The University of York Department of Computer Science

Submitted in part fulfilment for the degree of MEng.

A Method for Porting Parallel Algorithms for the DAP to Handel-C

Georgina Porter

Version 1.0, 31st May 2007

Supervisor: Susan Stepney

Number of words = 27,558, as counted by detex *filename* | wc -w.
This includes the body of the report only.

Abstract

The Distributed Array Processor (DAP) was a SIMD developed in the 1970s which comprised 64x64 processing elements, each with its own storage and linked via vertical and horizontal highways. It was a highly parallel processor, programmed using its own parallel language; DAP-Fortran. Much work was done in the twenty years of its use to find applications for the DAP and many highly parallel algorithms were developed for it.

Field Programmable Gate Arrays are modern day arrays of logic blocks linked by programmable interconnections. The configurations of these logic blocks are designed using the programming language Handel-C, which is compiled to the hardware.

The aim of this project was to investigate a means of translating the parallel algorithms for the DAP to Handel-C in order that knowledge and work done based on them may be preserved. Investigations showed that the two programming languages are too dissimilar in architecture for a transformation to be useful for the purposes given. However, it was found that algorithms written in DAP-Fortran were always inherently parallel and any translation of algorithms from DAP-Fortran to Handel-C were also highly parallel, despite their differences. DAP-Fortran was therefore discovered to be an effective tool for specifying parallel algorithms.

Contents

I Preliminaries **7**

1 Introduction **9**

2 Context **11**
2.1 The Evolution of Parallelism 11
2.1.1 SIMD Machines 12
2.1.2 The Distributed Array Processor (DAP) 13
2.1.3 DAP-Fortran 18
2.1.4 The rise and fall of the DAP 21
2.2 Modern Architectures and Parallelism 23
2.2.1 FPGAs 24
2.2.2 Handel-C 26
2.2.3 Context Summary 28

II Investigation **31**

3 Design **33**
3.1 Aims of the Project 33
3.2 Out of Scope 35
3.3 Methodology Employed 35
3.4 Method 36

4 Investigation **39**
4.1 Example 1: Matrix Comparison and Manipulation 39
4.1.1 Original DAP-Fortran Code 39
4.1.2 Version 1: Direct Conversion to Handel-C 40
4.1.3 Version 2: Revised Conversion to Handel-C 42
4.1.4 Version 3: Revised Conversion to Handel-C 43
4.1.5 Conclusion 45
4.2 Example 2: Matrix Multiplication 46

4.2.1 Original DAP-Fortran Code 46
4.2.2 Version 1: Direct Conversion to Handel-C 48
4.2.3 Version 2: Revised Conversion to Handel-C 50
4.2.4 Conclusion 51
4.3 Example 3: The Contour Problem 53
4.3.1 Original DAP-Fortran Code 53
4.3.2 Version 1: Direct Conversion to Handel-C 55
4.3.3 Version 2: Revised Conversion to Handel-C 60
4.3.4 Conclusion 61

III Conclusion **63**

5 Conclusion **65**

6 Evaluation **67**
6.1 Research 67
6.2 Conclusions 68
6.3 Method for Porting Algorithms 68
6.4 Project Review 69
6.5 Future Work 70

IV Appendices **77**

A DAP-Fortran Code: The Contour Problem **79**

B Handel-C Code: The Contour Problem **81**

C Process Steps for Porting DAP-Fortran to Handel-C **85**

D Mapping **87**

Part I

Preliminaries

1 Introduction

Throughout the lifetime of the field of computing, development has been both rapid and varied, with specialised branches and their associated new technologies emerging. The direction and longevity of these developments has inevitably been driven not only by academic curiosity, but by factors such as industrial need. Such factors have meant that money and interest have often been diverted away from less commercially successful or relevant areas, and once promising ideas have fallen by the wayside. Modern manufacturing processes, advanced algorithms and the changing needs of users has meant that many of these under and unexplored routes have been superseded. However, as current paths are exhausted, there may be merit in revisiting these abandoned areas to aid current and future work.

There are vast archives of research available on now defunct hardware, algorithms and languages. Likewise, much data has been lost as projects are abandoned, and printed and proprietary literature becomes unavailable. Whilst the implementations of these theories may now be redundant, many of the ideas behind them and subsequent research around them may still be applicable to modern day situations.

With this in mind, we look at the current trend towards the use of Field Programmable Devices (FPDs) in parallel processing, and investigate the relevance of work done in the late 1970s and early 1980s around the development and use of specialised parallel processing devices. Specifically, we focus on one of the most popular FPDs in use today, the Field Programmable Gate Array (FPGA) and the now little-known Single Instruction-stream, Multiple Data-stream (SIMD) machine, the Distributed Array Processor (DAP).

During the active life of the DAP, highly parallel algorithms were developed for use in fields such as physics, meteorology, mathematics and digital imaging. These algorithms demonstrated the benefits of ”thinking parallel” when approaching problems which required regular

operations over grids of data. The advantages to these approaches are still relevant today. A modernisation of these algorithms by porting them to a contemporary platform, such as the FPGA could therefore reinvigorate previous research into the field and the knowledge gained in the original research would not be lost.

In Section 2, we discuss the advancements of architectures over time from the SIMD of the 1970s to modern day FPDs, and the languages used by the two architectures to be investigated, the DAP and the FPGA. In Section 3 we discuss in more detail the aims of the project, and how these were to be met. We then investigate the steps required to port algorithms for the DAP to Handel-C in Section 4, discuss our conclusions in Section 5 and finally evaluate the processes found and project as a whole in Section 6.

2 Context

2.1 The Evolution of Parallelism

> "Historically supercomputers, or more generally any current state-of-the-art computers, have had their primary design criterion the provision of high-speed processing"
>
> Gostick (1979)

From the 1940s to the 1970s, the progress of computer systems could be marked by the changes in their component parts. In the 1940s it was relays and vacuum tubes, in the 1950s it was a move to discrete diodes and transistors, and in the 1960s and 1970s it was the scale of integrated circuits and devices (Hwang and Briggs, 1985). As the technologies improved, the speed of operations soared. "In 1944 an addition operation took 333 milliseconds on the MARK 1, in 1951 it took 282 microseconds on the UNIVAC 1, and in 1964 it took 300 nanoseconds on the CDC 6600" (Perrott and Zarea-Aliabadi, 1986).

