which is parallel processing or multitasking of procedures within a single process, and fine-grain parallelism which is the parallelism inherent in a single instruction or a data stream. Grain size may be referred to as the period between synchronization events for multiple processors [ECC]; synchronization is vital in parallel processing to initialise a task, parcel out work and then merge the results [Bri88]. The OSC optimising SISAL compiler developed for shared memory multiprocessors generates multiple parallel processes from threads of tasks, and processors are allocated dynamically to whatever processes currently have the highest priority. The first released Optimising SISAL Compiler OSC received for the research in early 1989 was used. The compiler only explores parallelism in SISAL's parallel loop constructs which therefore bounds the research to medium-grain parallelism. The descriptive material below is drawn in part from [Cann89, CO89].

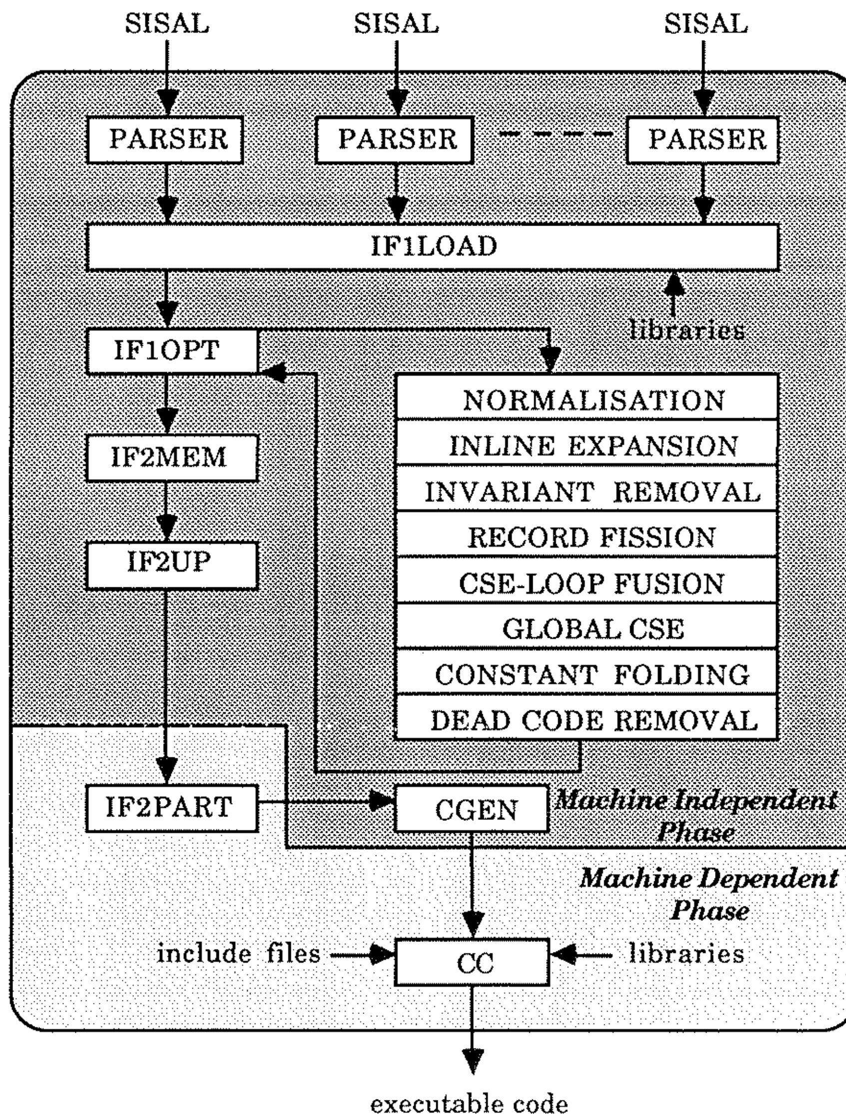

Figure 2.2: SISAL language processing in OSC

Diagrammatically illustrated in Figure 2.2, the front end compiler, *SISAL*, compiles SISAL source into IF1 [SG85], an intermediate form defining dataflow graphs that adhere to applicative semantics. This is followed by the formation of a monolithic IF1 program and the inlining of all functions except recursive ones and those exempted by user directives. The monolith is then read by *IF1OPT*, a machine independent optimiser which performs many conventional optimisations to produce a semantically equivalent but faster program. The optimisations include *graph normalisation*, *inline expansion*, *common subexpression elimination CSE*, *record fission*, *CSE-loop fusion*, *global CSE*, *constant folding* and *dead code removal*.

The above optimisations are followed by preallocations of array storage by *IF2MEM*, a build-in-place analyser [Ran87], where compile time analysis or compiler generated expressions executed at runtime can calculate the final size of an array. This optimisation attacks the incremental construction problem inherent in applicative language like SISAL. The result of this analysis is the production of a semantically equivalent program in IF2 [WS86], a superset of IF1 that is not applicative, supporting operations that directly reference and manipulate memory.

The next phase of compilation is update-in-place analysis, performed by *IF2UP* [Cann89]. Here some graphs are restructured to improve chances for in-place operation at runtime. The analyser will identify at compile time those aggregate modification operations that can execute in-place while preserving program correctness. It eliminates copying in the presence of nested aggregates, iteration and function boundaries. Recursion is not handled however. This analysis eliminates most reference counting.

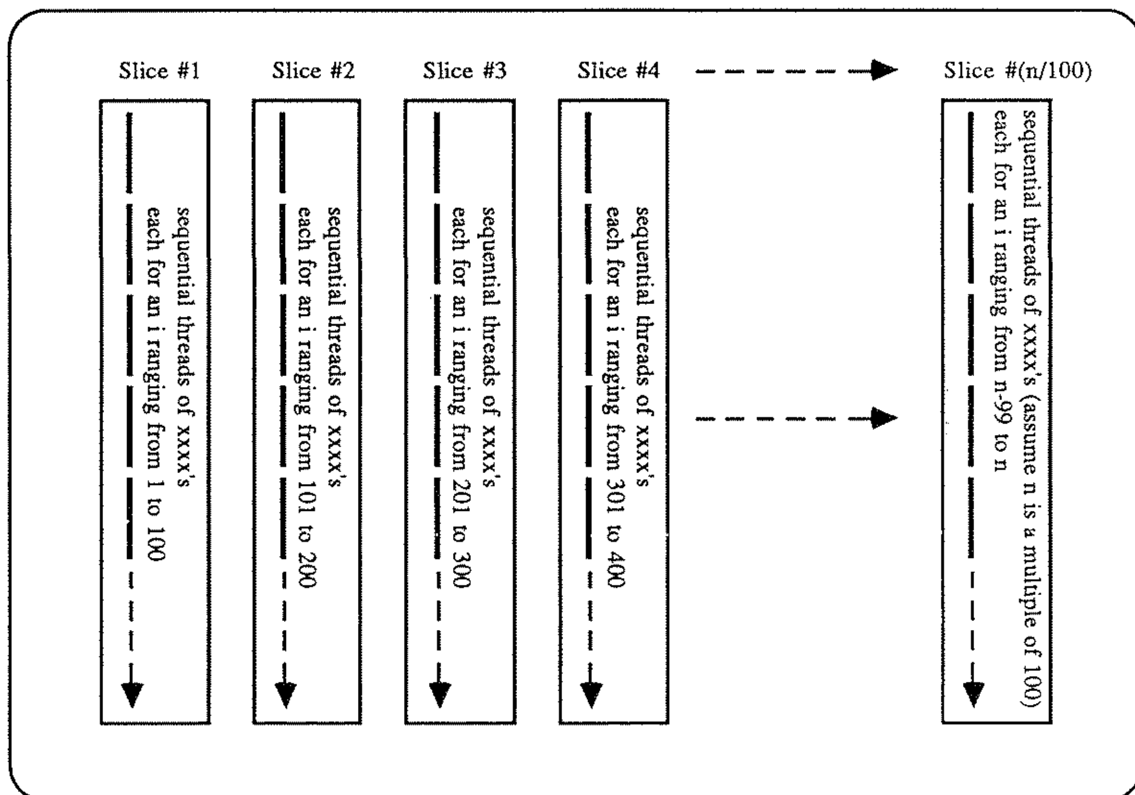

Figure 2.3: Loop slicing by OSC

After the update-in-place analysis, a machine dependent paralleliser called *IF2PART* is invoked to define the desired granularity of parallelism based on estimates of computational cost H and various other parameters such as I, number of iterations, and L, depth of nested loops to be sliced, that tune the analysis. These parameters can be modified at compile time to control parallelisation. Only parallel loop forms are subject to parallelisation. Parallel loops with estimated costs greater than H and nested no deeper than L (the default being all levels) are sliced. IF2PART weighs the bodies of loops by I, the default being 100 iterations; Figure 2.3 shows how loop slicing is, by default, performed for:

```
FOR i IN 1, upperbound
    body
END FOR
```

The optimised IF2 graphs are then translated into C and compiled using the target machine C compiler to produce executable code. Library software, linked during this phase of compilation provides support for parallel execution, storage management and interactions with the users. It is important to note that the ultimate code performance is limited by the quality of the C compiler on the target machines.

The runtime library used with OSC relies on variants of microtasking [CLOS87]. The dynamic storage system that it supports uses a parallelised boundary tag algorithm with five different entry points to reduce possible contention. It has a distributed storage cache in front of the boundary tag system to exploit locality of request sizes. The C code that OSC produces runs on multiprocessors as well as uniprocessors where parallel task execution decomposes naturally to sequential coroutine execution [CO88]. The computer that has been chosen in this research as representative of current parallel machines is the Encore Multimax multiprocessor.

2.3 Encore Multimax Multiprocessor [ECC]

Parallel computers adopt either shared memory or distributed memory architecture, or a hybrid of the two. They may be broadly classified as SIMD (Single Instruction stream Multiple Data stream) or MIMD (Multiple Instruction stream Multiple Data stream) machines. Every processor in a SIMD computers executes the same instruction at every cycle exploiting vector-pipeline parallelism whereas in a MIMD machine, each processor executes instructions independently of the others in exploiting true parallelism. Many current designs, for instance Cray-XMP, incorporates both these aspects where each node of the MIMD system is itself a vector processor, with the hope of achieving the shortest critical path [McB88]. The descriptive material below is drawn in part from [ECC, Bri88].

The Encore Multimax multiprocessor system is a shared memory tightly coupled architecture where all processors and programs share access to all of main memory, I/O interfaces and mass storage. The Multimax system provides a multiprocessing environment in which the operating system is not replicated for each processor. This provides more memory space for programs and data. Shared access to system memory also produces higher effective operating speed since processors need not pass messages in order to communicate. Memory is allocated dynamically to processes, not to processors, so as to increase the efficiency in the usage of available memory and improve interprocess communication. The system employs an extended UNIBUS-type interconnect, whereby all arithmetic and input/output processor modules can access memory modules. Processors are not dedicated to specific users or processes, but are allocated dynamically to whatever processes currently have the highest priority.

The system configuration used in the research incorporates 64 Mbytes of fast shared memory and 20 32-bit processors, each capable of executing 2 MIPS resulting in an aggregate performance rating of 40 MIPS. The processor cards of the Multimax are based on the National Semiconductor 32000 series of 32-bit, virtual memory microprocessors and associated floating point coprocessors. They are called APC, Advanced Dual Processor Cards, each of which incorporates two NS32332 microprocessors. The floating point support is provided by standard NS32081 chips. Cache memories attached to each processor handle approximately 95% of its requests, limiting the traffic on the common bus, the Nanobus, which has a data transfer bandwidth of 100 Mbytes per second.

The Nanobus is a high bandwidth bus. It provides up to 12.5 million bus transaction per second for high speed synchronous operation. All data transfers are synchronised with a 12.5 MHz bus clock. The memory bandwidth matches Nanobus capacity through *memory interleaving*, which allows contiguous longwords to be stored in up to 8 separate memory banks which can be accessed at system clock rates rather than at the slower memory clock rates. This technique allows data to move between processors, I/O devices, and memory at speeds of up to 100 Mbytes per second. Multiprocessor interrupt handling and efficient interprocessor communication via memory are two features provided by the Nanobus for fast synchronization between processors and I/O devices with negligible effect upon other system activities, a requirement in a high performance multiprocessing environment.

As Multimax is a shared-memory multiprocessor, its processors can take full advantage of memory caching which increases access speed for every processor in the system. Additionally, Multimax can exploit different degrees of parallelism: very coarse grain, coarse grain and medium grain, and also fine grain if specialised systolic or array processors are added to the system.

The operating system *UMAX V* is built to handle many service requests concurrently by supporting *multi-threading*, multiple and parallel streams of control incorporating a number of performance enhancement techniques such as interlocking individual system table entries rather than the entire table.

In a *processor-memory interlocked operation*, a processor assures rapid synchronization among processors through atomic *test&set* protocols among bus requesters and responders. *Interlocked Read-Modify-Write* bus cycle can overlap one another and other bus transactions without compromising atomicity, and other system activity is not delayed while interlocked operations are taking place.

2.4 Application Study: Spectral Barotropic Numerical Weather Prediction Model

In order to analyse the behaviour of physical phenomena in a variety of situations such as fluids, air flow, atmospheric patterns and the like, computational simulation models have been developed to enable scientists and researchers to evaluate their behaviour more closely. These models involve complicated mathematical expressions whose degree of complication was once constrained by the low computing capability of conventional computers. Today, the availability of very high performance computing machines has encouraged researchers to modify and extend these simulation models to become mathematically very complicated and computationally intensive, up to the level which was once either too costly or impossible to perform on the older computer systems [TB88]. The capability of these machines together with the very finely tuned models engender accurate and realistic forecasting of future events in areas like weather modelling, and in many cases superior designs in areas like car and aerodynamic industries. While there are many good applications that are available, the focus here is on the simulation of the global weather which has inherently very high level of parallelism.

The partial aim of the research is to investigate the feasibility of a parallel implementation of this class of models in the functional language SISAL, therefore instead of the very large multiple-level numerical weather model, a barotropic (one-level) model [Bou72] has been adopted as a representative spectral model [Sim77].

2.4.1 Mathematical Description of the Model

In contrast to the usual *grid points* models which represent parameters as grid points in space, the *spectral* model studied here represents these parameters in terms of spatial basis functions called the spherical harmonics [Sim78]. The model size is expressed in terms of the *spectral resolution number*, *J*, and the associated *number of Gaussian latitudes*, *ilat*, and *number of longitudinal points*, *ilong*, on each latitude where:

$$ilat \geq \frac{5 * J + 1}{2} \quad \text{and} \quad ilong \geq 3 * J + 1$$

A full description of the model setting out its advantages, mathematical algorithms and presenting the results which were obtained on an IBM 360/65, is given by Bourke in [Bou72]. Inspection of the equations describing the barotropic model suggests very high potential concurrency. In its primitive form, the model is expressed in terms of the vorticity and divergence of the horizontal wind field as shown in Equations 1 to 8.

Integration of the primitive equations is facilitated by a spectral grid transform technique which arises in the evaluation of the nonlinear products $U^2\psi$, $V^2\psi$, $U\Phi'$ and $V\Phi'$ in Equations 4, 5 and 6.

Definitions of symbols used:

$\underline{V}$ = *wind vector (east U and north V)*

ψ = *stream function*

∇ = *horizontal gradient operator*

χ = *velocity potential*

J = *wave number truncation (resolution)*

Φ = *geopotential height of the surface*

Φ' = *time dependent perturbation field*

D = *horizontal divergence*

$\underline{k}$ = *vertical unit vector*

Ω = *angular velocity of earth*

a = *radius of earth*

ζ = *vertical component of relative vorticity*

Φ^* = *global mean geopotential*

ϕ and λ are spherical coordinates

$$\zeta = \underline{k} \cdot \nabla x \underline{V} = \nabla^2 \psi \qquad (1)$$

$$\Phi = \Phi^* + \Phi' \qquad (2)$$

$$D = \nabla . \underline{V} = \nabla^2 \chi \qquad (3)$$

$$\frac{\delta(\nabla^2\psi)}{\delta t} = \frac{-1}{a\cos^2\phi}\left[\frac{\delta(U\nabla^2\psi)}{\delta l} + \cos\phi\frac{\delta(V\nabla^2\psi)}{\delta f}\right] - 2\Omega(\sin\phi\,\nabla^2\chi + \frac{V}{a}) \quad (4)$$

$$\frac{\delta(\nabla^2\chi)}{\delta t} = \frac{1}{a\cos^2\phi}\left[\frac{\delta(V\nabla^2\psi)}{\delta\lambda} - \cos\phi\frac{\delta(U\nabla^2\psi)}{\delta\phi}\right] + 2\Omega(\sin\phi\,\nabla^2\chi - \frac{U}{a})$$

$$- \nabla^2\left[\frac{U^2+V^2}{2\cos^2\phi} + \Phi'\right] \quad (5)$$

$$\frac{\delta\Phi'}{\delta t} = \frac{-1}{a\cos^2\phi}\left[\frac{\delta U\Phi'}{\delta\lambda} + \cos\phi\frac{\delta V\Phi'}{\delta\phi}\right] - \Phi^* D \quad (6)$$

$$U = \frac{-\cos\phi}{a}\frac{\delta\psi}{\delta t} + \frac{1}{a}\frac{\delta\chi}{\delta\lambda} \quad (7)$$

$$V = \frac{1}{a}\frac{\delta\psi}{\delta\lambda} + \frac{\cos\phi}{a}\frac{\delta\chi}{\delta\phi} \quad (8)$$

Briefly, the first step of this technique is to obtain the truncated expansions for approximating the stream function, geopotential height and two derived wind fields U and V illustrated in Equations 9 to 15.

$$\psi = a^2 \sum_{m=-J}^{+J} \sum_{r=|m|}^{|m|+J} \psi_r^m \; Y_r^m \quad (9)$$

$$\Phi = a^2 \sum_{m=-J}^{+J} \sum_{r=|m|}^{|m|+J} \Phi_r^m \; Y_r^m \quad (10)$$

$$U = a \sum_{m=-J}^{+J} \sum_{r=|m|}^{|m|+J+1} U_r^m \; Y_r^m \quad (11)$$

$$V = a \sum_{m=-J}^{+J} \sum_{r=|m|}^{|m|+J+1} V_r^m \; Y_r^m \quad (12)$$

where $Y_r^m = P_r^m(\sin\phi)\, e^{im\lambda}$

and $P_r^m(\sin\phi) = alp_r^m =$ Normalised Legendre Polynomial

$$\int_{-\pi/2}^{\pi/2} P_r^m(\sin\phi)\, P_r^m(\sin\phi)\, \cos\phi\, d\phi = 1 \qquad (13)$$

and
$$U_r^m = (r-1)\,\rho_r^m\,\psi_{r-1}^m - (r+2)\,\rho_{r+1}^m\,\psi_{r+1}^m + im\chi_r^m \qquad (14)$$

$$V_r^m = -(r-1)\,\rho_r^m\,\chi_{r-1}^m + (r+2)\,\rho_{r+1}^m\,\chi_{r+1}^m + im\psi_r^m \qquad (15)$$

where
$$\rho_r^m = \sqrt{\frac{(r^2 - m^2)}{(4r^2 - 1)}}$$

This is followed by a Fast Fourier Transformation (FFT) of these fields to the Gaussian latitude-longitude grid on the globe. The nonlinear products of Equations 4 to 6 are those on the left hand sides of Equations 16 to 20. They can now be obtained by direct multiplications in the grid domain.

$$U\nabla^2\psi = a \sum_{m=-J}^{+J} A_m\, e^{im\lambda} \qquad (16)$$

$$V\nabla^2\psi = a \sum_{m=-J}^{+J} B_m\, e^{im\lambda} \qquad (17)$$

$$U\Phi' = a^3 \sum_{m=-J}^{+J} C_m\, e^{im\lambda} \qquad (18)$$

$$V\Phi' = a^3 \sum_{m=-J}^{+J} D_m\, e^{im\lambda} \qquad (19)$$

$$\frac{U^2 + V^2}{2} = a^2 \sum_{m=-J}^{+J} E_m\, e^{im\lambda} \qquad (20)$$

An inverse FFT of these terms can now be performed, as described in Equations 16 to 20. The final step is to transform these fields back to the spectral domain. The final spectral forms of the model as a whole are shown in Equations 21 to 24.

$$-r(r+1)\frac{\delta\psi_r^m}{\delta t} = -\int_{-\pi/2}^{\pi/2} \frac{1}{a\cos^2\phi}\left[imA_m P_r^m(\sin\phi) - B_m\cos\phi \frac{\delta P_r^m(\sin\phi)}{\delta\phi}\right]\cos\phi\, d\phi$$

$$+ 2W\left[r(r-1)\rho_r^m \chi_{r-1}^m + (r+1)(r+2)\rho_{r+1}^m \chi_{r+1}^m - V_r^m \right] \qquad (21)$$

$$-r(r+1)\frac{\delta\chi_r^m}{\delta t} = \int_{-\pi/2}^{\pi/2} \frac{1}{\cos^2\phi}\left[imB_m P_r^m(\sin\phi) + A_m \cos\phi \frac{\delta P(\sin\phi)}{\delta\phi}\right]\cos\phi\, d\phi$$

$$- 2\Omega\left[r(r-1)\rho_r^m \psi_{r-1}^m + (r+1)(r+2)\rho_{r+1}^m \psi_{r+1}^m + U_r^m \right]$$

$$+ r(r+1)(E_r^m + \Phi_r^m) \qquad (22)$$

$$\frac{\delta\Phi_r^m}{\delta t} = -\int_{-\pi/2}^{\pi/2} \frac{1}{\cos^2\phi}\left[imC_m P_r^m(\sin\phi) - D_m \cos\phi \frac{\delta P_r^m(\sin\phi)}{\delta\phi}\right]\cos\phi\, d\phi$$

$$+ \Phi^* r(r+1)\chi_{r+1}^m \qquad (23)$$

$$E_r^m = \int_{-\pi/2}^{\pi/2} \frac{E_m}{\cos^2\phi} P_r^m(\sin\phi)\cos\phi\, d\phi \qquad (24)$$

2.4.2 Original Sequential Implementation: FORTRAN Realisation

The above mathematical formulation was originally implemented in FORTRAN. The program flow of the FORTRAN formulation as shown in the functional block diagram in Figure 2.4 has been derived in a direct transliteration of the code into SISAL, which will be discussed in the next chapter. The diagram and the following function descriptions amply illustrate, in real terms, how the simulation is performed in a sequential implementation:

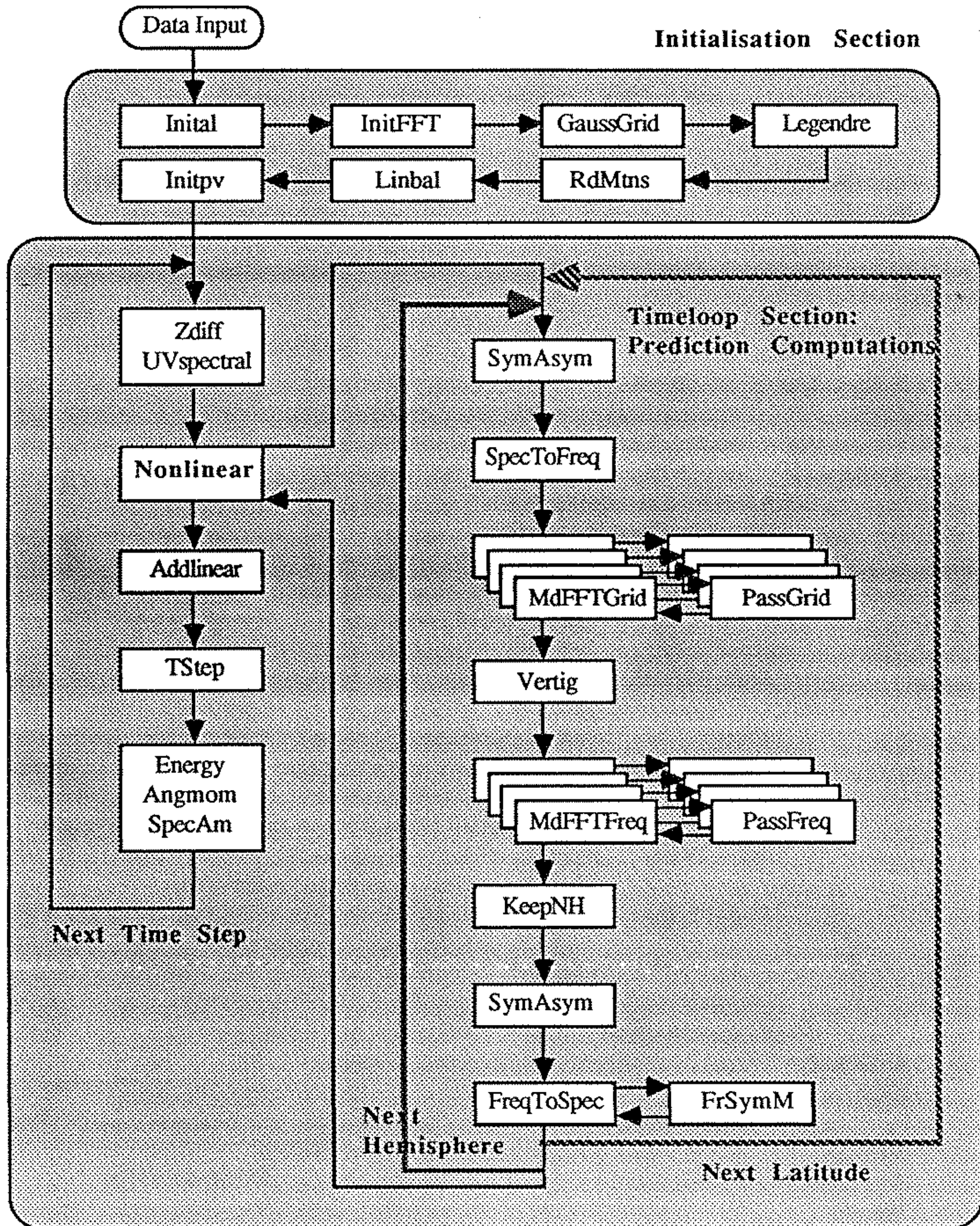

Figure 2.4: Flow chart for the FORTRAN and sequential SISAL implementations

Inital	*initialise essential model variables and generate indexing arrays of orthogonal spherical harmonics*
InitFFT	*generate tables for the FFTs*
GaussGrid	*generate the cosine of the colatitudes and the weights for the Gaussian quadrature*
Legendre	*compute the spherical harmonics for each latitude from the associated Legendre polynomial of the first kind*
RdMtns	*read the topography (mountains) of the globe in spectral form as the mean global geopotential field imposing linear balance condition (dD/dt)*
LinBal	*compute the starting geopotential field*
Initpv	*compute the starting tendencies of the spectral stream function, divergence and geopotential fields; the spectral forms of the fields are complex expansion coefficients.*

This completes the initialisation section which may in fact be partially *precomputed* for any forecast run.

In the other part, which is the timeloop section, the form of the computation for each time step is:

```
For each time step Iterate
Zdiff         compute the spectral time dependent geopotential perturbation field
UVspectral    compute the two spectral wind fields
Nonlinear     evaluation of the nonlinear products:
          For each hemisphere Iterate
             For each latitude Iterate
             SymAsym      if computing for southern hemisphere, simply copy by
                          reversing the northern hemisphere's symmetrical
                          spherical harmonics
             SpecToFreq   compute truncated expansions of each field
             MdFFTGrid    FFT transform each truncated field into the intermediate
                          domain
             Vertig       obtain nonlinear products by direct multiplications
             MdFFTFreq    inverse-FFT transform the products back to the
                          intermediate domain
             KeepNH       store values of fields if in northern hemisphere
             SymAsym      if in southern hemisphere, reverse polarity of the present
                          hemispherical spherical harmonics
             FreqToSpec   sum intermediate non linear terms from each latitude; the
                          results are the non linear terms of the fields in spectral
                          form
             Next latitude
          Next hemisphere
AddLinear     add the linear and nonlinear terms
TStep         perform a model timestep for the spectral model; the results are the
              final simulated values of the fields in spectral form after this timestep
Energy        check and ensure conservation of energy
AngMom        check and ensure conservation of angular momentum
Specam        check and ensure conservation of vorticity, divergence and height
Next time step
```

where the number of time step is dependent on the number of hours of forecast. This then completes the dynamics of the spectral barotropic numerical weather model.

Chapter 3

IMPLEMENTATIONS

This chapter consists of two main sections. The first section describes a direct transliteration of the code from FORTRAN to SISAL and the results of the implementation. This is followed by the second section which describes a parallel implementation of the model in SISAL; the parallel implementation was derived from the analysis of the computational algorithms of functions resulting from the direct transliteration approach as well as from some knowledge of the original mathematics and physics of the model. The heuristics developed for transforming a sequential loop to a parallel loop in SISAL will also be discussed.

3.1 Direct Transliteration Approach

This approach gave the first feel of the sequential realisation of the computational model. Correctness checking was performed for all mappings of subroutines to SISAL functions based on the corresponding FORTRAN results.

3.1.1 Encountered Problems

While many FORTRAN control structures mapped readily into SISAL, some difficulties were immediately encountered from FORTRAN's *common* and *equivalence* statements, and the implicit mappings from *real* to *complex* number representations because SISAL does not provide global structures, implicit mappings or an intrinsic complex number type. The *side effects* propagated from the use of *common* and *equivalence* statements made it even more difficult for the context of the model to be understood.

Another direct problem emerged when there were uncertainties of loop behaviour for some *DO* loops. For instance, a program execution may jump out of a *DO* loop irregularly. One was therefore forced to analyse the flow of data to find and eliminate inessential constraints on the sequencing of transliterated SISAL's function parts.

Described below are other encountered problems.

3.1.1.1 Late Availability of OSC

OSC arrived late in the research while DI (Dataflow Interpreter) [Yat88] and SC (SISAL Compiler) [Cann89] were always available. Compiling the transliterated SISAL functions using SC, and debugging the functions and viewing the concurrency achieved using DI created a false sense of performance of those functions. The reason was that DI traced dataflow parallelism while SC and OSC extracted parallelism only from product form *FOR* loops on MIMD multiprocessors. Both architectures exploited comparatively different grains of parallelism from the same program, in this case the transliterated functions, but these were misunderstood. This period was time wasting due to the late availability of the right compiler (OSC) and the software tools later built around OSC.

3.1.1.2 Debugging SISAL Programs

Working on a large application code, the critical problem in the direct transliteration process was the absence of an effective program debugging tool. It is to date impossible to debug a SISAL program at its source level. The best possible debugging tool available is DI, the IF1 interpreter, which interprets the corresponding IF1 graphs of a SISAL program and offers debugging from there. Unfortunately, even DI as a debugger had bugs which created problems in producing results from multiple-nested sequential loops.

In DI, the correctness of a focused variable whose value alters as it undergoes changes in different program states can only be checked by making it an input at function entries or an output at function exits. Thus it is necessary to create a function boundary around the variable to be investigated. Then in DI, a command "trace function data *function_name* " is entered at the start of the run so that the interpreter can spit off the values of the variable at every change in state both at the function entries and exits. The values are then compared with the FORTRAN output for the changes in state of the variable which may be effortlessly obtained from FORTRAN by a "*print *, variable_name*" statement in the sequential FORTRAN program. This *indirect* debugging in DI is both difficult and unreliable and requires additional lengthy, tedious and error-prone efforts. The reason is that one not only has to investigate program correctness as originally intended, but also has to deal with the correctness of the additional functions created as well as always beware of the integrity of the interpreter for complicated programs (Heisenberg uncertainty principle [Parker, Meyers]).

3.1.1.3 Accessing Out of Bound Array Elements

Other than the bad use of *common* and *equivalence* statements and ill-structuring of the FORTRAN code, another problem involving the code was that some parts of the program accessed array elements beyond declared array bounds. FORTRAN could tolerate this mistake while DI continued executing with error typed data. However, SC and OSC aborted the run immediately giving error messages like "*floating point exception error*" or "*segmentation fault*".

In function *LINBAL*, for instance, *epsi[]* and *p[]* are declared as having array sizes of *jxxmx* and *jxmx* respectively. However, the FORTRAN code accessed *epsi[jxxmx+1]* and *p[jxmx+1]* returning very large non zero numbers. The practice did not affect the overall results of the FORTRAN program, or else it would have long been corrected. In another example, *ksq[0]* in function *LINEAR* was also accessed although index *0* was not declared. The same result as the previous one occurred but one was then engaged in difficult debugging of the SISAL function.

3.1.1.4 Keeping Track with New Variable Names

Cautions on side effects in the FORTRAN code were needed. Converting from FORTRAN to SISAL involved mappings from multiple assignments to a variable which must satisfy the semantic single assignment rule of SISAL. This was attained by introducing a new name for each new assignment or update of a variable.

3.1.1.5 OLD Statements of SISAL: an Easy Mistake

SISAL's sequential loop constructs are associated with the use of *OLD* statements. As *OLD* is used on the right hand side, multiple accesses of an *OLD* variable are common place. So errors due to the coexistence of the variables evaluated in the present iteration and the *OLD* variables evaluated in the previous iteration can occur easily in multiple nested sequential loops. A particular example is in the sequential loops in the function *LEGENDRE*, (Appendix B) where once the *OLD* statement was missed out, the error was very difficult to be detected unless the results were checked to the finest detail using DI.

3.1.2 Separate Parallel and Sequential Loops of SISAL

In the transliteration process, FORTRAN *DO* loops, which treat array elements *successively* through loop iterations and perform array updates by one array element per iteration, are equivalent to product form *FOR* loops of SISAL, hence these loops are *inadvertently* parallelised in SISAL's *parallel loop* constructs. On the other hand, some FORTRAN *DO* loops do not update array elements successively but rather irregularly through loop iterations; others perform updates for multiple array elements per iteration, or some whose patterns of data dependencies are not obvious. All these loops are regarded as non product form loops and therefore are mapped into SISAL's *sequential loop* constructs.

3.1.3 Typical Mapping from FORTRAN to SISAL

A typical mapping from FORTRAN to SISAL for *SpecToFreq*, one of the few functions which constitutes the innermost loops in the subroutine *Nonlinear*, is given in Figure 3.1. This subroutine performs a transformation to compute the truncated expansions of the *pg*, *zg*, *ug* and *vg* fields. The mappings resulted in two inner parallel *For* loops which performed summations, inside an outer sequential loop which updated the arrays of the four fields. With many other similar sequential codes , some more costly than *SpecToFreq*, residing inside another two sequential *For* loops (*For each hemisphere **Iterate*** and *For each latitude **Iterate***) described in Section 2.4.2, the expected result would be a large number of complicated sequential array updates in deeply nested sequential loops which, with the version of OSC available, would lead in turn to a large amount of array copying.

The need to express operations on complex numbers explicitly in addition to the adoption of a direct transliteration of the original code resulted in a SISAL formulation which was 50% longer than the original FORTRAN code.

```
      DO 20 m = 1, mx
      mi = m + m
      mr = mi - 1
      pg(mr) = 0.0
      pg(mi) = 0.0
      zg(mr) = 0.0
      zg(mi) = 0.0

      DO 30 j = jx, 1, -1
      IF (j .eq. 1 .and. m .eq. 1) GO TO 30
      jm = kmjx(m) + j
      jmi = jm + jm
      jmr =jmi - 1
      jmx =kmjxx(m) + j
      pg(mr) = pg(mr) + alp(jmx) * pri(jmr)
      pg(mi) = pg(mi) + alp(jmx) * pri(jmi)
      zg(mr) = zg(mr) + alp(jmx) * zri(jmr)
      zg(mi) = zg(mi) + alp(jmx) * zri(jmi)
 30   CONTINUE

 20   CONTINUE

      DO 120 m = 1, mx
      mi = m + m
      mr = mi - 1
      ug(mr) = 0.0
      ug(mi) = 0.0
      vg(mr) = 0.0
      vg(mi) = 0.0

      DO 130 j = jxx, 1, -1
      jmx= kmjxx(m) + j
      jmi = jmx + jmx
      jmr = jmi - 1
      ug(mr )= ug(mr) + alp(jmx) * uri(jmr)
      ug(mi) = ug(mi) + alp(jmx) * uri(jmi)
      vg(mr) = vg(mr) + alp(jmx) * vri(jmr)
      vg(mi) = vg(mi) + alp(jmx) * vri(jmi)
130   CONTINUE

120   CONTINUE
```

(a) FORTRAN implementation

```
pg, zg, ug, vg :=
    FOR INITIAL
    m := 1;
    pg := ARRAY_fill(1, mx * 2, 0.0);
    zg := ARRAY_fill(1, mx * 2, 0.0);
    ug := ARRAY_fill(1, mx * 2, 0.0);
    vg := ARRAY_fill(1, mx * 2, 0.0);
    WHILE m <= mx REPEAT
    m := old m + 1;
    mi := old m * 2;
    mr := mi - 1;
    pgmr, pgmi, zgmr, zgmi :=
        FOR j IN 1, jx
        jm := kmjx[old m] + j;
        jmi := jm * 2;
        jmr := jmi - 1;
        jmx := kmjxx[old m] + j;
        pgr, pgi, zgr, zgi :=
            IF old m = 1 & j = 1
                THEN 0.0, 0.0, 0.0, 0.0
                ELSE   alp[jmx] * pri[jmr],
                       alp[jmx] * pri[jmi],
                       alp[jmx] * zri[jmr],
                       alp[jmx] * zri[jmi]
            END IF
        RETURNS  VALUE of SUM pgr
                 VALUE of SUM pgi
                 VALUE of SUM zgr
                 VALUE of SUM zgi
        END FOR;
    pg := old pg[mi: pgmi; mr: pgmr];
    zg := old zg[mi: zgmi; mr: zgmr];

    ugmr, ugmi, vgmr, vgmi :=
      FOR j IN 1, jxx
      jmx := kmjxx[old m] + j;
      jmi := jmx * 2;
      jmr := jmi - 1;
      RETURNS  VALUE of SUM
                   alp[jmx] * uri[jmr]
               VALUE of SUM
                   alp[jmx] * uri[jmi]
               VALUE of SUM
                   alp[jmx] * vri[jmr]
               VALUE of SUM
                   alp[jmx] * vri[jmi]
      END FOR;
    ug := old ug[mi : ugmi; mr : ugmr];
    vg := old vg[mi : vgmi; mr : vgmr];
    RETURNS  VALUE of pg
             VALUE of zg
             VALUE of ug
             VALUE of vg
    END FOR;
```

(b) Sequential SISAL implementation

Figure 3.1: Direct transliteration of the FORTRAN implementation of **SpecToFreq** to SISAL

3.1.4 Results of Implementation

The sequential computational algorithm of the model implemented in SISAL is still the same as that of FORTRAN shown in Figure 2.4. because the transliteration was kept as close as possible. The results of this were obtained using the standard f77 FORTRAN compiler released with the Encore Multimax (operating system UMAX V), and OSC. The FORTRAN compilations were performed with optimisation.

The sample model size chosen (from hereon) was a realistic spectral resolution with J=30 (corresponding approximately to a 460 km grid) which is also the largest model size in the available dataset. The run times of the model with one iteration of the timeloop at this resolution for multiple processors in Figure 3.2 shows that the SISAL run time on a single processor (773.0 seconds) was 11 times slower than the FORTRAN run time (70.0 seconds). The results with multiple processors sharing the workload indicated that this approach was inefficient for this code because the program iteratively progressed through the transformation section hemisphere by hemisphere, and in each hemisphere latitude by latitude, before obtaining the final non-linear terms of the new spectral fields. Confirmed by the concurrency profile in Figure 3.3, this sequential algorithm resulted in loss of parallelism due to excessive copying and sequential updating of arrays (36% of computation time from profile). The memory requirement for this implementation was thus many times larger than that for the FORTRAN implementation where update in place is intrinsic.

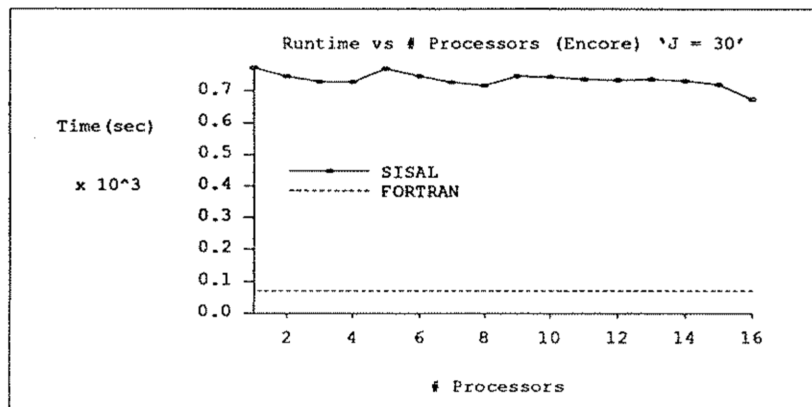

Figure 3.2: Multiple processor run times for FORTRAN and sequential SISAL

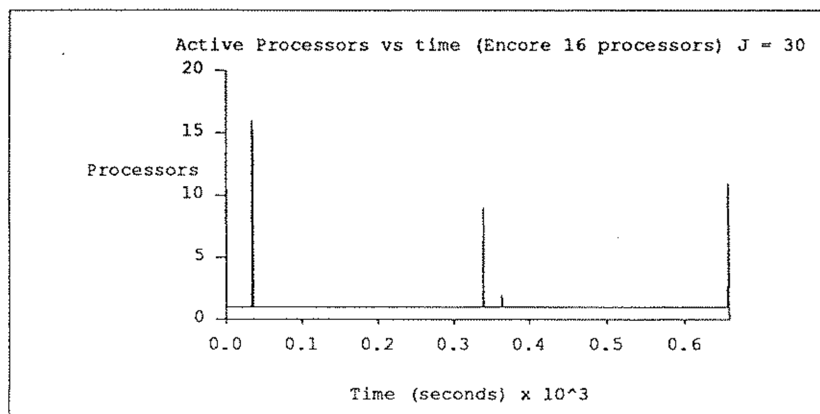

Figure 3.3: Parallelism profile of the directly transliterated SISAL version

3.2 Parallel Implementation

Allowing the specification of global state changes, FORTRAN leads to programs that are very difficult to analyse for parts that may be executed concurrently. Without a complete analysis of the entire program, it is generally impossible to trace the flow of the data. Once transliterated into SISAL, such analysis is made possible and easy by the modular and functional form of the SISAL program to find and eliminate inessential constraints on the sequencing of concurrent program parts. The direct transliteration results therefore led to the commencement of a parallel implementation process.

The process involved transformation of the individual child functions to product form loops, re-ordering the major events in the main functions to enable transformation of these events to product form loops and enforcing loop slicing for parallel loops which OSC identifies as costly to slice [CGMr90]. Observing closely, all these were principally guided by a loop transformation heuristic.

3.2.1 Heuristic of Transforming a Sequential Loop to a Parallel Loop in SISAL [Chang90]

SISAL's only means of exploiting true parallelism on general purpose MIMD machines is presently via its parallel loop constructs; parallelisation of codes must therefore be targeted to these constructs. This may be performed systematically by rewriting the sequential codes to satisfy the following conditions:

(1) only one array update is performed per iteration, and

(2) array updates are performed for successive array elements (whose indices are represented by "index" in the following example) through loop iterations.

The correctness of this rewritten sequential loop can be ensured by generating and checking the index of the updated array element in each iteration. So a typical desirable sequential code will look like:

```
FOR INITIAL
index := starting_index;
array := initialised;
WHILE index ≤ highest_index REPEAT
index := OLD index + 1;
array := OLD array[OLD index: updated];
etc.....
RETURNS VALUE OF array
END FOR
```

Any trace of data dependency is thus indicated by the **OLD** statements. If the data dependency is *direct*, in other words, if the data dependency is only between successive iterations to support the two required conditions above, such as *array := OLD array[OLD index: updated]*, then the sequential loop can be directly transformed into a parallel construct, for example one as shown below:

```
FOR index from starting_index to highest_index
etc.....
RETURNS ARRAY   OF xxxx
END FOR
```

Instead of performing update to a created array, as it was in the sequential loop form, the code now is constructing a new array in the parallel loop form.

The parallel expressive capability of SISAL's loop reduction operators such as SUM, PRODUCT and CATENATE are particularly useful in many cases to remove data dependency constrained by sequential loops. Starting from the innermost loop in a deeply nested sequential loop construct, this technique may be repeatedly applied outwardly to parallelise the whole construct.

In the case of the timeloop section in the weather code, the technique has enabled first loop transformations for individual child functions, and then *globalisations* for many of these functions.

3.2.2 Transformation of Individual Child Functions to Product Form Loops

This process focused on exploitation of *algorithmic parallelism* for the individual child functions. In their directly transliterated forms, these functions consisted of sequential loops or outer loops which were sequential because in the direct transliteration scheme, FORTRAN loops which performed updates irregularly or non-successively through loop iterations were mapped into SISAL's sequential loop constructs. The aim here was to restructure the sequential codes so as to enable the use of SISAL's parallel loop constructs. Then the bodies of the loops were checked to function similarly as those of sequential form. Final checking for correctness of results using DI was particularly important here to reconfirm the correctness of the new algorithms.

Functions whose algorithmic parallelisms had to be re-explored were those nested in the timeloop section of the model (Figure 2.4) namely *SymAsym*, *SpecToFreq*, *MdFFTGrid*, *MdFFTFreq*, *FreqToSpec*, *Linear*, *Tstep*, *Energy*, *Angmom* and *Specam*. They can be grouped into three types which are typified by *SymAsym*, *SpecToFreq* and *FreqToSpec*. This thus leads to the descriptions of parallelisation of these three functions, all of which are nested in subroutine *Nonlinear*.

```
lpfin := IF MOD(ir, 2) = 0 THEN ir + 2
         ELSE ir + 1 END IF;        % make even

alfa := FOR INITIAL
        alp1 := alp;
        mp := 1;
        WHILE mp <= ir + 1 REPEAT
        mp := OLD mp + 1;
        m := OLD mp - 1;
        ipm := m * (ir + 2);
        alp1 :=
          FOR INITIAL
          alp2 := OLD alp1;
          lp := 2;
          WHILE lp <= lpfin REPEAT
          lp := OLD lp + 2;
          ilm := ipm + OLD lp;
          alp2 := OLD alp2[ilm: - OLD alp2[ilm]];
          RETURNS VALUE OF alp2
          END FOR
        RETURNS VALUE OF alp1
        END FOR
```

(i) Sequential Implementation

```
alfa := FOR mp IN 1, ir + 1
        RETURNS VALUE of CATENATE

          FOR lp IN 1, ir + 2
          ilm := (mp - 1) * (ir + 2) + lp;
          RETURNS ARRAY of
            IF lp = 1 | MOD(lp, 2) ~= 0
            THEN REAL(alp[ilm])
            ELSE REAL(- alp[ilm]) END IF
          END FOR

        END FOR
```

(ii) Parallel Implementation

Figure 3.4: Parallelisation of function SymAsym

3.2.2.1 Parallelisation of Functions Outside Subroutine *Nonlinear*

SymAsym, for reasons which will become obvious later, may be removed from *Nonlinear* and shifted to the initialisation section. It is a typical and the simplest example of those functions not in *Nonlinear*. Parallelisation of the function was quite straightforward in that the rules of thumb could be directly applied as shown in Figure 3.4. The resultant code strikingly shows the formulation of *alfa* as:

$$alfa = \underset{mp=1}{\overset{ir+1}{Concatenate}} \left\{ \underset{lp=1}{\overset{ir+2}{Array}} \; (\, alp[\,]\,) \right\}$$

3.2.2.2 Parallelisation of Common Functions Nested in *Nonlinear*

Function *SpecToFreq* was depicted in Figure 3.1 to show how a typical direct code transliteration was performed from FORTRAN to SISAL. Incidentally it is representative of those common functions nested in *Nonlinear*. Applying the parallelisation heuristic discussed earlier, the outer loop of the directly transliterated version of *SpecToFreq*, namely the sequential version in Figure 3.5(i), may be parallelised. The resultant code as shown in Figure 3.5(ii) demonstrates a much better readability in contrast with the sequential version, in addition to its *emerging* formulation for *pg*, *zg*, *ug* and *vg*, that is:

$$pg = \underset{mrmi=1}{\overset{mx*2}{Array}} \left\{ \sum_{j=1}^{jx} (alp[\,] * pri[\,]) \right\}$$

$$zg = \underset{mrmi=1}{\overset{mx*2}{Array}} \left\{ \sum_{j=1}^{jx} (alp[\,] * zri[\,]) \right\}$$

$$ug = \underset{mrmi=1}{\overset{mx*2}{Array}} \left\{ \sum_{j=1}^{jxx} (alp[\,] * uri[\,]) \right\} \quad \text{and}$$

$$vg = \underset{mrmi=1}{\overset{mx*2}{Array}} \left\{ \sum_{j=1}^{jxx} (alp[\,] * vri[\,]) \right\}$$

```
pg, zg, ug, vg :=
   FOR INITIAL
   m := 1;
   pg := ARRAY_fill(1, mx * 2, 0.0);
   zg := ARRAY_fill(1, mx * 2, 0.0);
   ug := ARRAY_fill(1, mx * 2, 0.0);
   vg := ARRAY_fill(1, mx * 2, 0.0);
   WHILE m <= mx REPEAT
   m := old m + 1;
   mi := old m * 2;
   mr := mi - 1;
   pgmr, pgmi, zgmr, zgmi :=
     FOR j IN 1, jx
     jm := kmjx[old m] + j;
     jmi := jm * 2;
     jmr := jmi - 1;
     jmx := kmjxx[old m] + j;
     pgr, pgi, zgr, zgi :=
       IF old m = 1 & j = 1
       THEN 0.0, 0.0, 0.0, 0.0
       ELSE  alp[jmx] * pri[jmr],
             alp[jmx] * pri[jmi],
             alp[jmx] * zri[jmr],
             alp[jmx] * zri[jmi]
       END IF
     RETURNS   VALUE of SUM pgr
               VALUE of SUM pgi
               VALUE of SUM zgr
               VALUE of SUM zgi
     END FOR;
   pg := old pg[mi: pgmi; mr: pgmr];
   zg := old zg[mi: zgmi; mr: zgmr];

   ugmr, ugmi, vgmr, vgmi :=
     FOR j IN 1, jxx
     jmx := kmjxx[old m] + j;
     jmi := jmx * 2;
     jmr := jmi - 1;
     RETURNS   VALUE of SUM
               alp[jmx] * uri[jmr]
               VALUE of SUM
                   alp[jmx] * uri[jmi]
               VALUE of SUM
                   alp[jmx] * vri[jmr]
               VALUE of SUM
                     alp[jmx] * vri[jmi]
     END FOR;
   ug := old ug[mi : ugmi; mr : ugmr];
   vg := old vg[mi : vgmi; mr : vgmr];
   RETURNS   VALUE of pg
             VALUE of zg
             VALUE of ug
             VALUE of vg
   END FOR;
```

(i) Sequential version

```
pg, zg, ug, vg:=
   FOR mrmi IN 1, mx * 2
   m := (mrmi + 1) / 2;
   pg, zg :=
      FOR j IN 1, jx
      jm := kmjx[m] + j;
      jmx := kmjxx[m] + j;
      jmrjmi := jm * 2 - MOD(mrmi, 2);
      pgj, zgj := IF ~(m = 1 & j = 1)
                  THEN alp[jmx] * pri[jmrjmi],
                       alp[jmx] * zri[jmrjmi]
                  ELSE 0.0, 0.0 END IF;
      RETURNS VALUE of SUM pgj
                VALUE of SUM zgj
      END FOR;
   ug, vg :=
      FOR j IN 1, jxx
      jmx := kmjxx[m] + j;
      jmrjmi := jmx * 2 - MOD(mrmi, 2)
      RETURNS VALUE of SUM
                 alp[jmx] * uri[jmrjmi]
              VALUE of SUM
                 alp[jmx] * vri[jmrjmi]
      END FOR;
   RETURNS   ARRAY of pg
             ARRAY of zg
             ARRAY of ug
             ARRAY of vg
   END FOR;
```

(ii) Parallel version

Figure 3.5: Parallelisation of function SpecToFreq

3.2.2.3 Parallelisation of Function *FreqToSpeq* Nested in *Nonlinear*

The most difficult of all transformations was for *FreqToSpec* (Figure 3.8), which demanded an understanding of its functional role in the computational model, in addition to very careful analysis of its code pattern which was of a totally different form relative to other functions nested in *Nonlinear*. It was realised that parallelisation of this function at this stage was far from complete because, while the function still embedded inside a sequential loop (iterations of hemispheres), the needed variables on the right hand side were still data dependent. Hence parallelisation of this function is more appropriate to be discussed in the next section where a *globalisation* factor is introduced to widen the scope of investigations for parallelisation and therefrom to resolve the problem of code complexity.

3.2.3 *Globalisation* of Major Functions: Further Parallelisation

The parallelised child functions presently still embedded inside multiple sequential loops (the latitudes in both hemispheres) and therefore the model was as yet completely dominated by the costs of storage allocation and deallocation and array copying imposed by these loops. This led to the next stage of identifying and performing *geographical parallelism* in the computational model from a realisation that the individual longitudinal points of the same generation on each latitude of each hemisphere could be computed independently of each other. The tasks involved unravelling loop iterations through the latitudes, unravelling loop iterations for the North and South hemispheres, and thereby reordering of events in major functions particularly those inside the subroutine ***Nonlinear***; these evolved transformations of the events to product form loops with *globalisation* statements in the form of:

> ***FOR*** *both hemisphere* ***CROSS*** *all latitudes*
> ***RETURNS ARRAY of*** *individual child functions*

Hence, *SpecToFreqSphere*, *MdFFTGridSphere*, *VertigSphere* and *MdFFTFreqSphere* were created as functions of the same class wherein all points on the globe are dealt with concurrently. *FreqToSpecSphere* was evolved, however, as a separate class of function where summations were performed from field arrays on each latitude level; the concept of *globalisation* introduced here helped resolve the problem of code complexity and hence effect parallelisation. Productively, the realisation of the created function *SymAsymSphere* resulted in shortening of the critical path of *Nonlinear*, thus the timeloop as a whole.

```
alfa :=
  FOR hemi IN 1, 2 CROSS latlev IN 1,ilath
  RETURNS ARRAY of
        IF hemi = 1      THEN FOR specindex IN 1, jxxmx                 % North
                              RETURNS ARRAY of real(alp[latlev, specindex])
                              END FOR
        ELSE   FOR mp IN 1, ir + 1                                      % South
               RETURNS VALUE of CATENATE
                     FOR lp IN 1, irmax2
                     ilm := (mp - 1) * irmax2 + lp;
                     RETURNS ARRAY of
                           IF lp = 1 | MOD(lp, 2) ~= 0 THEN real(alp[latlev, ilm])
                           ELSE real(-alp[latlev, ilm]) END IF
                     END FOR
               END FOR
        END IF
  END FOR
```

Figure 3.6: SymAsymSphere: globalised version of SymAsym

3.2.3.1 Relocation of *SymAsym* to Model Initialisation Section

The evaluations for the symmetrical and anti-symmetrical spherical harmonics for all Fourier points on both hemispheres of the globe may be performed concurrently in the evolved function *SymAsymSphere* shown in Figure 3.6, thus producing *alfa*, a matrix which stores the spherical harmonics of all Fourier points on the whole sphere. Since the results are constants hereon and need not be recomputed in *Nonlinear* of the timeloop, the function may be relocated to the initialisation section of the model, rather than being re-evaluated iteratively in the timeloop section. Additionally, the *global* content of the function means that function *KeepNH* (Section 2.4.2) is hereby redundant. Excluding these functions, the critical path length of the timeloop was further shortened.

3.2.3.2 *Globalisation* of Common Functions Nested in *Nonlinear*

Globalisation for this class of functions were particularly straightforward. The existing child functions may simply be absorbed by the *globalisation* statements as typified by the function *SpecToFreqSphere* shown in Figure 3.7. The resultant field arrays were global dimensional which then enabled similar parallelisation of succeeding functions of the same class as well as *global-parallelisation* of function *FreqToSpec*.

```
pg, zg, ug, vg :=
    FOR hemi IN 1, 2 CROSS latlev IN 1, ilath
    pg, zg, ug, vg:=
          FOR mrmi IN 1, mx * 2
          m := (mrmi + 1) / 2;
          pg, zg :=     FOR j IN 1, jx
                        jm := kmjx[m] + j;
                        jmx := kmjxx[m] + j;
                        jmrjmi := jm * 2 - mod(mrmi, 2);
                        pgj, zgj :=  IF ~(m = 1 & j = 1)
                                     THEN alp[hemi, latlev, jmx] * pri[jmrjmi],
                                          alp[hemi, latlev, jmx] * zri[jmrjmi]
                                     ELSE  0.0, 0.0 END IF;
                        RETURNS  VALUE  of  SUM  pgj
                        VALUE  of  SUM  zgj
                        END  FOR;
          ug, vg :=     FOR j IN 1, jxx
                        jmx := kmjxx[m] + j;
                        jmrjmi := jmx * 2 - mod(mrmi, 2)
                        RETURNS   VALUE of SUM alp[hemi, latlev, jmx] * uri[jmrjmi]
                                  VALUE of SUM alp[hemi, latlev, jmx] * vri[jmrjmi]
                        END FOR
          RETURNS  ARRAY  of  pg
                   ARRAY  of  zg
                   ARRAY  of  ug
                   ARRAY  of  vg
          END  FOR
    RETURNS  ARRAY  of  pg
             ARRAY  of  zg
             ARRAY  of  ug
             ARRAY  of  vg
    END  FOR
```

Figure 3.7: SpecToFreqSphere: globalised version of SpecToFreq

3.2.3.3 *Globalisation* of Function *FreqToSpec* Nested in *Nonlinear*

The parallelisation of this function was simplified and effected by considering *globalisation*. This section describes how the transformations were evolved from a sequential version of the code to a globalised parallel version.

In the first part of the sequential version (marked "***Part 1***") shown in Figure 3.8, the assignments for *eg, pug, pvg, zug* and *zvg* imply that each of these array variables has two data independent sets of values for the conditions in the Northern hemisphere and in the Southern hemisphere respectively. Also, the *pugni* and ***old*** *pug* pair are actually *puf* fields for the Northern and Southern hemispheres respectively, the same case as the other four pairs of arrays of *_f* fields. The Northern case (symmetrical) may be evaluated by "North *_f* fields + South *_f* fields" whereas the Southern (anti-symmetrical) by "North *_f* fields - South *_f* fields", and these two evaluations may be evaluated independently, hence concurrently.

Each of the array variables *ctri, eri, ptri* and *ztri* that follow in the second part (marked "***Part 2***") undergo an array update for each array element in multiple nested sequential loops. Inside the first iteration, i.e. the Northern iteration, the first sequential summations for *ctri, eri, ptri* and *ztri* were obtained from condition *ja=1*, and these summations were added to the second sequential summations of *ctri, eri, ptri* and *ztri* from the condition *jb=2*. The results are summed with the summing results obtained in the second iteration, the Southern iteration, which operates the same way as the first iteration. The final results of *ctri, eri, ptri* and *ztri* were but a series of five conditional summations: when *ja=1* and *jb=2* and when *ja=2* and *jb=1*, and in each of the two cases there is a symmetrical condition and an anti-symmetrical condition, plus a condition at the boundary. A careful analysis shows that the overall summations may be performed from the Northern and Southern hemispherical variables concurrently, and also from the condition *ja=1* and *jb=2* and the condition *ja=2* and *jb=1* concurrently. This gives an approval signal for a parallel and less complicated computational algorithm.

Once the characteristics of both the routine *FreqToSpec* and its functional parts were clear, loop transformation of this function could proceed, guided by the rules of thumb described in Section 3.2.2 as well as the notion of parallelisation at an order of complexity higher, namely with *globalisation*. The multiple names introduced to keep track of multiple assignments in the transliteration scheme needed to be resolved in the code restructuring.

The resulting codes for ***Part 1*** became very simple when the field sources were preconstructed as three dimensional fields [hemisphere, latitude level, point of Fourier spectrum] (Figure 3.9). Here the *symmetrical and anti-symmetrical sources* can be evaluated concurrently, thereby they existed as distinct intermediate entities *eP, puP, pvP, zuP, zvP* and *eM, puM, pvM, zuM, zvM*.

Parallelisation of ***Part 2*** was greatly simplified and effected by the results of ***Part 1***. Instead of having the various conditions necessitating multiple deeply nested sequential loops, the results were evaluated in a reverse order by having multiple deeply nested parallel loops examining these conditions. The product of the transformation was a much shorter code which had been effectively parallelised. The beauty of the resultant code in Figure 3.9 is the closeness of the ***Part 2*** of *FreqToSpec* to its mathematical formulation which can be clearly seen as:

$$ctri = \underset{m=1}{\overset{mx}{\text{Concatenate}}} \left\{ \underset{jj=1}{\overset{jx*2}{\text{Array}}} \left(\sum_{latlev=1}^{ilath} (ctri_jm[latlev]) \right) \right\}$$

$$eri = \underset{m=1}{\overset{mx}{\text{Concatenate}}} \left\{ \underset{jj=1}{\overset{jx*2}{\text{Array}}} \left(\sum_{latlev=1}^{ilath} (eri_jm[latlev]) \right) \right\}$$

$$ptri = \underset{m=1}{\overset{mx}{\text{Concatenate}}} \left\{ \underset{jj=1}{\overset{jx*2}{\text{Array}}} \left(\sum_{latlev=1}^{ilath} (ptri_jm[latlev]) \right) \right\}$$

$$ztri = \underset{m=1}{\overset{mx}{\text{Concatenate}}} \left\{ \underset{jj=1}{\overset{jx*2}{\text{Array}}} \left(\sum_{latlev=1}^{ilath} (ztri_jm[latlev]) \right) \right\}$$

where each of *ctri_jm*, *eri_jm*, *ptri_jm* and *ztri_jm* is a function of *j*, *m*, *latlev* and the *symmetrical* and *anti-symmetrical sources.*

```
ctri, eri, ptri, ztri:=
FOR INITIAL                    % loop_2000
symme_antisymmetric := 1;
ctri, eri, ptri, ztri := ctr, er, ptr, ztr;
eg, pug, pvg, zug, zvg := egi, pugi, pvgi, zugi, zvgi;
WHILE symme_antisymmetric <= 2 REPEAT                % include 2
symme_antisymmetric := old symme_antisymmetric + 1;
ja, jb := IF old symme_antisymmetric = 1 THEN 1, 2 ELSE 2, 1 END IF;

%Part 1
eg, pug, pvg, zug, zvg :=
        IF old symme_antisymmetric = 2    % South
        THEN  FOR mri IN 1, mx2
              RETURNS      ARRAY of 2.0 * pugni[mri] - old pug[mri]
                           ARRAY of 2.0 * pvgni[mri] - old pvg[mri]
                           ARRAY of 2.0 * zugni[mri] - old zug[mri]
                           ARRAY of 2.0 * zvgni[mri] - old zvg[mri]
                           ARRAY of 2.0 * egni[mri] - old eg[mri]
              END FOR

        ELSE  FOR mri IN 1, mx2              % North
              RETURNS      ARRAY of old pug [mri]  +  pugni[mri]
                           ARRAY of old pvg [mri]  +  pvgni[mri]
                           ARRAY of old zug [mri]  +  zugni[mri]
                           ARRAY of old zvg [mri]  +  zvgni[mri]
                           ARRAY of old eg [mri]  +  egni[mri]
              END FOR
        END IF

%Part 2
eriisy :=  IF old symme_antisymmetric = 2
           THEN old eri  ELSE old eri[1 : old eri[1] + eg[1] * wocs[ihem] * alp[1]]
           END IF;

ctri, eri, ptri, ztri :=
        FOR INITIAL   % loop 110
        m := 1;
        ctri1, eri1, ptri1, ztri1 := old ctri, eriisy, old ptri, old ztri;
        WHILE m <= mx REPEAT
        m := old m + 1;
        mi := old m * 2;
        mr := mi - 1;
        realm := old m - 1;

        ctri1t, eri1t, ptri1t, ztri1t :=          % loop_100
              FOR INITIAL
              j := ja;
              ctri2, eri2, ptri2, ztri2 := old ctri1, old eri1, old ptri1, old ztri1;
              WHILE j <= jx REPEAT
              j := old j + 2;
              ctri2, eri2, ptri2, ztri2 :=
                    IF old j=1 & old m=1 THEN old ctri2, old eri2, old ptri2, old ztri2
                    ELSE LET
                          jm := kmjx[old m] + old j;
                          jmi := jm * 2;
                          jmr := jmi - 1;
                          jmx := kmjxx[old m] + old j;
                          gwplm := alp[jmx] * wocs[ihem];
                          b := real(realm) * gwplm
                          IN
                        old ctri2[jmr : old ctri2[jmr] - b * pvg[mi]; jmi : old ctri2[jmi] + b * pvg[mr]],
                        old eri2[jmr : old eri2[jmr] + gwplm * eg[mr]; jmi : old eri2[jmi] + gwplm * eg[mi]],
                        old ptri2[jmr : old ptri2[jmr] + b * pug[mi]; jmi : old ptri2[jmi] - b * pug[mr]],
                        old ztri2[jmr : old ztri2[jmr] + b * zug[mi]; jmi : old ztri2[jmi] - b * zug[mr]]
                          END LET
                    END IF;
```

```
            RETURNS VALUE of ctri2
                    VALUE of eri2
                    VALUE of ptri2
                    VALUE of ztri2
            END FOR;              % end loop_100

        eri1 := eri1t;

        ctri1, ptri1, ztri1 :=        % loop_201
            FOR INITIAL
            j := jb;
            ctri4, ptri4, ztri4 := ctri1t, ptri1t, ztri1t;

            WHILE j <= jx REPEAT
            j := old j + 2;
            ctri4, ptri4, ztri4 :=
            IF old j = 1 & old m = 1
            THEN old ctri4, old ptri4, old ztri4
            ELSE  LET
                  jm := kmjx[old m] + old j;
                  jmi := jm * 2;
                  jmr := jmi - 1;
                  jmx := kmjxx[old m] + old j;
                  realn := real(old j + old m - 2);
                  alpm := if old j ~= 1 then alp[jmx - 1] else 0.0 end if;
                  alpp := alp[jmx + 1];
                  a := ((realn + 1.0) * epsi[jmx] * alpm - realn * epsi[jmx + 1] * alpp) * wocs[ihem];
                  IN
                  old ctri4[jmr: old ctri4[jmr] + a * pug[mr]; jmi: old ctri4[jmi] + a * pug[mi]],
                  old ptri4[jmr: old ptri4[jmr] + a * pvg[mr]; jmi: old ptri4[jmi] + a * pvg[mi]],
                  old ztri4[jmr: old ztri4[jmr] + a * zvg[mr]; jmi: old ztri4[jmi] + a * zvg[mi]]
                  END LET
            END IF;

            RETURNS VALUE of ctri4
                    VALUE of ptri4
                    VALUE of ztri4
            END FOR;              % end loop_201

        RETURNS VALUE of ctri1
                VALUE of eri1
                VALUE of ptri1
                VALUE of ztri1
        END FOR;              % end loop_110

RETURNS VALUE of ctri
        VALUE of eri
        VALUE of ptri
        VALUE of ztri
END FOR                       % end loop_2000
```

Figure 3.8: Sequential version of FreqToSpec

```
%Part 1
eP, puP, pvP, zuP, zvP,                 % symmetrical sources: North _f + South _f
eM, puM, pvM, zuM, zvM:=                % anti-symmetrical sources: North _f - South _f
  FOR latlev IN 1, ilath CROSS mri IN 1, mx2
  RETURNS    ARRAY of ef[1, latlev, mri]  +  ef[2, latlev, mri]      % symmetric
             ARRAY of puf[1, latlev, mri]  +  puf[2, latlev, mri]
             ARRAY of pvf[1, latlev, mri]  +  pvf[2, latlev, mri]
             ARRAY of zuf[1, latlev, mri]  +  zuf[2, latlev, mri]
             ARRAY of zvf[1, latlev, mri]  +  zvf[2, latlev, mri]
                        % variable[hemisphere, latitude level, point of fourier spectrum]
             ARRAY of ef[1, latlev, mri] - ef[2, latlev, mri]        % anti-symmetric
             ARRAY of puf[1, latlev, mri] - puf[2, latlev, mri]
             ARRAY of pvf[1, latlev, mri] - pvf[2, latlev, mri]
             ARRAY of zuf[1, latlev, mri] - zuf[2, latlev, mri]
             ARRAY of zvf[1, latlev, mri] - zvf[2, latlev, mri]
  END FOR;

%Part 2
ctri, eri, ptri, ztri :=                % All-parallel loops
  FOR m IN 1, mx          % Concatenation of arrays
  mi := m * 2;
  mr := mi - 1;
  realm := m - 1;
  ctri_m, eri_m, ptri_m, ztri_m :=   % Build arrays
       FOR jj IN 1, jx * 2
       j := (jj + 1) / 2;
       jm := kmjx[m] + j;
       jmrjmi := jm * 2 - mod(jj, 2);
       jmx := kmjxx[m] + j;
       realn := real( j +  m - 2);
       ctri_jj, eri_jj, ptri_jj, ztri_jj:=       % Parallel summations
         FOR latlev IN 1, ilath
         ihem := iy + 1 - latlev;
         gwplm := alp[latlev, jmx] * wocs[ihem];                          %---------- for symmetric parts
         b := real(realm) * gwplm;
         alpm := IF j ~= 1  THEN alp[latlev, jmx - 1]  ELSE 0.0 END IF;  %--------- for anti-symmetric parts
         alpp := alp[latlev, jmx + 1];
         a := ((realn + 1.0) * epsi[jmx] * alpm  -  realn * epsi[jmx + 1] * alpp) * wocs[ihem]; % ------------
         ctri_jm, eri_jm, ptri_jm, ztri_jm :=
               IF ~(j = 1 & m = 1)
               THEN IF MOD(jm, 2) = 0
                       THEN   IF MOD(jmrjmi,2) = 0
                          THEN   a * puP[latlev, mi] +  b * pvM[latlev, mr], gwplm * eM[latlev, mi],
                                 a * pvP[latlev, mi] - b * puM[latlev, mr],
                                 a * zvP[latlev, mi] - b * zuM[latlev, mr]
                          ELSE   a * puP[latlev, mr] -  b * pvM[latlev, mi], gwplm * eM[latlev, mr],
                                 a * pvP[latlev, mr] + b * puM[latlev, mi],
                                 a * zvP[latlev, mr] + b * zuM[latlev, mi]
                          END IF
                    ELSEIF MOD(jmrjmi, 2) = 0
                         THEN  a * puM[latlev, mi] + b * pvP[latlev, mr],
                               gwplm * eP[latlev, mi],
                               a * pvM[latlev, mi] - b * puP[latlev, mr],
                               a * zvM[latlev, mi] - b * zuP[latlev, mr]
                    ELSE     a * puM[latlev, mr] - b * pvP[latlev, mi],
                             gwplm * eP[latlev, mr],
                             a * pvM[latlev, mr] + b * puP[latlev, mi],
                             a * zvM[latlev, mr] + b * zuP[latlev, mi]
                    END IF
               ELSE  0.0,      IF jj=1 THEN eP[latlev, 1] * wocs[ihem] * alp[latlev, 1] ELSE 0.0 END IF,
                     0.0,  0.0
               END IF
       RETURNS   VALUE of SUM ctri_jm
                 VALUE of SUM eri_jm
                 VALUE of SUM ptri_jm
                 VALUE of SUM ztri_jm
       END FOR
```

```
        RETURNS ARRAY of ctri_jj
                ARRAY of eri_jj
                ARRAY of ptri_jj
                ARRAY of ztri_jj
        END FOR
    RETURNS VALUE of CATENATE ctri_m
            VALUE of CATENATE eri_m
            VALUE of CATENATE ptri_m
            VALUE of CATENATE ztri_m
END FOR
```

Figure 3.9: FreqToSpecSphere: globalised version of FreqToSpec

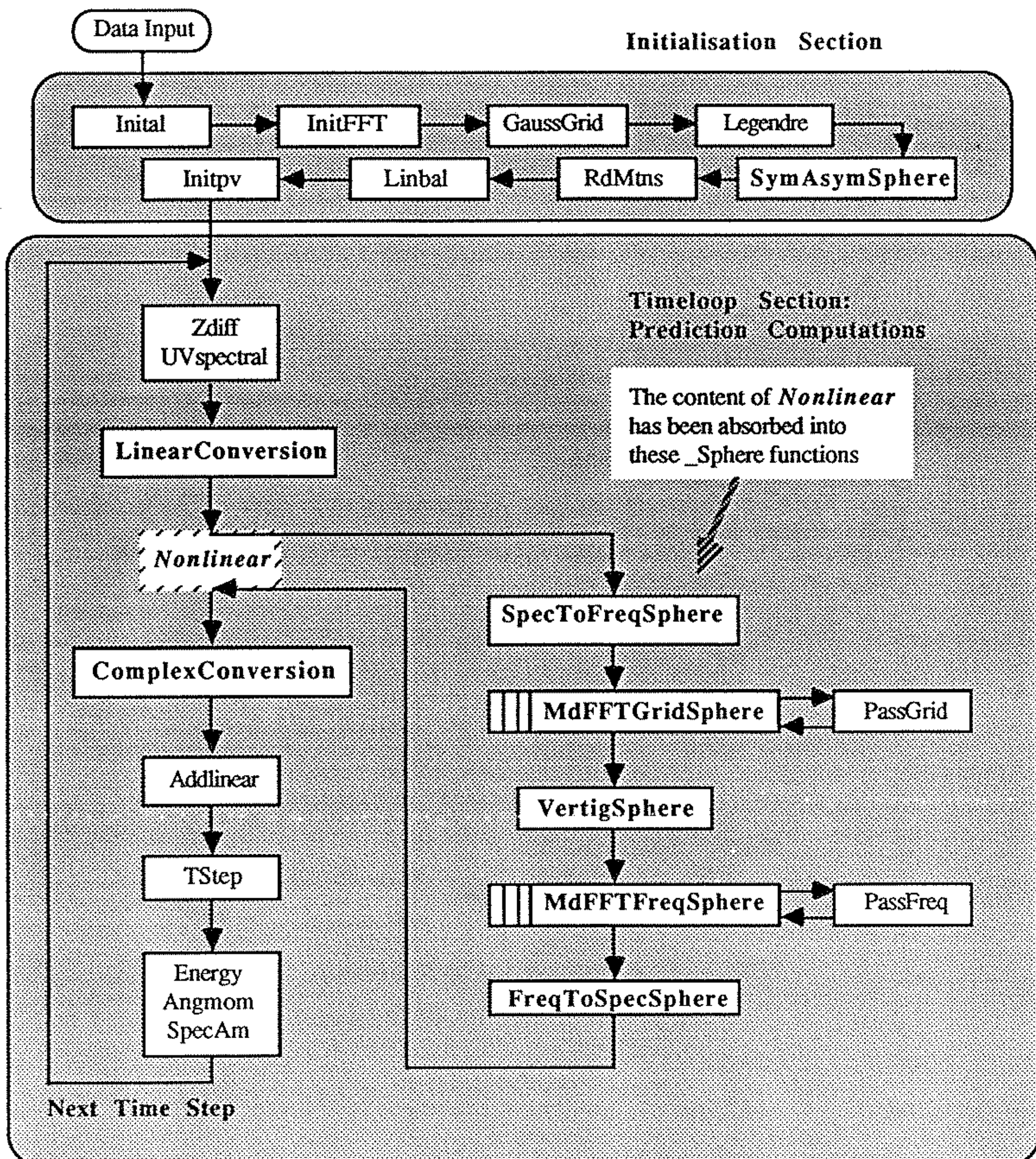

Figure 3.10: The new parallel computational algorithm of the spectral weather model

3.2.4 Resultant Computational Algorithm from Parallel Realisation [CG90]

With deeply nested parallel loops replacing deeply nested sequential loops, the product was a significant contrast in the outlook and length of subroutine *Nonlinear* thus the timeloop section and the computational model as a whole, as illustrated in the functional block diagram in Figure 3.10, where *Nonlinear* now consisted simply of:

SpecToFreqSphere();
MdFFTGridSphere();
VertigSphere();
MdFFTFreqSphere();
FreqToSphere();

3.2.5 Monitoring Bottlenecks with Concurrency Profiles [CGMr90]

Having implemented the parallel algorithm, the next stage was to investigate those parts of the implementation which, due to yet to be identified factors, may degrade the performance. Monitoring of these bottlenecks was then enabled by the availability of a software tool which traced the instantaneous activity of all the processors used, in other words the concurrency profile. The use of the tool not only showed distinctively the successfully parallelised program parts and functions, but also exposed the bottlenecks both in the initialisation and timeloop sections.

3.2.5.1 Sequential Code Sections in Timeloop

Shown in Figure 3.11, the concurrency profile for the implementation of the timeloop section of the model for a model size of J = 30 with 16 processors sharing the workload indicated that the ***inter-function*** sequential or Amdahl [Amd67] notches caused by data dependency have a second order effect on potential speedup. However, there remained three significant serial sections which consumed approximately 13% of the total execution time and grew with problem size.

The sequential sections were due to three specific functions, all of which were involved in building single dimensional arrays from singly nested parallel *FOR* loops containing small loop bodies. A number of subsequent experiments had shown that the OSC's slicing and parallelisation of this type of loops was globally determined by the compile time routine which estimated the parallel execution costs; and these loops were not sliced because the OSC cost estimator did not recognise the critical path significance of these functions.

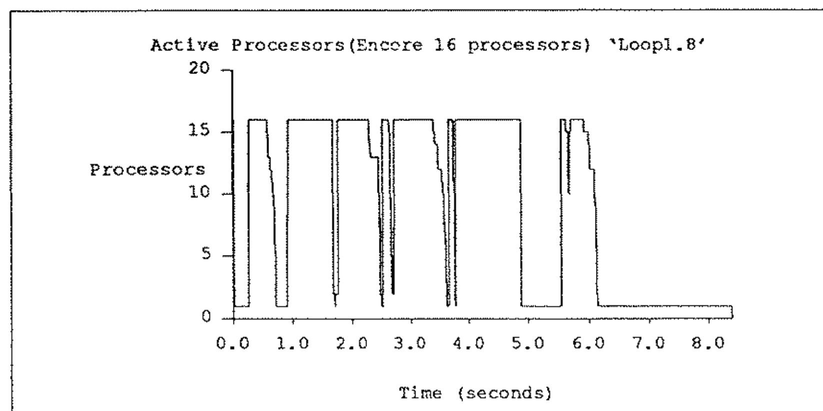

Figure 3.11: Concurrency profile of the timeloop section

3.2.5.1.1 Singly Nested Parallel Loop With Small Loop Body

SISAL presently does not implicitly support the data structure for complex numbers. Hence a complex number was represented by a *RECORD* of *Repart* and *Impart* in this implementation with the SISAL type-declaration of:

TYPE *CplexReal* = ***RECORD***[*Repart, Impart: REAL*]

Figure 3.12 shows one of these functions which explicitly converts four arrays of real numbers, each having an array size of *jx* * *mx* * *2*, to four corresponding arrays of complex numbers of array size *jx* * *mx* each. Regardless of the number of processors used, the total length of the serial code on the concurrency profile undesirably increased with the size of the loop bound.

Nevertheless, this code could be locally compiled with maximum slicing of the parallel *FOR* loop using the *-H1* pragma of OSC. The *Local Maximum Slicing* or *LMS* curves in Figures 3.14 and 3.15 illustrate the desired improvement to this code as a result. However, the present OSC does not support the linkage to a separately compiled routine, and therefore the amount of slicing of parallel *FOR* loops could only be specified as a *globally effective* OSC pragma at compile time. Unfortunately a pragma value of *-H1* led to slicing of the routines of interest but *over-parallelisation* of the rest of the program [2]. This in turn resulted in an execution time of 30 seconds compared with the original 8 seconds for the model size J = 30.

```
ctC, eC, ptC, ztC :=
    FOR complex_index IN 1, jxmx
    index := complex_index * 2
    RETURNS     ARRAY of RECORD CplexReal[Repart : ct[index - 1]; Impart : ct[index]]
                ARRAY of RECORD CplexReal[Repart : e[index - 1]; Impart : e[index]]
                ARRAY of RECORD CplexReal[Repart : pt[index - 1]; Impart : pt[index]]
                ARRAY of RECORD CplexReal[Repart : zt[index - 1]; Impart : zt[index]]
    END FOR
```

Figure 3.12. A singly nested parallel FOR loop with a small loop body

```
ctC, eC, ptC, ztC :=
    FOR m IN 1, mx
    ctC, eC, ptC, ztC :=
        FOR j IN 1, jx
        complex_index := jx * (m - 1) + j;
        index := complex_index * 2
        RETURNS   ARRAY of RECORD CplexReal[Repart : ct[index - 1]; Impart : ct[index]]
                  ARRAY of RECORD CplexReal[Repart : e[index - 1]; Impart : e[index]]
                  ARRAY of RECORD CplexReal[Repart : pt[index - 1]; Impart : pt[index]]
                  ARRAY of RECORD CplexReal[Repart : zt[index - 1]; Impart : zt[index]]
        END FOR
    RETURNS     VALUE of CATENATE ctC
                VALUE of CATENATE eC
                VALUE of CATENATE ptC
                VALUE of CATENATE ztC
    END FOR
```

Figure 3.13: A Quasi Doubly Nested loop

A possible solution to this problem is to augment the cost estimation of OSC by providing the number of processors available on the target machine as an additional pragma. The likely effect of this solution can be demonstrated by explicitly slicing these loops using a *Quasi Doubly Nested* (*QDN*) technique, which returns in this case the desired single dimensional arrays. The technique may be used effectively to force appropriate decisions from the current OSC cost estimator.