However, by the 1970s it was becoming apparent that this rate of improvement to the speed of components was unlikely to continue. At the same time the mainstream usage of computers was increasing, with huge amounts of data being generated for analysis. The need for a move away from the standard von Neumann model of processing was recognised and the use of parallelism was identified as a possible solution.

One approach was to use parallelism in uniprocessor systems. Here parallelism was achieved on a small scale through the use of multiple specialised functional units within each Arithmetic and Logic Unit (ALU), pipelining within the CPU, multiprogramming and timesharing. A second, and more adventurous, approach was the promotion of parallel processing.

Parallel processing machines of the 1970s could be divided into three categories: pipeline computers, multiprocessor systems and array processors, each with a different philosophy of parallelism. Pipe-line

computers focussed on *temporal parallelism*, through the overlapping of computations such as fetch operations. Multiprocessor systems achieved *a-synchronous parallelism* through the use of interactive processors sharing common resources. Finally, processor arrays enabled *spatial parallelism* by using ALUs called Processing Elements (PEs), "that can operate in parallel in a lock-step fashion" (Hwang and Briggs, 1985).

While aspects of all three have become commonplace in more modern computer architectures, it was the investigation and exploitation of array processor machines that became a dominant theme within industry. The 1970s and 1980s, one style of processor array introduced was the Single Instruction-stream, Multiple Data-stream (SIMD) architecture.

2.1.1 SIMD Machines

The family of SIMD machines are so named due to the model of information flow used. Instructions are taken from memory and streamed to a central Master Control Unit (MCU) which broadcasts them to each Processing Element (PE) within the array. Each PE then executes these commands in parallel on the stream of data retrieved from their separate memory modules. This model of processing can be seen in Figure 2.1.

It was the simplicity of this general architecture which contributed to its success, with the view of rival Multiple Input-streams, Multiple Data-Streams (MIMD) architectures being that their "programming and control seem impossibly complex and inefficient" (Reddaway, 1979).

Three of these popular SIMD machines were the STARAN (Batcher,

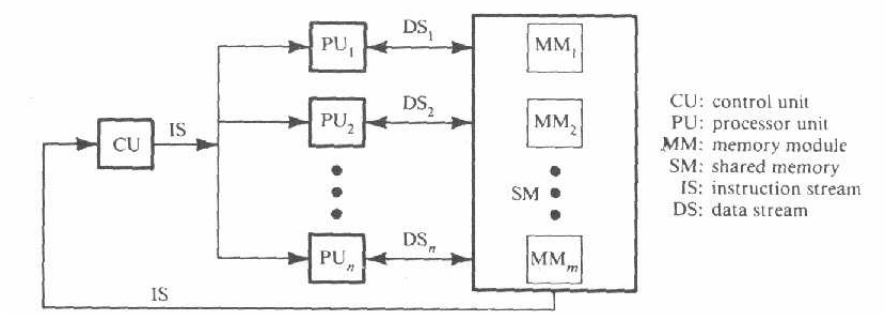

Figure 2.1: Flynn's Classification of a SIMD Computer taken from Hwang and Briggs (1985)

1979), the ILLIAC IV (Feierbach and Stevenson, 1979) and the Distributed Array Processor (DAP) (Reddaway, 1973). While the STARAN was one of the first SIMDs developed and served to create curiosity within the computing community, it was the development of the ILLIAC IV that really attracted international interest.

The ILLIAC IV was conceived in the mid-1960s at the University of Illinois with funding from the USA's Defense Advanced Research Projects Agency (DARPA). It was developed in collaboration with Texas Instruments, the Burroughs Corporation and Fairchild Camera and Instruments. The ILLIAC IV was delivered to the National Aeronautics and Space Administration (NASA) in 1975 and shortly thereafter, the Institute for Advanced Computation (IAC) "was created with an initial charter to bring the ILLIAC IV to operational status" (Feierbach and Stevenson, 1979). "The ILLIAC IV project was an ambitious, multidisciplinary approach to implementing a large-scale parallel processor" and attracted a huge amount of government funding and attention (Feierbach and Stevenson, 1979).

The ILLIAC IV's architecture was based on one central MCU working with 64 PEs capable of performing 64-bit or 32-bit floating point arithmetic. In contrast to the ILLIAC's small number of complex processors, the British made commercial SIMD, the DAP, comprised a large number of very simple processors.

2.1.2 The Distributed Array Processor (DAP)

> "cost-effectiveness rather than speed at any price."
>
> Reddaway (1979)

The Distributed Array Processor was one of the first commercially available SIMDs and was conceived in 1972 by a research team at International Computers Limited (ICL). "The origin of the DAP project lay in finding a match between large fairly regular number-crunching applications, such as meteorology, and a technology that could produce regular hardware very cheaply." (Reddaway, 1979).

The ”regular number-crunching applications” such as meteorology were ”a class of computations characterised by having largely regular operations on each element of a computing grid, but with the need for some local control, and various types of communication between the data elements" (Parkinson, 1990). The DAP was thus developed to meet this challenge and comprised an MCU which controlled a grid of PEs,

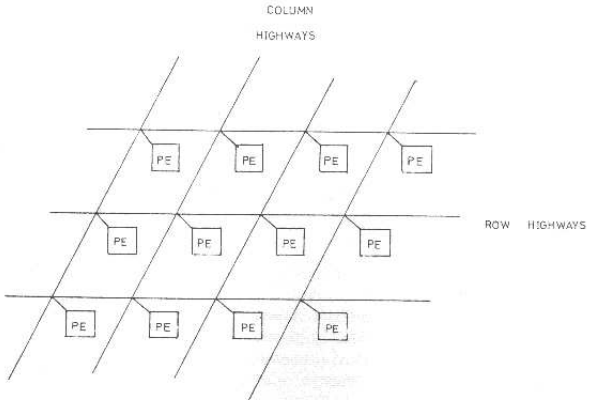

Figure 2.2: Processing Element Interconnections (Highways) taken from Parkinson (1983)

arranged so that each PE was able to communicate with each of the others.

A working prototype (32x32 PEs) was available from 1976, with delivery of the first generation DAP (64x64) made to Queen Mary College, University of London (QMC) in 1980. This first generation DAP was referred to in consequent papers by various names including: the Mainframe DAP; the QMC DAP; the MSI DAP; and the ICL DAP.