3.2.5.1.2 *Quasi Doubly Nested* Technique

Using the *QDN* technique, as shown in Figure 3.13, the inner parallel *FOR* loop of loop bound *jx* computes the correct array indices. This loop resides inside an outer parallel *FOR* loop, of loop bound *mx*, which concatenates every temporary array it produces. The loop body of this outer loop hence becomes larger and an order of complexity higher. This technique produces a slightly larger code size but the overhead is felt only when one processor is employed. Furthermore, the execution time and concurrency profiles produced using this technique are the same as those produced when the original code is locally compiled with maximum slicing, as already discussed in the previous section. Figures 3.14 and 3.15 illustrate the more efficient exploitation of concurrency and an execution time of 3.5 times faster for this code relative to the original non-parallelised code.

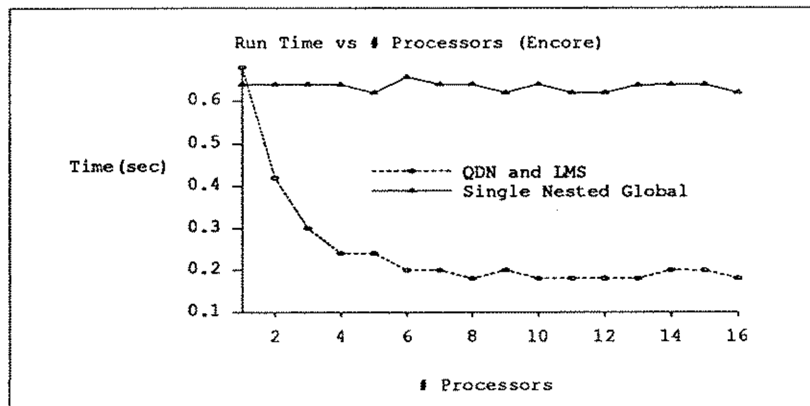

Figure 3.14: Comparison of execution time as a function of number of processors (J = 30)

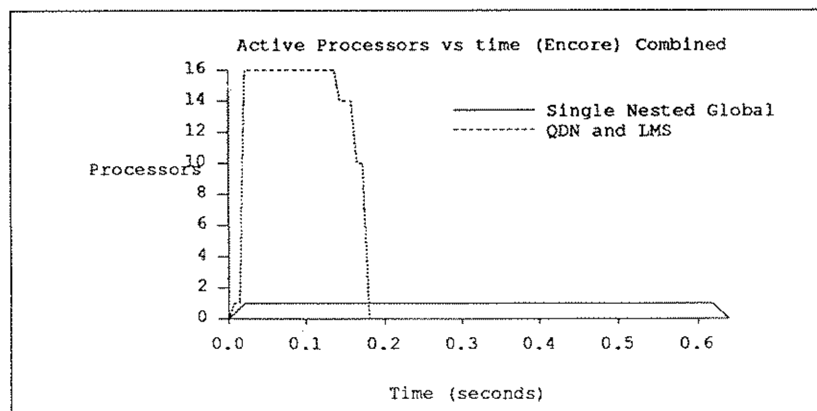

Figure 3.15: Comparison of concurrency profiles as a function of number of processors (J = 30)

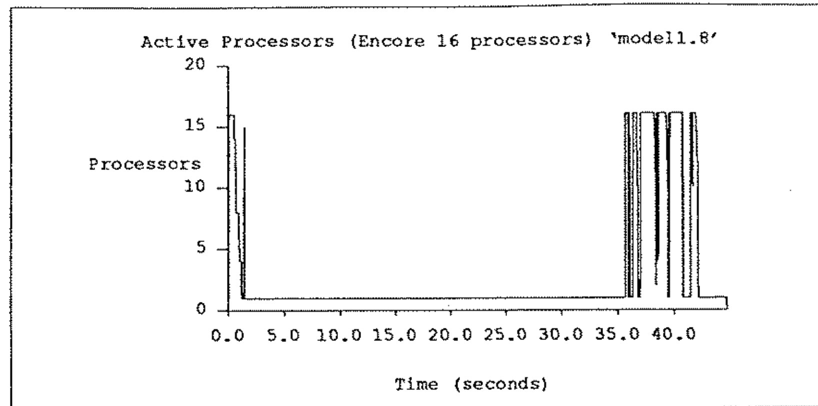

Figure 3.16: Concurrency profile of the full model at this stage

3.2.5.2 Serial Code Section in Initialisation Section

The serial computations for the Legendre polynomial of the first kind which produced the spherical harmonics of the globe [Sim78] dominated the initialisation sections of both the FORTRAN and SISAL models. These sequential computations when parallelised significantly sped up the initialisation section.

3.2.5.2.1 Serial Implementation

The model groups the latitudes of the globe into $\frac{ilat}{2}$ number of North-South latitude pairs. With this formulation, the function *LEGENDRE* computes $jxx * mx$ number of spherical harmonics for each latitude pair. However, the initial implementation was sequentially realised where both the computations for each of the harmonics on the same latitude pair and for each frame of latitude pairs were executed sequentially. In other words, there were a total of $\frac{ilat}{2} * jxx * mx$ harmonics produced and hence the same number of corresponding serial array updates performed. The SISAL equivalent of the FORTRAN version, from direct transliteration, is shown in Figure 3.17.

```
alp := FOR INITIAL
        WORKlgn   := ARRAY_fill(1, jxxmx, 0.0d0);
        lat_level  := 1;
        alp_LGN    := LEGENDRE(ir, irmax2, jxxmx, coaiy[1], siaiy[1], deltaiy[1], WORKlgn);
        WHILE lat_level < ilat / 2 REPEAT
        lat_level     := old lat_level + 1;
        alp_LGN      := LEGENDRE(ir, irmax2, jxxmx, coaiy[lat_level], siaiy[lat_level], deltaiy[lat_level],
                               old alp_LGN);
        RETURNS ARRAY of alp_LGN
        END FOR
```

Figure 3.17: Sequential computation for spherical harmonics

```
alp :=  FOR lat_level IN 1, ilat / 2
        alp_LGN   := LEGENDRE(ir, irmax2, jxxmx, coaiy[lat_level], siaiy[lat_level], deltaiy[lat_level])
        RETURNS ARRAY of alp_LGN
        END FOR
```

Figure 3.18: Parallel computation for spherical harmonics

3.2.5.2.2 Parallel Implementation

Having analysed the data dependency in the contents of *LEGENDRE*, it was realised that from direct transliteration, this routine was enforced to be sequential by arrays *WORKlgn* and *OLD alp_LGN* although all sequential threads of *LEGENDRE* could have been evaluated independently of each other. Figure 3.18 thus shows how this routine was conveniently parallelised. In this version, all frames of latitude pairs, or in other words all sequential threads of *LEGENDRE*, were computed concurrently. Figure 3.19 and 3.20 show the dramatic improvement in the run time and concurrency of the initialisation section of the model for J = 30 in contrast to the sequential implementation.

The improved performance over the initial implementation suggested that all processors were kept busy throughout. So although function *LEGENDRE* itself could be coded as a *wavefront* algorithm leading to additional improvement, this was not attempted due to the satisfactory gains already obtained.

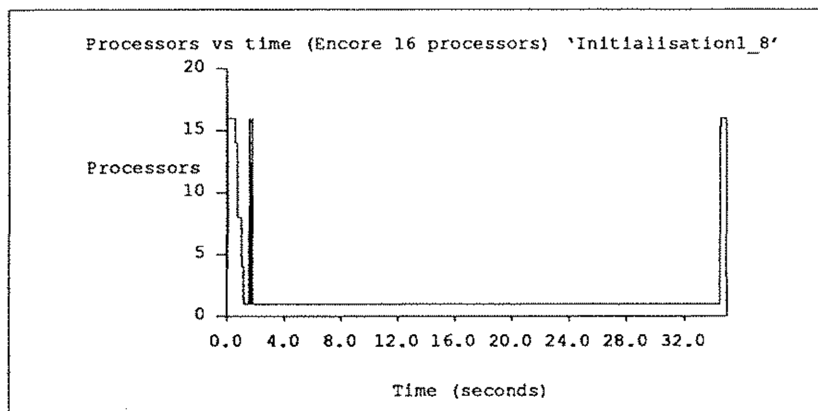

Figure 3.19: Concurrency profile of initialisation section from sequential implementation

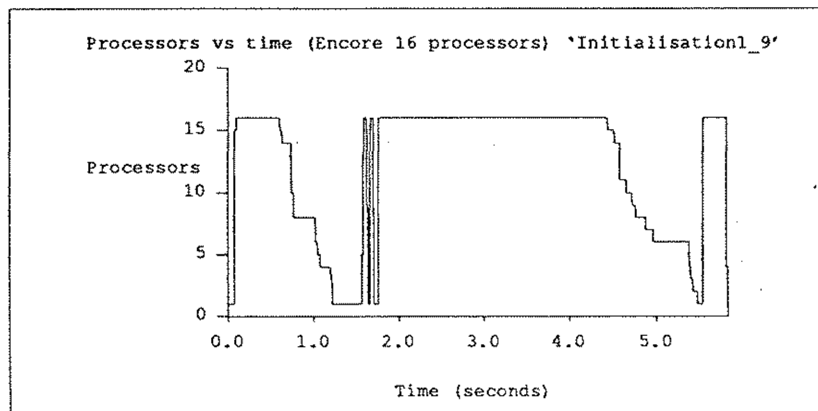

Figure 3.20: Improved concurrency profile from corresponding parallel implementation

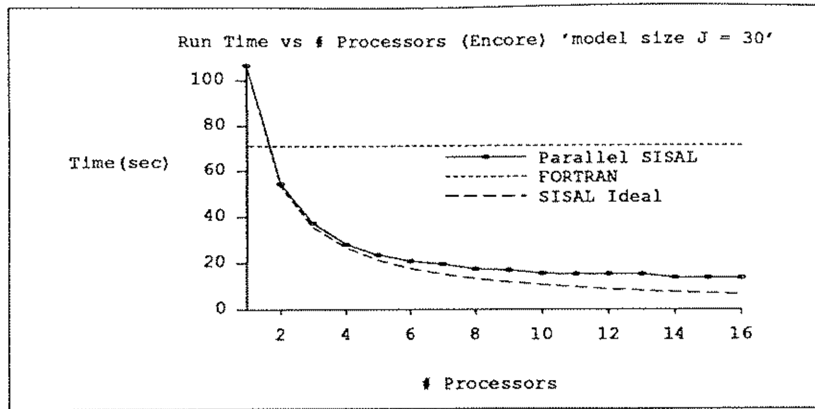

Figure 3.21: Execution time profile of the final implementation

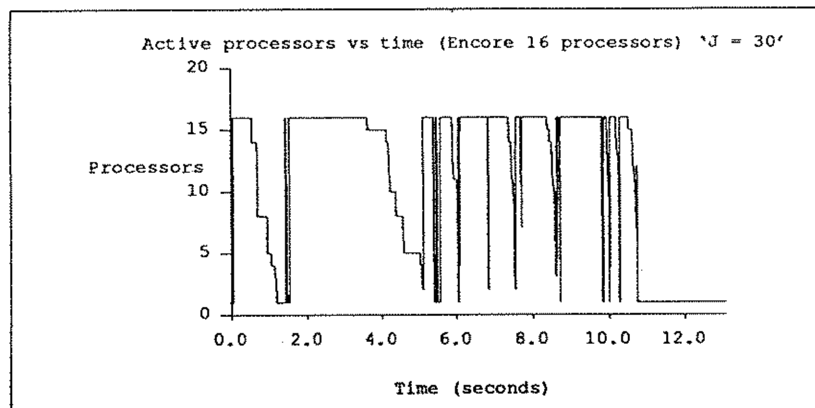

Figure 3.22: Concurrency profile for the final implementation

3.2.6 Results from Final Implementation

The run times of the model as a function of the number of processors used for the model size J = 30 with one iteration of the timeloop is plotted in Figure 3.21; the curve indicates that SISAL's parallel implementation (106.7 seconds) executes 7 times faster than its sequential implementation (Figure 3.2) on a single Multimax processor. The plot shows additionally that with 16 processors sharing the workload, the run time for this model size has been reduced to 13.7 seconds. The corresponding achieved parallelism in Figure 3.22 indicates that the parallelisation of the body of the timeloop, commencing at approximately 5.5 seconds, has been successful.

The overall results confirm the feasibility of an implementation of the weather model. The results also demonstrate that the parallel computational algorithm derived in this research (Figure 3.10) performs faster and is better suited for parallelisation than the original sequential one. This leads to further investigation of these results and performance analysis of the parallel implementation of the model in SISAL in the next chapter.

Chapter 4

PERFORMANCE ANALYSIS

The concurrency result displayed in the previous section indicates that up to this stage, all intended parallelisation of individual child functions and the timeloop section have been discretely installed and have taken effect. Following this, it is necessary to investigate in further detail how this code performs and the impact the new parallel computational algorithm of the spectral barotropic weather model could have on large model sizes, a real application situation [CG90]. The analysis leads to definitions of various speedup ratios which clarify the parallelism performance of the model as well as highlight the loss of parallelism caused by a dynamic memory management routine of OSC. Although independent of the parallel algorithm, the latter is a potential deficiency of OSC which requires a synthetical analysis into its effect and possible solutions. This chapter comprises these discussions.

4.1 General Execution Time Curves for Varying Model Sizes

The available dataset only allows the model size to be increased up to $J = 30$. In order to show the capability of the new SISAL implementation in exploiting concurrency of larger model sizes, J has been extended beyond 30 by building additional dummy datasets which still result in the same amount of computation.

The execution time for the model profiling with variable model sizes in Figure 4.1 indicates that the runtime of a small model size settles quickly with increasing number of processors because there is not enough available parallelism to be exploited. However, when the model size grows larger, the increased available concurrency lowers the rate of saturation. The "*Ideal*" line shown assumes three unrealistic conditions that is:

(a) the SISAL code is perfectly parallel,
(b) the runtime system of the compiler is overhead free, and
(c) the ENCORE Multimax architecture is fully capable of exploiting this parallelism.

It is included to show the contrast of the actual run time for model size $J = 30$ with the ideal.

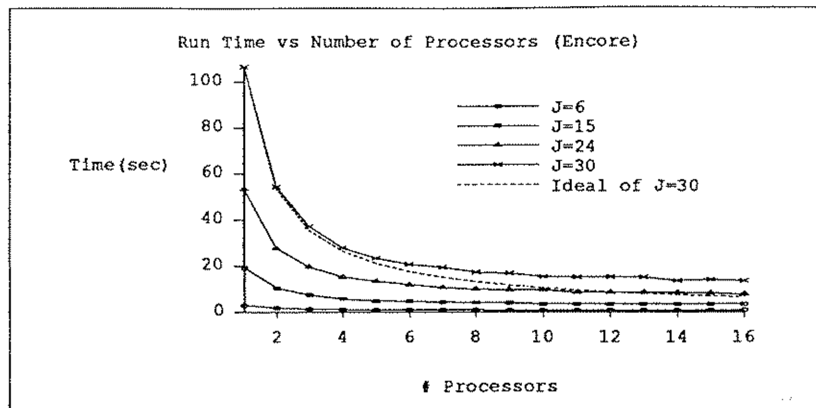

Figure 4.1: Execution time profile of the model

4.2 Analysis of Timeloop Performance

For a model size of $J = 30$, each time step is 30 minutes; hence 48 iterations of the timeloop are needed to perform a 24-hour forecast. The performance of the timeloop therefore is critical since it dominates the computation time of the model. It follows that one iteration of the timeloop is sufficient and adequate for the following analysis.

4.2.1 General Speedup Profiles for Varying Model Sizes

The $Sp(n)$ speedup curves for n number of processors with varying model sizes [EZL89], defined by

$$Sp(n) = \frac{T(1)}{T(n)} \qquad \text{where } T = \text{execution time}$$

are plotted in Figure 4.2. Representing the general principle of parallel processing, the speedups increase non-linearly, in accordance with *Amdahl's Law* [Amd67], as n is increased. Amdahl's Law defines the speedup of a code as

$$Speedup(n) = \frac{1}{s + \frac{1-s}{n}}$$

where s (a constant) refers to the sequential fraction of the code. Therefore, if s is negligible, the speedup is ideal, giving

$$Speedup_{ideal} = n$$

However,

$$Speedup_{max} \rightarrow \frac{1}{s}$$

if s is significant and if n is large. In Figure 4.2, however, as more processors are added to share the workload, instead of a horizontal *asymptote* for maximum speedups predicted by the Law, the curves reach their *points* of maximum speedup *Spmax*, before they decrease, effected by the increased *OH overhead factor*, where

$$Spmax \;=\; (Sp_{max}, n_{max})$$

and $$OH \;=\; \frac{OSC\ run\ time\ overheads}{useful\ computation}$$

The evidence is shown in the curve for $J = 6$, a small model size from which the run time system struggles to exploit a limited amount of potential concurrency (Figure 4.3). Larger amount of concurrency is present with increasing model sizes (Figure 4.4), so the loci of *Spmax* in Figure 4.2 are expected to move upward and rightward; the research is unable to prove this further due to limited number of available processors.

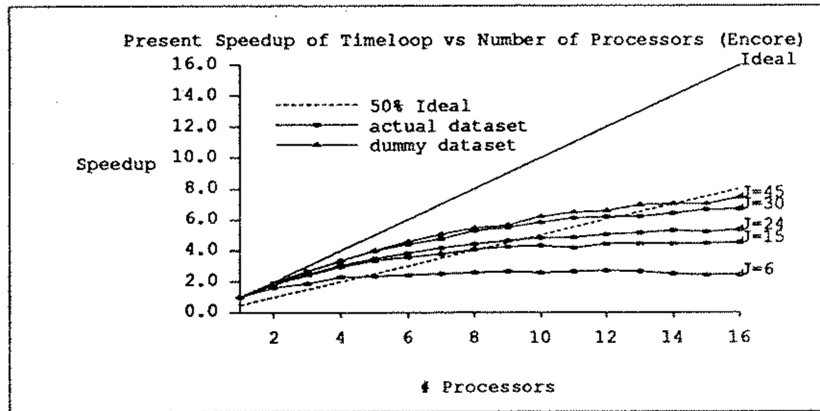

Figure 4.2: Speedup profile of the timeloop for the present implementation

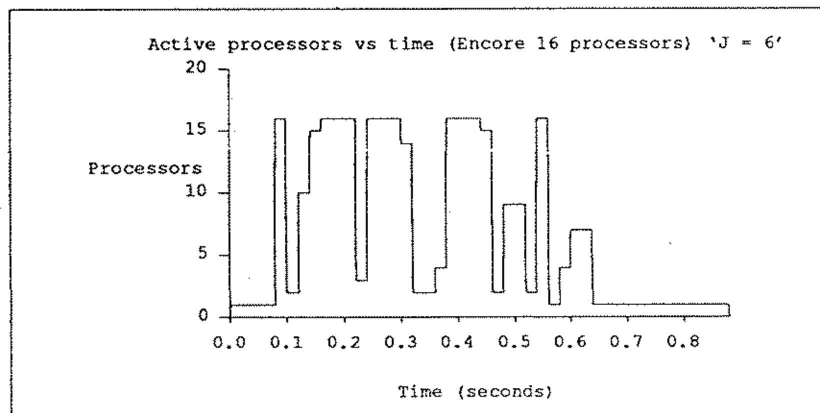

Figure 4.3: Concurrency profile of timeloop for J = 6 (small model size)

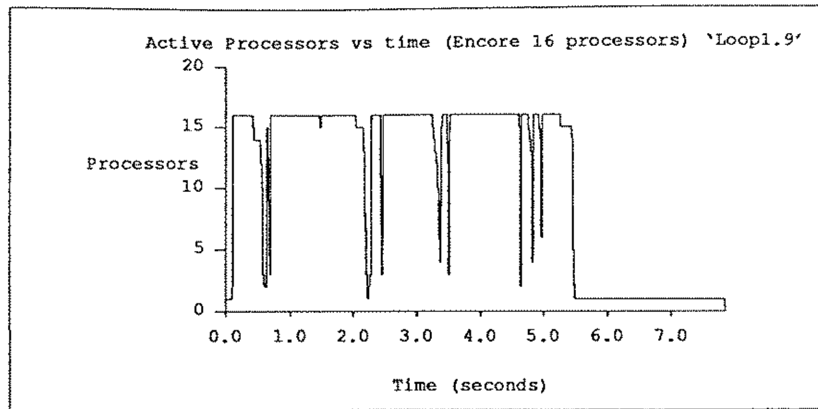

Figure 4.4: Concurrency profile of timeloop for J = 30 (largest model size in real dataset)

4.2.2 Analysis in terms of Model Sizes

The 1 processor FORTRAN (*F1*), 1 processor SISAL (*S1*) and 16 processor SISAL (*S16*) execution time curves of the timeloop section for varying model sizes are depicted in Figure 4.5. Similar to that observed by Bourke [Bou72], the FORTRAN curve up to $J = 30$ is approximately proportional to J^2. The run times for SISAL beyond $J = 30$ are, however, obtained from additional dummy data sets. The same datasets for FORTRAN are however difficult to arrange, and so the FORTRAN curve is extrapolated tangentially, thus conservatively, from $J = 30$.

The *S1* and *F1* curves show that the parallel implementation in SISAL has a single processor execution time curve which is very close to the original sequential implementation in FORTRAN, though slightly slower.

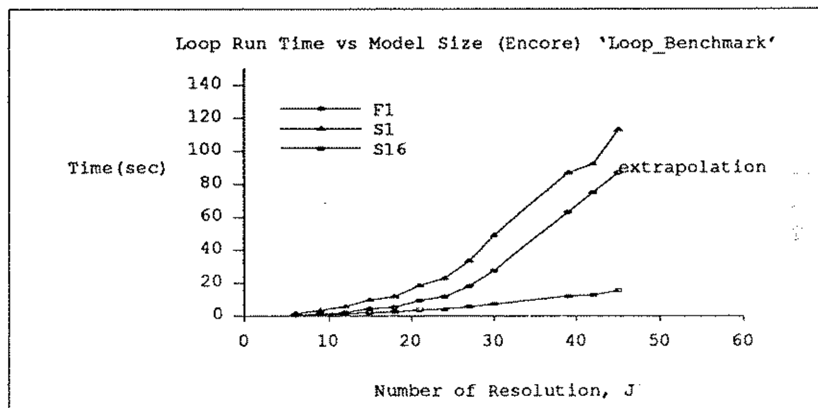

Figure 4.5: Execution time curve of FORTRAN and SISAL implementations as a function of model size

Representative of SISAL executing on multiple processors, the growth in execution time with increasing model size for the *S16* curve is much slower than that for a single processor for either FORTRAN or SISAL. This curve when extrapolated is particularly important since the model resolution for multi-level spectral models (whose curves in this case are proportional to J^3) has been attempted up to 213 in an attempt to postpone the onset of CHAOS [Lor79]; in this case the third dimension will bring the complexity of major parallel

loop bodies in the model an order of degree higher, thus expanding further the parallelism in the computation. Hence the parallel algorithm could have significant impact on spectral numerical weather modelling.

4.2.3 Speedup Analysis from Benchmark Ratios

The above execution time benchmarks may be expressed as *S1/F1*, *F1/S16* and *S1/S16* speedup ratios as a function of model size as plotted in Figure 4.6.

The *S1/F1* curve refers to the speedup of the FORTRAN implementation over the SISAL implementation for varying model size. It indicates that the implementation in SISAL on conventional computer systems, on equal terms, executes competitively with the FORTRAN implementation as the model size is increased. The negative gradient suggests that the run time could possibly be faster than the FORTRAN implementation.

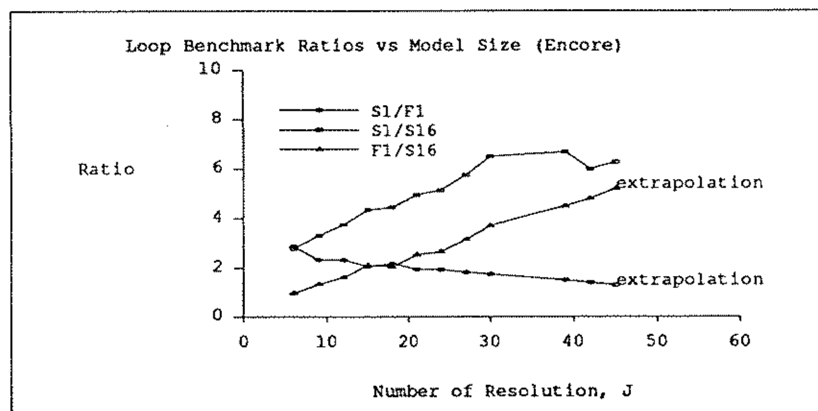

Figure 4.6: Benchmark ratios as a function of model size

The F1/S16 curve refers to the speedup of the SISAL implementation in a multiprocessor environment where 16 processors share the workload simultaneously, over a sequential implementation in FORTRAN where a single processor performs the whole task. The positive gradient of the *F1/S16* curve suggests that the speedup of the 16 processor SISAL execution time over the FORTRAN execution time can be even more substantial when a larger model size is adopted. The ratio justifies the intention of a parallel implementation of the weather model.

Finally, the *S1/S16* curve refers to the speedup of the SISAL implementation in a multiprocessor environment, where 16 processors concurrently share the workload, over the execution time of the same task by a single processor. It shows that the larger the model size, the higher will be the achieved speedup because the level of parallelism is higher (Figures 4.3 and 4.4); Figure 4.4 indicates that the fractional time spent in sequential regions is reduced as excess available tasks in the SISAL run time task queue are transferred forward in time, thus extending the parallel regions.

Interestingly, the *S1/S16* curve also highlights the impact of a problem in OSC; the falling gradient of this curve at the extended model sizes $J = 39, 42$ and 45 presents itself as a symptom of an inefficient dynamic memory management scheme of the OSC run time system which will be discussed in later sections. Disregard this problem, since it can be solved, the *S1/S16* curve indicates that employing SISAL on multiprocessors can provide efficient speedup over a single processor on the Multimax multiprocessor.

4.3 Effect of Memory Deallocation Overhead

Returning to Figures 4.3 and 4.4, close comparisons of the concurrency profiles for executions of different model sizes indicate that the fraction of the serial tail section becomes more significant, thus critical, as more parallelism is obtained, contributing to the overhead shown in the *S1/S16* ratio in Figure 4.6. This tail has been found to be produced by the eager memory deallocation routine of the OSC run time system in which the storage structures used are automatically but sequentially deallocated at the end of every cycle of the timeloop section. The resulting overhead consumes a large proportion of the loop time as demonstrated in Figure 4.4 (approximately 28%).

The same observations have been reported by Cann and Oldehoeft in their experience in executing the *SIMPLE* hydrodynamics code [AE87] on a Sequent Balance 21000 [CO89], although the fraction of the tail was much smaller in that case. The computational experiments of a *Shallow Water* weather model performed by Egan on a Multimax also reported similar observations [Egan90].

This then suggests a general deficiency of the OSC dynamic performance. Supplementary computational experiments were then devised to investigate and resolve this problem. The study, which will be described in the next sections, suggests positively that better performance for this weather model as well as for the experiments of Cann, Oldehoeft and Egan can be attained by storage reuse, instead of iterative storage allocations and deallocations.

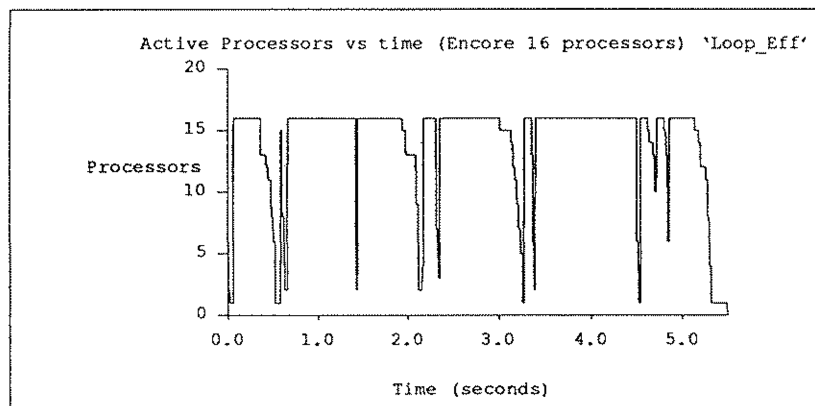

Figure 4.7: Achievable timeloop concurrency (J = 30) with an efficient dynamic memory deallocation

Figures 4.7 and 4.8 show the performance that more efficient dynamic memory deallocation could achieve. In this example the deallocation code in the C level was removed entirely resulting in the removal of the serial tail section. The effected total concurrency (Figure 4.7) corresponds entirely to the originally intended parallelisation of individual child functions and the timeloop section. The product is a loop iteration with significantly upgraded speedup characteristics relative to the previous ones; with a fixed number of processors sharing the workload concurrently, 16 for example, the speedup curves increase, with better gradients, with model size. At the same time, the percentage machine utilisation, or efficiency, also improves. These features also indicate that more processors may be added, if necessary, to provide further speedup with good machine utilisation for larger model sizes.

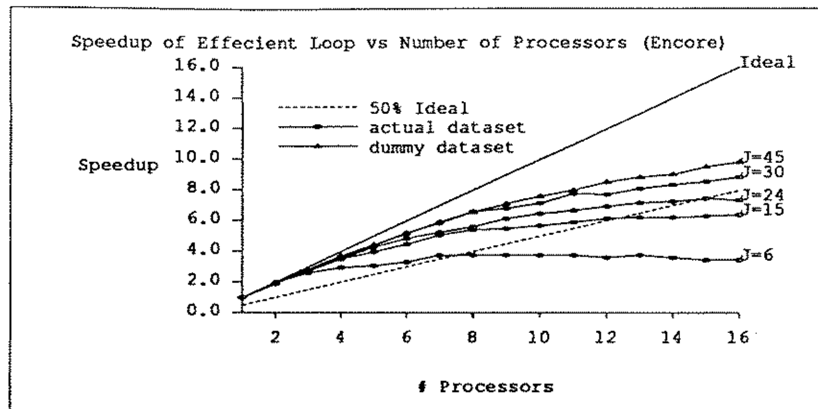

Figure 4.8: Achievable speedup of timeloop with an efficient dynamic memory deallocation

4.3.1 Investigations for Dynamic Memory Management Scheme of OSC

In an attempt to confirm the general performance of the OSC runtime in this aspect in scientific codes, supplementary application study was undertaken to specifically highlight and define the actual problems as well as to search for possible remedies [CGMy90]. The algorithm adopted for such investigations was a two dimensional Fast Fourier Transformation routine [GW] coded in C and targeted for feature detection; the intended path was to translate the routine from C to SISAL.

A two dimensional Fast Fourier Transformation routine sequentially implemented in C was readily available. It was an engineering code actually used in practice. The code was small but possessed large amount of parallelism both at medium-grain and fine-grain levels, hence may be representative of large scientific codes. The routine was sequentially realised at fine-grain level, but the medium-grain *geographical parallelism* can be directly implemented in SISAL by the loop transformation heuristic described earlier. Hence the routine may be coded to concurrently execute multiple sequential threads consisting of nested sequential loops of function *FFT*, thus enabling magnification of the anticipated impact of the dynamic storage management problems. These features made the routine a suitable choice. The diagrams in Figure 4.9 briefly illustrates the mechanism in the two dimensional FFT routine.

4.3.1.1 Direct Transliteration from FFT in C

The routine written in C is listed in Section C.1 in Appendix C. In the SISAL version of the routine listed in Section C.2, the sequential FFT consists of deeply nested sequential loops performing *successive doubling* through n stages sequentially, where the mesh size is $2^n x 2^n$, and similar loops performing bit-reversal transformation. Nonetheless, multiple threads of FFTs, each performing for a row or a column, may be executed concurrently. The strength of the parallel implementation, as far as the SISAL source level parallelism goes, is therefore in the expressions of:

Forall rows Do
FFT(row)

and *Forall columns Do*
FFT(column)

which are typical of scientific codes.

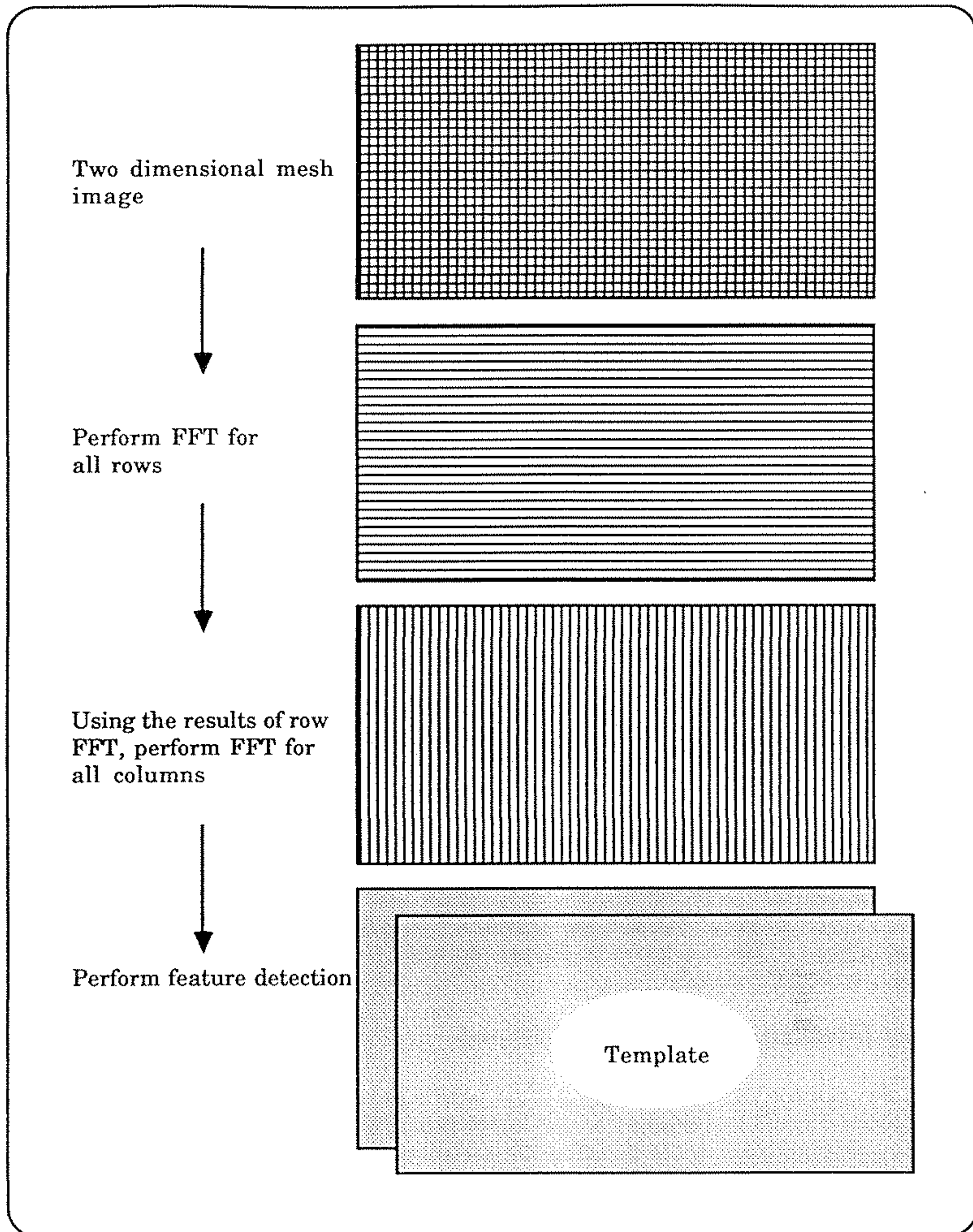

Figure 4.9: Two dimensional FFT routine for feature detection

However, the emerging deficiency of the dynamic routines of OSC is not controllable at the SISAL source level, as shown in Figures 4.11 and 4.12 for the case of a 512 x 512 mesh image. The results shown would have been implausible if not for the preconceived knowledge of the runtime problems.

Although there were other factors involved such as aggregate construction costs and copying of rows on first iterations, the dominant factor which contributed to the much degraded results for the SISAL code was the problem due to the dynamic storage management. The profiling tool *eprof* on UMAX recorded a total fraction of 30% and 10% of time spent on the dynamic allocation and deallocation routines of OSC respectively. This information helped in the derivation of *ABD synthesis*, which synthetically analyses the negative effect of the present dynamic storage management scheme adopted by OSC.

4.3.1.2 ABD Synthesis for Effect of Dynamic Memory Management Scheme of OSC

The synthesis does not attempt to give an accurate analysis of the behaviour of a SISAL parallel loop concurrently executing multiple threads of sequential loops, but to derive with approximation, an upper bound performance for such typical codes in large scientific applications.

Best visualised as in Figure 4.13, there are a single allocator and a single deallocator in the runtime system in the present scheme for sequential loops. In the initialisation (SISAL's ***For Initial***) of a sequential loop, the allocator allocates free storage for variables to be initialised (denoted by *allocate(old_x)*). Then in the loop body, the allocator again allocates another free storage for the new version of the variables (*allocate(x)*) followed by evaluations for the new variable values as a function of the old variable values. At the end of the loop body, the storage structures for the old variable are deallocated (*deallocate(old_x)*) and the newly evaluated variable values are renamed as old (*rename old_x := x*). Then, the program returns to the beginning of the loop body for the next iteration.

For the purpose of ABD synthesis, a sequential loop may be divided into three separate routines namely allocation (*A*), body (*B*) and deallocation (*D*). For some data structures, it is expected that the *A* and *D* operations may be performed partially overlapping each other. While the best case is if they were allowed to run concurrently, the worst case is if they were mutually exclusive and had to run serial to each other. The former is a simpler situation to analyse for the intended purpose, which results in the conception of ABD synthesis.

In the synthesis, the execution time of each iteration is normalised to 1; so for the 512 x 512 two dimensional Fast Fourier Transformation routine, *A* and *D* will have to be normalised to 0.3 and 0.1 respectively. Now lets assume that:

(i) the time for initialising a process is insignificant,
(ii) all processors execute equal length of processes,
(iii) work scheduling propagates orderly from left to right,
(iv) allocator and deallocator may operate concurrently, and
(v) each iteration consists of an A, B and D operations.

These assumptions are sufficient to define an upper bound speedup of the two dimensional Fast Fourier Transformation routine, Sp_{upper}, where:

$$Sp_{upper} = \frac{1}{A}$$

so for $A = 0.3$, the definition gives $Sp_{upper} = 3.333$.