The DAP's PEs interacted via a network of column and row highways (see Figure 2.2). At the boundaries of the arrays, these highways wrapped around forming cyclic connections. Through this method of interconnection, each PE was able to take its input from either its own input from data store or from its north, east, south or west neighbour via its top multiplexor, with its output sent to its neighbours or store via the bottom multiplexor. This can be seen in Figure 2.3.

The structure of the PE shown in Figure 2.3 is of the first generation DAP, a simplified version of the original prototype described in Reddaway (1973). Because of this simplification and several other minor alterations, many of the papers between these dates describe a slightly different structure.

"Any SIMD machine with many PEs requires high-bandwidth data storage so that processing power of the PEs is not wasted waiting for the data." (Batcher, 1979). This problem was faced by every SIMD designer, and was met by the DAP by providing each PE with its own store of 4096 bits. The combined storage of the 64x64 DAP was

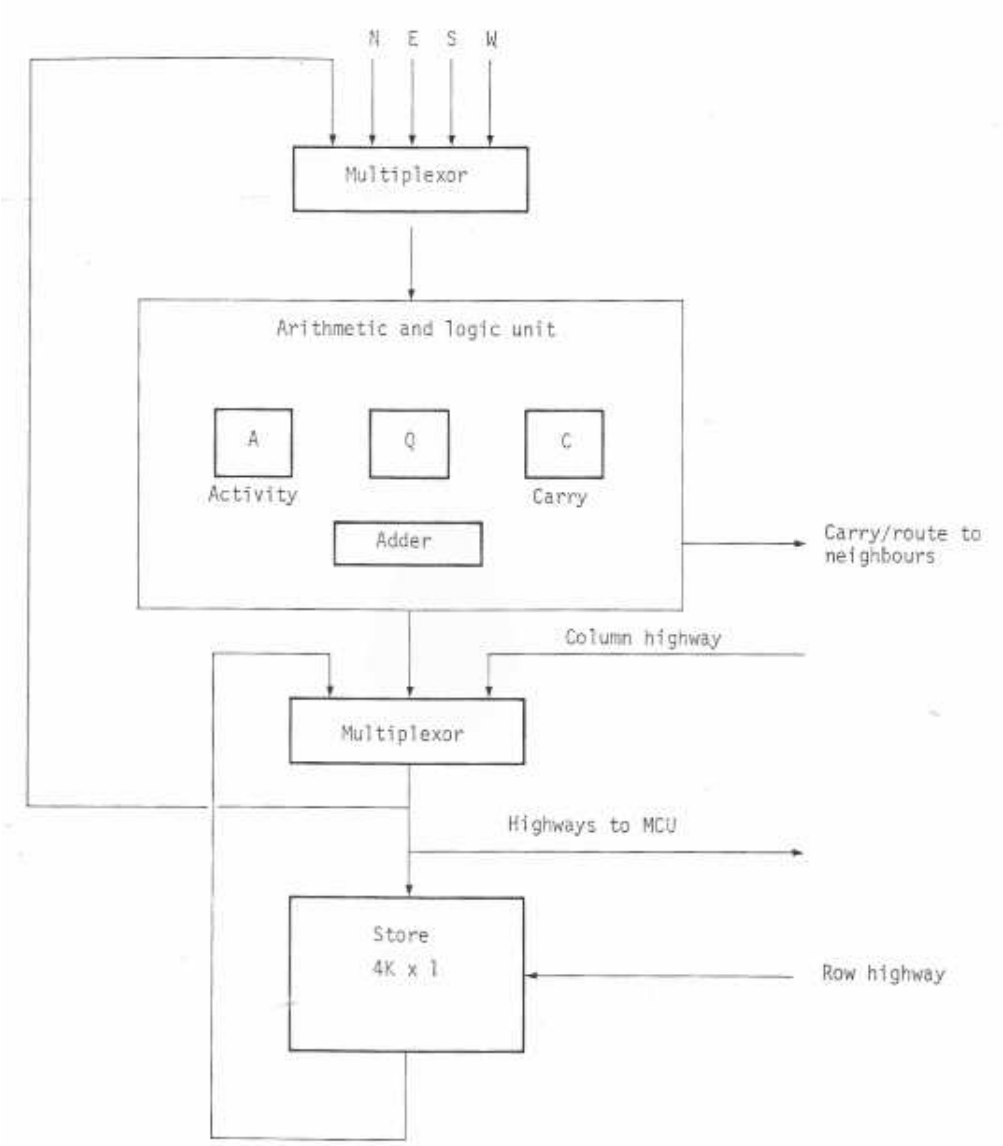

Figure 2.3: Processing Element taken from Reddaway (1979)

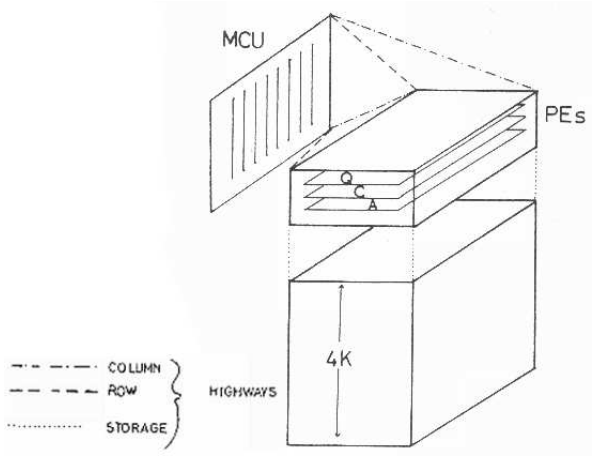

Figure 2.4: DAP Overview taken from Parkinson (1983)

therefore 2 Mbytes, but with no addressing limitations it was anticipated that once improved storage chips were available this capacity could be increased. In 1983 there were plans to upgrade the QMC DAP to 8 Mbytes (Parkinson, 1983), but there is no evidence that this was done. The grid of storage blocks could be used to store data in two modes: Horizontal Mode (originally called Array Mode) and Vertical Mode (originally called Main Store Mode). This allowed arrays of data to be stored with one element at each PE or with all elements stored in one. With the original requirement to apply regular operations across a grid, Horizontal mode allowed for operations to be applied to each element of a grid simultaneously.