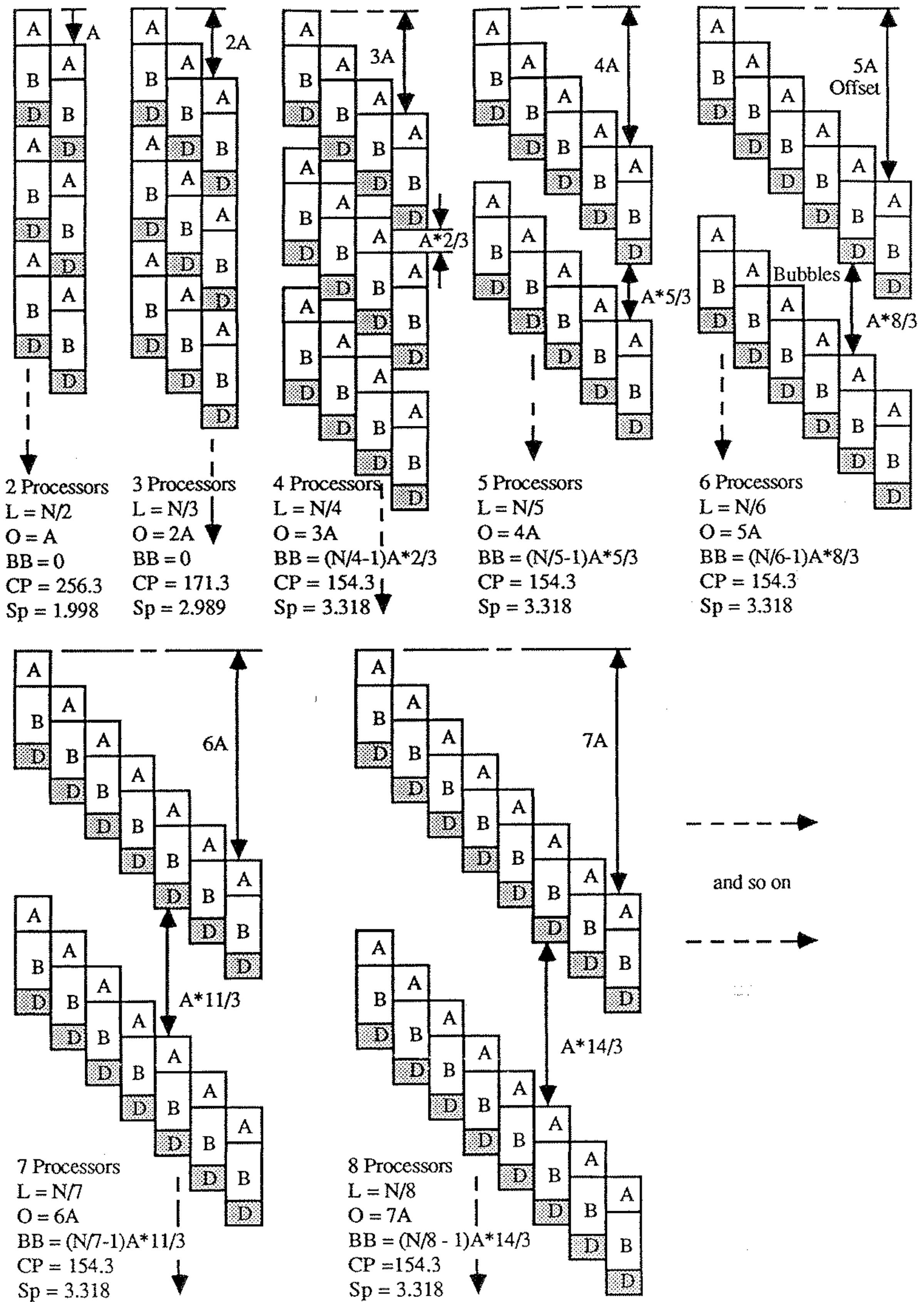

Figure 4.10: ABD synthesis for upper bound performance

However, these are not yet sufficient to predict when the upper bound occurs, an information which may be beneficial to have for maximising machine utilisation. The synthetical analysis is therefore useful. First lets define:

L = *the length of each process*
O = *the sum of process offsets*
BB = *the sum of bubbles*
N = *the total number of iteration*
CP = *the length of critical path*
Sp = *speedup*

Then, from the synthesis described diagrammatically in Figure 4.10, CP and the corresponding Sp may be approximated respectively as:

$$CP = L + O + BB$$

and $$Sp = \frac{N}{CP}$$

The synthesis demonstrates how *bubbles* are generated by the mere presence of A's. As more processors are added to share the workload, the process length L becomes shorter but the bubbles BB become larger and penetrate through all parts of the processes to counter the intended shortening of the critical path CP; the offset O becomes insignificant here. Depending on the fraction of A, the speedup may be dramatically effected by these bubbles rather than Amdahl's Law. While the diagram exhibits the behaviour for $A = 0.3$, it should be noted that each value of A results in a unique solution of CP, hence Sp.

The unnormalised results of the synthesis and those from actual runs are plotted in Figures 4.11 and 4.12. The analytical execution time curve from the synthesis is a relatively strict lower bound which settles at just above 40 seconds from 4 processors onwards. From the inversed angle, the corresponding speedup curve which saturates at 3.318 from 4 processors onwards sets a relatively strict upper bound in comparison to the actual maximum speedup of about 2.7. As well as predicting the maximum number of processors which will provide maximum speedup, the ABD synthesis thus validates the definition of Sp_{upper} above.

In this investigation, the actual speedup curve confirms the earlier discussed breakdown of Amdahl's Law in the actual multiprocessor environment effected by the *OH factor*. The investigation also proves the deficiency of the present memory management scheme as well as the impact that it has on the performance of codes written in a typical algorithmic pattern of large scientific codes. This leads to the next section in which the problems of the present dynamic memory management scheme will be defined and resolved.

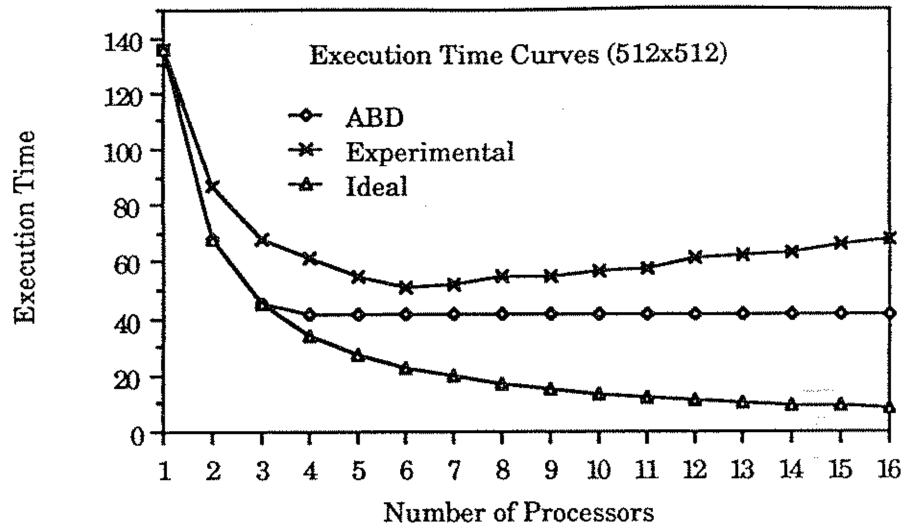

Figure 4.11: Execution time curves for 512x512 mesh FFT routine

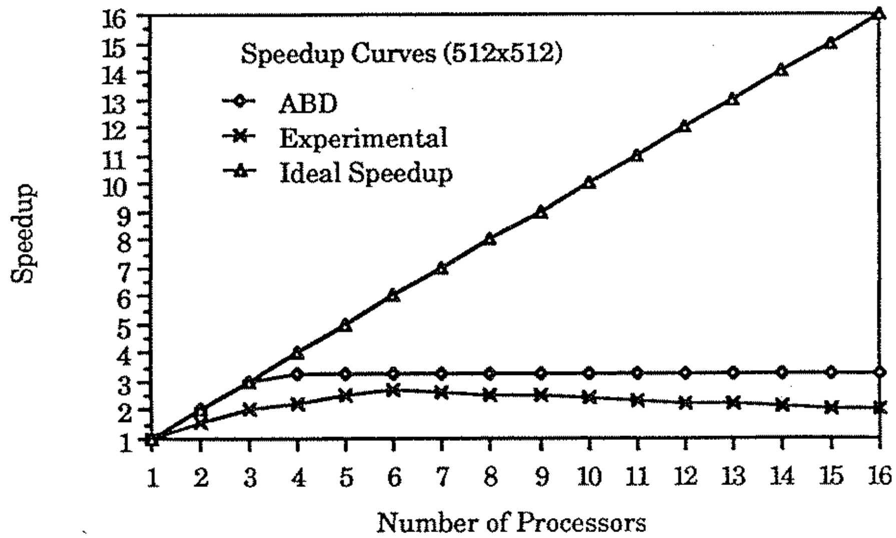

Figure 4.12: Corresponding speedup curves for 512x512 mesh FFT routine

4.3.2 Proposals for Better Dynamic Memory Management Schemes

The present dynamic memory management scheme of OSC as summarised in Figure 4.13 is a general solution for variable sized arrays. However the studies performed have shown its inadequacy for general scientific applications which overwhelmingly consist of fixed size array structures. In order to reduce the proven high cost associated with the scheme, it may be necessary to isolate these conditions in two categories:

Condition 1: Array sizes fixed through loop iterations.
Condition 2: Array sizes varies through loop iterations

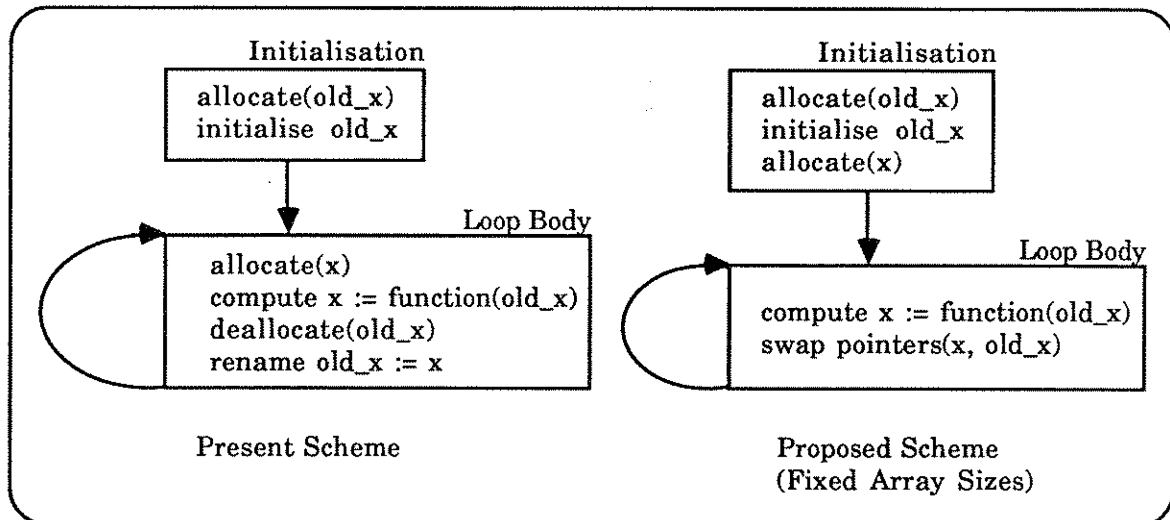

Figure 4.13: Dynamic memory management schemes of OSC for a sequential loop

4.3.2.1 Fixed Array Sizes Through Loop Iterations

An efficient solution for Condition 1 may be implemented more easily relative to Condition 2 since the array sizes are fixed. Illustrated in Figure 4.13, the repetitive allocation in the loop body has been relocated so that storage structures which are needed in the loop body are preallocated in the initialisation section (*allocate(old_x)* and *allocate(x)*). The deallocation, *deallocate(old_x),* at the end of the body is then replaced by a swap of pointers to the two storage locations, *swap pointers (x, old_x)*. Utilising such code motion and data structure pointer reassignment, this scheme cancels out the deallocation in the present iteration and the allocation in the next iteration.

Doing without repetitive storage allocations and deallocations in the loop body, this scheme is effective for parallel computing of such codes (parallel executions of multiple threads of sequential iterations), as shown diagrammatically in Figure 4.14; the diagram shows that each process is now entirely dominated by loop bodies which perform useful computations. As scientific application codes commonly are, and can be, expressed as having fixed array sizes, the performance degradation problems which were previously demonstrated in the serial tail section and the S1/S16 curves of the weather model, and evaluated in the ABD analysis, may therefore be effectively solved using this scheme.

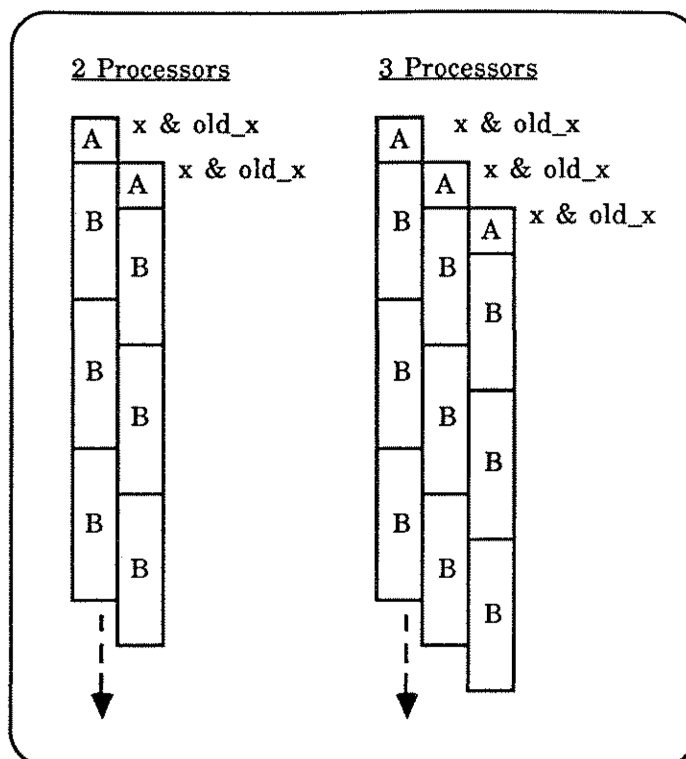

Figure 4.14: Effective parallel computation of sequential iterations for fixed array sizes using code motion and data structure pointer reassignment

4.3.2.2 Varying Array Sizes Through Loop Iterations

Condition 2 includes those cases in which data structure sizes cannot be determined statically. It is primarily for this that the present dynamic memory management scheme is designed. Our initial analysis proposed that *lazy* deallocation of memory structures in parallel with the main computation, only when the storage is exhausted, would lead to substantial gains [CG89]. However, Garsden [Gar90] has demonstrated that this solution is unfortunately incomplete, and the actual solution to this problem is more complicated than first thought. He has shown that by removing the deallocation of old structures, the tradeoff resulted from the consequent reduction in the hit rate of the software cache may worsen the already degraded performance.

The complexity of the problem suggests that dedicated research encompassing various aspects of the compiler is needed before a more appropriate scheme can be eventuated for parallel executions of multiple sequential threads with varying array sizes through loop iterations. A possible indirect solution to this problem will be discussed in Chapter 5.

Chapter 5

FINAL DISCUSSIONS AND CONCLUSIONS

Experience in transliteration of FORTRAN to SISAL and the identification of appropriate refinement heuristics has highlighted some problems with the expressive power of SISAL. The results from the weather model implementations have also indicated significant deficiencies in the run time performance of OSC. As will be shown in this chapter, which leads to the conclusions of this thesis, these two issues are interrelated. First the user issues will be discussed followed by the associated runtime issues. As a consequence of this research [CGA90], these issues have been acknowledged and addressed in the recent proposal for the SISAL 2.0 language definition [COBGF] and related compiler design [CV90].

5.1 Programming in SISAL

It has been claimed that SISAL functional programs can achieve sequential and parallel execution performance *competitive* with programs written in conventional languages; this research on a large scientific application code supports this claim. With additional language features and improvements in the runtime system, and subject to the qualifications which will be discussed below, SISAL can be a powerful functional language for scientific computations and can execute efficiently on conventional multiprocessors.

5.1.1 Parallelism in SISAL

The parallelism exploited by SISAL is dependent on the underlying machine architecture upon which a SISAL program is executed. On a dataflow machine for which SISAL is originally targeted, theoretically the concurrency in a SISAL program is limited only by data dependencies. In other words, the parallelism (fine-grain) is implicit. On a shared memory Encore Multimax multiprocessor, however, the Optimising SISAL Compiler, OSC, which generates C code from SISAL, only slices or parallelises, based on the cost estimation at compile time, those parts of a function which are in the form of parallel loop constructs (stream construct concurrency is exploited in the latest version of OSC [CV90]). In other words, parallelism in SISAL programs is not machine independent as generally implied. Viewed from conventional MIMD machines' frame of reference, the specification of parallelism is explicit rather than implicit, thus the implementation of parallelism also becomes explicit rather than implicit. These aspects are often confused in SISAL related articles.

Nevertheless, work scheduling is always implicit, in that programmers are unloaded from the burden of scheduling work tasks to the processors available; this is the beauty of the language implementation. The only explicit task whose degree of difficulty very much depends on individual routines is to express the available concurrency in SISAL's parallel loop forms.

On dataflow machines, the execution control is data dependent rather than statement ordering dependent thus the control is implicit. On conventional machines, conversely, the control is explicit depending on the ordering of statements, but this does not increase the task of programming. What does contribute to the difficulty of SISAL programming for MIMD machines is the constraint imposed by the single assignment rule of the language, which often makes the expressions of parallelism in parallel loop constructs inefficient .

5.1.2 Parallelisation at SISAL Level

The original weather simulation code in FORTRAN may be representative of existing large scientific application FORTRAN codes. It is mature but was developed from many inputs, resulting in not only inconsistent programming styles throughout the program, but a program unsuitable for loop slicing or parallelisation. The initial suspicion that desirable results may not eventuate from parallelising the code at the FORTRAN level has been confirmed by the availability of a parallel FORTRAN compiler after the research phase of this work was completed.

The results obtained from using the Encore Parallelising FORTRAN EPF compiler [ECCE] is shown in Table 5.1 for a 4 processor XPC based Multimax. The XPC based processors are generally regarded as being twice the performance of the APC based processors used for the major part of this study. The time for a single APC processor is also given as a reference point. The results indicate that no speedup is achieved and thus confirm the conjecture that the FORTRAN formulation is sequential.

Number of Processors	Execution Time (seconds) *1 timeloop iteration*
1	32.0 (70.0 on APC)
2	34.0
3	33.5
4	33.8

Table 5.1: Execution times of original weather prediction model in FORTRAN using EPF compiler on an XPC based Multimax

The experience of transliterating from the original FORTRAN was difficult due to the non structured programming styles adopted by its joint programmers. As a consequence, in part, of the varying styles adopted, uncovering problem parallelism in the FORTRAN formulation also proved to be difficult. Re-expressing the computation in a structured form using SISAL made the problem formulation clear. What was also clear was that the results from the direct transliteration of this formulation were significantly worse than anticipated. The process however uncovered the deficiencies in the original formulation and led quickly to a refinement process which through its loop transformation heuristics exposed the inherent problem parallelism.

5.1.2.1 Loop Transformation Heuristics for SISAL

The loop transformation heuristics may be performed, to a greater degree, by rewriting SISAL's sequential loops to satisfy two conditions, which are:

(1) only one array update is performed per iteration, and
(2) array updates are performed for successive array elements through loop iterations,

which then enables the loops to be directly recoded into SISAL's parallel loop constructs. Starting from the innermost loop in a deeply nested sequential loop construct, this technique may be repeatedly applied outwardly to parallelise the whole construct.

The initial stage of the transliteration process is largely mechanical while the refinement step, although requiring effort, is tractable. It is felt that this process may prove appropriate for many large existing codes. The alternative of identifying the kernel of a computation and applying essentially the same process to the kernel only has now been adopted in the multi-language facility of Sisal 2.0.

5.1.3 Debugging SISAL Programs

The most serious problem in the development of SISAL programs is the lack of debugging support at the SISAL source level. The indirect program debugging with DI is both difficult and unreliable and requires additional lengthy, tedious and error-prone efforts, as already discussed. The alternative program debugging at the C code level is sometimes useful too except that the C code generated from SISAL must be assumed as perfectly correct, which is not always true! As a result, program development of large scientific codes in SISAL is presently difficult. Research into source level debugging aids for SISAL is therefore needed. This research may not be attractive, yet the reality is that few large application codes are correct by design and even less codes work first time.

5.1.4 Problems of SISAL

SISAL is a relatively new functional language whose efficacy in expressing the potential concurrency of scientific computational models has been investigated with the practical application studies in this research. Nonetheless, it has been been found necessary to add some features to the language to improve its expressive capability, particularly for scientific computations. Discussed below are the deficiencies in the expressive power of SISAL features that had direct impact on the research.

5.1.4.1 Language Support for Complex Numbers

Computations involving complex numbers are extremely common in scientific applications; FORTRAN recognises this but unfortunately SISAL presently does not. The alternative adopted here was to explicitly express complex numbers as records of two numbers representing the real and imaginary parts. An array of complex numbers is thus an array of records. As a result, the arithmetic involving complex numbers have to be explicitly performed with the aid of additional subgraphs of functions representing arithmetic operators. For instance, a mathematical expression of complex arithmetic in function TStep of the spectral weather model is:

$$ccv = \frac{cm[jm] + deltt2*(ct[jm] + kl*\{zm[jm] + deltt*(zt[jm] - zmean*cm[jm]*0.5)\}\ )}{1.0 + deltt*deltt*kl*zmean}$$

where *cm*, *ct*, *zm*, *zt* and *zm* are complex array variables. In FORTRAN, it can be implicitly expressed as:

```
ccv = (cm(jm) + deltt2*(ct(jm) + kl*(zm(jm) + deltt*(zt(jm) - zmean*cm(jm)*0.5))))
        / (1.0 + deltt*deltt*kl*zmean)
```

with good program readability and closeness to the original mathematical expression. However, in SISAL, one is first forced to explicitly define a simple record type to enable the representation of complex single precision numbers such as:

```
TYPE complex = RECORD[Repart, Impart: real]
```

One also has to build records for the complex number representations for *cm*, *ct*, *zm*, *zt* and *zm* by, for example:

```
cm := RECORD complex[Repart: real_number; Impart: real_number]
```

Then functions representing arithmetic operators have to be created. For example *Crmul* will be expressed as:

```
FUNCTION Crmul( constant: real; cnum: complex RETURNS complex)
RECORD complex[Repart : constant * cnum.Repart; Impart : constant * cnum.Impart]
END FUNCTION
```

The resulting SISAL expression for the above equation hence becomes:

```
ccv := Crdiv(Cadd(cm[jm], Crmul(deltt2, Cadd(ct[jm], Crmul(kl,
        Cadd(zm[jm], Crmul(deltt, Csub(zt[jm], Crmul(0.5 * zmean, cm[jm]))))))),
        1.0 + deltt * deltt * kl * zmean)
```

The statement consists of multiple function calls for complex arithmetic which serve only to obscure the underlying algorithm from the user's perspective.

Although OSC performs a record fission optimisation at compile time, the subgraphs of additional functions for complex number arithmetic serve as a complication which may have contributed to the *Normalisation* error discussed in Section A.4 of Appendix A. In most cases, particularly when complex arithmetic constitutes a major part of a program, the explicit tasks in the treatment of complex numbers as records contribute to additional execution cost as previously shown in the sample code in Figure 3.12.

Another alternative is to express a complex number as two separate numbers, and then an array of complex numbers as two separate arrays of numbers. This too is an explicit task which may result in additional execution cost. The link with the original equation is even more obscure in this case as the computations have to be performed separately for the resulting real parts and imaginary parts.

As a consequence of this research, the inclusion of a complex number type is being considered for inclusion in the SISAL 2.0 definition [DV90].

5.1.4.2 Matrices in SISAL

Matrices are expressed in SISAL as arrays of arrays where arrays in SISAL are more strictly vectors; this was adopted as a general mechanism for variable sized objects. It has however significant impact on runtime performance in scientific codes which overwhelmingly use fixed size matrices. The specific performance impact is in OSC's dynamic storage allocation mechanisms.

5.1.4.2.1 Dope Vector Scheme: Declaration of Matrices as Arrays of Arrays

In this scheme, array elements are accessed by using pointers and dope vectors. For example, a two dimensional array of complex single precision numbers is declared as:

TYPE twodim = ARRAY[ARRAY[complex]]

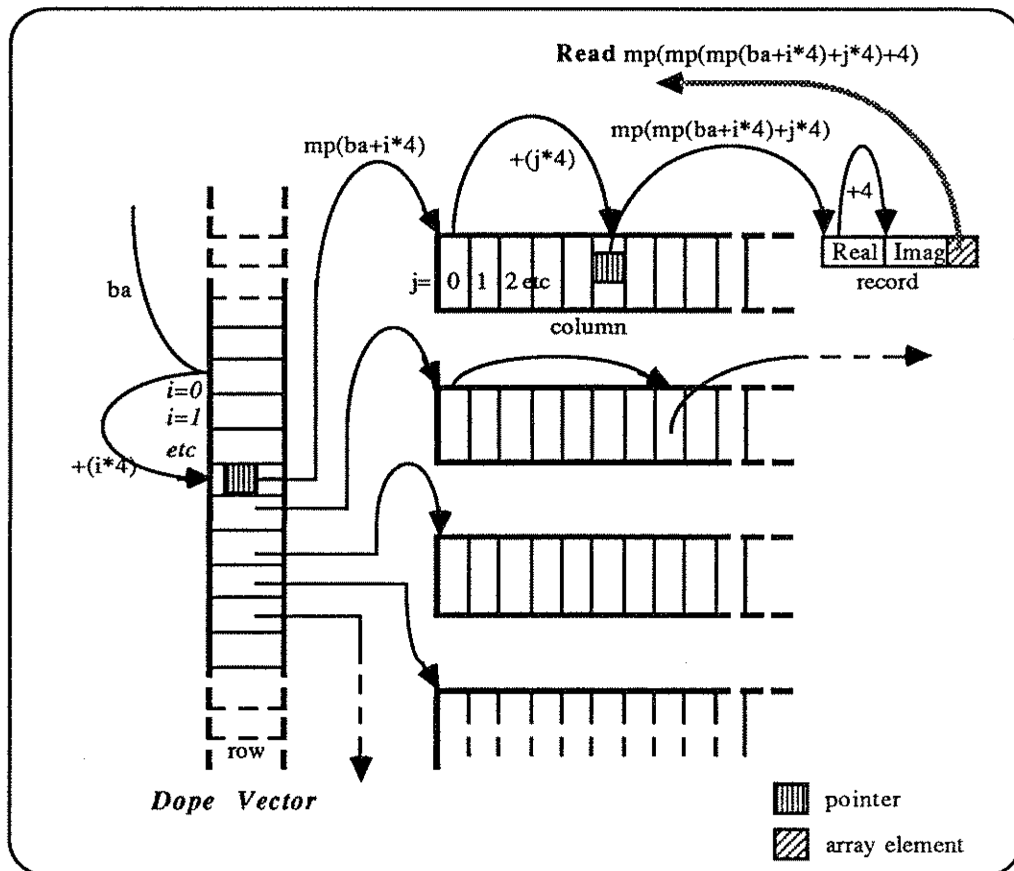

Figure 5.1: Dope Vector Scheme

In Figure 5.1, *mp(x)* denotes the content of address *x* and *ba* denotes base address. For a simple SISAL statement of *array_variable[i,j].Imag*, assuming byte addressing, the diagram shows that the read is *mp(mp(mp(ba + i*4) + j*4) + 4)*. If the cost is not yet obvious, then let us assume that this statement lies inside a doubly nested loop, that is:

```
FOR DO
        FOR DO
        .... := array_variable[i,j].Imag etc
        etc
        END FOR
END FOR
```

The actual computation performed is thus:

```
FOR DO
column_address := mp(ba + i*4)
        FOR j DO
        record_address := mp(column_address + j*4);
        imaginary_part_address := record_address + 4;
        READ mp(imaginary_part_address)
        END FOR
END FOR
```

In this scheme, the dope vector first supplies the pointer to the address space of column, which with an offset of *j*4* provides the pointer to the address of the record. The value of *array_variable[i,j].Imag* is then read at an offset from this address. Such use of pointers to pointers to access memory addresses of array elements is very costly although invariant removal is performed at a very fine level (one below IF1 and IF2) because the inner loop still performs many operations. In the memory management scheme extensively investigated in Chapter 4, storage allocations and deallocations are actually performed in this way, which contributes no doubt to the deficiency of the dynamic storage management of OSC, and the possibility that a contributing solution could be found from here. One such solution is Matrix scheme, which treats a matrix as a matrix.

5.1.4.2.2 Matrix Scheme: Declaration of Matrices as Matrices

A more rational approach is to determine that the matrices are of a fixed size, by analysis or declaration, and use the more conventional FORTRAN like representation.

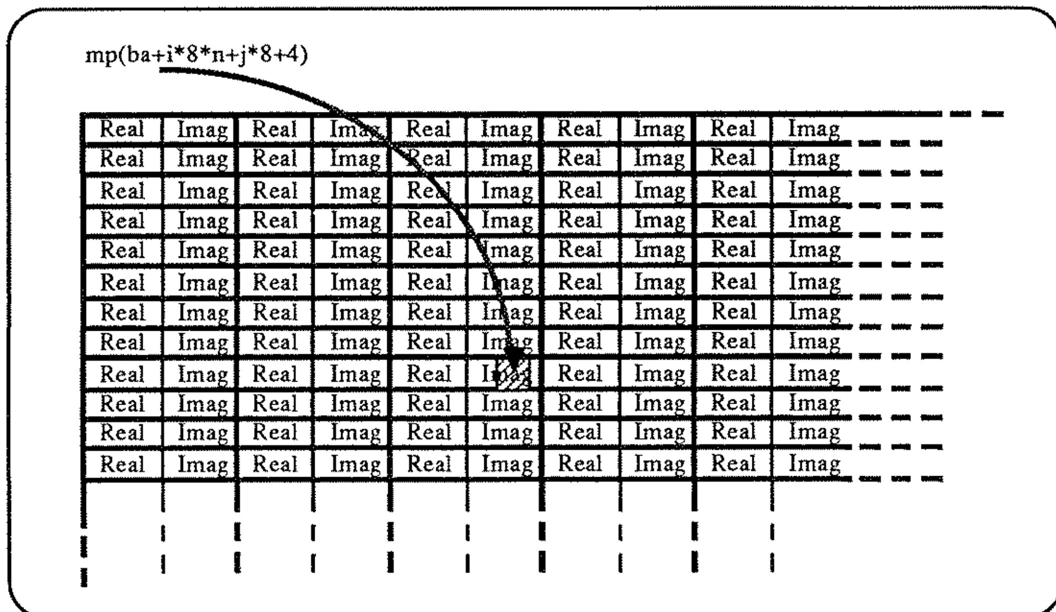

Figure 5.2: Matrix Scheme

Diagrammatically illustrated in Figure 5.2, the values are directly mapped into a matrix of memory locations. The statement *array_variable[i,j].Imag* in this case only needs a one step access that is *mp[ba+i*8*n+j*8+4]* for a read. In the case of the doubly nested loop, the computation involved is:

```
FOR i DO
row := ba + i*8*n + 4

    FOR j DO
    READ mp(row + j*8)
    END FOR

END FOR
```

In addition to the advantage of single step memory access, most computations can be performed in the outer loop for *row* resulting in less busy inner loop thus less number of overall computations. This scheme is therefore much more efficient and faster than Dope Vector scheme. It is clear from here that Matrix scheme will significantly enhance the present inefficient dynamic memory management scheme; the serial tail caused by the deallocation of data structure in the timeloop of the spectral weather model (Figure 4.4), even if the array sizes varied through loop iterations, is expected to be significantly shortened with the implementation of Matrix scheme.

5.2 Problems of OSC

The first released Optimising SISAL Compiler OSC used for the research has incorporated in it many optimisation stages. Nevertheless, given the newness of the compiler, there are still a number of improvements necessary to make the compiler more effective for providing support for parallelisation of SISAL programs and effecting intended speedup on conventional MIMD computers.

5.2.1 Exploitation of Coarser Grain Parallelism for Conventional Multiprocessors

OSC only exploits parallelism from the parallel loop constructs of SISAL. However, some parts in a program may consist of big blocks of mutually independent sequential loops which occupy significant fraction of the program's critical path. For example:

```
variable_a :=        FOR INITIAL
                     .....
                     WHILE ... REPEAT
                     etc
                     END FOR;        % Large sequential loop construct

independent_variable :=        FOR INITIAL
                               .....
                               WHILE ... REPEAT
                               etc
                               END FOR;  % Large sequential loop construct
... etc ...
```

should be able to be processed concurrently to reduce the time by half if the the two loops have equal execution time. OSC does not currently exploit the concurrency available in data independent unfusible sequential loops, forcing unnecessary loop transformations if the code is in the critical path.

5.2.2 Effective Loop Slicing

The initial findings relating to the cost estimation routine of the compiler runtime system in Section 3.2.6.1 indicated that the compiler did not identify the critical path importance of low complexity singly nested parallel loops which resided in highly parallel critical path of the weather code. The attempted solution was at the source level to enlarge the loop body.

5.2.2.1 Programmer's Effort: QDN

The solution was meant to *trick* the cost estimator using QDN technique consisting of a reduction operator, *CATENATE,* which was:

```
FOR array RETURNS VALUE OF CATENATE
    FOR array RETURNS ARRAY OF
        xxxxxxx
    END FOR
END FOR
```

The results in Figure 3.14 and 3.15 while showing improved performance, nevertheless demonstrate the cost of concatenation operations in the reduction of a parallel loop. As illustrated in the diagram in Figure 5.3, where each of a, b, c, and d is an array produced by the inner loop of a QDN loop, the concatenations were actually performed not in parallel but in serial.

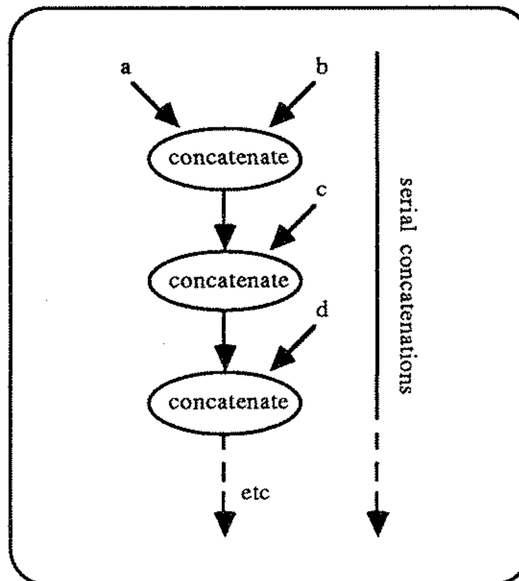

Figure 5.3: Reduction concatenations for a parallel loop

As a result, while the concurrency profile in Figure 3.15 shows that the QDN loop has 60% concurrency when 16 processors are used to share the workload, the execution time curve in Figure 3.14 gives a contradicting achieved speedup, or average parallelism, of only about 3.5; in other words, the QDN technique used in this case was only 22% efficient. This suggests therefore that QDN technique is not an efficient or long term solution. Rather, the solution should come from a better cost estimation scheme.

5.2.2.2 Better Cost Estimation Routine for OSC's Runtime System

Presently, cost estimates theoretically are performed relying on the number of loop iterations, I, and the complexity of the loop body and further information entered by a programmer at compiler time: the *H* cost parameter is the total cost of the loop below which loop slicing will not be performed; the *L* parameter is the depth of the nested loops that the compiler should consider slicing. So only these factors are known to the compiler to estimate the costs of, and to make the decision to slice.

Once slicing has been performed, the slice templates are fixed. It is then up to an application user to increase the problem size (if scalable speedup can be attained) to fill up the templates so as to maximise the *efficiency of parallelised processes,* which may be defined as:

$$\textit{efficiency of parallelised processes} = \frac{\textit{actual work performed in a process}}{\textit{work required to create the process}}$$

Thus, an important parameter that has been overlooked is the number of processors sharing the workload which presently is entered at runtime. It should be included as an additional pragma at compile time to improve cost estimation; this will enable cost estimation to be determined statically thus reducing the amount of runtime decision making. Logically, cost saving from subsequent reduction of runtime overhead may therefore be achieved. Another factor which may be incorporated to contribute to more effective loop slicing is to allow modular compilation with local cost pragmas.

5.2.2.3 Modular Compilation with Local Cost Pragmas

In the previous attempt to effect slicing of small body singly nested loops (in the same experience as above), the original code was locally compiled with maximum slicing imposed by using the *-H1* pragma of OSC. Discussed in Section 3.2.6.1.1, the desired improvement to the code was then achieved as shown by the LMS curves in Figures 3.14 and 3.15. Unfortunately, the present OSC does not support linkages of separately compiled routines, and therefore the amount of slicing of SISAL's parallel loop constructs can only be specified as a globally effective *-H* pragma at compile time. A global pragma value of *-H1* resulted in slicing of the routines of interest but also undesirably led to over-parallelisation of all other routines resulting in a worse performance.

5.2.3. Dynamic Memory Management Scheme

The problems relating to the dynamic memory management scheme of OSC has previously been discussed in detail in various sections including ABD Synthesis for a two dimensional FFT routine and Matrix scheme for better manipulation of array structures.

For the case where array sizes are invariant through loop iteration, it should be possible using code motion and data structure pointer reassignment to remove the allocation and deallocation of fixed size array structures from within iterative loops; the appropriate optimisation by hand at the C level is relatively easy to perform for simple examples. Where the data structure size cannot be determined statically, it is desirable that data deallocation be overlapped with the main computation of the loop body i.e. *lazily* when the storage is exhausted; while this scheme requires further research and in particular its impact on the performance of the runtime software cache, it is possible that Matrix scheme for array manipulation discussed above may improve the performance of the present dynamic memory management scheme.

5.3 Numerical Weather Prediction Model

The realisation that functions within subroutine *Nonlinear* are computationally most intensive and dominate the critical path of the model computation is of vital importance. It led to the research being focused on parallelising and *globalising* these functions to improve the execution speed from the original formulation (Section 2.4.2) which was:

```
For each hemisphere Iterate
    For each latitude Iterate
        SymAsym();
        SpecToFreq();
        MdFFTGrid();
        Vertig();
        MdFFTFreq();
        KeepNH();
        SymAsym();
        FreqToSpec();
    Next latitude
Next hemisphere
```

to a parallel formulation (Section 3.2.3) which is simply:

```
SpecToFreqSphere();
MdFFTGridSphere();
VertigSphere();
MdFFTFreqSphere();
FreqToSpecSphere();
```

The overall results (Section 3.2.7) have confirmed the feasibility of a parallel implementation of the adopted weather model in SISAL. The results have also demonstrated that the parallel computational algorithm derived in this research (Figure 3.10) is faster and is better suited for parallelisation than the original sequential version (Figure 2.4).

The results of parallel execution of the new computational algorithm on multiple processors (the *S16* curve in Figure 4.5) has shown that its growth in execution time with increasing model size is much slower than that for a single processor. This is particularly important for very large model sizes, where very accurate modelling is being considered. It is believed that the impact that the parallel algorithm could have on spectral numerical weather modelling is significant.

5.4 Conclusions

A barotropic numerical weather prediction model has been used as an experimental vehicle to explore the parallelisation of computationally intensive scientific applications. Experiments have been conducted including the direct transliteration of the model from FORTRAN to SISAL to expose the model formulation and its deficiencies. The experiments have resulted in the development of SISAL loop transformation heuristics; the use of these heuristics has led to a highly parallel formulation of the model.

The performance of the parallel implementation has been analysed leading to the identification of the limiting factors to performance due to the formulation of the application, SISAL language, SISAL compiler and OSC runtime system. Solutions to these deficiencies have been recommended and are being considered for incorporation into the current OSC and its runtime system, and SISAL 2.0. The research has found that many of the claims of the SISAL developers are justified subject to the qualifications presented here.

The SISAL formulation of the spectral barotropic numerical weather prediction model is being used as a benchmark code for refining the current OSC and in the development of SISAL 2.0. It has been made available to the parallel computing research community for other computational experiments and development. In particular there has been strong interest from the researchers at the Massachusetts Institute of Technology who intend to recode the SISAL version into ID Nouveau for the Monsoon dataflow computer. Their studies will supplement comparative studies between ID and SISAL currently being conducted on the CSIRAC II dataflow computer.

The research presented in this thesis leads to continuing application studies in the parallelisation of a next generation multi-level numerical weather prediction grid model under a joint collaboration research between the Laboratory for Concurrent Computing Systems and the Melbourne Bureau of Meteorology Research Centre.

Appendix A

BUGS IN OSC

The OSC compiler used during the course of the research was the first released version which we received in mid 1989. Presented in this appendix are the bugs encountered in the experience in implementing a two dimensional FFT routine and a spectral barotropic numerical weather prediction model in SISAL [CGA90]. These bugs were fixed [CV90] recently after they have been notified to, and acknowledged by, the OSC's developers at the Asilomar SISAL Workshop in May 1990.

A.1 Starting Index of "FOR array RETURNS VALUE OF CATENATE"

The expected results would be an array with a starting index of 0 if one wrote:

```
FOR i IN 0, bound
RETURNS VALUE OF CATENATE i
END FOR
```

However, the front end SISAL compiler generated IF1 graphs which had a starting index of 1. Additionally, both the Dataflow Interpreter and the C code generated by OSC gave the results with a starting index of 1 even if the lower bound in the IF1 graphs was manually set to 0. This is disastrous for computations which habitually consisted of arrays whose intended starting indices were 0, such as FFT routine.

The case study as shown in Figures A.1, A.2 and A.3 shows that the IF1 code generated by SISAL frontend sets the lower bound of the concatenation result to 1 regardless. Further, even if the low bound was altered to 0 in the IF1 code, both DI and OSC did not check this lower bound given in the IF1 code, but rather simply set it, again regardlessly, to 1.