Figure 2.4 shows an overview of the structure of the DAP and its connections to the MCU. The Q register was used to mark each PE as active for each operation sent by the MCU. If Q were not set for a PE then the associated operation would not be performed on that element.

The DAP was a peripheral device, connected to a host machine—usually an ICL 2900. "The totality of PE stores form a standard store module of a conventional computer. Hence there is no need for separate transfer of data between host and DAP" (Flanders et al., 1977). This shared section of storage could thus be written to and read from by both the DAP and its host, with its host free to use any programming language it pleased, so long as the data was stored in the correct format.

The simplicity of the DAP's structure allowed for great flexibility.

"Very few decisions are built into the hardware", so "the details of a function are left to the software. This means that the hardware does not dictate, for example, the functions available, the precision or the rounding algorithm" (Reddaway, 1979). This was seen as a great advantage over the more complicated MIMD architectures and those SIMDs with complex PEs, such as the ILLIAC IV. The bit-organised PEs also allowed the exploitation of parallelism in operations such as table look-up, data scanning, and image and symbol processing.

Papers of the time discuss the DAP's affordability and flexibility, rather than focussing on its speed. When its speed is discussed it is seen to perform favourably over architectures such as the ILLIAC IV, but the DAP's unique nature meant that little comparative analysis was done and these were carried out using the regular algorithms the DAP was designed to handle. "The performance of a program is measured by counting the number of instructions executed by the DAP" and there is "no exact way to measure exactly the CPU time used by the DAP" (Kindervater and Trienekens, 1988). Instead analysis is often done by comparing the DAP to itself with different sizes of data supplied and is used to show the efficiency of using greater parallelism within algorithms (Bhagavathi et al., 1992).

In general, papers on the DAP were written within and for a small community and are often lacking in context. Thus, timings are given which would give an indication to a reader of the day that an operation is exceedingly fast or exceedingly slow, but to a reader with no prior knowledge of comparable machines these timings mean little.

"As a research project, hardware, applications analysis and software were advanced together, resulting in a well integrated system." (Reddaway, 1979). These advances included the development of a macro assembly language, APAL, which was used to program the prototype DAP, and eventually DAP-Fortran which was used from the first generation onwards. After more than a year of use at QMC it was reported that "few problems benefit from using the low level assembler language (APAL) and so assembly level programming is now strongly discouraged" (Parkinson, 1982), and so DAP-Fortran became the primary programming language for the DAP.

2.1.3 DAP-Fortran

> ”DAP-Fortran provides a good match between problems and hardware and encourages users to rethink algorithms to good effect.”
>
> Flanders et al. (1977)

When facing the challenge of providing a high level language for the DAP, the designers had to decide upon the approach they wished to take. One way would be to take an existing language and provide a compiler which detected the potential for parallelism within the code and converted it. An attempt had previously been made by the team behind the ILLIAC IV to ”detect parallelism in ordinary Fortran programs and generate code for the machine exploiting this parallelism.” (Feierbach and Stevenson, 1979). This ambitious attempt had proved a costly endeavour and the IVTRAN compiler produced was ”neither fully developed nor adequately maintained”.

A second approach, and that eventually followed by the ILLIAC IV team and adopted by its users, was ”developing programming languages which explicitly express the parallelism in the syntax.” (Feierbach and Stevenson, 1979). The DAP team also chose to follow this second option. This was a course of action later vindicated by Perrott and Zarea-Aliabadi (1986) who concluded this to be the correct course of action.

While this course of action may seem natural from the perspective of a modern computer scientist, moving computer users from a ”sequential” way of thinking to a ”parallel” way of thinking was a big issue at the time (Flanders, 1979)(Hunt, 1979). The ease of transition from one to the other was crucial to the future success of a parallel programming language. Familiarity of structure and syntax were seen to be key by the DAP designers and so DAP-Fortran was developed based on the syntax of FORTRAN 2900, an ICL flavour of the popular standard FORTRAN. This philosophy of following a familiar syntax to aid user uptake is seen to this day in programming languages such as Handel-C (Bowen, 1998).

DAP-Fortran was initially designed at the time of the prototype DAP and developed with the DAP until it release in 1981. It has since evolved into what we know as Fortran-Plus. The language ”was designed to provide a good match between the capabilities of the hardware and the requirements of applications" (Parkinson, 1990) and so meets the requirements of the DAP project that had been met by the hardware

itself. "DAP-Fortran provides a good match between problems and hardware and encourages users to rethink algorithms to good effect" (Flanders et al., 1977), thus providing users with a smooth transition to parallel thought.

DAP-Fortran is based on the parallel manipulation of arrays and matrices. "Any SIMD machine suffers some loss of efficiency when the number of elements in a data array is not a multiple of the number of PEs" (Batcher, 1979) and so DAP-Fortran takes the approach of limiting all of its arrays to be of 64 elements and its matrices to be of 64x64 elements. Thus, where in FORTRAN an integer array would be declared as INTEGER A(64), DAP-Fortran simply omits the length and declares it as INTEGER A( ). Similarly, matrices are declared as INTEGER A(,) as its dimensions are fixed to be 64x64. The anticipated expansion of the size of the DAP arrays would be handled such that a 128x128 array would be handled as four 64x64 blocks.

Part of DAP-Fortran's inherent parallelism can be seen in the way it applies functions to data structures. The code

```
A = B + 1
```

can have multiple meanings in DAP-Fortran, dependent on how A and B are declared. As simple scalars, the value of A would be assigned the value of the sum B + 1. As vectors, each element in A would be assigned the sum of the corresponding element of B and 1. Likewise if A and B were both matrices it would perform elementwise addition. This simple method allows users to exploit the parallel abilities of the DAP hardware without having to think of the implementation details. In reality, this simple assignment would be done by storing each matrix element by element within the PE array. Each PE then simply carries out a sum on the element from B and stores it to A.

The use of the activity register, Q, of each PE is seen in the use of indexing by masks within DAP-Fortran. If an operation should be carried out only on those elements of a matrix meeting some criteria, then the following can be used

```
A(B.LT.8) = 1
```

Here, elements of A where the corresponding elements of B are less than 8 are assigned the value 1. All other elements are left unchanged. The hardware achieves this by carrying out a less than operation in each

PE on B and where the result is true, the Q register is set. The MCU then sends the assignment operation to all PEs, but only those where Q is set will carry out the operation. The programmer using DAP-Fortran is, however, shielded from these details.

Another feature of the DAP exploited by DAP-Fortran is the use of the PE highways and close ties between neighbouring PEs. These allow data stored or produced by one PE to be used by another. In terms of the data structures, this allows a simple method of using and comparing elements of a matrix or array.

```
A(B.LT.B(+,)) = 1
```

In the above code, a mask is created based on B.LT.B(+,). This less than comparison is done between all elements of B and that element's northern neighbour. Similarly, the comparison could have been made with the element's south, east or west neighbour using B(−,), B(,+) or B(,−) respectively. As with other shift functions, these can be implemented as a linear or cyclic shift. The DAP can be set to four geometries whereby values in the east-west plane and south-west plane are set independently. They are set so as to either shift in values cyclicly (CYCLIC mode) using the hard wired wrap around highways, or shift in the value 0 (PLANE mode) (ICL, 1978). This system has been significantly simplified from earlier versions where 32 different geometry settings were proposed (Reddaway, 1973).

A more detailed and thorough description of DAP-Fortran and its features is available in "DAP: Introduction to FORTRAN Programming", ICL (1978), with detailed algorithm examples and analysis available in Clint et al. (1985) and Stewart et al. (1991).

DAP-Fortran subroutines could be used within existing FORTRAN programs and so areas could be identified where parallelism should be introduced. Calls could then be made to these new subroutines, extending the lifetime of existing code and improving performance without huge rewrites being required. Efforts were made at QMC to produce a Subroutine Library (Liddell and Bowgen, 1982), but the results of this attempt and the library itself are no longer available.

At the time of DAP-Fortran's development there was an investigation undertaken on behalf of the Department of Energy in the USA to determine what its users wanted from a programming language (Wetherell, 1980). The aim of the study was to design and implement a common language across its laboratories. This study determined that the best way

forward was an extension to Fortran 77 which included array processing and powerful data structure manipulation abilities. This way forward had at this time already been taken by the writers of DAP-Fortran.

A further paper based on Wetherell's work goes on to discuss a survey done of users of the ILLIAC IV and concludes that users also wish to be able to express parallelism and to be hidden from the details of the hardware on which it is implemented (Perrott and Stevenson, 1981). Again, this philosophy can be seen implemented in the DAP-Fortran, where it "allows users to write software that takes full advantage of the ICL DAP parallel processing features without incorporating its architectural details within program statements" (Perrott and Zarea-Aliabadi, 1986).

2.1.4 The rise and fall of the DAP

> "Some of the factors affecting the rate of expansion are:
>
> - The extent to which users and purchasers grasp and exploit the improved problem solving opportunities offered by DAP
> - The rate at which DAP-like hardware and software products are developed and marketed, and the pricing policies followed"
>
> Reddaway (1979)

The DAP started out promisingly, with the publication of many papers praising the DAP; its attempt to move people towards parallelism and the DAP-Fortran language, even before the first generation of machines became available.

Gostick (1979) praises the DAP in terms of the range of diagnostic features that were available. In itself, this may seem mundane, but its importance is that other supercomputers that were available were lacking in this simple element. The DAP was being recognised for its superior usability. The paper states that most supercomputers do not offer such things due to "market pressures". The DAP, one of the first commercially available SIMDs, designed to be a cheap alternative to other machines, was seen to be providing what users needed and had been asking for.

Around the same time, there were many papers being published urging people to "think parallel" and proclaiming the genuine advantages that could be experienced by those willing to take the plunge. In these

papers the DAP was used as an example of how to achieve these results at minimal time and monetary expense (Hunt, 1979).

Hand in hand with that discussion was the more in depth study of the difficulties experienced by those attempting to create parallel algorithms (Flanders et al., 1977). The problems encountered were mostly due to the approaches taken by such people. In many cases, algorithms would be envisaged as sequential and converted step by step. DAP-Fortran was introduced as an effective way of bridging the gap, with its inherently parallel structure.

This enthusiasm continued after the DAP's release, with its expansion within the target domain of meteorology to its use in satellite climatology (Wilkinson and Burge, 1982). Expansion into other application domains were also frequently recommended by users, such as in computational physics at QMC (Parkinson, 1983), image processing (Arnot et al., 1982) and seismic migration (Krzeczkowski et al., 1982). The initial results for these industrial domains were extremely encouraging. The DAP and array processors were seen as a new breed, prompting investigations into new methods of comparing parallel computers, in addition to the existing Megaflops measurements (Hockney, 1982).

The academic community continued its praise and the DAP was applied to problems such as solving linear equations (Webb et al., 1982). Much attention was also given to the conversion of known algorithms to parallel versions, such as the WZ Algorithm (Hellier, 1982), non-hierarchic document clustering (Rasmussen and Willett, 1987), and combinatorial problems (Kindervater and Trienekens, 1988).

However, by the later 1980s, the enthusiasm was dwindling and had become limited to the academic community. Even though availability of the machines was an issue (Wunderlich, 1985), the DAP was still being discovered and loved by new users (Toral, 1988). However, ten years after it had been held as an ideal for parallelism, the DAP was still primarily being used as an example of why a move to parallelism should be made. Industry had by this time invested too much in sequential code, training and serial machinery.

The range of the appeal for the DAP was also limited geographically. While the UK and later mainland Europe found interest in the processor, no evidence could be found of any interest farther afield. This could be, in part, explained by the prominence of well funded and "well-publicised" (Hwang and Briggs, 1985) American projects such as the ILLIAC IV. Also at the time when ICL were hoping that the DAP would go into larger production, NASA bought a Goodyear Massively Parallel

Processor (MPP) (Wunderlich, 1985).

In 1987, two of the DAPs greatest advocates, QMC and Edinburgh University, necessarily decommissioned and replaced the DAP's ICL 2900 host machines. It was anticipated that they would replace the DAPs with the new AMT DAPs, but it does not seem that they were able to secure funding (Wallace, 1989). Around this time, the DAP patent changed hands and became a product of Active Memory Technology Ltd (AMT) who reworked it to the AMT DAP 500 series. This new generation had a grid of 32x32 more complex PEs, in direct contradiction to the ideals of the original designers.