One was therefore forced to always use *array_setl* to set the desired lower bound when any loop returning 'value of catenate' was used.

```
FOR i IN 0, 10
RETURNS VALUE OF CATENATE
      FOR j IN 0,10
      RETURNS VALUE OF j
      END FOR
END FOR
```

Figure A.1: SISAL code: parallel loop returning value of catenate with intended starting index 0

```
[ 1:  0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5 6 7 8 9 10
      0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5 6 7 8 9 10
      0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5 6 7 8 9 10
      0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5 6 7 8 9 10
      0 1 2 3 4 5 6 7 8 9 10 0 1 2 3 4 5 6 7 8 9 10
      0 1 2 3 4 5 6 7 8 9 10 ]
```

Figure A.2: Results produced by DI and OSC

```
T 1 1 0   %na=Boolean
T 2 1 1   %na=Character
T 3 1 2   %na=Double
T 4 1 3   %na=Integer
T 5 1 4   %na=Null
T 6 1 5   %na=Real
T 7 1 6   %na=WildBasic
T 8 10
T 9 0        4
T 10 8       9        0
T 11 3       0        10
T 12 4       4
T 13 8       9        10
T 14 3       13       10
T 15 4       9
C$ C Faked IF1CHECK
C$ D Nodes are DFOrdered
C$  E Common Subs Removed
C$  F Livermore Frontend   Version1.8
C$  G Constant Folding Completed
C$  L Loop Invars Removed
C$  O Offsets Assigned
X    1 1      "main"  %ar=13  %sl=3
E    4 1      0 1      9        %of=1   %mk=V
{ Compound  1  0
G    0        %fq=  0.00000000000000e+00        %ep=0
E    1 1      0 1      12        %na=j  %of=2    %mk=V
N 1  142
L             1 1      4 "0"    %of=3   %mk=V
L             1 2      4 "10"   %of=4   %mk=V
G    0        %fq=  0.00000000000000e+00        %ep=0
G    0        %fq=  0.00000000000000e+00        %ep=0
E    1 1      0 1      9        %of=5   %mk=V
N 1  107
L             1 1      4 "1"    %of=6   %mk=V
E    0 1      1 2      12        %na=j  %of=2    %mk=V  %sl=7
} 1 0 3 0 1 2
N 2  103
L             2 1      4 "1"    %of=7   %mk=V
N 3  115
E    1 1      3 1      9        %of=5   %mk=V
L             3 2      4 "0"    %of=8   %mk=V
{ Compound  4  0
G    0        %sl=5
E    1 1      0 3      12        %na=i  %of=11  %mk=V
```

```
N 1  142        %sl=5
L               1 1     4 "0"     %of=12  %mk=V
L               1 2     4 "10"    %of=13  %mk=V
G    0          %sl=5
E    0 2        0 4     9         %of=10  %mk=V
G    0          %sl=5
E    1 1        0 1     9         %of=1   %mk=V
N 1  149        %sl=9
L               1 1     14 "CATENATE"     %mk=V
E    0 1        1 2     9         %of=9   %mk=V
E    0 4        1 3     15        %of=10  %mk=V
} 4 0 3 0 1 2   %sl=5
E    2 1        4 1     9         %of=9   %mk=V
E    3 1        4 2     9         %of=10  %mk=V
```

Figure A.3: Corresponding IF1 code

A.2 "FOR array RETURNS VALUE OF CATENATE of concatenations of vectors"

This example arose from coding a two dimensional FFT in SISAL. At compilation time, the process passed through SISAL frontend compiler and the optimisation stages without any indication of problems, but during CC, the CGEN generated several errors relating to pointers. The problem is shown in Figure A.4.

The compilation of the code passed through the SISAL frontend compiler and all of the optimisation stages, but during CC, it was terminated due to some *struct/union* errors (Figure A.5) generated by CGEN. The problem embedded in:

```
FOR loop
RETURNS ARRAY OF
        FOR index IN lowerbound, upperbound
        vecvec := vector || vector              % concatenation
        RETURNS VALUE OF CATENATE vecvec
        END FOR
END FOR
```

```
DEFINE main
TYPE ArrIntl = ARRAY [integer];
TYPE ArrReal = ARRAY [real];
TYPE ArrReal2 = ARRAY [ArrReal]
GLOBAL SIN(num: real returns real)
GLOBAL COS(num: real returns real)
GLOBAL ATAN(num: real returns real)
GLOBAL SQRT(num: real returns real)

FUNCTION main(RETURNS ArrReal,ArrReal)
LET     n := 4; pi := 3.141593;
        twopow :=      FOR INITIAL          i:=0; pow:=1; two := array_fill(0,n,1);
                       WHILE i<n REPEAT     i:=old i+1; pow := old pow*2;
                       two := old two[i: pow];
                       RETURNS VALUE OF two
                       END FOR;
        Areal,Aimag:=  FOR row IN 0, twopow[n] - 1 CROSS col IN 0, twopow[n] - 1
                       RETURNS ARRAY OF  IF row<twopow[n]/2 THEN 5.0 ELSE 0.0 END IF
                               ARRAY OF  IF row<twopow[n]/2 THEN 5.0 ELSE 0.0 END IF
                       END FOR;
IN      LET
        stage := 2; off := twopow[n - stage];
        upperboundjump := twopow[stage - 1] - 1; jumpby := twopow[n - stage + 1];
        AR1, AI1:=    FOR indexjump IN 0, upperboundjump
                      jump := indexjump * jumpby;
                      Rwing1R, Rwing1I, Rwing2R, Rwing2I :=
                          FOR x IN 0, off - 1
                          p1 := x + jump; p2 := p1 + off;
                          W := pi * REAL(x) / REAL(off); cosine, sine := COS(W), SIN(W);
                          Lwing1R, Lwing1I, Lwing2R, Lwing2I :=
                                 Areal[1, p1], Aimag[1, p1], Areal[1, p2], Aimag[1, p2];
                          realm, imagm := Lwing1R - Lwing2R, Lwing1I - Lwing2I;
                          RETURNS      ARRAY OF Lwing1R + Lwing2R
                                       ARRAY OF Lwing1I + Lwing2I
                                       ARRAY OF realm*cosine + imagm*sine
                                       ARRAY OF imagm*cosine - realm*sine
                          END FOR;

                      % Error spot: The focus is on concatenations
                      groupR := Rwing1R || Rwing2R;       % This creates error in cc
                      groupI := Rwing1I || Rwing2I;
                      % The inexplicable solution:
                      %  groupR,groupI := for kk in 0, 2*off - 1
                      %  grR, grI := IF kk < off THEN Rwing1R[kk],Rwing1I[kk]
                      %                         ELSE Rwing2R[kk-off],Rwing2I[kk-off] END IF;
                      %  RETURNS       ARRAY of grR
                      %                ARRAY OF grI
                      %  END FOR;
                      %
                      % The drawback here is that one needs to know the actual array size of
                      % "groupR" and "groupI" ie 2*off - 1

                      RETURNS         VALUE OF CATENATE groupR
                                      VALUE OF CATENATE groupI
                      END FOR;
        IN AR1, AI1
        END LET
END LET
END FUNCTION
```

Figure A.4: Error producing SISAL code

```
osc chip.sis -v
sisal -noopt -nooff -dir /usr/local/sisal chip.sis
 LL Parse, using binary files
 * Reading file: chip.sis...

version 1.8      (Mar 28, 1989)

 accepted
   81 lines in program
   0 errors ( calls to corrector)
    0 tokens inserted;    0 tokens deleted.
   0 semantic errors

if1ld -o chip.mono -e main chip.if1
if1opt chip.mono chip.opt -l -e
unlink chip.mono
if2mem chip.opt chip.mem
unlink chip.opt
if2up chip.mem chip.up
unlink chip.mem
if2part /y/rco/rcodf/sisal/release/OSC_csu/bin/s.costs chip.up chip.part -L0
unlink chip.up
if2gen chip.part chip.c -b
unlink chip.part
cc -I/y/rco/rcodf/sisal/release/OSC_csu/bin -DSUN3 -f68881 -O -S chip.c
%"chip.c", line 229: nonunique name demands struct/union or struct/union pointer
%"chip.c", line 230: nonunique name demands struct/union or struct/union pointer
%"chip.c", line 262: nonunique name demands struct/union or struct/union pointer
%"chip.c", line 264: nonunique name demands struct/union or struct/union pointer
 ** COMPILATION ABORTED **
```

Figure A.5: Error messages given at compile time

A.3 Incomplete Graph Normalisation

In the initialisation section of the weather simulation implementation in SISAL, loops of similar loop bound were forced to be coupled together in order to be accepted and to pass through the OSC compiler. The full code listed in Section A.3.1 belongs to an older version of the initialisation routines but adequately exhibits the fault. The focus of this example is in the calculation of the variance *var* and the average potential height *h*. Figure A.6 is an extract of the error producing code. In attempting to simplify the program in order to isolate the source of error, the problem disappeared. This suggests that the *complexity* of the program could be a factor. When these two statements were stated separately in the program, IF1OPT [Cann89] failed, giving the error message shown in Figure A.8. This seems to suggest that *Graph Normalisation* was incomplete within IF1OPT.

```
var :=  FOR diffindex IN 2, jxmx
        RETURNS VALUE OF SUM CabsSqr(zt_mountain[diffindex])
        END FOR;
h :=    FOR index IN 1, jxmx
        RETURNS ARRAY OF Crmul(constant, zt_mountain[index])
        END FOR;
```

Figure A.6: Error producing region in the initialisation section

One way to get around this problem was to *couple* loops of similar loop bound together as shown in Figure A.7. This avoided any error.

```
var, h :=  FOR index IN 1, jxmx
           RETURNS  VALUE OF SUM  IF index = 1 THEN 0.0
                                  ELSE CabsSqr(zt_mountain[index]) END IF
                    ARRAY OF Crmul(constant, zt_mountain[index])
           END FOR;
```

Figure A.7: Immediate solution

```
osc main.sis -IF1 -double_real
LL Parse, using binary files
* Reading file: main.sis...

version 1.8     (Mar 28, 1989)

 accepted
  226 lines in program
 0 errors ( calls to corrector)
 0 tokens inserted;   0 tokens deleted.
 0 semantic errors
osc -v -o prefft main.ifl IntrFuncs.ifl complex.ifl  Inital.ifl
                InitFFT.ifl gaussg.ifl  legendre.ifl SasAlfa.ifl
if1ld -o main.mono -e main main.ifl IntrFuncs.ifl complex.ifl
     Inital.ifl InitFFT.ifl gaussg.ifl legendre.ifl SasAlfa.ifl
if1opt main.mono main.opt -l -e

if1opt: E - FORALL RETURN SUBGRAPHS NOT NORMALIZED

** COMPILATION ABORTED **

*** Error code 1
stop.
```

Figure A.8: Error message for the subgraph normalisation error

A.3.1 Joint Routines Which Produces *Normalisation Error*

% Author: Pau S. Chang
% Laboratory for Concurrent Computing Systems
% Revised: 2/2/1990
% Module: The initialisation stage of the Spectral Weather Model.

% filename: Makefile

```
# makefile for the SISAL codes barotropic model Version 1.8

# Let
iflfiles =      main.ifl IntrFuncs.ifl complex.ifl \
                Inital.ifl InitFFT.ifl gaussg.ifl \
                legendre.ifl SasAlfa.ifl

.SUFFIXES:
.SUFFIXES:      .sis .ifl

# Compile from .sis files to .ifl files
.sis.ifl:
                osc $*.sis -IF1 -double_real

prefft:         $(iflfiles)
                osc -v -o prefft $(iflfiles)
```

%Filename: IntrFuncs.sis

```
DEFINE ASINR, ACOSR, SQRTR, DSIN, DCOS

% ---------Intrinsic Functions
global SIN(num : real RETURNS real)
global COS(num : real RETURNS real)
global ASIN(num : double_real RETURNS double_real)
global ACOS(num : double_real RETURNS double_real)
global SQRT(num : double_real RETURNS double_real)

% Catering for real operations of Intrinsic functions
function ASINR(num : real RETURNS real)
real(ASIN(double_real(num)))
end function

function ACOSR(num : real RETURNS real)
real(ACOS(double_real(num)))
end function

function SQRTR(num : real RETURNS real)
real(SQRT(double_real(num)))
end function

% Catering for double_real operations of Intrinsic functions
function DSIN(num : double_real RETURNS double_real)
double_real(SIN(real(num)))
end function

function DCOS(num : double_real RETURNS double_real)
double_real(COS(real(num)))
end function
```

```
%Filename: complex.sis

DEFINE Cadd, Csub, Cmul, Cdiv, Crmul, Crsub, Crdiv, Conjg, Cneg, Csqrt, Cabs, CabsSqr

type CplexReal = Record[Repart,Impart:real];
type ArrCplexReal = Array[CplexReal];

% ---------Intrinsic Functions
global SIN(num : real RETURNS real)
global COS(num : real RETURNS real)
global ATAN(num : real RETURNS real)
global SQRTR(num : real RETURNS real)

% These subroutines do the arithmatics of complex numbers:

% cnum1 + cnum2
function Cadd(cnum1, cnum2 : CplexReal RETURNS CplexReal)
record CplexReal[ Repart : cnum1.Repart + cnum2.Repart; Impart : cnum1.Impart + cnum2.Impart ]
end function

% cnum1 - cnum2
function Csub(cnum1, cnum2 : CplexReal RETURNS CplexReal)
record CplexReal[ Repart : cnum1.Repart - cnum2.Repart;  Impart : cnum1.Impart - cnum2.Impart ]
end function

% cnum1 * cnum2
function Cmul(cnum1, cnum2 : CplexReal RETURNS CplexReal)
record CplexReal[ Repart : cnum1.Repart * cnum2.Repart - cnum1.Impart * cnum2.Impart;
                  Impart : cnum1.Repart * cnum2.Impart  +  cnum1.Impart * cnum2.Repart  ]
end function

% cnum1/cnum2
function Cdiv(cnum1, cnum2 : CplexReal RETURNS CplexReal)
LET dnom:= cnum2.Repart * cnum2.Repart + cnum2.Impart * cnum2.Impart;
IN record CplexReal[ Repart : (cnum1.Repart*cnum2.Repart + cnum1.Impart*cnum2.Impart) / dnom;
                     Impart : (cnum1.Impart*cnum2.Repart -  cnum1.Repart*cnum2.Impart) / dnom ]
end LET
end function

% Real_constant*cnum
function Crmul(cons : real; cnum : CplexReal RETURNS CplexReal)
record CplexReal[ Repart : cons * cnum.Repart; Impart : cons * cnum.Impart ]
end function

% cnum-Real_constant
function Crsub(cnum : CplexReal; cons : real RETURNS CplexReal)
record CplexReal[ Repart : cnum.Repart-cons; Impart : cnum.Impart ]
end function

% cnum/Real_constant
function Crdiv(cnum : CplexReal; cons : real RETURNS CplexReal)
record CplexReal[ Repart : cnum.Repart / cons; Impart : cnum.Impart / cons ]
end function

% conjugate(cnum)=Repart-Impart
function Conjg(cnum : CplexReal RETURNS CplexReal)
record CplexReal[ Repart : cnum.Repart; Impart : -cnum.Impart ]
end function

% Cneg(cnum)
function Cneg(cnum : CplexReal RETURNS CplexReal)
record CplexReal[ Repart : -cnum.Repart; Impart : -cnum.Impart ]
end function
```

```
% Csqrt(cnum)
function Csqrt(cnum:CplexReal RETURNS CplexReal)
LET    RR := cnum.Repart;
       II := cnum.Impart;
       mag := SQRTR( SQRTR(RR * RR + II * II) );
       angle := ATAN(II / RR) / 2.0;
       Re, Im := mag * COS(angle), mag * SIN(angle);
IN record CplexReal[Repart : Re; Impart : Im]
end LET
end function

% Cabs(cnum) refers to the MAGNITUDE of the complex number.
function Cabs(cnum : CplexReal RETURNS real)
SQRTR(cnum.Repart * cnum.Repart + cnum.Impart * cnum.Impart)
end function

% CabsSqr(cnum) refers to the MAGNITUDE Square of the complex number.
function CabsSqr(cnum : CplexReal RETURNS real)
cnum.Repart * cnum.Repart + cnum.Impart * cnum.Impart
end function

%Filename: Inital.sis_______________________________________________

DEFINE Inital

type ArrIntl = Array[integer];
type ArrReall = Array[real];

global SQRTR(num : real RETURNS real)

FUNCTION Inital(ir, ilong, ilat, mx, jx, jxx : integer; zmeanl : real
        RETURNS integer, integer, integer, integer, real, real, real, real, real, arrintl, arrintl, arrintl, arrreall)
LET
ww := 7.292E-5;
tw := ww * 2.0;
irmax:= ir;

ilath, irmax1, irmax2 := ilat / 2, irmax + 1, irmax + 2;

asq, grav := 6371.22E3 * 6371.22E3, 9.80616;
zmean:= zmeanl * grav / asq;

kmjx, kmjxx :=      FOR m IN 1, mx
                    RETURNS ARRAY of (m - 1) * jx
                            ARRAY of (m - 1) * jxx
                    END FOR;

ksq :=   FOR j IN 1, ir * 2
         RETURNS ARRAY of j * (j + 1)
         END FOR;

epsilon_a :=        FOR mp IN 1, mx
                    RETURNS VALUE of CATENATE
                       FOR j IN 1, jxx    % epsilon_size is 1-272
                       l := j + mp - 2;
                       m := mp - 1;
                       t := real((l + m) * (l - m));
                       b := real(4 * l * l - 1);
                        RETURNS ARRAY of SQRTR(t / b)
                       END FOR
                    END FOR;

epsilon := epsilon_a[1 : 0.0];

IN ir, ilong, ilath, irmax2, ww, zmean, tw, asq, grav, kmjx, kmjxx, ksq, epsilon

END LET
END FUNCTION
```

```
%Filename: InitFFT.sis

DEFINE InitFFT

type ArrIntl = Array[integer];
type ArrReall = Array[real]

global SIN(num : real RETURNS real)
global COS(num : real RETURNS real)

%----------------factr4/facStep/facRecur
function facRecur(nparti, idiv, ifTi : integer; ifacti : ArrIntl
                        RETURNS integer, integer, integer, ArrIntl)
FOR INITIAL
npart := nparti;
iquot := npart / idiv;
ifT := ifTi;
ifactl := ifacti;

WHILE npart - idiv * iquot = 0 REPEAT
npart := old iquot;
iquot := npart / idiv;
ifT := old ifT + 1;
ifactl := old ifactl[ifT : idiv]

RETURNS VALUE of npart
        VALUE of iquot
        VALUE of ifT
        VALUE of ifactl
END FOR
END function %facRecur

%----------------factr4/facStep
function Loop_id(n : integer RETURNS integer, integer, ArrIntl)
FOR INITIAL % loop_id
id := 1;
ifT := 0;
npart := n;
ifact := ARRAY_fill(1, 20, 0)    % NOTE: wild guess

WHILE id <= n REPEAT
idiv := IF old id - 1 <= 0 THEN 2 ELSE old id
        END IF;

npart, iquot, ifT, ifact := facRecur(old npart, idiv, old ifT, old ifact);

id := IF iquot - idiv <= 0 THEN n + 1 % just to make it greater than n ELSE old id + 2
      END IF;

RETURNS VALUE of npart
            VALUE of ifT
            VALUE of ifact
END FOR
END function          % Loop_id
```

```
%------------------------
function FACTR4(n : integer RETURNS integer, arrintl)
LET
npart, ifT, ifactl := Loop_id(n);

iff := if npart - 1 > 0 then ifT + 1else ifT END if;
ifact2 := if npart - 1 > 0 then ifactl[iff : npart] else ifactl END if;

nfactT := iff;

n2 :=   FOR INITIAL
        n2 := 0;
        i := 1;

        % n2 includes case i=nfactT
        WHILE i <= nfactT REPEAT
        i := old i + 1;
        n2 := if ifact2[old i] = 2 then old n2 + 1 else old n2 END if
        RETURNS VALUE of n2
        END FOR;        % NOTE: very ineffecient!

n4 := n2 / 2;
ifact3 := ARRAY_fill(1, n4, 4)
        ||
        for i in n4 + 1, nfactT - n4
        RETURNS ARRAY of ifact2[n4 + i]
        END for
        ||
        ARRAY_fill(nfactT - n4 + 1, nfactT, 0);

nfact := nfactT - n4;

IN nfact, ifact3

END LET
END function %factr4

% Subroutine InitFFT does the initialisations necessary so that the
% FFT's can be used. It factorises the number of longitudinal points.
% TRIGF are for forward transforms while TRIGB are for reverse.

function InitFFT(n : integer RETURNS boolean, boolean, integer, arrintl, ArrReall, ArrReall)
LET
Abortinitfft :=     IF (MOD(n, 2) ~= 0 | n > 200)       THEN true ELSE false
                    END IF;
AbortFFT := IF n > 96 THEN true ELSE false END IF;
pi := 3.14159265;

nfax, ifax := FACTR4(n);

trigf, trigb :=
  IF Abortinitfft
  THEN array ArrReall [], array ArrReall []

  ELSE  FOR Lp IN 1, n
          k := (Lp + 1) / 2;
          Cargument := - 2.0 * pi * real(k - 1)/real(n);
          COStheta := COS(Cargument);     % Repart
          SINtheta := SIN(Cargument);     % Impart

          RETURNS ARRAY of IF MOD(Lp, 2) = 0 THEN SINtheta ELSE COStheta END IF
                  ARRAY of IF MOD(Lp, 2) = 0 THEN - SINtheta ELSE COStheta END IF
        END FOR
  END IF

IN AbortFFT, Abortinitfft, nfax, ifax, trigf, trigb

END LET
END function
```

```
%Filename: gaussg.sis_______________________________________________

DEFINE gaussg

type ArrReal1 = Array[real];
type ArrDreal1 = Array[double_real]

global ACOS(num : double_real RETURNS double_real)
global SQRT(num : double_real RETURNS double_real)

global SIN(num : double_real RETURNS double_real)
global COS(num : double_real RETURNS double_real)

FUNCTION ORDLEG(ir : integer; coa : double_real RETURNS double_real)
LET
irpp, irppm := ir+1, ir;
delta := ACOS(coa);
sqr2 := SQRT(2.0d0);
theta := delta;
c1 := sqr2 *        FOR n IN 1, irppm
                    fn := n;
                    fn2 := fn * 2;
                    fn2sq := double_real(fn2 * fn2);
                    RETURNS VALUE of product SQRT(1.0d0 - 1.0d0 / fn2sq)
                    END FOR;

s1 :=      FOR INITIAL
             n := irppm;
             fn := double_real(irppm);
             fn2 := fn * 2.0d0;
             ang := fn * theta;
             s1T := 0.0d0;
             c4 := 1.0d0;
             a := -1.0d0;
             b := 0.0d0;
             n1 := n + 1;
             kk := 1;
           WHILE kk <= n1 REPEAT
             kk := old kk + 2;
             k := old kk - 1;
             c4T := IF k=n THEN 0.5d0 * old c4 ELSE old c4 END IF;
             s1T := old s1T + c4T * COS(old ang);
             a := old a + 2.0d0;
             b := old b + 1.0d0;
             fk := double_real(k);
             ang := theta * (fn - fk - 2.0d0);
             c4 := a * (fn - b + 1.0d0) / (b * (fn2 - a)) * c4T;
             RETURNS VALUE OF s1T
             END FOR;
sx := s1 * c1;
IN sx
END LET
END FUNCTION


%-----------------gaussg/cycle
FUNCTION CYCLE(ir, irm, irp : integer; ft, a, b, xlim : double_real RETURNS double_real)
LET    g := ORDLEG(ir, ft);
       gm := ORDLEG(irm, ft);
       gp := ORDLEG(irp, ft);
       gt := (ft * ft - 1.0d0)/(a * gp - b * gm);
       ftemp := ft - g * gt;
       gtemp := ft - ftemp;
       ftnew := ftemp;
IN IF ABS(gtemp) - xlim > 0.0d0 THEN CYCLE(ir, irm, irp, ftnew, a, b, xlim) ELSE ftnew END IF
END LET
END FUNCTION
```

```
%-------------------gaussg
FUNCTION gaussg(nzero : integer RETURNS ArrReal1, ArrReal1, ArrReal1, ArrReal1, ArrReal1, ArrDreal1)
LET
xlim := 1.0d-12;
ir := nzero * 2;
fi := double_real(ir);
fi1 := fi + 1.0d0;
pi := 3.141592653589793d0;
piov2 := pi * 0.5d0;
fn := piov2/double_real(nzero);

wt :=    FOR lat IN 1,nzero
         RETURNS ARRAY of double_real(lat) - 0.5d0
         END FOR;

f :=     FOR lat IN 1,nzero
         RETURNS ARRAY of SIN( wt[lat] * fn + piov2 )
         END FOR;

dn := fi/SQRT(4.0d0 * fi * fi - 1.0d0);
dn1 := fi1/SQRT(4.0d0 * fi1 * fi1 - 1.0d0);
a := dn1 * fi;
b := dn * fi1;
irp := ir + 1;
irm := ir - 1;

fnew := FOR lat IN 1, nzero
        RETURNS ARRAY of CYCLE(ir, irm, irp, f[lat], a, b, xlim)
        END FOR;

wtnew, radnew, coangnew, sianew :=
        FOR lat IN 1, nzero
        a1 := 2.0d0 * (1.0d0 - fnew[lat] * fnew[lat]);
        bo := ORDLEG(irm, fnew[lat]);
        b1 := bo * bo * fi * fi;
        wtt := a1 * (fi - 0.5d0) / b1;
        radt := ACOS(fnew[lat]);
        coangt := radt * 180.0d0 / pi;
        siat := SIN(radt);
        RETURNS ARRAY of wtt
                ARRAY of radt
                ARRAY of coangt
                ARRAY of siat
        END FOR;

WORKiyh := fnew || wtnew || sianew || radnew || coangnew;

fs, wts, sias, rads, coangs :=
        FOR lat IN 1, nzero
        RETURNS ARRAY of real(fnew[lat])
                ARRAY of real(wtnew[lat])
                ARRAY of real(sianew[lat])
                ARRAY of real(radnew[lat])
                ARRAY of real(coangnew[lat])
        END FOR;

IN fs, wts, sias, rads, coangs, WORKiyh
END LET
END FUNCTION
```

```
%Filename: legendre.sis

DEFINE legendre

type ArrDreal1 = Array[Double_real]

global SIN(num : double_real RETURNS double_real)
global COS(num : double_real RETURNS double_real)
global SQRT(num : double_real RETURNS double_real)

FUNCTION legendre(ir, irmax2, jxxmx : integer; coas, sias, deltas : real; RETURNS ArrDreal1)
LET
p := LET
   coa := double_real(coas);
   sia := double_real(sias);
   delta := double_real(deltas);
   irpp := ir + 2;
   theta := delta;
   sqr2 := SQRT(2.0d0);
   pp := FOR INITIAL
         n := 1;
         c1 := sqr2;
         pLoop1 := ARRAY ArrDreal1[1: 1.0d0 / sqr2]  ||  FOR jm IN 2, jxxmx
                                                         RETURNS ARRAY of 0.0d0
                                                         END FOR;
         WHILE n <= irpp REPEAT
         n := old n + 1;
         fn := double_real(old n);
         fn2 := 2.0d0 * fn;
         fn2sq := fn2 * fn2;
         c1 := old c1 * SQRT(1.0d0 - 1.0d0 / fn2sq);
         c3 := c1 / SQRT(fn * (fn + 1.0d0));

         s1, s2 := FOR INITIAL
                   kk := 1;
                   ang := fn * theta;
                   n1 := old n + 1;
                   ss1, ss2 := 0.0d0, 0.0d0;
                   c4, c5 := 1.0d0, fn;
                   a, b := - 1.0d0, 0.0d0;

                   WHILE kk <= n1 REPEAT
                   kk := old kk + 2;
                   k := old kk - 1;
                   ss2 := old ss2 + old c5 * SIN(old ang) * old c4;
                   c4t :=    if k = old n  then 0.5d0 * old c4  else old c4
                             end if;
                   ss1 := old ss1 + c4t * COS(old ang);
                   a := old a + 2.0d0;
                   b := old b + 1.0d0;
                   fk := double_real(k);
                   ang := theta * (fn - fk - 2.0d0);
                   c4 := (a * (fn - b + 1.0d0) / (b * (fn2 - a))) * c4t;
                   c5 := old c5 - 2.0d0

                   RETURNS VALUE of ss1
                           VALUE of ss2        % to s1 and s2
                   END FOR;

         pLoop1 := IF old n - irpp < 0
                   THEN old pLoop1[old n + 1 : s1 * c1; old n + irmax2 : s2 * c3]
                   ELSEIF old n - irpp = 0 THEN old pLoop1[old n + irmax2 : s2 * c3]
                   ELSE old pLoop1
                   END IF

         RETURNS VALUE of pLoop1     % to pp
         END FOR;
```

```
p2 :=   IF ir = 2 THEN pp
        ELSE  FOR INITIAL
              m := 2;
              ppp := pp

              WHILE m <= ir REPEAT
              m := old m + 1;
              fm := double_real(old m);
              fm1, fm2, fm3 := fm - 1.0d0, fm - 2.0d0, fm - 3.0d0;
              mm1 := old m - 1;
              m1 := old m + 1;
              c6 := SQRT((2.0d0 * fm + 1.0d0) / (2.0d0 * fm));
              p5 := old ppp[irmax2 * old m + 1 : c6 * sia * old ppp[irmax2 * mm1 + 1]];
              mpir := old m + ir + 1;
              mt := old m;

              ppp :=
                FOR INITIAL
                l := m1;
                p4 := p5;

                WHILE l <= mpir REPEAT
                l := old l + 1;
                fn := double_real(old l);
                c7 := (fn * 2.0d0 + 1.0d0) / (fn * 2.0d0 - 1.0d0);
                c8 := (fm1 + fn) / ((fm + fn) * (fm2 + fn));
                c := SQRT((fn * 2.0d0 + 1.0d0) /  (fn * 2.0d0 - 3.0d0) * c8 * (fm3 + fn));
                d := SQRT(c7 * c8 * (fn - fm1));
                e := SQRT(c7 * (fn - fm) / (fn + fm));
                lm := irmax2 * mt + old l - mt + 1;
                lmm2 := irmax2 * (mt - 2) + old l - mt + 3;
                lm1mm2 := lmm2 - 1;
                lm2mm2  :=  lm1mm2 - 1;
                lm1m := lm - 1;

                p4 :=  IF old l - mpir < 0
                       THEN old p4[lm:c * old p4[lm2mm2] - d * old p4[lm1mm2] * coa
                                         + e * old p4[lm1m] * coa]
                       ELSEIF old l - mpir > 0 THEN old p4
                       ELSE LET       a := SQRT((fn * fn - 0.25d0) / (fn * fn - fm * fm));
                                      b := SQRT((2.0d0 * fn + 1.0d0) * (fn - fm - 1.0d0) * (fn + fm1)
                                                 / ((2.0d0 * fn - 3.0d0) * (fn - fm) * (fn + fm)));
                                      lm2m := lm1m - 1;

                              IN old p4[lm : 2.0d0 * a * coa * old p4[lm1m]  - b * old p4[lm2m]]
                              END LET
                       END IF

                 RETURNS VALUE of p4          % to p6
                END FOR;
              RETURNS VALUE of ppp                            % to p2
              END FOR
        END IF
  IN p2
  END LET;             % RETURNS p2 to p
IN p
END LET
END FUNCTION
```

```
%Filename: SasAlfa.sis

DEFINE SasAlfa
TYPE arrDreal1 = ARRAY [double_real];
TYPE arrDreal2 = ARRAY [arrDreal1];
TYPE arrDreal3 = ARRAY [arrDreal2];
TYPE arrreal1 = ARRAY [real];
TYPE arrreal2 = ARRAY [arrreal1];
TYPE arrreal3 = ARRAY [arrreal2]

FUNCTION SasAlfa(ir, irmax2, jxxmx, ilath : integer; alp : ArrDReal2
            RETURNS ArrReal3)
LET
lpfin := IF MOD(ir, 2) = 0 THEN ir + 1 ELSE ir + 2 END IF;

alfa :=   FOR hemi IN 1, 2 CROSS latlev IN 1,ilath
        RETURNS ARRAY of IF hemi = 1          % North
                THEN  FOR specindex IN 1, jxxmx
                      RETURNS ARRAY of real(alp[latlev, specindex])
                      END FOR
                ELSE  FOR mp IN 1, ir + 1      % South
                      RETURNS VALUE of CATENATE
                            FOR lp IN 1, lpfin
                            ilm := (mp - 1) * irmax2 + lp;
                            RETURNS ARRAY of IF lp = 1 | MOD(lp, 2) ~= 0
                                              THEN real(alp[latlev, ilm])
                                              ELSE real(-alp[latlev, ilm])
                                              END IF
                            END FOR
                      END FOR
                END IF
        END FOR
IN alfa
END LET
END FUNCTION
```

```
% Filename: main.sis
% Main Program

DEFINE MAIN

type ArrInt1 = Array[integer];
type ArrReal1 = Array[real];
type ArrReal2 = Array[ArrReal1];
type ArrReal3 = Array[ArrReal2];
type ArrDreal1 = Array[Double_real];
type ArrDreal2 = Array[ArrDreal1];
type CplexReal = Record[Repart,Impart:real];
type ArrCplexReal = Array[CplexReal];

global SIN(num : real RETURNS real)
global ACOSR(num : real RETURNS real)

global Cadd(cnum1, cnum2 : CplexReal RETURNS CplexReal)
global Crmul(cons : real; cnum : CplexReal RETURNS CplexReal)
global CabsSqr(cnum : CplexReal RETURNS real)

global Inital(ires, ix, iy, mx, jx, jxx : integer; zmean1 : real
        RETURNS integer, integer, integer, integer, real, real, real, real, real, arrint1, arrint1, arrint1, arrreal1)

global InitFFT( n : integer RETURNS boolean, boolean, integer, arrint1, ArrReal1, ArrReal1)

global gaussg(nzero : integer RETURNS ArrReal1, ArrReal1, ArrReal1, ArrReal1, ArrReal1, ArrDreal1)

global legendre(ir, irmax2, jxxmx : integer; coas, sias, deltas : real RETURNS ArrDreal1)

global SasAlfa(  ir, irmax2, jxxmx, ilath : integer;  alp_double: ArrDReal2   RETURNS ArrReal3)
```

```
function MAIN(
        ires, ix, iy,
        ktotal, idelt, idumpt_i, nrun, imp, istart, izon, ilin:integer;
        zmean_1, hdiff, hdrag, vnu:real;
        p_in, c_in, z_in, zt_mountain:ArrCplexReal
        RETURNS integer,
        integer, integer, integer, integer, integer, integer,
         integer, integer, integer, integer, integer, integer, integer,
         integer, integer, integer, integer, integer, integer, integer,
         integer, integer, integer, real, real, real, real, real, real,
         real, real, ArrIntl, ArrIntl, ArrIntl, ArrIntl, ArrReall, ArrReall,
         ArrReal3, ArrCplexReal, ArrCplexReal, ArrCplexReal, ArrCplexReal,
         ArrReall, ArrCplexReal, ArrCplexReal, ArrCplexReal,
         ArrReall, ArrReall)

LET
ixh := ix/2;
iyh := iy/2;
jxx := ires + 2;
jx := ires + 1;
mx := ires + 1;
jxxmx := jxx * mx;
jxmx := jx * mx;
mxmx := mx * mx;
mx2 := mx * 2;
jxmx2 := jxmx * 2;
jxxmx2 := jxxmx * 2;

ifirst := 1;
itflag := 1;
iglobe := 2;
delt := idelt;
idumpt := IF idumpt_i = 0 THEN 1000 ELSE idumpt_i END IF;
zero := record CplexReal[Repart : 0.0; Impart : 0.0];

ir, ilong, ilath, irmax2, ww, zmean, tw, asq, grav,
kmjx, kmjxx, ksq_1_uncared_for, epsi := Inital(ires, ix, iy, mx, jx, jxx, zmean_1);

ksq := ARRAY[0 : 0] || ksq_1_uncared_for || ARRAY[1 : 0, 0];

AbortFFT, AbortInitFFT, nfax, ifax, trigf, trigb := InitFFT(ix);

coa, w, sia, delta, wocs, WORKiyh := gaussg(ilath);     % size iyh

wix :=  IF ilin = 0                     % Indeed
        THEN  FOR lat_level IN 1, ilath
                RETURNS ARRAY of w[lat_level] / real(ix)
                END FOR
        ELSE  w
        END IF;                         % size iyh; of the North

winv, coainv :=  FOR lat_level IN 1, ilath
                 winv := wix[iy / 2 + 1 - lat_level];
                 coainv := -coa[iy / 2 + 1 - lat_level]
                 RETURNS ARRAY of winv
                         ARRAY of coainv
                 END FOR;
wiy, coaiy := wix || winv, coa || coainv;       % size iy; of North & South

deltaiy, siaiy, wocsiy :=     % size iy; of North & South
        FOR lat_level IN 1, iy
                deltai := ACOSR(coaiy[lat_level]);
                siai := SIN(deltai);
                wocsi := wiy[lat_level] / (siai * siai);
        RETURNS ARRAY of deltai
                ARRAY of siai
                ARRAY of wocsi
        END FOR;
```

```
wocsilath, wilath :=  IF iglobe = 2            % Indeed, highlight the South
                      THEN wocsiy, wiy

                      ELSE  FOR lat_level in 1, ilath
                            wocsiyhalf := 2.0 * wocsiy[lat_level]
                            RETURNS ARRAY of wocsiyhalf
                            END FOR                   || ARRAY_adjust(wocsiy, ilath + 1, iy),

                            FOR lat_level in 1, ilath
                            wiyhalf := 2.0 * wiy[lat_level]
                            RETURNS ARRAY of wiyhalf
                            END FOR                  || ARRAY_adjust(wiy, ilath + 1, iy)
                      END IF;

alp_double :=
  FOR lat_level IN 1, ilath
   alp_LGN := legendre(ir, irmax2, jxxmx, coaiy[lat_level], siaiy[lat_level], deltaiy[lat_level]);
  RETURNS ARRAY of alp_LGN
  END FOR;                    % arraysize [iyh levels, spectral_indices]

alp := SasAlfa(ir, irmax2, jxxmx, ilath, alp_double);

constant := grav / asq;

% Here is the Trouble Spot:
% When these two are put out seperately, if1opt disallows:
var :=  FOR diffindex IN 2, jxmx
        RETURNS VALUE OF SUM CabsSqr(zt_mountain[diffindex])
        END FOR;
h :=    FOR index IN 1, jxmx
        RETURNS ARRAY OF Crmul(constant, zt_mountain[index])
        END FOR;

% The inexplicable solution:
% var, h :=    FOR index IN 1, jxmx
%              RETURNS VALUE OF SUM IF index = 1 THEN 0.0
%                                   ELSE  CabsSqr(zt_mountain[index])
%                                   END IF
%                                  ARRAY OF Crmul(constant, zt_mountain[index])
%              END FOR;

hnew := IF ilin = 0        % Indeed
        THEN h ELSE    ARRAY_fill(1, jx, zero) || ARRAY_adjust(h, jx + 1, jxmx)
        END IF;

p, c_taken, z :=    FOR row IN 1, jxmx
                     p, c, z:= IF row > 256 THEN zero, zero, zero ELSE p_in[row], c_in[row], z_in[row]
                            END IF;
                    RETURNS ARRAY of p
                            ARRAY of c
                            ARRAY of z
                    END FOR;

c :=    IF istart = 0 THEN ARRAY_FILL(1, jxmx, zero)     % Indeed
        ELSEIF ARRAY_SIZE(c_in) = 0 THEN ARRAY_FILL(1, jxmx, zero)
        ELSE c_taken
        END IF;
```

```
znew := IF istart = 0                   % Indeed
           %-------------------------------------Linear Balance Equation
        THEN  For m IN 1, mx
              RETURNS  VALUE  of  CATENATE
                FOR j IN 1, jx
                jm := kmjx[m] + j;
                jmx := kmjxx[m] + j;
                realn := real(m + j - 2);
                realn1 := realn + 1.0;
                zj:     IF (j = 1 & m = 1) THEN zero ELSEIF (j = jx & m = mx)
                        THEN Crmul( - tw / realn / realn * epsi[jmx], p[jm - 1])
                        ELSE Crmul( - tw / realn / realn1,  Cadd(Crmul(realn1 / realn * epsi[jmx], p[jm - 1]),
                                Crmul(realn / realn1 * epsi[jmx + 1], p[jm + 1])))
                        END IF
                RETURNS  ARRAY  of  zj
                END FOR
              END FOR
          %----------------------------
        ELSEIF  array_size(z_in) = 0 THEN ARRAY_FILL(1, jxmx, zero)
        ELSE z END IF;
pm := p;
p1 :=   FOR j IN 1, jxmx
        RETURNS  ARRAY  of  p[j].Repart
        END FOR;
cm := c;
zm := znew;
th_time_step:=1;

IN
1, mx, jx, jxx, ilin, mx2, jxmx, jxxmx, nfax, ilath, imp,
istart, idumpt, ir, irmax2, ires, ix, ixh, iy, delt, ilong, izon, ifirst, th_time_step,
hdiff, hdrag, tw, zmean, vnu, asq, ww, grav, kmjx, kmjxx, ksq, ifax, epsi, wocsilath, alp,
p, c, znew, hnew, p1, pm, cm, zm, trigb, trigf
END LET
END FUNCTION
```

Appendix B

SISAL PROGRAM OF SPECTRAL BAROTROPIC NUMERICAL WEATHER PREDICTION MODEL

This appendix consists of a listing of the parallellised spectral barotropic numerical weather prediction code in SISAL. The original FORTRAN code and the SISAL code directly transliterated from FORTRAN are not listed here as they are too long. They are listed only in the original thesis.