Into the 1990s, far fewer people were using DAPs, but those that still were, spoke highly of them (Boyle et al., 1992), (Kontoghiorghes, 1999) and were using them in far more complex applications (Fitzpatrick et al., 1995), (Fitzpatrick et al., 1996), (Fitzpatrick et al., 1997), although these were still confined to academia. This trend toward academic use was somewhat ironic. One of DAPs original creators had put forward the argument in favour of SIMDs to be that a "MIMD may provide interesting work for computer scientists, but not for cost-effective computing, especially for a wide range of large jobs" (Reddaway, 1979).

Today, DAPs are no longer available. Their production ceased when AMT was bought out by Fujitsu Ltd in 2003. They are however still in use, with a new course on parallel programming on the DAP now being offered at the Parallel Computer Centre at Queen's University, Belfast (PCC, 2007).

2.2 Modern Architectures and Parallelism

Whilst the interests of some moved towards the design of new processor architectures, the improvement of component parts continued. Very Large Scale Integration (VLSI) technology allowed a full custom approach to high density chip design, where some or all parts of a chip could be specified to meet individual requirements. Application-Specific Integrated Circuit (ASIC) chips could be designed by users and sent off to be manufactured. However, as the complexity of these circuits increased, as did the non-repeatable cost and the time taken to manufacture. This meant that such chips were only affordable in large scale production (Brown et al., 1992).

The need for chips which could be used for prototyping and low volume production was identified, and in the early 1970s Philips

introduced the Programmable Logic Array (PLA), a Programmable Logic Device (PLD). The PLA had two layers of programmable logic, one of connected AND gates and one of OR gates. These versatile structures provided the user with the ability to fuse connections between the logic gates according to their own chip designs (Brown and Rose, 1996). Unfortunately, the dual layer proved difficult to manufacture and the connections between them caused significant propagation delays. Because of this, the plane of OR gates was fixed, with only the AND gate plane remaining programmable, and so Programmable Array Logic (PAL) was developed.

PALs and other PLDs provided gate level programmability, but with the increasing complexity and density of the Field Programmable Device (FPD) chips a higher level approach was needed. The Field Programmable Gate Array (FPGA) was developed to "resolve the conflict between standard parts and ASICs and provide the engineer with intimate and immediate access to all resources for system building" (Brown et al., 1995).

2.2.1 FPGAs

The Field Programmable Gate Array (FPGA) was developed in 1985 and "comprised an array of uncommitted circuit elements, called *logic blocks*, and interconnect resources" (Brown and Rose, 1996) programmable by the designer. An overview of an FPGA architecture can be seen in Figure 2.5.

Logic blocks are arranged regularly about the chip with interconnections programmable via the use of fuses, antifuses and now more commonly EEPROM and SRAM transistors (Parnell and Mehta, 2002). These allow the inputs and outputs of each block to be routed between blocks and the chips own IO. The use of transistors enabled the devices to not only be programmable, but reprogrammable in the field. This in collaboration with the configurable logic blocks meant that FPGAs were highly versatile, allowing FPGAs to become an alternative to custom Integrated Circuits.

It is easy to see that this high degree of programmability allows the FPGA to be used for parallel processing. Given an appropriate device description language, different input data can be routed concurrently through independent logic blocks applying the same or different operations to the data. This is demonstrated in Figure 2.6.

There are many hardware description languages available for the

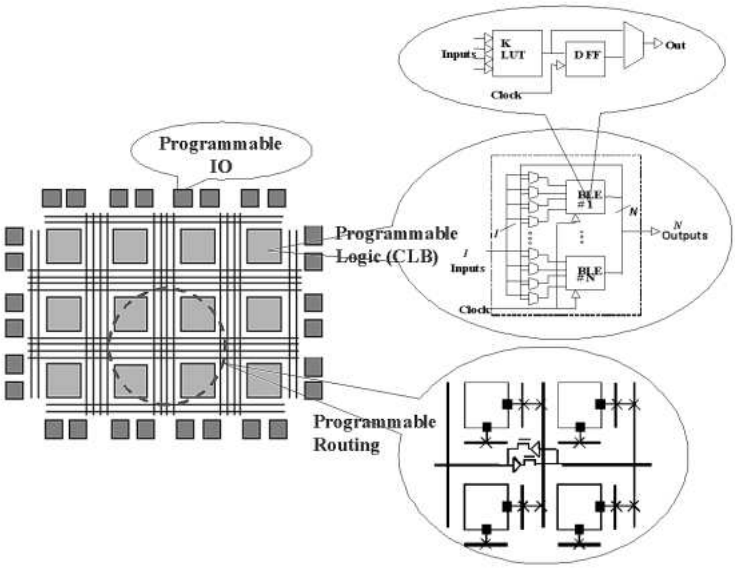

Figure 2.5: FPGA Overview taken from Audsley (2006)

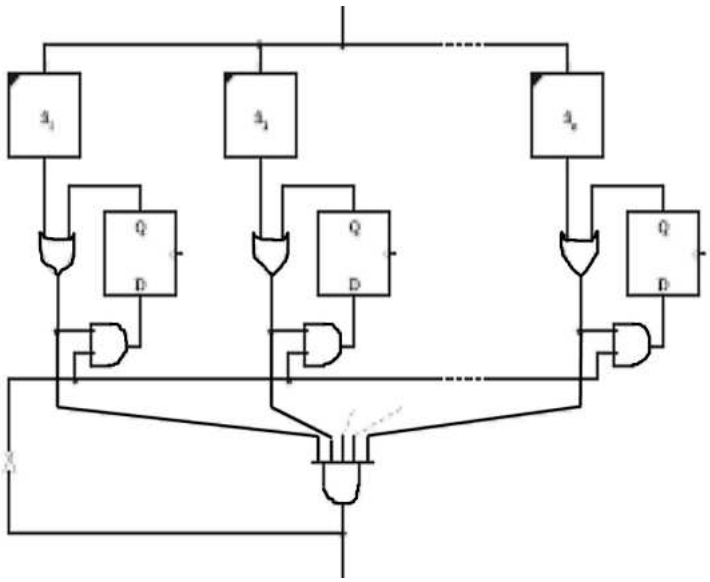

Figure 2.6: Parallel Composition on an FPGA taken from Audsley (2006)

programming of FPGAs, the most popular of which are ABEL and VHDL. However, we look at the higher level, algorithm focused Handel-C.