B.1 Parallel SISAL Version

Listed below is the parallel version of the weather model in SISAL, which was implemented through the loop transformation heuristics described in Chapter 3.

```
# Author: Pau Sheong Chang
# Laboratory for Concurrent Computing Systems
# Spectral Barotropic Numerical Weather Prediction Model in SISAL, Version 1.9
# 1989

# makefile for the SISAL codes barotropic model Version 1.9
# Let
if1files =      send.if1 SecondaryIntrFuncs.if1 ComplexFuncs.if1 \
                Inital.if1 InitFFT.if1 GaussianQuadrature.if1 \
                LegendrePolyOf1stKind.if1 SasAlfaSphere.if1 Loop_TimeStep.if1 \
                U_V_Spectral.if1 ComplexConversion.if1 SpecToFreqSphere.if1 \
                MdFFTGrid.if1 MdFFTFreq.if1 VertigSphere.if1 \
                PassGrid.if1 IFACTg_2ETC.if1 IFACTg_3.if1 IFACTg_4.if1 \
                PassFreq.if1 IFACTm_2ETC.if1 IFACTm_3.if1 IFACTm_4.if1 \
                FreqToSpecSphere.if1 \
                Linear.if1 TStep.if1 Energy.if1 AngMom.if1 Specam.if1

.SUFFIXES:      .sis .if1

# Compile from .sis files to .if1 files
.sis.if1:       osc $*.sis -IF1

model1.9:       $(if1files)
                osc -v -inter -J -o model1.9 $(if1files)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

SecondaryIntrFuncs.sis

```
------------------------------------------------------------------------------------------ SecondaryIntrFuncs.sis
DEFINE ASINR, ACOSR, SQRTR, DSIN, DCOS

% ---------Intrinsic Functions
GLOBAL SIN(num : real RETURNS real)
GLOBAL COS(num : real RETURNS real)
GLOBAL ASIN(num : double_real RETURNS double_real)
GLOBAL ACOS(num : double_real RETURNS double_real)
GLOBAL SQRT(num : double_real RETURNS double_real)

% Catering for real operations of Intrinsic functions
FUNCTION ASINR(num : real RETURNS real)
real(ASIN(double_real(num)))
END FUNCTION

FUNCTION ACOSR(num : real RETURNS real)
real(ACOS(double_real(num)))
END FUNCTION

FUNCTION SQRTR(num : real RETURNS real)
real(SQRT(double_real(num)))
END FUNCTION

% Catering for double_real operations of Intrinsic functions
FUNCTION DSIN(num : double_real RETURNS double_real)
double_real(SIN(real(num)))
END FUNCTION

FUNCTION DCOS(num : double_real RETURNS double_real)
double_real(COS(real(num)))
END FUNCTION
```

-- **ComplexFuncs.sis**

```
DEFINE Cadd, Csub, Cmul, Cdiv, Crmul, Crsub, Crdiv, Conjg, Cneg, Csqrt, Cabs, CabsSqr

TYPE CplexReal = RECORD[Repart,Impart:real];
TYPE ArrCplexReal = ARRAY[CplexReal];

% ---------Intrinsic Functions
GLOBAL SIN(num : real RETURNS real)
GLOBAL COS(num : real RETURNS real)
GLOBAL ATAN(num : real RETURNS real)
GLOBAL SQRTR(num : real RETURNS real)

% These subroutines do the arithmatics of complex numbers:

% cnum1 + cnum2
FUNCTION Cadd(cnum1, cnum2 : CplexReal RETURNS CplexReal)
RECORD CplexReal[ Repart : cnum1.Repart + cnum2.Repart; Impart : cnum1.Impart + cnum2.Impart ]
END FUNCTION

% cnum1 - cnum2
FUNCTION Csub(cnum1, cnum2 : CplexReal RETURNS CplexReal)
RECORD CplexReal[ Repart : cnum1.Repart - cnum2.Repart; Impart : cnum1.Impart - cnum2.Impart ]
END FUNCTION

% cnum1 * cnum2
FUNCTION Cmul(cnum1, cnum2 : CplexReal RETURNS CplexReal)
RECORD CplexReal[ Repart : cnum1.Repart * cnum2.Repart - cnum1.Impart * cnum2.Impart;
                  Impart : cnum1.Repart * cnum2.Impart + cnum1.Impart * cnum2.Repart ]
END FUNCTION

% cnum1/cnum2
FUNCTION Cdiv(cnum1, cnum2 : CplexReal RETURNS CplexReal)
LET dnom:= cnum2.Repart * cnum2.Repart + cnum2.Impart * cnum2.Impart;
IN RECORD CplexReal[ Repart : (cnum1.Repart*cnum2.Repart + cnum1.Impart*cnum2.Impart) / dnom;
                     Impart : (cnum1.Impart*cnum2.Repart - cnum1.Repart*cnum2.Impart) / dnom ]
END LET
END FUNCTION

% Real_constant*cnum
FUNCTION Crmul(cons : real; cnum : CplexReal RETURNS CplexReal)
RECORD CplexReal[ Repart : cons * cnum.Repart; Impart : cons * cnum.Impart ]
END FUNCTION

% cnum-Real_constant
FUNCTION Crsub(cnum : CplexReal; cons : real RETURNS CplexReal)
RECORD CplexReal[ Repart : cnum.Repart-cons; Impart : cnum.Impart ]
END FUNCTION

% cnum/Real_constant
FUNCTION Crdiv(cnum : CplexReal; cons : real RETURNS CplexReal)
RECORD CplexReal[ Repart : cnum.Repart / cons; Impart : cnum.Impart / cons ]
END FUNCTION

% conjugate(cnum)=Repart-Impart
FUNCTION Conjg(cnum : CplexReal RETURNS CplexReal)
RECORD CplexReal[ Repart : cnum.Repart; Impart : -cnum.Impart ]
END FUNCTION

% Cneg(cnum)
FUNCTION Cneg(cnum : CplexReal RETURNS CplexReal)
RECORD CplexReal[ Repart : -cnum.Repart; Impart : -cnum.Impart ]
END FUNCTION
```

```
% Csqrt(cnum)
% This routine receives a complex number, then stripes it into Repart &
% Impart. It then calculates the magnitude and argument(angle) of the
% square root of this number and then reconstruct them into the
% corresponding Real and Imaginary parts. The result is then structured
% as a RECORD of these two parts before it is returned.
FUNCTION Csqrt(cnum:CplexReal RETURNS CplexReal)
LET   RR := cnum.Repart;
      II := cnum.Impart;
      mag := SQRTR( SQRTR(RR * RR + II * II) );
      angle := ATAN(II / RR) / 2.0;
      Re, Im := mag * COS(angle), mag * SIN(angle);
IN    RECORD CplexReal[Repart : Re; Impart : Im]
END LET
END FUNCTION

% Cabs(cnum) refers to the MAGNITUDE of the complex number.
FUNCTION Cabs(cnum : CplexReal RETURNS real)
SQRTR(cnum.Repart * cnum.Repart + cnum.Impart * cnum.Impart)
END FUNCTION

% CabsSqr(cnum) refers to the MAGNITUDE Square of the complex number.
FUNCTION CabsSqr(cnum : CplexReal RETURNS real)
cnum.Repart * cnum.Repart + cnum.Impart * cnum.Impart
END FUNCTION
```

-- Inital.sis

```
DEFINE Inital

TYPE ArrIntl = Array[integer];
TYPE ArrReal1 = Array[real];

GLOBAL SQRTR(num : real RETURNS real)

FUNCTION Inital(ir, ilong, ilat, mx, jx, jxx : integer; zmean1 : real
        RETURNS integer, integer, integer, integer, real, real, real, real, real, arrint1, arrint1, arrint1, arrreal1)
LET
ww := 7.292E-5;
tw := ww * 2.0;
irmax:= ir;
ilath, irmax1, irmax2 := ilat / 2, irmax + 1, irmax + 2;
asq, grav :=  6371.22E3 * 6371.22E3,  9.80616;
zmean:= zmean1 * grav / asq;
kmjx, kmjxx :=    FOR m IN 1, mx
                  RETURNS ARRAY of (m - 1) * jx
                          ARRAY of (m - 1) * jxx
                  END FOR;
ksq :=    FOR j IN 1, ir * 2
          RETURNS ARRAY of j * (j + 1)
          END FOR;

epsilon_a :=      FOR mp IN 1, mx
                  RETURNS VALUE of CATENATE
                     FOR j IN 1, jxx    % epsilon_size is 1-272
                     l := j + mp - 2;
                     m := mp - 1;
                     t := real((l + m) * (l - m));
                     b := real(4 * l * l - 1);
                      RETURNS ARRAY of SQRTR(t / b)
                      END FOR
                  END FOR;

epsilon := epsilon_a[1 : 0.0];

IN ir, ilong, ilath, irmax2, ww, zmean, tw, asq, grav, kmjx, kmjxx, ksq, epsilon

END LET
END FUNCTION
```

--- **InitFFT.sis**

```
DEFINE InitFFT

TYPE ArrInt1 = ARRAY[integer];
TYPE ArrReal1 = ARRAY[real]

GLOBAL SIN(num : real RETURNS real)
GLOBAL COS(num : real RETURNS real)

% Subroutine FACTR4 factors an integer n into its prime factors.
% For example, n=1960 is factorised into six prime factors(thus
% nfact=6), which are(thus ifact(ifT)=) 2, 2, 5, 7 and 7.
% The version below is modified to give the factor 4, so as to
% be used by the new FFT. Thus, the subsequent factors are 4, 2,
% 5, 7 and 7, and nfact=5.

%----------------factr4/facStep/facRecur
FUNCTION facRecur(nparti, idiv, ifTi : integer; ifacti : ArrInt1
                    RETURNS integer, integer, integer, ArrInt1)
FOR INITIAL
npart := nparti;
iquot := npart / idiv;
ifT := ifTi;
ifact1 := ifacti;

WHILE npart - idiv * iquot = 0 REPEAT
npart := old iquot;
iquot := npart / idiv;
ifT := old ifT + 1;
ifact1 := old ifact1[ifT : idiv]

RETURNS         VALUE of npart
                VALUE of iquot
                VALUE of ifT
                VALUE of ifact1
END FOR
END function  % facRecur

%-----------------factr4/facStep
FUNCTION Loop_id(n : integer  RETURNS integer, integer, ArrInt1)
FOR INITIAL   % loop_id
id := 1;
ifT := 0;
npart := n;
ifact := ARRAY_FILL(1, 20, 0)   % NOTE: wild guess

WHILE id <= n REPEAT
idiv := IF old id - 1 <= 0  THEN 2 ELSE old id   END IF;

npart, iquot, ifT, ifact :=  facRecur(old npart, idiv, old ifT, old ifact);

id :=     IF iquot - idiv <= 0 THEN n + 1  % just to make it greater than n
          ELSE old id + 2    END IF;

RETURNS         VALUE of npart
                VALUE of ifT
                VALUE of ifact
END FOR
END FUNCTION          % Loop_id

%-----------------------
function FACTR4(n : integer RETURNS integer, arrint1)
LET
npart, ifT, ifact1 :=  Loop_id(n);

iff := IF npart - 1 > 0  THEN ifT + 1 ELSE ifT END IF;
ifact2 := IF npart - 1 > 0  THEN ifact1[iff : npart] ELSE ifact1 END IF;
nfactT := iff;
```

```
n2 :=    FOR INITIAL
         n2 := 0;
         i := 1;
         % n2 includes case i=nfactT
         WHILE i <= nfactT REPEAT
         i := old i + 1;
         n2 := IF ifact2[old i] = 2 THEN old n2 + 1 ELSE old n2 END IF
         RETURNS VALUE of n2
         END FOR;        % NOTE: very ineffecient!

n4 := n2 / 2;
ifact3 := ARRAY_FILL(1, n4, 4)
         ||       for i in n4 + 1, nfactT - n4
                  RETURNS ARRAY of ifact2[n4 + i]
                  END for
         || ARRAY_FILL(nfactT - n4 + 1, nfactT, 0);

nfact := nfactT - n4;

IN nfact, ifact3

END LET
END FUNCTION          % factr4
%--------------------------------------------------------------

% Subroutine InitFFT does the initialisations necessary so that the
% FFT's can be used. It factorises the number of longitudinal points.
% TRIGF are for forward transforms while TRIGB are for reverse.

FUNCTION InitFFT(n : integer RETURNS boolean, boolean, integer, arrint1, ArrReal1, ArrReal1)
LET
Abortinitfft :=    IF (MOD(n, 2) ~= 0 | n > 200) THEN true ELSE false END IF;
AbortFFT := IF n > 96 THEN true ELSE false END IF;

pi := 3.14159265;

nfax, ifax := FACTR4(n);

trigf, trigb :=    IF Abortinitfft THEN array ArrReal1 [], array ArrReal1 []
                   ELSE   FOR Lp IN 1, n
                          k := (Lp + 1) / 2;
                          Cargument := - 2.0 * pi * real(k - 1)/real(n);
                          COStheta := COS(Cargument);      % Repart
                          SINtheta := SIN(Cargument);      % Impart
                          RETURNS ARRAY of IF MOD(Lp, 2) = 0 THEN SINtheta ELSE COStheta
                                           END IF
                                  ARRAY of IF MOD(Lp, 2) = 0 THEN - SINtheta ELSE COStheta
                                           END IF
                          END FOR
                   END IF
IN AbortFFT, Abortinitfft, nfax, ifax, trigf, trigb
END LET
END function
```

```
---------------------------------------------------------- GaussianQuadrature.sis (GaussGrid)
DEFINE GaussianQuadrature

TYPE ArrReal1 = Array[real];
TYPE ArrDreal1 = Array[double_real]

GLOBAL ACOS(num : double_real RETURNS double_real)
GLOBAL SQRT(num : double_real RETURNS double_real)

GLOBAL SIN(num : double_real RETURNS double_real)
GLOBAL COS(num : double_real RETURNS double_real)

FUNCTION ORDLEG(ir : integer; coa : double_real RETURNS double_real)
LET     irpp, irppm := ir+1, ir;
        delta := ACOS(coa);
        sqr2 := SQRT(2.0d0);
        theta := delta;
        c1 := sqr2 *        FOR n IN 1, irppm
                            fn := n;
                            fn2 := fn * 2;
                            fn2sq :=  double_real(fn2 * fn2);
                            RETURNS VALUE of product SQRT(1.0d0 - 1.0d0 / fn2sq)
                            END FOR;

        s1 :=    FOR INITIAL
                 n := irppm;
                 fn := double_real(irppm);
                 fn2 := fn * 2.0d0;
                 ang := fn * theta;
                 s1T := 0.0d0;
                 c4 := 1.0d0;
                 a := -1.0d0;
                 b := 0.0d0;
                 n1 := n + 1;
                 kk := 1;

                 WHILE kk <= n1 REPEAT
                  kk := old kk + 2;
                  k := old kk - 1;
                 c4T := IF k=n THEN 0.5d0 * old c4 ELSE old c4 END IF;
                 s1T := old s1T + c4T * COS(old ang);
                 a := old a + 2.0d0;
                 b := old b + 1.0d0;
                 fk := double_real(k);
                 ang := theta * (fn - fk - 2.0d0);
                 c4 := a * (fn - b + 1.0d0) / (b * (fn2 - a)) * c4T;

                 RETURNS VALUE OF s1T
                 END FOR;
        sx := s1 * c1;
IN sx
END LET
END FUNCTION
%-------------------------------------------------------------

% Subroutine GaussianQuadrature calculates the cosine of the
% Colatitudes(f//fs) and the weights(wt//wts) for the
% Gaussian Quadrature with NZERO Gaussian points between the pole
% and the Equator. Calculations are done in double
% precision, but the results are returned in single
% precision.
```

```
%----------------gaussg/cycle
FUNCTION CYCLE(ir, irm, irp : integer;  ft, a, b, xlim : double_real RETURNS double_real)
LET     g := ORDLEG(ir, ft);
        gm := ORDLEG(irm, ft);
        gp := ORDLEG(irp, ft);
        gt := (ft * ft - 1.0d0)/(a * gp - b * gm);
        ftemp := ft - g * gt;
        gtemp := ft - ftemp;
        ftnew := ftemp;
IN      IF ABS(gtemp) - xlim > 0.0d0 THEN CYCLE(ir, irm, irp, ftnew, a, b, xlim)
        ELSE ftnew END IF
END LET
END FUNCTION


%------------------GaussianQuadrature
FUNCTION GaussianQuadrature(nzero : integer
        RETURNS ArrReal1, ArrReal1, ArrReal1, ArrReal1, ArrReal1, ArrDreal1)
LET     xlim := 1.0d-12;
        ir := nzero * 2;
        fi := double_real(ir);
        fi1 := fi + 1.0d0;
        pi := 3.141592653589793d0;
        piov2 := pi * 0.5d0;
        fn := piov2/double_real(nzero);
        wt :=   FOR lat IN 1,nzero
                RETURNS ARRAY of double_real(lat) - 0.5d0
                END FOR;
        f :=    FOR lat IN 1,nzero
                RETURNS ARRAY of SIN( wt[lat] * fn + piov2 )
                END FOR;
        dn := fi/SQRT(4.0d0 * fi * fi - 1.0d0);
        dn1 := fi1/SQRT(4.0d0 * fi1 * fi1 - 1.0d0);
        a := dn1 * fi;
        b := dn * fi1;
        irp := ir + 1;
        irm := ir - 1;
        fnew := FOR lat IN 1, nzero
                RETURNS ARRAY of CYCLE(ir, irm, irp, f[lat], a, b, xlim)
                END FOR;
        wtnew, radnew, coangnew, sianew :=
                FOR lat IN 1, nzero
                a1 := 2.0d0 * (1.0d0 - fnew[lat] * fnew[lat]);
                bo := ORDLEG(irm, fnew[lat]);
                b1 := bo * bo * fi * fi;
                wtt := a1 * (fi - 0.5d0) / b1;
                radt := ACOS(fnew[lat]);
                coangt := radt * 180.0d0 / pi;
                siat := SIN(radt);
                RETURNS ARRAY of wtt
                        ARRAY of radt
                        ARRAY of coangt
                        ARRAY of siat
                END FOR;

        WORKiyh := fnew || wtnew || sianew || radnew || coangnew;

        fs, wts, sias, rads, coangs :=
                FOR lat IN 1, nzero
                RETURNS ARRAY of real(fnew[lat])
                        ARRAY of real(wtnew[lat])
                        ARRAY of real(sianew[lat])
                        ARRAY of real(radnew[lat])
                        ARRAY of real(coangnew[lat])
                END FOR;

IN fs, wts, sias, rads, coangs, WORKiyh
END LET
END FUNCTION
```

-- **LegendrePolyOf1stKind.sis (Legendre)**

```
DEFINE LegendrePolyOf1stKind

TYPE ArrDreal1 = Array[Double_real]

GLOBAL SIN(num : double_real RETURNS double_real)
GLOBAL COS(num : double_real RETURNS double_real)
GLOBAL SQRT(num : double_real RETURNS double_real)

% The COMMON statement(WORK) CATENATEs into ARRAY P, the following
% ARRAYs in the order of F(20), WT(20), SIA(20), RAD(20) and
% COANG(20), followed by 172 new empty cells. This has to be taken
% into consideration.
% Hence from start, an ARRAY of "family" P has to be created to be
% as such.
% NOTE:
% However, this should be done in the main program beforehand,
% before the main program calls LegendrePolyOf1stKind ILATH times, because once
% the program enters subroutine LegendrePolyOf1stKind, it includes WORK to update
% the containt of WORK. This should be investigated again because
% of the presence of SIA in the variable-passing and WORK. Is SIA
% passed the updated SIA in WORK?
% p should be returned as double_real for cocsistency in MAIN, in
% contrast to the Fortran codes

FUNCTION LegendrePolyOf1stKind(ir, irmax2, jxxmx : integer; coas, sias, deltas : real
              RETURNS ArrDreal1)
LET
p := LET
   coa := double_real(coas);
   sia := double_real(sias);
   delta := double_real(deltas);
   irpp := ir + 2;
   theta := delta;
   sqr2 := SQRT(2.0d0);
   pp := FOR INITIAL
         n := 1;
         c1 := sqr2;
         pLoop1 := ARRAY ArrDreal1 [1: 1.0d0 / sqr2]      || FOR jm IN 2, jxxmx
                                                             RETURNS ARRAY of 0.0d0
                                                             END FOR;

         WHILE n <= irpp REPEAT
         n := old n + 1;
         fn := double_real(old n);
         fn2 := 2.0d0 * fn;
         fn2sq := fn2 * fn2;
         c1 := old c1 * SQRT(1.0d0 - 1.0d0 / fn2sq);
         c3 := c1 / SQRT(fn * (fn + 1.0d0));

         s1, s2 := FOR INITIAL
                   kk := 1;
                   ang := fn * theta;
                   n1 := old n + 1;
                   ss1, ss2 := 0.0d0, 0.0d0;
                   c4, c5 := 1.0d0, fn;
                   a, b := - 1.0d0, 0.0d0;
                   WHILE kk <= n1 REPEAT
                   kk := old kk + 2;
                   k := old kk - 1;
                   ss2 := old ss2 + old c5 * SIN(old ang) * old c4;
                   c4t := if k = old n then 0.5d0 * old c4 else old c4 END if;
                   ss1 := old ss1 + c4t * COS(old ang);
                   a := old a + 2.0d0;
                   b := old b + 1.0d0;
                   fk := double_real(k);
                   ang := theta * (fn - fk - 2.0d0);
                   c4 := (a * (fn - b + 1.0d0) / (b * (fn2 - a))) * c4t;
                   c5 := old c5 - 2.0d0
                   RETURNS      VALUE of ss1
```

```
                              VALUE of ss2   % to s1 and s2
             END FOR;
      pLoop1 := IF old n - irpp < 0  THEN old pLoop1[old n + 1 : s1 * c1; old n + irmax2 : s2 * c3]
                ELSEIF old n - irpp = 0  THEN old pLoop1[old n + irmax2 : s2 * c3]
                ELSE old pLoop1
                END IF
      RETURNS VALUE of pLoop1     % to pp
      END FOR;

  p2 := IF ir = 2 THEN pp
      ELSE   FOR INITIAL
             m := 2;
             ppp := pp
             WHILE m <= ir REPEAT
             m := old m + 1;
             fm := double_real(old m);
             fm1, fm2, fm3 := fm - 1.0d0, fm - 2.0d0, fm - 3.0d0;
             mm1 := old m - 1;
             m1 := old m + 1;
             c6 := SQRT((2.0d0 * fm + 1.0d0) / (2.0d0 * fm));
             p5 := old ppp[irmax2 * old m + 1 : c6 * sia *  old ppp[irmax2 * mm1 + 1]];
             mpir := old m + ir + 1;
             mt := old m;

             ppp :=   FOR INITIAL
                      l := m1;
                      p4 := p5;
                      WHILE l <= mpir REPEAT
                      l := old l + 1;
                      fn := double_real(old l);
                      c7 := (fn * 2.0d0 + 1.0d0) / (fn * 2.0d0 - 1.0d0);
                      c8 := (fm1 + fn) / ((fm + fn) * (fm2 + fn));
                      c := SQRT((fn * 2.0d0 + 1.0d0) /   (fn * 2.0d0 - 3.0d0) * c8 * (fm3 + fn));
                      d := SQRT(c7 * c8 * (fn - fm1));
                      e := SQRT(c7 * (fn - fm) / (fn + fm));
                      lm := irmax2 * mt + old l - mt + 1;
                      lmm2 := irmax2 * (mt - 2) + old l - mt + 3;
                      lm1mm2 := lmm2 - 1;
                      lm2mm2  :=  lm1mm2 - 1;
                      lm1m := lm - 1;

                      p4 :=    IF old l - mpir < 0
                               THEN old p4[lm:c * old p4[lm2mm2]  - d * old p4[lm1mm2] * coa
                                       + e * old p4[lm1m] * coa]
                               ELSEIF old l - mpir > 0  THEN old p4
                               ELSE LET a := SQRT((fn * fn - 0.25d0) /  (fn * fn - fm * fm));
                                            b := SQRT((2.0d0 * fn + 1.0d0) *  (fn - fm - 1.0d0) * (fn + fm1)
                                                  / ((2.0d0 * fn - 3.0d0) * (fn - fm) * (fn + fm)));
                                            lm2m := lm1m - 1;
                                        IN old p4[lm : 2.0d0 * a * coa * old p4[lm1m] - b * old p4[lm2m]]
                                        END LET
                               END IF

                      RETURNS VALUE of p4          % to p6
                      END FOR;

             RETURNS VALUE of ppp                        % to p2
             END FOR
      END IF

   IN p2
   END LET;            % RETURNS p2 to p

IN p
END LET
END FUNCTION
```

SasAlfaSphere.sis (SymAsymSphere)

```
------------------------------------------------------------------------ SasAlfaSphere.sis (SymAsymSphere)
DEFINE SasAlfaSphere

TYPE arrDreal1 = ARRAY [double_real];
TYPE arrDreal2 = ARRAY [arrDreal1];
TYPE arrDreal3 = ARRAY [arrDreal2];
TYPE arrreal1 = ARRAY [real];
TYPE arrreal2 = ARRAY [arrreal1];
TYPE arrreal3 = ARRAY [arrreal2]

% The parallelism in .ASS. is improved here by rewriting the original
% two nested .while. loops in .forall. loops.
% Here .lpfin. in reversely defined to suit the forall environment.

FUNCTION SasAlfaSphere(ir, irmax2, jxxmx, ilath : integer; alp : ArrDReal2 RETURNS ArrReal3)
LET
%lpfin :=        IF MOD(ir, 2) = 0 THEN ir + 1
%                ELSE ir + 2
%                END IF;

alfa :=   FOR hemi IN 1, 2 CROSS latlev IN 1,ilath
          RETURNS ARRAY of
                IF hemi = 1        % North
                THEN  FOR specindex IN 1, jxxmx
                      RETURNS ARRAY of real(alp[latlev, specindex])
                      END FOR
                ELSE  FOR mp IN 1, ir + 1        % South
                      RETURNS VALUE of CATENATE
                            % FOR lp IN 1, lpfin
                            FOR lp IN 1, irmax2
                            ilm := (mp - 1) * irmax2 + lp;
                            RETURNS ARRAY of
                                  IF lp = 1 | MOD(lp, 2) ~= 0
                                  THEN real(alp[latlev, ilm])
                                  ELSE real(-alp[latlev, ilm])
                                  END IF
                            END FOR
                      END FOR
                END IF
          END FOR
IN alfa
END LET
END FUNCTION
```

U_V_Spectral.sis (UVspectral)

```
------------------------------------------------------------------ U_V_Spectral.sis (UVspectral)
DEFINE U_V_Spectral

TYPE ArrReal1 = Array[real];
TYPE CplexReal = Record[Repart,Impart:real];
TYPE ArrCplexReal = Array[CplexReal];
GLOBAL Cadd(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Csub(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Cmul(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Crmul(cons : real; cnum : CplexReal RETURNS CplexReal)
```

```
FUNCTION U_V_Spectral( mx,jx,jxx: integer; epsi: ArrReal1;  p, c: ArrCplexReal
                       RETURNS  ArrCplexReal, ArrCplexReal)
LET
iz := RECORD CplexReal[Repart : 0.0; Impart : 1.0];
zero := RECORD CplexReal[Repart : 0.0; Impart : 0.0];
u, v :=    FOR m IN 1, mx                          % size of jxxmx=272
        realm := real(m - 1);
        u1, v1 := FOR j IN 1, jxx
                nreal := j + m - 2;
                realn := real(nreal);
                realn1 := realn + 1.0;
                jm := (m - 1) * jx + j;
                jmx := (m - 1) * jxx + j;
                coeffd, coeffc, coeffu, pd, pc, pu, cd, cc, cu :=
                        IF j = 1  THEN zero, IF nreal = 0 THEN zero
                                             ELSE zero replace [Impart : realm / realn / realn1]
                                             END IF,
                                        zero replace [Repart:epsi[jmx + 1] / realn1],
                                        zero, p[jm], p[jm + 1], zero, c[jm], c[jm + 1]
                        ELSEIF j = jx     THEN   zero replace [Repart : epsi[jmx] / realn],
                                                 zero replace [Impart : realm / realn / realn1],
                                                 zero replace [Repart : epsi[jmx + 1] / realn1],
                                                 p[jm - 1], p[jm], zero, c[jm - 1], c[jm], zero
                        ELSEIF j=jxx THEN zero replace [Repart : epsi[jmx] / realn],
                                              zero, zero, p[jm - 1], zero, zero, c[jm-1], zero, zero
                        ELSE    zero replace [Repart : epsi[jmx] / realn],
                                zero replace [Impart : realm / realn / realn1],
                                zero replace [Repart : epsi[jmx + 1] / realn1],
                                p[jm - 1], p[jm], p[jm + 1], c[jm - 1], c[jm], c[jm + 1]
                        END IF;
                ujm :=  Cadd(Crmul( - 1.0, Cmul(coeffd, pd)), Csub(Cmul(coeffu, pu), Cmul(coeffc, cc)));
                vjm :=  Csub(Csub(Cmul(coeffd, cd), Cmul(coeffu, cu)), Cmul(coeffc, pc));
                RETURNS ARRAY of ujm
                        ARRAY of vjm
                END FOR
        RETURNS VALUE of CATENATE u1
                VALUE of CATENATE v1
        END FOR
In u, v
END LET
END FUNCTION
```

---------------- ComplexConversion.sis (LinearConversion & ComplexConversion)

```
DEFINE Complexing_ct_e_pt_ztSp, Decomplexing_p_zdiff_u_v
TYPE ArrInt1 = ARRAY [integer];
TYPE ArrReal1 = ARRAY [real];
TYPE CplexReal = RECORD [Repart, Impart : real];
TYPE ArrCplexReal = ARRAY[CplexReal]
FUNCTION Complexing_ct_e_pt_ztSp  (jx, mx : integer; kmjx : ArrInt1; ct, e, pt, zt : ArrReal1
        RETURNS ArrCplexReal, ArrCplexReal, ArrCplexReal, ArrCplexReal)
LET ctC, eC, ptC, ztC :=
        FOR m in 1, mx
        ctC, eC, ptC, ztC :=
                FOR j in 1, jx
                complex_index := kmjx[m] + j;
                index := complex_index * 2
                RETURNS ARRAY of RECORD CplexReal[Repart : ct[index - 1]; Impart : ct[index]]
                        ARRAY of RECORD CplexReal[Repart : e[index - 1]; Impart : e[index]]
                        ARRAY of RECORD CplexReal[Repart : pt[index - 1]; Impart : pt[index]]
                        ARRAY of RECORD CplexReal[Repart : zt[index - 1]; Impart : zt[index]]
                END FOR
        RETURNS VALUE of CATENATE ctC
                VALUE of CATENATE eC
                VALUE of CATENATE ptC
                VALUE of CATENATE ztC
        END FOR
IN ctC, eC, ptC, ztC
END LET
END FUNCTION
```

```
FUNCTION Decomplexing_p_zdiff_u_v (jx, mx, jxx : integer; p, zdiff, u, v : ArrCplexReal
         RETURNS ArrReal1, ArrReal1, ArrReal1, ArrReal1)
LET  pri, zri, uri, vri :=
        FOR m IN 1, mx * 2
        pri, zri :=  FOR j IN 1, jx
                     real_index := jx * (m - 1) + j;
                     complex_index := (real_index + 1) / 2;
                     pri, zri := IF MOD(real_index, 2) = 0
                             THEN p[complex_index].Impart, zdiff[complex_index].Impart
                             ELSE p[complex_index].Repart, zdiff[complex_index].Repart
                             END IF
                     RETURNS ARRAY of pri
                             ARRAY of zri
                     END FOR;
        uri, vri :=  FOR j in 1, jxx
                     real_index := jxx * (m -1) + j;
                     complex_index := (real_index + 1) / 2;
                     uri, vri := IF MOD(real_index, 2) = 0
                              THEN u[complex_index].Impart, v[complex_index].Impart
                              ELSE u[complex_index].Repart, v[complex_index].Repart
                              END IF
                     RETURNS ARRAY of uri
                             ARRAY of vri
                     END FOR
        RETURNS VALUE of CATENATE pri
                VALUE of CATENATE zri
                VALUE of CATENATE uri
                VALUE of CATENATE vri
        END FOR

IN pri, zri, uri, vri
END LET
END FUNCTION
```

--- **SpecToFreqSphere.sis**

```
DEFINE SpecToFreqSphere
TYPE arrint1=array[integer];
TYPE arrreal1=array[real];
TYPE arrreal2=array[arrreal1];
TYPE arrreal3=array[arrreal2]

FUNCTION SpecToFreqSphere (jx, mx, jxx, ilath, ixh : integer; kmjx, kmjxx : ArrInt1;
        alp : ArrReal3; pri, zri, uri, vri : ArrReal1 RETURNS ArrReal3, ArrReal3, ArrReal3, ArrReal3)
LET
% The rest_0 below is to set the right size, ie (ix + 1) * 2, for MDFFTG
% rest_0 := ARRAY_FILL(1, (ixh * 2 + 1 - mx) * 2, 0.0);
% But let MdFFTG worries for itself
% However the actual computation of MDFFTG can live even with size of n
% for pg, zg, ug and vg. Therefore
% rest_0 := ARRAY_FILL(1, (ixh - mx) * 2, 0.0);
% This is a trial:
% rest_0 := array_fill(1, (ixh-mx)*2, 0.0);
pg, zg, ug, vg :=
        FOR hemi IN 1, 2 CROSS latlev IN 1, ilath
        pg, zg, ug, vg:=
                FOR mrmi IN 1, mx * 2
                m := (mrmi + 1) / 2;
                pg, zg :=FOR j IN 1, jx
                        jm := kmjx[m] + j;
                        jmx := kmjxx[m] + j;
                        jmrjmi := jm * 2 - mod(mrmi, 2);
                        pgj, zgj := IF ~(m = 1 & j = 1) THEN alp[hemi, latlev, jmx] * pri[jmrjmi],
                                                   alp[hemi, latlev, jmx] * zri[jmrjmi]
                                ELSE 0.0, 0.0  END IF;
                        RETURNS        VALUE of SUM pgj
                                       VALUE of SUM zgj
                        END FOR;
```

```
            ug, vg :=FOR j IN 1, jxx
                     jmx := kmjxx[m] + j;
                     jmrjmi := jmx * 2 - mod(mrmi, 2)
                     RETURNS VALUE of SUM alp[hemi, latlev, jmx] * uri[jmrjmi]
                                VALUE of SUM alp[hemi, latlev, jmx] * vri[jmrjmi]
                     END FOR;

            RETURNS ARRAY of pg
                         ARRAY of zg
                         ARRAY of ug
                         ARRAY of vg
            END FOR;

      RETURNS ARRAY of pg %|| rest_0
                  ARRAY of zg %|| rest_0    % Without this catenation,
                  ARRAY of ug %|| rest_0        % their sizes will be mx * 2
                  ARRAY of vg %|| rest_0
      END FOR;

IN pg, zg, ug, vg
END LET
END FUNCTION
```

--- **IFACTg_2ETC.sis**

```
DEFINE IFACTg_2ETC
TYPE ArrReal1 = ARRAY [real]

FUNCTION IFACTg_2ETC(m, la, iink, jink, jump, inc1, inc2 : integer;  a, ci, trigs : ArrReal1
                     RETURNS ArrReal1)
FOR INITIAL
k := 0;
ia := 1;
ja := 1;
c := ci;

WHILE k <= m / 2 REPEAT
k := old k + la;
ia, ja, c :=
 IF old k = 0
 THEN LET ia1, ja1, c1 :=

        FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        jb := old ja1 + jink;
        c1 := old c1[old ja1 : a[old ia1] + a[ib]; jb : a[old ia1] - a[ib]];

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1, ja1 + jump, c1
        END LET

 ELSEIF 2 * old k < m
 THEN LET ia1, ja1, c1 :=
```

```
        FOR INITIAL
        kb := old k * 2;
        lower := iink + 2;
        lupper := iink + jink * 2 - 2;
        l := lower;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= lupper REPEAT
        l := old l + 4;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        jb := old ja1 + jink;
        ibxx := old l - old ia1;
        cjb := a[old ia1] - a[ibxx];
        djb := a[old ia1 + 1] + a[ibxx + 1];
        tempr := cjb * trigs[kb + 1] - djb * trigs[kb + 2];
        tempi := cjb * trigs[kb + 2] + djb * trigs[kb + 1];
        c1 := old c1[old ja1 : a[old ia1] + a[ibxx]; jb : tempr;
                old ja1 + 1 : a[old ia1 + 1] - a[ibxx + 1]; jb + 1 : tempi];

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1, ja1 + jump, c1
        END LET

ELSE    FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        jb := old ja1 + jink;
        c1 := old c1[old ja1 : 2.0 * a[old ia1]; jb : -2.0 * a[old ia1 + 1]];

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

 END IF;

RETURNS VALUE of c
END FOR
END FUNCTION
```

IFACTg_3.sis

```
DEFINE IFACTg_3

TYPE ArrReal1 = Array[real];

FUNCTION IFACTg_3(m, la, iink, jink, jump, inc1, inc2 : integer;  a, ci, trigs : arrReal1
                        RETURNS arrReal1)
FOR INITIAL
sin60 := 0.866025403784437;
k := 0;
ia := 1;
ja := 1;
c := ci;

WHILE k <= m / 2 REPEAT
k := old k + la;
ia, ja, c :=
 IF old k = 0
 THEN LET ia1, ja1, c1 :=

        FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        jb := old ja1 + jink;
        jc := jb + jink;
        a1 := a[old ia1] - a[ib];
        b1 := 2.0 * sin60 * a[ib + 1];
        c1 := old c1[old ja1:a[old ia1] + 2.0 * a[ib]; jb:a1 - b1; jc:a1 + b1]

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1, ja1 + jump, c1
        END LET

 ELSEIF 2 * old k < m
 THEN  LET ia1, ja1, c1 :=

        FOR INITIAL
        kb := old k * 2;
        kc := kb * 2;
        lower := iink + 2;
        lupper := iink + jink * 2 - 2;
        l := lower;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= lupper REPEAT
        l := old l + 4;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        icxx := old l - old ia1;
        jb := old ja1 + jink;
        jc := jb + jink;
        a1 := a[ib] + a[icxx];
        b1 := a[ib + 1] - a[icxx + 1];
```

```
        a2 := a[old ia1] - 0.5 * a1;
        b2 := a[old ia1 + 1] - 0.5 * b1;
        a3 := sin60 * (a[ib] - a[icxx]);
        b3 := sin60 * (a[ib + 1] + a[icxx + 1]);
        cjb := a2 - b3;
        djb := b2 + a3;
        cjc := a2 + b3;
        djc := b2 - a3;
        tempr1 := cjb * trigs[kb + 1] - djb * trigs[kb + 2];
        tempi1 := cjb * trigs[kb + 2] + djb * trigs[kb + 1];
        tempr2 := cjc * trigs[kc + 1] - djc * trigs[kc + 2];
        tempi2 := cjc * trigs[kc + 2] + djc * trigs[kc + 1];
        c1  := old c1[old ja1 : a[old ia1] + a1; jb : tempr1; jc : tempr2; old ja1 + 1 : a[old ia1 + 1] + b1;
                    jb + 1 : tempi1; jc + 1 : tempi2];

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1,ja1 + jump,c1
        END LET