2.2.2 Handel-C

Handel-C is "a programming language aimed at compiling high level algorithms directly into gate level hardware" (Bowen, 1998), developed by Embedded Systems for Celoxica Ltd. By approaching the construction of the hardware from the perspective of the algorithm, Handel-C aims to allow the user to focus more on the algorithm itself than the workings of the hardware. However, even if users chooses to ignore the lower level aspects of Handel-C, they are not protected from the implementation entirely. Handel-C does not provide the same degree of abstraction from the hardware enjoyed by the users of DAP-Fortran.

Like DAP-Fortran, Handel-C is based on a familiar syntax. In this case it is the syntax of the popular ANSI C programming language (Bowen, 1998). The standard operations such as assignment and statements should be familiar to users of conventional C and so are not discussed here. However, there are several important differences between C and Handel-C. Firstly, standard Handel-C has only one fundamental type for variables, the integer. As these variables are to be represented in hardware, efficiency is key, and so integers have user defined arbitrary width. Secondly, pointers do not exist in Handel-C. Thirdly, it is possible to use Handel-C to "instruct the compiler to build hardware to execute statements in parallel" (Bowen, 1998). A more detailed description of the differences can be found in the Handel-C Reference Manual, Bowen (1998).

Although Handel-C is inherently sequential, it includes the use of parallel structures inspired by occam2 (Brookes and Stewart, 1989). The below shows the syntax of a typical parallel operation in Handel-C.

```
par
{  x = 1;
   {  y = 2;
      z = 3; }
}
```

Here, the operation to assign a value to the variables x is compiled to execute in parallel to the sequential assignment of values to y and then

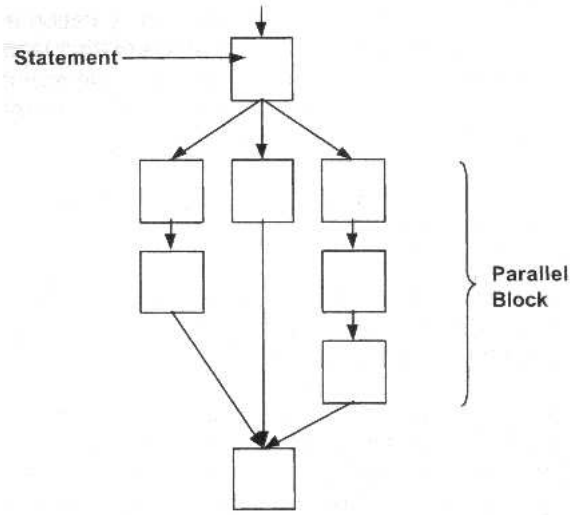

Figure 2.7: Parallelism in Handel C taken from Bowen (1998)

z. The repeated par structure, key to efficient parallel programming in occam2 (Pountain and May, 1988) was included in the DK2 Handel-C advanced optimisation manual in 2003 (Celoxica, 2003).

```
par (i=0; i<10; i++)
{
    x[i] = i;
}
```

The body of the repeated par structure is simply copied by the compiler, with the value of the accumulator i inserted into the code. All statements are then compiled to operate in parallel on the hardware.

Figure 2.7 shows Handel-C's treatment of parallelism as parallel blocks of sequential operations. The similarity between this and the eventual hardware implementation shown in Figure 2.6 should be noted. In reality, the way in which the Handel-C code is used to generate hardware is a proprietary process and includes many optimisation steps. The link between the model used by code and the final implementation is variable.

These parallel blocks contain sequences of operations that are applied independently and the Handel-C Reference Manual (Bowen, 1998) states that no variable must be accessed by more than one of these parallel branches. The only legal method of communication between parallel

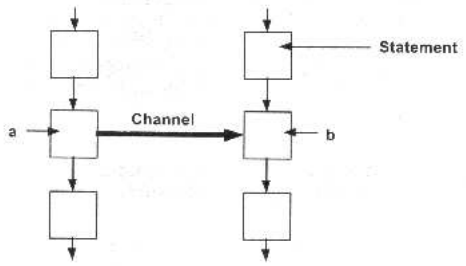

Figure 2.8: Handel-C Channels taken from Bowen (1998)

branches is by channels. These channels act in the same way as those of occam2, using ! and ? to denote a channel output and input (Hoare, 1988). An example of this communication can be seen in Figure 2.8.

```
x[4] ! 3;  // Output 3 on channel x[4]
x[3] ? y;  // Input to y from channel x[3]
```

The above code shows channel x[4] being set to output the value 3 and value from channel x[3] being assigned to variable y. The use of channels means that the programmer is forced to think about the flow of data through the hardware and of timing and deadlock considerations. These details are often considered the most difficult part of programming in Handel-C, and are the type of pitfall the DAP-Fortran user is protected from.

2.2.3 Context Summary

The DAP was a parallel processor developed in the 1970s alongside a specialist programming language, DAP-Fortran which enabled the production of highly parallel code. The DAP was applied to many different fields in both industry and academic research, with many important parallel algorithms developed. Over time, the use of the DAP largely ceased, but much of the work done for it remains.

FPGAs are highly versatile modern day pieces of logic-gate level programmable devices. Algorithms may be written and compiled to form hardware that executes in parallel using Handel-C.

The possibility exists that Handel-C could be used to replicate the operations of the DAP and eventually the structure itself on the FPGA. This may be used to prevent the loss of important information gained by

the development and use of DAP-Fortran algorithms. The first step of this process would be to translate the parallel algorithms of DAP-Fortran to the inherently parallel Handel-C.

Part II

Investigation

3 Design

3.1 Aims of the Project

The overall focus of this project is to provide a method for porting algorithms written in the parallel language DAP-Fortran to Handel-C. Specifically, the project aims to provide answers to the following questions.

1. What processes can be followed in order to efficiently convert algorithms by hand from the original DAP-Fortran to Handel-C?

2. What, if any, problems are encountered when translating the high-level algorithms designed for the DAP architecture to the lower level hardware independent Handel-C?

3. What, if any, problems are encountered when translating algorithms for an inherently parallel language to a sequential language which has support for parallelism?

4. Are there benefits to porting parallel algorithms for the DAP to Handel-C?

What processes can be followed in order to efficiently convert algorithms by hand from the original DAP-Fortran to Handel-C?