  ELSE FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        jb := old ja1 + jink;
        jc := jb + jink;
        a1 := 0.5 * a[old ia1] - a[ib];
        b1 := sin60 * a[old ia1 + 1];
        c1 := old c1[old ja1 : a[old ia1] + a[ib]; jb : a1 - b1; jc : - a1 - b1]

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR
  END IF;

RETURNS VALUE of c
END FOR
END FUNCTION


----------------------------------------------------------------------------IFACTg_4.sis

DEFINE IFACTg_4

TYPE ArrReal1 = Array[real];

FUNCTION IFACTg_4(m, la, iink, jink, jump, inc1, inc2 : integer;
                  a, ci, trigs : ArrReal1
                  RETURNS ArrReal1)
FOR INITIAL
sin45  := 0.7071067812;
k := 0;
ia := 1;
ja := 1;
c := ci;
```

```
WHILE k <= m / 2 REPEAT
k := old k + la;
ia, ja, c :=
 IF old k = 0
 THEN LET ia1, ja1, c1 :=

        FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        ic := ib + iink;
        jb := old ja1 + jink;
        jc := jb + jink;
        jd := jc + jink;
        a1 := a[old ia1] + a[ic];
        a2 := 2.0 * a[ib];
        a3 := a[old ia1] - a[ic];
        a4 := 2.0 * a[ib + 1];
        c1 := old c1[old ja1:a1 + a2; jb:a3 - a4; jc:a1 - a2; jd:a3 + a4];

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1, ja1 + jump, c1
        END LET

 ELSEIF 2 * old k < m
 THEN LET ia1, ja1, c1 :=

        FOR INITIAL
        lower := iink + 2;
        lupper := iink + jink * 2 - 2;
        l := lower;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= lupper REPEAT
        l := old l + 4;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        idxx := old l - old ia1;
        icxx := idxx + iink;
        jb := old ja1 + jink;
        jc := jb + jink;
        jd := jc + jink;
        kb := old k * 2;
        kc := kb * 2;
        kd := kc + kb;
        a0 := a[old ia1] + a[icxx];
        a1 := a[ib] + a[idxx];
        a2 := a[old ia1] - a[icxx];
        a3 := a[ib] - a[idxx];
        b0 := a[old ia1 + 1] - a[icxx + 1];
        b1 := a[ib + 1] - a[idxx + 1];
        b2 := a[old ia1 + 1] + a[icxx + 1];
        b3 := a[ib + 1] + a[idxx + 1];
        cjb := a2 - b3;
        djb := b2 + a3;
```

```
        cjc := a0 - a1;
        djc := b0 - b1;
        cjd := a2 + b3;
        djd := b2 - a3;
        tempr1 := cjb * trigs[kb + 1] - djb * trigs[kb + 2];
        tempi1 := cjb * trigs[kb + 2] + djb * trigs[kb + 1];
        tempr2 := cjc * trigs[kc + 1] - djc * trigs[kc + 2];
        tempi2 := cjc * trigs[kc + 2] + djc * trigs[kc + 1];
        tempr3 := cjd * trigs[kd + 1] - djd * trigs[kd + 2];
        tempi3 := cjd * trigs[kd + 2] + djd * trigs[kd + 1];
        c1 := old c1[old ja1 : a0 + a1; jb : tempr1; jc : tempr2; jd : tempr3;
                old ja1 + 1 : b0 + b1; jb + 1 : tempi1; jc + 1 : tempi2;  jd + 1 : tempi3];

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1, ja1 + jump, c1
        END LET

  ELSE FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        jb := old ja1 + jink;
        jc := jb + jink;
        jd := jc + jink;
        a1 := sin45 * (a[old ia1 + 1] + a[ib + 1]);
        a2 := sin45 * (a[old ia1] - a[ib]);
        c1 := old c1[old ja1:2.0 * (a[old ia1] + a[ib]); jb:2.0 * (a2 - a1);
                jc:2.0 * (a[ib + 1] - a[old ia1 + 1]); jd: - 2.0 * (a1 + a2)]

        RETURNS VALUE of ia1
              VALUE of ja1
              VALUE of c1
        END FOR

  END IF;

RETURNS VALUE of c
END FOR
END FUNCTION
```

-- **PassGrid.sis**

```
DEFINE PassGrid

TYPE ArrReal1 = Array[real]

GLOBAL IFACTg_2ETC(m, la, iink, jink, jump, inc1, inc2 : integer;  a, ci, trigs : ArrReal1
                   RETURNS ArrReal1)

GLOBAL IFACTg_3(m, la, iink, jink, jump, inc1, inc2 : integer;   a, ci, trigs : arrReal1
                   RETURNS arrReal1)

GLOBAL IFACTg_4(m, la, iink, jink, jump, inc1, inc2 : integer;  a, ci, trigs : ArrReal1
                   RETURNS ArrReal1)
```

```
FUNCTION PassGrid(inc1, inc2, n, ifac, la : integer;  a, c, trigs : ArrReal1 RETURNS ArrReal1) % c
LET      m := n / ifac;
         iink := inc1 * m;
         jink := inc2 * la;
         jump := (ifac - 1) * jink;
         igo := ifac - 1;
         c_return :=          IF igo = 2  THEN IFACTg_3(m, la, iink, jink, jump, inc1, inc2, a, c, trigs)
                              ELSEIF igo = 3  THEN IFACTg_4(m, la, iink, jink, jump, inc1, inc2, a, c, trigs)
                              ELSE IFACTg_2ETC(m, la, iink, jink, jump, inc1, inc2, a, c, trigs)
                              END IF;
IN c_return
END LET
END FUNCTION
```

-- **MdFFTGrid.sis**

```
DEFINE MdFFTGrid

TYPE ArrInt1 = Array[integer];
TYPE ArrReal1 = Array[real]

GLOBAL PassGrid(inc1, inc2, n, ifac, la : integer;  a, c, trigs : ArrReal1 RETURNS ArrReal1) % c

FUNCTION MdFFTGrid( nlev, nwave, nfax, n : integer;          %nwave=mx
                    ifax : ArrInt1; trigs, gridii : ArrReal1 RETURNS ArrReal1)
LET
% Need to fix itself, hence: gridii has size mx * 2; grid n*2+2
gridi := gridii || ARRAY_FILL(1, n * 2 + 2 - nwave * 2, 0.0);

x := ARRAY_FILL(1, (n + 1) * 2, 0.0);
work := ARRAY_FILL(1, (n + 1) * 2, 0.0);
grid :=  IF nfax <= 1 THEN PassGrid(2, 2, n, ifax[1], 1, gridi, x, trigs)
         ELSE LET grid_2dim:=
                        FOR ilev IN 1, nlev
                        next_R, x_R,  work_R :=
                               IF MOD(nfax, 2) = 1
                               THEN 40, PassGrid(2, 2, n, ifax[1], 1, gridi, x, trigs), work
                               ELSE 50, x, PassGrid(2, 2, n, ifax[1], 1, gridi, work, trigs)
                               END IF;
                        la3,  work_la3 :=
                               FOR INITIAL
                               loop := 2;
                               la3 := 1;
                               next3 := next_R;
                               x_la3 := x_R;
                               work_la3 := work_R;
                               WHILE loop <= (nfax - 1) REPEAT
                               loop := old loop + 1;
                               la3 :=  old la3 * ifax[old loop - 1];
                               next3, x_la3, work_la3:=
                                   IF old next3 = 50 THEN  40, PassGrid(2, 2, n, ifax[old loop], la3,
                                                                 old work_la3, old x_la3, trigs),
                                                        old work_la3
                                   ELSE 50, old x_la3,
                                        PassGrid(2, 2, n, ifax[old loop], la3, old x_la3, old work_la3, trigs)
                                   END IF;
                        RETURNS  VALUE of la3
                                   VALUE of work_la3
                        END FOR;
                  la := la3  *  ifax[nfax - 1];
                  grid:=PassGrid(2, 1, n, ifax[nfax], la, work_la3, gridi, trigs);
                  RETURNS  ARRAY  of grid
                  END  FOR
              IN grid_2dim[1]        % Here only the first layer is needed
             END LET
         END IF;
IN grid
END LET
END FUNCTION
```

-- **VertigSphere.sis**

```
DEFINE VertigSphere
TYPE ArrReal1 = ARRAY[real];
TYPE ArrReal2 = ARRAY[ArrReal1];
TYPE ArrReal3 = ARRAY[ArrReal2]

FUNCTION VertigSphere (longitude_END, ilath : integer;  pg, zg, ug, vg : ArrReal3
                RETURNS ArrReal3, ArrReal3, ArrReal3, ArrReal3, ArrReal3)
LET
% The rest_0 below is catenated with the longitude point array in order to
% satisfy the requirement for MDFFTM which needs arraysize of ix * 2 + 2
% rest_0 := ARRAY_FILL(1, longitude_END + 2, 0.0);
%
% But MdFFTGrid supplies Grid arrays of size ix * 2 + 2 already, so this
% is unnecessary here

eg, pvg, pug, zvg, zug :=
        FOR hemi IN 1, 2 CROSS latlev IN 1, ilath
        eg, pvg, pug, zvg, zug :=
                FOR longpt IN 1, longitude_END * 2 + 2      % modified bound
                RETURNS ARRAY of ug[hemi, latlev, longpt] * ug[hemi, latlev, longpt]
                                        + vg[hemi, latlev, longpt] * vg[hemi, latlev, longpt]
                        ARRAY of pg[hemi, latlev, longpt] * vg[hemi, latlev, longpt]
                        ARRAY of pg[hemi, latlev, longpt] * ug[hemi, latlev, longpt]
                        ARRAY of zg[hemi, latlev, longpt] * vg[hemi, latlev, longpt]
                        ARRAY of zg[hemi, latlev, longpt] * ug[hemi, latlev, longpt]
                END FOR
                                % hence unnecessary
        RETURNS ARRAY of eg      %||     rest_0
                ARRAY of pvg     %||     rest_0
                ARRAY of pug     %||     rest_0
                ARRAY of zvg     %||     rest_0
                ARRAY of zug     %||     rest_0
        END FOR
IN eg, pug, pvg, zug, zvg
END LET
END FUNCTION
```

-- **IFACTm_2ETC.sis**

```
DEFINE IFACTm_2ETC

TYPE ArrReal1 = Array[real]

FUNCTION IFACTm_2ETC(m, la, iink, jink, jump, inc1, inc2 : integer;  a, ci, trigs : ArrReal1
                RETURNS ArrReal1)
FOR INITIAL
k := 0;
ia := 1;
ja := 1;
c := ci;

WHILE k <= m / 2 REPEAT
k := old k + la;
ia, ja, c :=
 IF old k = 0
 THEN LET ia1, ja1, c1 :=

        FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;
```

```
        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        jb := old ja1 + jink;
        c1 := old c1[old ja1 : a[old ia1] + a[ib]; jb : a[old ia1]-a[ib]]

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1 + jump, ja1, c1
        END LET

ELSEIF 2 * old k < m
THEN    LET ia1, ja1, c1 :=

        FOR INITIAL
        kb := old k * 2;
        lower := jink + 2;
        lupper := iink * 2 + jink - 2;
        l := lower;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;
        WHILE l <= lupper REPEAT
        l := old l + 4;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        jbxx := old l - old ja1;
        tempr := a[ib] * trigs[kb + 1] - a[ib + 1] * trigs[kb + 2];
        tempi := a[ib] * trigs[kb + 2] + a[ib + 1] * trigs[kb + 1];
        c1 := old c1[old ja1 : a[old ia1] + tempr; jbxx : a[old ia1] - tempr;
                old ja1 + 1 : a[old ia1 + 1] + tempi;  jbxx + 1 : tempi - a[old ia1 + 1]];
        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1 + jump, ja1, c1
        END LET

ELSE    FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        c1 := old c1[old ja1 : a[old ia1]; old ja1 + 1 : - a[ib]];

        RETURNS VALUE of ia1
              VALUE of ja1
              VALUE of c1
        END FOR

END IF;

RETURNS VALUE of c
END FOR
END FUNCTION
```

-- **IFACTm_3.sis**

```
DEFINE IFACTm_3

TYPE ArrReal1 = ARRAY [real]

FUNCTION IFACTm_3(m, la, iink, jink, jump, inc1, inc2 : integer;  a, ci, trigs : ArrReal1
                  RETURNS ArrReal1)
FOR INITIAL
sin60 := 0.866025403784437;
k := 0;
ia := 1;
ja := 1;
c := ci;

WHILE k <= m / 2 REPEAT
k := old k + la;
ia, ja, c :=
 IF old k = 0
 THEN LET ia1, ja1, c1 :=

        FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        ic := ib + iink;
        jb := old ja1 + jink;
        a1 := a[ib] + a[ic];
        a2 := a[old ia1] - 0.5 * a1;
        a3 := sin60 * (a[ic] - a[ib]);
        c1 := old c1[old ja1 : a[old ia1] + a1; jb : a2; jb + 1 : a3];

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1 + jump, ja1, c1
        END LET

ELSEIF 2 * old k < m
THEN   LET ia1, ja1, c1 :=

        FOR INITIAL
        kb := old k * 2;
        kc := kb * 2;
        lower := jink + 2;
        lupper := iink * 2 + jink - 2;
        l := lower;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= lupper REPEAT
        l := old l + 4;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        ic := ib + iink;
        jb := old ja1 + jink;
        jcxx := old l - old ja1;
        tempr1 := a[ib] * trigs[kb + 1] - a[ib + 1] * trigs[kb + 2];
```

```
        tempi1 := a[ib] * trigs[kb + 2] + a[ib + 1] * trigs[kb + 1];
        tempr2 := a[ic] * trigs[kc + 1] - a[ic + 1] * trigs[kc + 2];
        tempi2 := a[ic] * trigs[kc + 2] + a[ic + 1] * trigs[kc + 1];
        a1 := tempr1 + tempr2;
        b1 := tempi1 + tempi2;
        a2 := a[old ia1] - 0.5 * a1;
        b2 := a[old ia1 + 1] - 0.5 * b1;
        a3 := sin60 * (tempr2 - tempr1);
        b3 := sin60 * (tempi2 - tempi1);
        c1 :=   old c1[old ja1:a[old ia1] + a1; jb:a2 - b3; jcxx:a2 + b3;
                old ja1 + 1:a[old ia1 + 1] + b1; jb + 1:b2 + a3; jcxx + 1:a3 - b2];

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1 + jump, ja1, c1
        END LET

ELSE    FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        ic := ib + iink;
        jb := old ja1 + jink;
        a1 := a[ib] - a[ic];
        c1 := old c1[old ja1 : a[old ia1] + 0.5 * a1; jb : a[old ia1] - a1;  old ja1 + 1 : -sin60 * (a[ib] + a[ic])];

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR
END IF;

RETURNS VALUE of c
END FOR
END FUNCTION
```

```
------------------------------------------------------------------------------------------ IFACTm_4.sis

DEFINE IFACTm_4

TYPE ArrReal1 = ARRAY [real]

FUNCTION IFACTm_4(m, la, iink, jink, jump, inc1, inc2 : integer;   a, ci, trigs : ArrReal1
                RETURNS ArrReal1)
FOR INITIAL
sin45 := 0.7071067812;
k := 0;
ia := 1;
ja := 1;
c := ci;
```

```
WHILE k <= m / 2 REPEAT
k := old k + la;
ia, ja, c :=
 IF old k = 0
 THEN LET ia1, ja1, c1 :=

        FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        ic := ib + iink;
        id := ic + iink;
        jb := old ja1 + jink;
        jc := jb + jink;
        a1 := a[old ia1] + a[ic];
        a2 := a[ib] + a[id];
        a3 := a[old ia1] - a[ic];
        a4 := a[id] - a[ib];
        c1 := old c1[old ja1 : a1 + a2; jb : a3; jc : a1 - a2; jb + 1 : a4]

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1 + jump, ja1, c1
        END LET

ELSEIF 2 * old k < m
THEN    LET ia1, ja1, c1 :=

        FOR INITIAL
        lower := jink + 2;
        lupper := iink * 2 + jink - 2;
        l := lower;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;

        WHILE l <= lupper REPEAT
        l := old l + 4;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        ic := ib + iink;
        id := ic + iink;
        jb := old ja1 + jink;
        jdxx := old l - old ja1;
        jcxx := jdxx + jink;
        kb := old k * 2;
        kc := kb * 2;
        kd := kc + kb;
        tempr1 := a[ib] * trigs[kb + 1] - a[ib + 1] * trigs[kb + 2];
        tempi1 := a[ib] * trigs[kb + 2] + a[ib + 1] * trigs[kb + 1];
        tempr2 := a[ic] * trigs[kc + 1] - a[ic + 1] * trigs[kc + 2];
        tempi2 := a[ic] * trigs[kc + 2] + a[ic + 1] * trigs[kc + 1];
        tempr3 := a[id] * trigs[kd + 1] - a[id + 1] * trigs[kd + 2];
        tempi3 := a[id] * trigs[kd + 2] + a[id + 1] * trigs[kd + 1];
        a0 := a[old ia1] + tempr2;
        a1 := tempr1 + tempr3;
        a2 := a[old ia1] - tempr2;
        a3 := tempr1 - tempr3;
```

```
        b0 := a[old ia1 + 1] + tempi2;
        b1 := tempi1 + tempi3;
        b2 := a[old ia1 + 1] - tempi2;
        b3 := tempi1 - tempi3;
        c1 := old c1[old ja1 : a0 + a1; jb : a2 + b3;  jcxx : a0 - a1; jdxx : a2 - b3;
                    old ja1 + 1 : b0 + b1; jb + 1 : b2 - a3;  jcxx + 1 : b1 - b0; jdxx + 1 : - b2 - a3];

        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

        IN ia1 + jump, ja1, c1
        END LET

  ELSE FOR INITIAL
        l := 1;
        ia1 := old ia;
        ja1 := old ja;
        c1 := old c;
        WHILE l <= la REPEAT
        l := old l + 1;
        ia1 := old ia1 + inc1;
        ja1 := old ja1 + inc2;
        ib := old ia1 + iink;
        ic := ib + iink;
        id := ic + iink;
        jb := old ja1 + jink;
        a1 := sin45 * (a[ib] - a[id]);
        a2 := sin45 * (a[ib] + a[id]);
        c1 := old c1[old ja1 : a[old ia1] + a1; jb : a[old ia1] - a1;  old ja1 + 1 : - a[ic] - a2; jb + 1 : a[ic] - a2];
        RETURNS VALUE of ia1
                VALUE of ja1
                VALUE of c1
        END FOR

  END IF;

RETURNS VALUE of c
END FOR
END FUNCTION
```

-- **PassFreq.sis**

```
DEFINE PassFreq

TYPE ArrReal1 = Array [real]

GLOBAL IFACTm_2ETC(m, la, iink, jink, jump, inc1, inc2 : integer;   a, ci, trigs : ArrReal1
                RETURNS ArrReal1)
GLOBAL IFACTm_3(m, la, iink, jink, jump, inc1, inc2 : integer;  a, ci, trigs : ArrReal1
                RETURNS ArrReal1)
GLOBAL IFACTm_4(m, la, iink, jink, jump, inc1, inc2 : integer;  a, ci, trigs : ArrReal1
                RETURNS ArrReal1)

FUNCTION PassFreq(inc1, inc2, n, ifac, la : integer;   a, c, trigs : ArrReal1 RETURNS ArrReal1) % c
LET     m := n / ifac;
        jink := inc2 * m;
        iink := inc1 * la;
        jump := (ifac - 1) * iink;
        igo := ifac - 1;
        c_return := IF igo=2  THEN IFACTm_3(m, la, iink, jink, jump, inc1, inc2, a, c, trigs)
                ELSEIF igo=3  THEN IFACTm_4(m, la, iink, jink, jump, inc1, inc2, a, c, trigs)
                ELSE IFACTm_2ETC(m, la, iink, jink, jump, inc1, inc2, a, c, trigs)
                END IF;
IN c_return
END LET
END FUNCTION
```

-- **MdFFTFreq.sis**

```
DEFINE MdFFTFreq

TYPE ArrIntl = Array [integer];
TYPE ArrReal1 = Array [real]

GLOBAL PassFreq(inc1, inc2, n, ifac, la : integer;  a, c, trigs : ArrReal1 RETURNS ArrReal1) % c

FUNCTION MdFFTFreq( nlev, nwave, nfax, n : integer; ifax : ArrIntl; trigs, grid : ArrReal1
                    RETURNS ArrReal1)
LET
la1 := n / ifax[1];
x_0 := ARRAY_FILL (1,n*2+2,0.0);
work_0 := ARRAY_FILL (1,n*2+2,0.0);
fourier_2dim:=
  FOR level IN 1, nlev
  x :=   IF nfax <= 1  THEN PassFreq(1, 2, n, ifax[1], la1, grid, x_0, trigs)
         ELSE LET  next1, x1, work1 :=
                        IF MOD(nfax, 2) = 1
                        THEN 40, PassFreq(1, 2, n, ifax[1], la1, grid, x_0, trigs), work_0
                        ELSE 50, x_0, PassFreq(1, 2, n, ifax[1], la1, grid, work_0, trigs)
                        END IF;
                   x := FOR INITIAL
                        loop := 2;
                        la := la1;
                        next := next1;
                        x := x1;
                        work := work1;
                        WHILE loop <= nfax REPEAT
                        loop := old loop + 1;
                        la := old la / ifax[old loop];
                        next, x, work :=
                           IF old next = 50
                           THEN 40, PassFreq(2, 2, n, ifax[old loop], la, old work, old x, trigs), old work
                           ELSE 50, old x, PassFreq(2, 2, n, ifax[old loop], la,  old x, old work, trigs)
                           END IF;
                        RETURNS VALUE of x
                        END FOR;
           IN x
           END LET
           END IF;

  fourier :=        FOR truncated_index IN 1, nwave * 2
                    RETURNS ARRAY of x[truncated_index]
                    END FOR
                    || ARRAY_FILL(nwave * 2 + 1, n, 0.0)

   RETURNS ARRAY of fourier
   END FOR

IN fourier_2dim[1]          % Only one layer is needed
END LET
END FUNCTION
```

-- **FreqToSpecSphere.sis**

```
DEFINE FreqToSpecSphere

TYPE arrreal1=ARRAY [real];
TYPE ArrReal2=ARRAY [ArrReal1];
TYPE ArrReal3=ARRAY [ArrReal2];
TYPE arrint1=ARRAY [integer]

FUNCTION FreqToSpecSphere(jx, mx, mx2, ilath, iy : integer; kmjx, kmjxx : ArrInt1;
                          wocs, epsi : ArrReal1; alp : ArrReal2; ef, puf, pvf, zuf, zvf : ArrReal3
               RETURNS ArrReal1, ArrReal1, ArrReal1, ArrReal1)
LET
eP, puP, pvP, zuP, zvP,           % symmetric: North + South
eM, puM, pvM, zuM, zvM:=          % anti-symmetric: North - South
        FOR latlev IN 1, ilath CROSS mri IN 1, mx2
        RETURNS % symmetric
                ARRAY of ef[1, latlev, mri]  +  ef[2, latlev, mri]
                ARRAY of puf[1, latlev, mri]  +  puf[2, latlev, mri]
                ARRAY of pvf[1, latlev, mri]  +  pvf[2, latlev, mri]
                ARRAY of zuf[1, latlev, mri]  +  zuf[2, latlev, mri]
                ARRAY of zvf[1, latlev, mri]  +  zvf[2, latlev, mri]

                % anti-symmetric
                ARRAY of ef[1, latlev, mri] - ef[2, latlev, mri]
                ARRAY of puf[1, latlev, mri] - puf[2, latlev, mri]
                ARRAY of pvf[1, latlev, mri] - pvf[2, latlev, mri]
                ARRAY of zuf[1, latlev, mri] - zuf[2, latlev, mri]
                ARRAY of zvf[1, latlev, mri] - zvf[2, latlev, mri]
         END FOR;

ctri, eri, ptri, ztri :=
  FOR m IN 1, mx
  mi := m * 2;
  mr := mi - 1;
  realm := m - 1;
   ctri_m, eri_m, ptri_m, ztri_m :=   % loop
        FOR jj IN 1, jx * 2
        j := (jj + 1) / 2;
        jm := kmjx[m] + j;
        jmrjmi := jm * 2 - mod(jj, 2);
        jmx := kmjxx[m] + j;
        realn := real( j +  m - 2);

        ctri_jj, eri_jj, ptri_jj, ztri_jj:=
          FOR latlev IN 1, ilath
          ihem := iy + 1 - latlev;
%         ------------------------ for symmetric parts
          gwplm := alp[latlev, jmx] * wocs[ihem];
          b := real(realm) * gwplm;
%         ------------------------ for antisymmetric parts
          alpm := IF j ~= 1  THEN alp[latlev, jmx - 1] ELSE 0.0 END IF;
          alpp := alp[latlev, jmx + 1];
          a := ((realn + 1.0) * epsi[jmx] * alpm  -  realn * epsi[jmx + 1] * alpp) * wocs[ihem];
%         ------------------------

           ctri_jm, eri_jm, ptri_jm, ztri_jm :=
                IF ~(j = 1 & m = 1)
                THEN  IF MOD(jm, 2) = 0
                      THEN  IF MOD(jmrjmi,2) = 0
                            THEN   a * puP[latlev, mi] + b * pvM[latlev, mr],
                                   gwplm * eM[latlev, mi],
                                   a * pvP[latlev, mi] - b * puM[latlev, mr],
                                   a * zvP[latlev, mi] - b * zuM[latlev, mr]
                            ELSE   a * puP[latlev, mr] - b * pvM[latlev, mi],
                                   gwplm * eM[latlev, mr],
                                   a * pvP[latlev, mr] + b * puM[latlev, mi],
                                   a * zvP[latlev, mr] + b * zuM[latlev, mi]
                            END IF
```

```
                    ELSEIF MOD(jmrjmi, 2) = 0 THEN  a * puM[latlev, mi] + b * pvP[latlev, mr],
                                                    gwplm * eP[latlev, mi],
                                                    a * pvM[latlev, mi] - b * puP[latlev, mr],
                                                    a * zvM[latlev, mi] - b * zuP[latlev, mr]
                    ELSE    a * puM[latlev, mr] - b * pvP[latlev, mi],
                            gwplm * eP[latlev, mr],
                            a * pvM[latlev, mr] + b * puP[latlev, mi],
                            a * zvM[latlev, mr] + b * zuP[latlev, mi]
                    END IF
                ELSE    0.0,    IF jj=1
                                THEN eP[latlev, 1] * wocs[ihem] * alp[latlev, 1]
                                ELSE 0.0 END IF,
                        0.0, 0.0
                END IF

         RETURNS        VALUE of SUM ctri_jm
                        VALUE of SUM eri_jm
                        VALUE of SUM ptri_jm
                        VALUE of SUM ztri_jm
         END FOR
       RETURNS          ARRAY of ctri_jj
                        ARRAY of eri_jj
                        ARRAY of ptri_jj
                        ARRAY of ztri_jj
       END FOR
   RETURNS    VALUE of CATENATE ctri_m
              VALUE of CATENATE eri_m
              VALUE of CATENATE ptri_m
              VALUE of CATENATE ztri_m
   END FOR

IN ctri, eri, ptri, ztri
END LET
END FUNCTION
```

-- **Linear.sis (AddLinear)**

```
DEFINE Linear

TYPE ArrInt1 = ARRAY [integer];
TYPE ArrReal1 = ARRAY [real];
TYPE CplexReal = RECORD [Repart,Impart:real];
TYPE ArrCplexReal = ARRAY [CplexReal]

GLOBAL Csub(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Cadd(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Crmul(cons : real; cnum : CplexReal RETURNS CplexReal)

% This subroutine adds the linear terms to the time-derivatives.
```

```
FUNCTION Linear(mx, jx : integer; kmjx, kmjxx, ksq : ArrInt1;  tw : real; epsi : ArrReal1;
                c, p, u, v, ctin, e, ptin : ArrCplexReal
                RETURNS ArrCplexReal, ArrCplexReal)
LET
zero := record CplexReal[Repart : 0.0; Impart : 0.0];

pt,  ct := FOR m IN 1, mx
        pt_m, ct_m :=
                FOR j IN 1, jx
                l := j + m - 2;
                kl := real(ksq[l]);
                jm := kmjx[m] + j;
                jm_p1 := jm + 1;
                jm_m1 := jm - 1;
                jmx := kmjxx[m] + j;

                pjm_p1, cjm_p1 :=  IF j = jx THEN zero, zero ELSE p[jm_p1], c[jm_p1] END IF;
                pjm_m1, cjm_m1 :=  IF j = 1 THEN zero, zero ELSE p[jm_m1], c[jm_m1] END IF;

                RETURNS ARRAY of Csub( ptin[jm], Crmul(tw,  Cadd(Crmul(epsi[jmx], cjm_m1),
                                                Cadd(Crmul(epsi[jmx + 1], cjm_p1), v[jmx]))) )
                        ARRAY of
                            Cadd( ctin[jm], Cadd(Crmul(tw, Csub(Cadd( Crmul(epsi[jmx], pjm_m1),
                                Crmul(epsi[jmx + 1], pjm_p1) ), u[jmx])),  Crmul(0.5 * kl, e[jm])) )
                END FOR;

        RETURNS VALUE of CATENATE pt_m
                VALUE of CATENATE ct_m
        END FOR;

IN pt, ct
END LET
END FUNCTION
```

-- **TStep.sis**

```
DEFINE TStep

TYPE ArrInt1 = ARRAY [integer];
TYPE ArrReal1 = ARRAY [real];
TYPE CplexReal = RECORD [Repart,Impart:real];
TYPE ArrCplexReal = ARRAY [CplexReal]

GLOBAL Csub(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Cadd(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Crmul(cons : real; cnum : CplexReal RETURNS CplexReal)
GLOBAL Crsub(cnum : CplexReal; cons : real RETURNS CplexReal)
GLOBAL Crdiv(cnum : CplexReal; cons : real a CplexReal)
%---------------------------
FUNCTION TStep(jx, mx, delt, izon, ifirst, imp, istart : integer; hdiff, hdrag, zmean, vnu : real;
                kmjx, kmjxx, ksq : ArrInt1; p1 : ArrReal1; c, p, z, cm, pm, zm, ct, pt, zt : ArrCplexReal
    RETURNS integer, ArrCplexReal, ArrCplexReal, ArrCplexReal,
        ArrCplexReal, ArrCplexReal, ArrCplexReal,  ArrCplexReal, ArrCplexReal, ArrCplexReal)
LET
deltt2 :=  IF ifirst = 0 THEN real(delt) * 2.0 ELSE real(delt) END IF;
deltt := deltt2 * 0.5;
zero := RECORD CplexReal[Repart : 0.0; Impart : 0.0];

newc, newp, newz, newcm, newpm, newzm, newct, newpt, newzt :=
  FOR m IN 1, mx
   c_m, p_m, z_m, cm_m, pm_m, zm_m, ct_m, pt_m, zt_m :=
        FOR j IN 1, jx
        jm := kmjx[m] + j;
        kl := real(ksq[j + m - 2]);
        dkl := kl - 2.0;
        c_j, p_j, z_j, cm_j, pm_j, zm_j, ct_j, pt_j, zt_j :=
                IF (m = 1 & izon = 1) | jm = 1
                THEN c[jm], p[jm], z[jm], cm[jm], pm[jm], zm[jm], ct[jm], pt[jm], zt[jm]
```

```
            ELSE   LET
                   ptjm := Csub( Csub(pt[jm], Crmul(dkl * hdiff, pm[jm])),
                           Crmul(hdrag, Crsub(pm[jm], p1[jm])) );
                   ctjm := Csub( ct[jm],  Crmul(hdrag + dkl * hdiff, cm[jm]) );
                   ztjm := Csub( zt[jm],  Crmul(dkl * hdiff, zm[jm]) );
                   ppv := Cadd( pm[jm],  Crmul(deltt2, pt[jm]) );
                   ccv, zzv :=  IF imp=1
                                THEN LET
                           ccv1 := Crdiv(Cadd(cm[jm], Crmul(deltt2, Cadd(ctjm,
                                   Crmul(kl, Cadd(zm[jm], Crmul(deltt,
                                   Csub(ztjm, Crmul(0.5 * zmean, cm[jm]))))))),
                                   1.0 + deltt * deltt * kl * zmean );

                           zzv1 := Cadd(zm[jm], Crmul(deltt2, Csub(ztjm,
                                   Crmul(0.5 * zmean, Cadd(cm[jm], ccv1)))))

                                   IN ccv1, zzv1
                                   END LET

                             ELSE Cadd(cm[jm], Crmul(deltt2, Cadd(ctjm,  Crmul(kl, z[jm])))),
                                  Cadd( zm[jm], Crmul(deltt2, Csub(ztjm, Crmul(zmean, c[jm]))) )

                             END IF;

            pmjm,  cmjm,  zmjm,  pjm,  cjm,  zjm :=
               IF ifirst = 0
               THEN Cadd(p[jm], Crmul(vnu, Cadd(Csub(pm[jm],  Crmul(2.0, p[jm])), ppv))),

                     Cadd(c[jm], Crmul(vnu, Cadd(Csub(cm[jm],  Crmul(2.0, c[jm])), ccv))),

                     Cadd(z[jm], Crmul(vnu, Cadd(Csub(zm[jm], Crmul(2.0, z[jm])), zzv))),

                     ppv,  ccv,  zzv

               ELSE  pm[jm],  cm[jm],
                      IF istart = 0 THEN Crdiv(ctjm, - kl) ELSE zm[jm] END IF,
                      ppv,
                      IF istart = 0 THEN zero ELSE ccv END IF,
                      IF istart = 0 THEN Crdiv(ctjm, - kl) ELSE zzv END IF
               END IF;

            IN cjm, pjm, zjm, cmjm, pmjm, zmjm, ctjm, ptjm, ztjm
            END LET
    END IF;

    RETURNS        ARRAY  of c_j
                   ARRAY  of p_j
                   ARRAY  of z_j
                   ARRAY  of cm_j
                   ARRAY  of pm_j
                   ARRAY  of zm_j
                   ARRAY  of ct_j
                   ARRAY  of pt_j
                   ARRAY  of zt_j
    END FOR;
  RETURNS    VALUE  of CATENATE c_m
             VALUE  of CATENATE p_m
             VALUE  of CATENATE z_m
             VALUE  of CATENATE cm_m
             VALUE  of CATENATE pm_m
             VALUE  of CATENATE zm_m
             VALUE  of CATENATE ct_m
             VALUE  of CATENATE pt_m
             VALUE  of CATENATE zt_m
  END FOR;
ifirst_R := 0;
IN ifirst_R, newc, newp, newz, newcm, newpm, newzm, newct, newpt, newzt
END LET
END  FUNCTION
```

-- **Energy.sis**

```
DEFINE Energy
TYPE CplexReal = RECORD [Repart,Impart:real];
TYPE ArrCplexReal = ARRAY [CplexReal]
GLOBAL Csub(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Cmul(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Conjg(cnum : CplexReal RETURNS CplexReal)

FUNCTION Energy(jx, jxmx : integer; zmean, asq : real;  e, h, zm : arrCplexReal
                RETURNS real, real, real)
LET
% The following asSUMes h[1] and e[1] not having non - zero Impart.
% Otherwise, the following has to be rewritten.
gmass := 4.0 * (zmean - h[1].Repart) / asq;
backdown := 2 + jxmx;
ptot1, ktot1 :=  FOR j IN 2, jxmx
                 k := backdown - j;
                 zm_R := zm[k].Repart;
                 zm_I := zm[k].Impart;
                 conjg_e := Conjg(e[k]);
                 potential := zm_R * zm_R+zm_I * zm_I;
                 kinetic := Cmul(Csub(zm[k], h[k]), conjg_e).Repart;
                 RETURNS VALUE of SUM  IF k > jx THEN 2.0 * potential ELSE potential END IF
                         VALUE of SUM IF k > jx THEN 2.0 * kinetic ELSE kinetic END IF

                 END FOR;
ptot := ptot1/gmass;
ktot := (ktot1 + e[1].Repart * 1.4142136 * (zmean - h[1].Repart)) / gmass;
total := ptot + ktot;
IN ptot, ktot, total
END LET
END FUNCTION
```

-- **AngMom.sis**

```
DEFINE AngMom
TYPE CplexReal = RECORD [Repart, Impart : real];
TYPE ArrCplexReal = ARRAY [CplexReal]
GLOBAL SQRTR(num : real RETURNS real)
GLOBAL Csub(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Cmul(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Conjg(cnum : CplexReal RETURNS CplexReal)

FUNCTION AngMom(jx, jxmx : integer; zmean,  asq, ww : real;  u, h, zm, z : ArrCplexReal
                RETURNS real, real, real, real, real)
LET
c1, c2, c3, c4 := 0.942809,  0.421637,  1.4142136,  1E-5;
gmass := 4.0 * (zmean - h[1].Repart) / asq;

atot1 := u[1].Repart * c3 * (zmean - h[1].Repart);
backdown := 2 + jxmx;
atotup := FOR j IN 2, jxmx
          k := backdown - j;
          conjg_u := Conjg(u[k]);
          relative := Cmul(Csub(zm[k], h[k]), conjg_u).Repart;
         RETURNS VALUE of SUM IF k > jx THEN 2.0 * relative ELSE relative END IF
         END FOR;

atot := (atot1 + atotup) / gmass * c4;
atot_1 := atot1 / gmass * c4;
wtot := ww * ( - c2 * (z[3].Repart - h[3].Repart)) / gmass * c4;
total := atot + wtot;
total1 := atot_1 + wtot;

IN atot, atot_1, wtot, total, total1
END LET
END FUNCTION
```

-- **Specam.sis**

```
DEFINE Specam

TYPE ArrIntl = ARRAY [integer];
TYPE ArrReal1 = ARRAY [real];
TYPE CplexReal = RECORD [Repart,Impart:real];
TYPE ArrCplexReal = ARRAY [CplexReal]

GLOBAL SQRTR(num : real RETURNS real)

%----------------------------
FUNCTION Specam(jx, mx : integer; kmjx : ArrIntl;  asq, ww, grav :  real;  c, p, z : ArrCplexReal
            RETURNS ArrReal1, ArrReal1, ArrReal1)
LET  ampk, ampvor, ampz :=
        FOR m IN 1, mx
        ampk_m, ampvor_m, ampz_m :=
            FOR j IN 1, jx
            jm := kmjx[m] + j;
            c_R,c_I := c[jm].Repart, c[jm].Impart;
            p_R,p_I := p[jm].Repart, p[jm].Impart;
            z_R,z_I := z[jm].Repart, z[jm].Impart;
            div := c_R * c_R + c_I * c_I;
            vor := p_R * p_R + p_I * p_I;
            sq := z_R * z_R + z_I * z_I;
            divsq, vorsq, zsq :=  IF m > 1 THEN 2.0 * div, 2.0 * vor, 2.0 * sq
                                  ELSE div, vor, sq  END IF;
            RETURNS VALUE of SUM divsq
                    VALUE of SUM vorsq
                    VALUE of SUM zsq
            END FOR;

        RETURNS ARRAY of SQRTR(ampk_m) / ww * 10.0
                ARRAY of SQRTR(ampvor_m) / ww
                ARRAY of SQRTR(ampz_m) * asq / grav
        END FOR;
IN ampk, ampvor, ampz
END LET
END FUNCTION
```

-- **Loop_TimeStep.sis (Timeloop Section)**

```
DEFINE Loop_TimeStep

TYPE ArrIntl = ARRAY [integer];
TYPE ArrReal1 = ARRAY [real];
TYPE ArrReal2 = ARRAY [ArrReal1];
TYPE ArrReal3 = ARRAY [ArrReal2];
TYPE CplexReal = RECORD [Repart,Impart:real];
TYPE ArrCplexReal = ARRAY [CplexReal];

GLOBAL Csub(cnum1, cnum2 : CplexReal RETURNS CplexReal)

GLOBAL U_V_Spectral(mx, jx, jxx: integer; epsi: ArrReal1; p, c: ArrCplexReal
            RETURNS  ArrCplexReal, ArrCplexReal)

GLOBAL Decomplexing_p_zdiff_u_v (jx, mx, jxx : integer; p, zdiff, u, v : ArrCplexReal
        RETURNS ArrReal1, ArrReal1, ArrReal1, ArrReal1)

GLOBAL SpecToFreqSphere(jx, mx, jxx, ilath, ixh : integer;  kmjx, kmjxx : ArrIntl;
                  alp : ArrReal3; pri, zri, uri, vri : ArrReal1
                  RETURNS ArrReal3, ArrReal3, ArrReal3, ArrReal3)

GLOBAL MdFFTGrid(  nlev, nwave, nfax, n : integer;  ifax : ArrIntl; trigb, gridi : ArrReal1
             RETURNS ArrReal1)
```

```
GLOBAL VertigSphere(longitude_END, ilath : integer;  pg, zg, ug, vg : ArrReal3
                    RETURNS ArrReal3, ArrReal3, ArrReal3, ArrReal3, ArrReal3)

GLOBAL MdFFTFreq(  nlev, nwave, nfax, n : integer;  ifax : ArrInt1; trigf, grid : ArrReal1
                    RETURNS ArrReal1)

GLOBAL FreqToSpecSphere(
        jx, mx, mx2, ilath, iy : integer; kmjx, kmjxx : ArrInt1; wocs, epsi : ArrReal1; alp : ArrReal2;
        ef, puf, pvf, zuf, zvf : ArrReal3 RETURNS ArrReal1, ArrReal1, ArrReal1, ArrReal1)

GLOBAL Complexing_ct_e_pt_ztSp (jx, mx : integer; kmjx : ArrInt1; ct, e, pt, zt : ArrReal1
          RETURNS ArrCplexReal, ArrCplexReal, ArrCplexReal, ArrCplexReal)

GLOBAL Linear(  mx, jx : integer; kmjx, kmjxx, ksq : ArrInt1;  tw : real; epsi : ArrReal1;
                c, p, u, v, ctin, e, ptin : ArrCplexReal RETURNS ArrCplexReal, ArrCplexReal)

GLOBAL TStep(jx, mx, delt, izon, ifirst, imp, istart : integer; hdiff, hdrag, zmean, vnu : real;
                kmjx, kmjxx, ksq : ArrInt1; p1 : ArrReal1; c, p, z, cm, pm, zm, ct, pt, zt : ArrCplexReal
        RETURNS integer, ArrCplexReal, ArrCplexReal, ArrCplexReal,  ArrCplexReal,
                ArrCplexReal, ArrCplexReal,  ArrCplexReal, ArrCplexReal, ArrCplexReal)

GLOBAL Energy(  jx, jxmx : integer; zmean, asq : real;  e, h, zm : arrCplexReal
                RETURNS real, real, real)

GLOBAL AngMom(  jx, jxmx : integer; zmean,  asq, ww : real;  u, h, zm, z : ArrCplexReal
                RETURNS real, real, real, real, real)

GLOBAL Specam(  jx, mx : integer; kmjx : ArrInt1;  asq, ww, grav :  real;  c, p, z : ArrCplexReal
                RETURNS ArrReal1, ArrReal1, ArrReal1)

FUNCTION Loop_TimeStep (mx, jx, jxx, ilin, mx2, jxmx, jxxmx, nfax, ilath, imp,  istart, idumpt,
        ir, irmax2, ires, ix, ixh, iy, delt, ilong,  izon, ifirst, ihkont : integer;
        hdiff, hdrag, tw, zmean, vnu, asq, ww, grav : real;
        kmjx, kmjxx, ksq, ifax : ArrInt1; epsi, wocs : ArrReal1; alp : ArrReal3;
        p, c, z, h : ArrCplexReal; p1 : ArrReal1; pm, cm, zm : ArrCplexReal; trigb, trigf : ArrReal1
        RETURNS         Integer, ArrCplexReal, ArrCplexReal,
                        ArrCplexReal, ArrCplexReal, ArrCplexReal, ArrCplexReal,
                        ArrCplexReal, ArrCplexReal, ArrCplexReal, ArrCplexReal,
                        %ArrReal2, ArrReal2, ArrReal2, ArrReal2,
                        ArrReal1, ArrReal1, ArrReal1, real, real, real, real, real, real, real, real)
LET
%zdiff :=   FOR jm IN 1, jxmx
%           RETURNS ARRAY of Csub(z[jm], h[jm])
%           END FOR;
% Introducing static indexing array kmjx:

zdiff :=  FOR m IN 1, mx
          RETURNS VALUE of CATENATE
                FOR j IN 1,jx
                jm := kmjx[m] + j
                RETURNS ARRAY of Csub(z[jm], h[jm])
                END FOR
          END FOR;

u, v :=   U_V_Spectral(mx, jx, jxx, epsi, p, c);          % The spectral U- and V-fields

pDec, zDec, uDec, vDec := Decomplexing_p_zdiff_u_v(jx, mx, jxx, p, zdiff, u, v);

pIF, zIF, uIF, vIF := SpecToFreqSphere(jx, mx, jxx, ilath, ixh, kmjx, kmjxx, alp, pDec, zDec, uDec, vDec);

p_Grid, z_Grid, u_Grid, v_Grid :=
        FOR hemi IN 1, 2 CROSS latlev IN 1,ilath
        RETURNS ARRAY of MdFFTGrid(1, mx, nfax, ix, ifax, trigb, pIF[hemi, latlev])
                ARRAY of MdFFTGrid(1, mx, nfax, ix, ifax, trigb, zIF[hemi, latlev])
                ARRAY of MdFFTGrid(1, mx, nfax, ix, ifax, trigb, uIF[hemi, latlev])
                ARRAY of MdFFTGrid(1, mx, nfax, ix, ifax, trigb, vIF[hemi, latlev])
        END FOR;

e_Grid, pu_Grid, pv_Grid, zu_Grid, zv_Grid := VertigSphere(ilong, ilath, p_Grid, z_Grid, u_Grid, v_Grid);
```

```
eIF, puIF, pvIF, zuIF, zvIF :=
        FOR hemi IN 1, 2 CROSS latlev IN 1,ilath
        RETURNS ARRAY of MdFFTFreq(1, mx, nfax, ix, ifax, trigf, e_Grid[hemi, latlev])
                    ARRAY of MdFFTFreq(1, mx, nfax, ix, ifax, trigf, pu_Grid[hemi, latlev])
                    ARRAY of MdFFTFreq(1, mx, nfax, ix, ifax, trigf, pv_Grid[hemi, latlev])
                    ARRAY of MdFFTFreq(1, mx, nfax, ix, ifax, trigf, zu_Grid[hemi, latlev])
                    ARRAY of MdFFTFreq(1, mx, nfax, ix, ifax, trigf, zv_Grid[hemi, latlev])
        END FOR;

ctSp, eSp, ptSp, ztSp := FreqToSpecSphere(jx, mx, mx2, ilath, iy, kmjx, kmjxx,
                                    wocs, epsi, alp[1], eIF, puIF, pvIF, zuIF, zvIF);

ct, e, pt, zt := Complexing_ct_e_pt_ztSp(jx, mx, kmjx, ctSp, eSp, ptSp, ztSp);

ptlin, ctlin := Linear(mx, jx, kmjx, kmjxx, ksq, tw, epsi, c, p, u, v, ct, e, pt);  % c, e, p, u, v, pt, ct);

ifirst_ttp, c_ttp, p_ttp, z_ttp, cm_ttp, pm_ttp, zm_ttp,  ct_ttp, pt_ttp, zt_ttp :=
        TStep(jx, mx, delt, izon, ifirst, imp, istart, hdiff, hdrag, zmean, vnu, kmjx, kmjxx, ksq,
                    p1, c, p, z, cm, pm, zm, ctlin, ptlin, zt);

ptot, ktot, TotalEnergy := Energy(jx, jxmx, zmean, asq, e, h, zm_ttp);

atot, atot1, wtot, totalangmom, totalangmom1 := AngMom(jx, jxmx, zmean, asq, ww, u, h, zm_ttp, z_ttp);

ampk, ampvor, ampz :=       IF mod(ihkont, idumpt) = 0
                            THEN Specam(jx, mx, kmjx, asq, ww, grav, c_ttp, p_ttp, z_ttp)
                            ELSE ARRAY ArrReal1 [], ARRAY ArrReal1 [], ARRAY ArrReal1 []
                            END IF;

IN      ifirst_ttp, ct_ttp, pt_ttp, zt_ttp, e, c_ttp, p_ttp, z_ttp, cm_ttp, pm_ttp, zm_ttp,
%p_Grid[1] || p_Grid[2], z_Grid[1] || z_Grid[2],
%u_Grid[1] || u_Grid[2], v_Grid[1] || v_Grid[2],
        ampk, ampvor, ampz, atot, atot1, wtot, totalangmom, totalangmom1, ptot, ktot, totalenergy

END LET
END FUNCTION
```

-- **send.sis (Main Program)**

```
DEFINE MAIN

TYPE ArrInt1 = ARRAY [integer];
TYPE ArrReal1 = ARRAY [real];
TYPE ArrReal2 = ARRAY [ArrReal1];
TYPE ArrReal3 = ARRAY [ArrReal2];
TYPE ArrDreal1 = ARRAY [Double_real];
TYPE ArrDreal2 = ARRAY [ArrDreal1];
TYPE CplexReal = RECORD [Repart,Impart:real];
TYPE ArrCplexReal = ARRAY [CplexReal];

GLOBAL SIN(num : real RETURNS real)
GLOBAL ACOSR(num : real RETURNS real)

GLOBAL Cadd(cnum1, cnum2 : CplexReal RETURNS CplexReal)
GLOBAL Crmul(cons : real; cnum : CplexReal RETURNS CplexReal)
GLOBAL CabsSqr(cnum : CplexReal RETURNS real)

GLOBAL Inital (ires, ix, iy, mx, jx, jxx : integer; zmean1 : real
                RETURNS integer, integer, integer, integer, real, real, real, real, real,
                            arrint1, arrint1, arrint1, arrreal1)

GLOBAL InitFFT (n : integer RETURNS boolean, boolean, integer, arrint1, ArrReal1, ArrReal1)

GLOBAL GaussianQuadrature(nzero : integer
        RETURNS ArrReal1, ArrReal1, ArrReal1, ArrReal1, ArrReal1, ArrDreal1)

GLOBAL LegendrePolyOf1stKind(ir, irmax2, jxxmx : integer; coas, sias, deltas : real
                RETURNS ArrDreal1)
```

```
GLOBAL SasAlfaSphere(   ir, irmax2, jxxmx, ilath : integer; alp_double: ArrDReal2 RETURNS ArrReal3)

GLOBAL Loop_TimeStep( mx, jx, jxx, ilin, mx2, jxmx, jxxmx, nfax, ilath, imp,
        istart, idumpt, ir, irmax2, ires, ix, ixh, iy, delt, ilong, izon, ifirst, ihkont : integer;
        hdiff, hdrag, tw, zmean, vnu, asq, ww, grav : real; kmjx, kmjxx, ksq, ifax : ArrInt1;
        epsi, wocs : ArrReal1; alp : ArrReal3; p, c, z, h : ArrCplexReal; p1 : ArrReal1;
        pm, cm, zm : ArrCplexReal; trigb, trigf : ArrReal1
        RETURNS Integer, ArrCplexReal, ArrCplexReal, ArrCplexReal, ArrCplexReal,
              ArrCplexReal, ArrCplexReal, ArrCplexReal, ArrCplexReal, ArrCplexReal, ArrCplexReal,
                   %ArrReal2, ArrReal2, ArrReal2, ArrReal2,
              ArrReal1, ArrReal1, ArrReal1,  real, real, real, real, real, real, real, real)

FUNCTION MAIN (ires, ix, iy, ktotal, idelt, idumpt_i, nrun, imp, istart, izon, ilin:integer;
        zmean_1, hdiff, hdrag, vnu:real; p_in, c_in, z_in, zt_mountain:ArrCplexReal
        RETURNS Boolean, Boolean, % ArrReal3, ArrReal3, ArrReal3, ArrReal3,    % _Grid fields
        ArrCplexReal, ArrCplexReal, ArrCplexReal, ArrCplexReal, % tendency
        ArrReal1, ArrReal1, ArrReal1,                        % energy
        ArrReal1, ArrReal1, ArrReal1, ArrReal1, ArrReal1, % Angularmomentum
        ArrReal1, ArrReal1, ArrReal1)                        % specam
LET
ixh := ix/2;
iyh := iy/2;
jxx := ires + 2;
jx := ires + 1;
mx := ires + 1;
jxxmx := jxx * mx;
jxmx := jx * mx;
mxmx := mx * mx;
mx2 := mx * 2;
jxmx2 := jxmx * 2;
jxxmx2 := jxxmx * 2;
ifirst := 1;
itflag := 1;
iglobe := 2;
delt := idelt;
idumpt := IF idumpt_i = 0 THEN 1000 ELSE idumpt_i END IF;
zero := RECORD CplexReal[Repart : 0.0; Impart : 0.0];
ir, ilong, ilath, irmax2, ww, zmean, tw, asq, grav, kmjx, kmjxx, ksq_1_uncared_for, epsi :=
        Inital(ires, ix, iy, mx, jx, jxx, zmean_1);

ksq := ARRAY [0 : 0] || ksq_1_uncared_for || ARRAY [1 : 0, 0];

AbortFFT, AbortInitFFT, nfax, ifax, trigf, trigb := InitFFT(ix);

coa, w, sia, delta, wocs, WORKiyh := GaussianQuadrature(ilath);          % size iyh

wix :=  IF ilin = 0                 % Indeed
        THEN  FOR lat_level IN 1, ilath
              RETURNS ARRAY of w[lat_level] / real(ix)
              END FOR
        ELSE w
        END IF;                     % size iyh; of the North

winv, coainv :=  FOR lat_level IN 1, ilath
                 winv := wix[iy / 2 + 1 - lat_level];
                 coainv := -coa[iy / 2 + 1 - lat_level]
                 RETURNS ARRAY of winv
                         ARRAY of coainv
                 END FOR;
wiy, coaiy := wix || winv,  coa || coainv;      % size iy; of North & South
deltaiy, siaiy, wocsiy :=     % size iy; of North & South
        FOR lat_level IN 1, iy
        deltai := ACOSR(coaiy[lat_level]);
        siai := SIN(deltai);
        wocsi := wiy[lat_level] / (siai * siai);
        RETURNS ARRAY of deltai
                ARRAY of siai
                ARRAY of wocsi
        END FOR;
```

```
wocsilath, wilath :=
        IF iglobe = 2       % Indeed, highlight the South
        THEN  wocsiy, wiy          ELSE   FOR lat_level IN 1, ilath
                                          wocsiyhalf := 2.0 * wocsiy[lat_level]
                                          RETURNS ARRAY of wocsiyhalf
                                          END FOR
                                          || ARRAY_ADJUST (wocsiy, ilath + 1, iy),

                                          FOR lat_level IN 1, ilath
                                          wiyhalf := 2.0 * wiy[lat_level]
                                          RETURNS ARRAY of wiyhalf
                                          END FOR
                                          || ARRAY_ADJUST(wiy, ilath + 1, iy)
        END IF;

alp_double := FOR lat_level IN 1, ilath
                alp_LGN := LegendrePolyOf1stKind (ir, irmax2, jxxmx, coaiy[lat_level], siaiy[lat_level],
                                                                    deltaiy[lat_level]);
              RETURNS ARRAY of alp_LGN
              END FOR;                          % arraysize [iyh levels, spectral_indices]

alp := SasAlfaSphere (ir, irmax2, jxxmx, ilath, alp_double);

% When these two are put out seperately, iflopt disallows.
%constant := grav / asq;
%var := for diffindex in 2, jxmx returns value of sum CabsSqr(zt_mountain[diffindex])        end for;
%h := for index in 1, jxmx returns array of Crmul(constant, zt_mountain[index])    end for;

constant := grav / asq;
var, h :=  FOR index IN 1, jxmx
        RETURNS VALUE OF SUM IF index ~= 1 THEN CabsSqr(zt_mountain[index])
                                          ELSE 0.0 END IF
                  ARRAY OF Crmul(constant, zt_mountain[index])
        END FOR;

hnew := IF ilin = 0          % Indeed
        THEN h
        ELSE    ARRAY_FILL(1, jx, zero) || ARRAY_ADJUST(h, jx + 1, jxmx) END IF;

p, c_taken, z :=    FOR row IN 1, jxmx
                    p, c, z:= IF row <= 256 THEN p_in[row], c_in[row], z_in[row] ELSE zero, zero, zero
                            END IF;
                    RETURNS ARRAY of p
                            ARRAY of c
                            ARRAY of z
                    END FOR;

c :=      IF istart = 0 THEN ARRAY_FILL(1, jxmx, zero)      % Indeed
        ELSEIF ARRAY_SIZE(c_in) = 0 THEN ARRAY_FILL(1, jxmx, zero)
        ELSE c_taken  END IF;

znew := IF istart = 0                 % Indeed
        THEN  FOR m IN 1, mx  %---------------- Linear Balance Equation
              zjm :=   FOR j IN 1, jx
                       jm := kmjx[m] + j;
                       jmx := kmjxx[m] + j;
                       realn := real(m + j - 2);
                       realn1 := realn + 1.0;
                       zj:=IF ~(j = 1 & m = 1) & ~(j = jx & m = mx)
                           THEN Crmul( - tw / realn / realn1, Cadd(Crmul(realn1 / realn * epsi[jmx], p[jm - 1]),
                                            Crmul(realn / realn1 * epsi[jmx + 1], p[jm + 1])))
                           ELSEIF (j = jx & m = mx) THEN Crmul( - tw / realn / realn * epsi[jmx], p[jm - 1])
                           ELSE zero END IF
                       RETURNS  ARRAY  of zj
                       END FOR
              RETURNS VALUE of CATENATE zjm
              END FOR        %---------------------------
        ELSEIF ARRAY_SIZE(z_in) = 0 THEN ARRAY_FILL(1, jxmx, zero)
        ELSE z             END IF;
```

```
pm := p;
p1 :=   FOR j IN 1, jxmx
        RETURNS ARRAY of p[j].Repart
        END FOR;
cm := c;
zm := znew;

% Loop_TimeStep Procedure
% ktotal denotes the total number of timesteps intended.

%p_GridSphere, z_GridSphere, u_GridSphere, v_GridSphere,
newct, newpt, newzt, newe,         ptot, ktot, totalenergy,
atot, atot1, wtot, totalangmom, totalangmom1,       ampk, ampvor, ampz :=

        FOR INITIAL
        emptyArrCplexReal := ARRAY ArrCplexReal []; %emptyArrReal2 := ARRAY ArrReal2 [];
        emptyArrReal1 := ARRAY ArrReal1 [];
        Ifirst_loop := ifirst;
        newct, newpt, newzt, newe := emptyArrCplexReal, emptyArrCplexReal,
                                     emptyArrCplexReal, emptyArrCplexReal;
        newc, newp, newz, newcm, newpm, newzm := c, p, znew, cm, pm, zm;
        %p_GridSphere, z_GridSphere, u_GridSphere, v_GridSphere:=emptyArrReal2;
        ampk, ampvor, ampz := emptyArrReal1, emptyArrReal1, emptyArrReal1;
        atot, atot1, wtot, totalangmom, totalangmom1,
                ptot, ktot, totalenergy := 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0;
        th_time_step := 1;

        WHILE th_time_step <= ktotal REPEAT
        th_time_step := old th_time_step + 1;

        Ifirst_loop,      newct, newpt, newzt, newe,        newc, newp, newz, newcm, newpm, newzm,
        %p_GridSphere, z_GridSphere, u_GridSphere, v_GridSphere,
        ampk, ampvor, ampz,     atot, atot1, wtot, totalangmom, totalangmom1,
        ptot, ktot, totalenergy :=

          Loop_TimeStep(mx, jx, jxx, ilin, mx2, jxmx, jxxmx, nfax, ilath, imp,
                istart, idumpt, ir, irmax2, ires, ix, ixh, iy, delt, ilong, izon, old Ifirst_loop, old th_time_step,
                hdiff, hdrag, tw, zmean, vnu, asq, ww, grav, kmjx, kmjxx, ksq, ifax, epsi, wocsilath, alp,
                old newp, old newc, old newz, hnew, p1, old newpm, old newcm, old newzm, trigb, trigf);

        RETURNS %ARRAY of p_GridSphere     %ARRAY of z_GridSphere
                %ARRAY of u_GridSphere     %ARRAY of v_GridSphere
                VALUE of newct
                VALUE of newpt
                VALUE of newzt
                VALUE of newe
                 ARRAY of ptot
                 ARRAY of ktot
                ARRAY of TotalEnergy
                ARRAY of atot
                ARRAY of atot1
                ARRAY of wtot
                ARRAY of totalangmom
                ARRAY of totalangmom1
                VALUE of ampk
                VALUE of ampvor
                VALUE of ampz
        END FOR;

IN      AbortFFT, AbortInitFFT,  %p_GridSphere, z_GridSphere, u_GridSphere, v_GridSphere,
        newct, newpt, newzt, newe,  ptot, ktot, totalenergy,
        atot, atot1, wtot, totalangmom, totalangmom1, ampk, ampvor, ampz

END LET
END FUNCTION % -- Main
```

Appendix C

FAST FOURIER TRANSFORMATION (FFT) CODES

This appendix consists of the listings of Fast Fourier Transformation codes. Sections C.1 and C.2 list the original C version and the SISAL version respectively.

C.1 Original Code in C

The two dimensional FFT routines are readily available in image processing textbooks. Listed below is one recoded from an example in [GW].

```
   /*  program to perform Fast Fourier Transform */
              /* two dimentional data */

#include<stdio.h>
#include<math.h>
#define pi 3.141593
#define n 512
#define ln 9
struct complx{ float re,im; };

double  sine[512],cosine[512];
unsigned char image[n][n];        /* input-output image */
struct complx ff[n][n];        /* input - output array */
struct complx f[n];           /* row - column decompositions array */
struct complx  z,z1,z2,z3,w1;

main()
{
    int i,j;
    char  filename[10];
    FILE *fopen(),*fp;

 /* get input data */
  printf(" Enter input filename: ");
  scanf("%s",filename);
  fp=fopen(filename,"r");
   fread(&image[0][0],sizeof(image[0][0]),n*n,fp);
  fclose(fp);
  conversion();
```

```
/* perform forward Fourier Transform */
  sin_cos();
  do_fft();

/* create a Fourier spectrum image */
  printf(" Enter Fourier spectrum filename: ");
  scanf("%s",filename);
  fp=fopen(filename,"w");
  make_mag(fp);

/* create a phase spectrum image */
  printf(" Enter phase spectrum filename: ");
  scanf("%s",filename);
  fp=fopen(filename,"w");
  make_phase(fp);
}

/* convert real image into complex image with imaginary part is zero */
conversion()
{
 int i,j;

 for(i=0; i<n; i++)
  for(j=0; j<n; j++)
  {
    complex((float)image[i][j],0.0);
   ff[i][j]=w1;
  }
}

/* function to compute sin_cos table */
sin_cos()
{
 int m;
 double me,me1;

 for(m=0; m<=ln-1; m++)
  {
    me=pow(2.0,(float)m);
    cosine[m]=cos(pi/me);
    sine[m]=sin(pi/me);
   }
 }

/* do_fft */
do_fft()
{
 int i,j;

/* do column transformation */
 for(j=0; j<n; j++)
  {
    for(i=0; i<n; i++)
     f[i]=ff[i][j];
    fft(-1);
    for(i=0; i<n; i++)
     ff[i][j]=f[i];
  }

/* do row transformation */
 for(i=0; i<n; i++)
   {
    for(j=0; j<n; j++)
     f[j]=ff[i][j];
    fft(-1);
```

```
/* divide results by n */
    for(j=0; j<n; j++)
    {
      ff[i][j].re=f[j].re/(float)n;
      ff[i][j].im=f[j].im/(float)n;
   }
  }
}

/* create an image file */
make_mag(fp)
FILE *fp;
{
 int i,j;

 for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    image[i][j]=sqrt(ff[i][j].re*ff[i][j].re+ff[i][j].im*ff[i][j].im);
 fwrite(&image[0][0],1,n*n,fp);
 fclose(fp);
}

/*
  ************************************************************
 *
 *              create an phase image file
 *
************************************************************
*/

make_phase(fp)
FILE *fp;
{
 int i,j;

 for(i=0; i<n; i++)
  for(j=0; j<n; j++)
   image[i][j]=atan(ff[i][j].im/ff[i][j].re);
  fwrite(&image[0][0],1,n*n,fp);
  fclose(fp);
}


/* fft function */
fft(sign)   /* sign = -1 for forward FFT, 1 for inverse FFT */
int sign;
{
 struct complx u,w,t,cmpl;
 double le,le1;
 int i1,nv2,le2,le3,l,k,ip;
 int jn=0;

/* Reordered the input array */
  nv2=n/2;
 for(i1=0; i1<n-1; i1++)
  {
   k=nv2;
   if(i1<jn)
   {
    t=f[jn];
    f[jn]=f[i1];
    f[i1]=t;
   }
    while(k<=jn)
    {
     jn=jn-k;
     k=k/2;
    }
    jn=jn+k; }
```

```
/* Perform successive_doubling calculations */
 for(l=0; l<=ln-1; l++)
  {
    le=pow(2.0,(float)l);
    le1=le*2.0;
    complex(1.0,0.0);
   u=w1;
    complex(cosine[l],sign*sine[l]);
   w=w1;
   for(jn=0; jn<=le-1; jn++)
   {
    for(i1=jn; i1<=n-1; i1=i1+le1)
    {
      ip=i1+le;
            mult(f[ip],u);
      t=z1;
            sub(f[i1],t);
      f[ip]=z2;
      add(f[i1],t);
      f[i1]=z;
    }
    mult(u,w);
    u=z1;
   }
  }
}

complex(val1,val2)
float  val1,val2;
 {
   w1.re=val1;
   w1.im=val2;
 }

add(x,y)
struct complx x,y;
 {
   z.re=x.re + y.re;
   z.im=x.im + y.im;
 }

sub(x,y)
struct complx x,y;
 {
   z2.re=x.re - y.re;
   z2.im=x.im - y.im;
 }

mult(x,y)
struct complx x,y;
 {
   z1.re=x.re*y.re - x.im*y.im;
   z1.im=x.re*y.im + x.im*y.re;
 }
```

C.2 SISAL Version

The SISAL code listed below is, in most parts, a translated version from the C code :

```
% Author: Pau Sheong Chang
% let the input be (8, 3) or (4, 2)
DEFINE MAIN
TYPE ArrInt = ARRAY [integer];
TYPE ArrReal = ARRAY [real];
TYPE ArrReal2 = ARRAY [ArrReal];
TYPE Cplex = RECORD [Repart, Impart: real];
TYPE ArrCplex = ARRAY [Cplex];
TYPE ArrCplex2 = ARRAY [ArrCplex]

GLOBAL SIN(num: real RETURNS real)
GLOBAL COS(num: real RETURNS real)
GLOBAL ATAN(num: real RETURNS real)
GLOBAL SQRT(num: real RETURNS real)

FUNCTION Cadd(x,y: Cplex RETURNS Cplex)
RECORD Cplex [Repart: x.Repart + y.Repart; Impart: x.Impart + y.Impart]
END FUNCTION

FUNCTION Csub(x,y: Cplex RETURNS Cplex)
RECORD Cplex [Repart: x.Repart - y.Repart; Impart: x.Impart - y.Impart]
END FUNCTION

FUNCTION Cdiv(x: Cplex; y: real RETURNS Cplex)
RECORD Cplex [Repart: x.Repart / y; Impart: x.Impart / y]
END FUNCTION

FUNCTION Cmult(x, y : Cplex RETURNS Cplex)
LET     xre, xim, yre, yim := x.Repart, x.Impart, y.Repart, y.Impart
IN      RECORD Cplex [Repart: xre * yre - xim * yim; Impart: xre * yim + xim * yre]
END LET
END FUNCTION

FUNCTION fft(sign:real; ln, n: integer; sine, cosine: ArrReal; twotothepower:ArrInt;
                input_f: ArrCplex RETURNS ArrCplex)
% ReorderInputArray
LET
f :=    FOR INITIAL
        i1:=0;
        jn:=0;
        f:=input_f;
        WHILE i1 < n - 1 REPEAT
        k:=n/2;
        i1:=OLD i1 + 1;
        f:=     IF OLD i1<OLD jn
                THEN OLD f[OLD i1: OLD f[OLD jn]; OLD jn: OLD f[OLD i1]]
                ELSE OLD f
                END IF;
        jn:=    FOR INITIAL
                kk:=k;
                jnjn:=OLD jn;
                WHILE kk<=jnjn REPEAT
                jnjn:=OLD jnjn-OLD kk;
                kk:=OLD kk/2;
                RETURNS VALUE of jnjn+kk
                END FOR
        RETURNS VALUE of f
        END FOR;
```

```
%/* Perform successive_doublINg calculations */

 fnew :=  FOR INITIAL
          l := 0;
          f1 := f;
          WHILE l <= ln - 1 REPEAT
          le := twotothepower[OLD l];
          le1 := le * 2;
          w := RECORD Cplex [Repart: cosine[OLD l]; Impart: sign * sine[OLD l]];
          f1 :=     FOR INITIAL
                    jn := 0;
                    u := RECORD Cplex [Repart: 1.0; Impart: 0.0];
                    f2 := OLD f1;
                    WHILE jn <= le - 1 REPEAT
                    f2 :=     FOR INITIAL
                              i1 := OLD jn;
                              f3 := OLD f2;
                              WHILE i1 <= n - 1 REPEAT
                              ip := OLD i1 + le;
                              t := Cmult(OLD f3[ip], OLD u);
                              f3 := OLD f3[OLD i1: Cadd(OLD f3[OLD i1], t); ip: Csub(OLD f3[OLD i1], t)];
                              i1 := OLD i1 + le1;
                              RETURNS VALUE of f3
                              END FOR;
                    u := Cmult(OLD u, w);
                    jn := OLD jn + 1;
                    RETURNS VALUE of f2
                    END FOR;
          l := OLD l + 1;
          RETURNS VALUE of f1
          END FOR;
IN fnew
END LET
END FUNCTION
%--------------------------------


FUNCTION do_fft(ln, n: integer; sine, cosine: ArrReal; twotothepower: ArrInt; ff: ArrCplex2
               RETURNS ArrCplex2)
LET
% /* do row transformation */
realn:= REAL(n);
ffrow:=   FOR i IN 0, n - 1
          RETURNS ARRAY of fft(-1.0, ln, n, sine, cosine, twotothepower, ff[i])
          END FOR;
%/* do column transformation *costly/
newff:=   FOR j IN 0, n - 1
          column:=  FOR i IN 0, n - 1
                    RETURNS ARRAY of Cdiv(ffrow[i, j], realn)
                    END FOR;
          RETURNS ARRAY of fft(-1.0, ln, n, sine, cosine, twotothepower, column)
          END FOR;       % here the matrix has been transposed
% /* divide the results by n */
IN newff
END LET
END FUNCTION
```

```
FUNCTION MAIN(n, ln:integer RETURNS ArrReal2, ArrReal2)
LET
% /* get input data */
pi:= 3.141593;

%conversion()
%ff :=   FOR i IN 0, n - 1 CROSS j IN 0, n - 1
%        RETURNS ARRAY of  RECORD Cplex [Repart: 1.0; Impart: 0.0]
%        END FOR;

twotothepower:= FOR pow IN 0, ln - 1
                RETURNS ARRAY of  IF pow ~= 0
                                  THEN  FOR j IN 1, pow
                                        RETURNS VALUE of PRODUCT 2
                                        END FOR
                                  ELSE 1   END IF
                END FOR;

% /* perform forward Fourier Transform */
%        /* FUNCTION to compute SIN_COS table *% SIN_COS()/
cosine, sine :=  FOR m IN 0, ln - 1
                 pionme := pi / REAL(twotothepower[m]);
                 RETURNS      ARRAY of COS(pionme)
                              ARRAY of SIN(pionme)
                 END FOR;
newff := do_fft(ln, n, sine, cosine, twotothepower, ff);

%/* create a Fourier spectrum image */
image_magnitude, image_phase :=  FOR j IN 0, n - 1 CROSS i IN 0, n - 1
                                 re, im := newff[i, j].Repart, newff[i, j].Impart;
                                 RETURNS ARRAY of SQRT(re * re + im * im)
                                         ARRAY of IF re=0.0 THEN 0.0 ELSE ATAN(im / re) END IF
                                 END FOR;
IN image_magnitude, image_phase
END LET
END FUNCTION
```

References

[AD79] W. Ackerman and J. Dennis, "VAL - A Value-Oriented Algorithmic Language", Technical Report LCS/TR-218, Massachusetts Institute of Technology, June 1979.

[AA82] T. Agerwala and Arvind, "Data Flow Systems", *IEEE Computer Journal*, pp. 10-13, February 1982.

[AG] G.Alamasi and A. Gottlieb, *Highly Parallel Computing*, pp. 31-41, The Benjamin/Cummings Publishing Company, Inc., 1989

[Amd67] G. Amdahl, "Validity of the Single-Processor Approach to Achieving Large-Scale Computer Capabilities", *Proceedings of AFIPS Spring Joint Computer Conference*, pp. 483-485, Atlantic City, NJ, 1967.

[AE87] Arvind and K. Ekanadham, "Future Scientific Programming on Parallel Machines", Computation Structures Group Memo 272, Massachusetts Institute of Technology, March 1987.

[Bou72] W. Bourke, "An Efficient, One-Level, Primitive-Equation Spectral Model", *Monthly Weather Review*, Vol. 100, No. 9, pp. 683-689, September 1972.

[Bri88] F. Briggs, "Synchronization, Coherence and Event Ordering in Multiprocessors", *IEEE Computer Journal*, pp. 9-21, February 1988.

[Cann89] D. Cann, "Compilation Techniques for High Performance Applicative Computation", Technical Report CS-89-108, Colorado State University, May 1989.

[Chang90] P. Chang, "Implementation of a Numerical Weather Prediction Model in SISAL", Technical Report 31-017, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology, June 1990.

[CA88] K. Chandy and J. Austin, "Architecture Independent Programming", *Proceedings of the Third International Conference on Supercomputers*, Vol. 3, pp. 345-351, International Supercomputing Institute Inc., 1988.

[CLOS87] D. Cann, C. Lee, R. Oldehoeft and S. Skedzielewski, "SISAL Multiprocessing Support", Technical Report, Lawrence Livermore National Laboratory, Livermore, CA, 1987.

[CO89] D. Cann and R. Oldehoft, "High Performance Parallel Applicative Computing", Technical Report CS-89-104, CSU, Feb 1989.

[CO88] D. Cann and R. Oldehoeft, "Porting Multiprocessor SISAL Software", Technical Report CS-88-104, Computer Science Department, CSU, Fort Collins, CO, February 1988.

[COBGF] D. Cann, R. Oldehoft, A.P.W.Bohm, D. Grit and J. Feo, "SISAL Reference Manual, Language Version 2.0", Colorado State University and Lawrence Livermore National Laboratory, 1990.

[CV90] Verbal communication with D. Cann, June 1990.

[CGA89] P. Chang and G. Egan, "A Parallel Implementation of a Barotropic Numerical Weather Prediction Model in SISAL", TR 118 088 R, Joint RMIT/CSIRO Parallel Systems Architecture Project, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, August 1989

[CG89] P. Chang and G. Egan, "Performance Evaluation of a Parallel Implementation of Spectral Barotropic Numerical Weather Prediction Model in the Functional Dataflow Language SISAL", (TR 118 091 R, Joint RMIT/CSIRO Parallel Systems Architecture Project, Department of Communication and Electrical Engineering, RMIT, October 1989), *Proceedings of Second Australian Supercomputer Conference*, University of Wollongong, December 1989.

[CG90] P. Chang and G. Egan, "A Parallel Implementation of a Barotopic Spectral Numerical Weather Prediction Model in the Functional Language SISAL", (TR 118 088 R, Joint RMIT/CSIRO Parallel Systems Architecture Project, Department of Communication and Electrical Engineering, RMIT, 1989), *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP*, Seattle, Washington, pp. 109-117, SIGPLAN Notices, Vol. 25, No. 3, March 1990.

[CGMr90] P. Chang and G. Egan, "Analysis of a Parallel Implementation of a Numerical Weather Model in the Functional Language SISAL", Technical Report 31-012, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology, March 1990. (To be published in the *Proceedings of Parallel '90 - First International Conference on Parallel Processing for Computational Mechanics*, September 1990, England, Computational Mechanics Publications)

[CGMy90] P. Chang and G. Egan, "Analysis of Dynamic Memory Management Schemes for OSC", Technical Report 31-015, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology, May 1990.

[CGA90] P. Chang and G. Egan, "Proposals for SISAL and OSC", Technical Report 31-014, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology, April 1990.

[Egan90] G. Egan, "Some Shallow Experiences: The Shallow Water Numerical weather Prediction Program", Technical Report 31-004, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology, 1990.

[ECC] Encore Computer Corporation, *Multimax Technical Summary*.

[ECCE] Encore Computer Corporation, *Encore Parallel Fortran Manual*, December 1988.

[EZL89] D. Eager, J. Zahorjan and E. Lazowska, "Speedup Versus Efficiency in Parallel Systems", *IEEE Transaction on Computers*, pp. 408-423, Vol. 38, No. 3, March 1989.

[EWB90] G. Egan, N. Webb and A.P.W. Bohm, "Some Architectural Features of the CSIRAC II Dataflow Computer", Technical Report 31-007, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology, 1990.

[Feo87] J. Feo, "The Livermore Loops in SISAL", UCID-21159, Lawrence Livermore National Laboratory, August 1987.

[Gar90] H. Garsden, "Data Allocation and Deallocation in OSC", Department of Computer Science, University of Adelaide, May 1990.

[GW] R. Gonzalez and P. Wintz, *Digital Image Processing*, p. 100, Second Edition, Addison-Wesley, 1987.

[GB88] J. Gurd and A.P.W. Bohm, "Implicit Parallel Processing: SISAL on the Manchester Dataflow Computer", *Parallel Systems and Computation*, pp. 179-204, Elservier Science Publishers B.V. (North-Holland), 1988.

[Has90] *The Haskell Report*, Version 1.0, April 1990.

[HP89] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages", *ACM Computing Surveys*, Vol. 21, No. 3, pp. 359-411, September 1989.

[LSF88] C. Lee, S Skedzielewski and J. Feo, "On the Implementation of Applicative Languages on Shared-Memory, MIMD Multiprocessors", pp. 188-197, *Proceedings of the First ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New Haven, July 1988.

[Lor79] Lorenz, "On the Prevalence of Aperiodicity in Simple Systems", pp. 53-75, *Global Analysis*, eds. Mgrmela and Marsden, New York, Springer-Verlag, 1979.

[McB88] O. McBryan, "New Architectures: Performance Highlights and New Algorithms", *Journal of Parallel Computing*, No. 7, pp. 477-499, North-Holland, 1988.

[Meyers] R. Meyers (Editor), *Encyclopedia of Physical Science and Technology*, pp. 444, 520-521, Academic Press, Inc., 1987.

[MS85] J. McGraw, S. Skedzielewski, S. Allan, R Oldehoeft, J. Glauert, C. Kirkham, Bill Noyce and R Thomas, "SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2", Memo 146, Lawrence Livermore National Laboratory, March 1985.

[Nik88] R. Nikhil, "ID (Version 88.0) Reference Manual", Computation Structures Group Memo 284, Massachusetts Institute of Technology, March 1988.

[NPA86] R. Nikhil, K. Pingali and Arvind, "ID Nouveau", Computation Structures Group Memo 265, Massachusetts Institute of Technology, July 1986.

[Parker] S. Parker (Editor), *McGraw-Hill Dictionary of Science and Engineering*, p. 898, McGraw-Hill Book Company, 1984.

[Ran87] J. Ranelletti, "Graph Transformation Algorithms for Array Memory Optimisation in Applicative Languages", PhD thesis, University of California at Davis, Computer Science Department, Davis, California, 1987.

[Sharp] J. Sharp, *Data Flow Computing*, pp. 1-17, Elis Harwood Series, Computers and Their Applications.

[Sim77] I. Simmonds, "Introduction to Spectral Methods", Lecture Notes, Department of Meteorology, University of Melbourne, 1977.

[Sim78] I. Simmonds, "The Spectral Modelling Project", Report No.1, Department of Meteorology, University of Melbourne, August 1978.

[TB88] J. Tuccillo and F. Balint, "Futures Needs for Supercomputers in Weather Forecasting", *Proceedings of the Third International Conference on Supercomputers*, Vol. 2, pp. 20-24, International Supercomputing Institute Inc., 1988.

[Turn86] D. Turner, "An Overview of Miranda", SIGPLAN Notices, *Programming Languages*, Vol. 21, No. 12, pp. 158-166, December 1986.

[WS86] M. Welcome, S. Skedzielewski, R. Yates, J. Ranelletti, "IF2: An Applicative Language Intermediate Form with Explicit Memory Management", Manual M195, University of California, Lawrence Livermore National Laboratory, November 1986.

[Whi88] P. Whiting, "IDA: A Dataflow Programming Language", Technical Report 112 075 R, Joint RMIT/CSIRO Parallel Systems Architecture Project, October 1988.

[Yat88] R. Yates, "DI Tutorial", Lawrence Livermore National Laboratory, March 1988.