Faced with the different programming paradigms of DAP-Fortran and Handel-C, it is expected that a naïve direct translation of an algorithm from one to the other will not lead to an efficient Handel-C implementation. There may be cases where this method is sufficient, but for most it is likely to be necessary that different processes be followed. These processes would differ dependent on the methods used by the original algorithm. This project aims to provide processes which could be followed by a developer wishing to port DAP-Fortran algorithms.

What, if any, problems are encountered when translating the high-level algorithms designed for the DAP architecture to the lower level hardware independent Handel-C?

DAP-Fortran is a high level programming language which provides many built in componential and aggregate functions. These functions provide programmers with a degree of abstraction from the details of how the parallelism is implemented. The details of the algorithms used by these built in functions have mostly been lost, but can be seen to be dependent on features provided by the architecture of the DAP. Conversely, whilst Handel-C was written for use on FPGAs, it is described as being "designed to enable the compilation of programs into synchronous hardware" (Bowen, 1998) and so is independent of any one piece of hardware. Whilst it is a level above a hardware description language, its algorithms must still be low level enough as to "instruct the compiler to build hardware to execute statements in parallel" (Bowen, 1998). An aim of this project is to discover what problems are faced by a developer wishing to port a high-level algorithm written in DAP-Fortran to the lower level language, Handel-C.

What, if any, problems are encountered when translating algorithms for an inherently parallel language to a sequential language which has support for parallelism?

When designing the programming language for the DAP, the designers took the conscious decision to write DAP-Fortran as a parallel language, based on the syntax of an existing sequential language its target base would be familiar with (Flanders, 1979) (Perrott and Stevenson, 1981) (Perrott and Zarea-Aliabadi, 1986). As a result, DAP-Fortran programs are generally written as a sequence of parallel operations. In contrast, Handel-C was written as a restricted version of the sequential programming language C, with aspects of parallelism—inspired by occam—added to it. As a result, Handel-C parallel programs tend to have sequential processes operating in parallel blocks, synchronised by inter-process communications via channels. Therefore, a further aim of this project is to discover what problems are encountered by a developer when porting an algorithm between languages with such different paradigms of parallelism.

Are there benefits to porting parallel algorithms for the DAP to Handel-C?

In many cases there will already be existing equivalent algorithms available in Handel-C to those written in DAP-Fortran. Such algorithms are likely to exploit specific features of the targeted FPGAs which could not easily be considered in a general porting process. A final aim of the project is to determine whether there are benefits to using the processes developed to port algorithms from DAP-Fortran to Handel-C.

3.2 Out of Scope

A direct translation from one language to another is usually a matter of following rules and mappings between subsets of the two languages. Such a process should produce target code with the same meaning and outcomes as the original. Further optimisation steps may then be applied to improve the efficiency of this target code. This project does not seek to provide detailed mappings between DAP-Fortran and Handel-C. Its interest lies in the algorithms employed.

There are currently Fortran to C converters available e.g. f2c (Feldman et al., 1990), but to use these, the DAP-Fortran would first need to be converted to Fortran and then the C code produced would undergo a further transformation to Handel-C. These conversions would require the algorithms to be translated to sequential algorithms. Such a process would therefore negate the reasons for attempting the translations, and so this approach is also not attempted.

3.3 Methodology Employed

The method of development used employs techniques taken from Rapid Application Development (RAD) (Martin, 1991). The project was found suitable for this IT development process as it demonstrated many features that matched the criteria for RAD and the working practices of the developer.

The criteria matched by the project were as follows.

- The project is high level and constrained by a short, inflexible timeframe.

- Small project team of fewer than six people—in this case, one.
- The requirements for the project have been previously established.
- Data for the project is already available, with the main focus being on analysing and reporting on that data.
- Technologies in use are well defined and clear.

The main aspects of the RAD method used are prototyping, iterating and timeboxing. These methods mean that the project can be tackled in short bursts of intensive work, a way of working well suited to the implementer.

The project lends itself to the use of prototyping as it requires an investigation into possible processes. Using prototyping, a process can be tried, evaluated and refined over several iterations. At each step, the results of the iteration can be used to influence the direction of the investigation.

As the investigation has a flexible scope, the timescale of the project and availability of the developer are the limiting factors. Using timeboxing, the work done towards the project is focussed towards the stages of the project, such as producing the prototype translations and analysing the processes which emerge, with the number of iterations limited by the time allocated.

3.4 Method

The aims of the project require that processes for converting DAP-Fortran algorithms to Handel-C be produced and the reasons for the approaches taken be analysed. Using the RAD processes described in Section 3.3 an obvious approach to this investigation would be to use prototyping to attempt the translation of an algorithm. The information gained by this process would then feed into the decision to either attempt to refine the prototype and thus the process of the translation, or to start the translation of another algorithm and a new prototype cycle.

The method to be followed during the investigation is thus:

1. Identify a suitably simple DAP-Fortran algorithm to be translated

2. Ascertain the method followed by the algorithm in terms of the data structures used and the steps taken to move between these
3. Construct a Handel-C program which directly follows the method of the DAP-Fortran algorithm, including the intermediate steps and data structures where possible. Identify where direct translations are not possible
4. Compare the Handel-C program produced with the original DAP-Fortran code and determine a process for translation
5. Identify problems encountered and note aspects which require further investigation
6. Analyse the Handel-C program produced and ascertain the potential for efficiency improvements, whilst preserving meaning. If potential improvements are identified, produce the new code and repeat the analysis of steps 4 and 5.
7. When no further improvements can be found, consult notes for the aspects of the translations which were identified as requiring further investigation.
8. Identify an algorithm in DAP-Fortran which would enable any necessary further investigation and repeat.

Once all avenues of investigation have been exhausted, or the time allotted to the investigation has expired, the translation processes used should be analysed and evaluated.

4 Investigation

4.1 Example 1: Matrix Comparison and Manipulation

4.1.1 Original DAP-Fortran Code

This first piece of example code demonstrates parallel matrix assignment and the use of a mask. These two operations are fundamental to many DAP-Fortran algorithms. The code shown in Listing 4.1 is adapted from an example given in (ICL, 1978).

Listing 4.1: Example 1: A Simple DAP-FORTRAN Algorithm

```
1         CHARACTER MATRIX FUNCTION MAP(HEIGHT)
2         INTEGER HEIGHT(,), SEA_LEVEL
3   C
4   C       CODE TO CHECK A MATRIX OF HEIGHTS AND CREATE
5   C       A MATRIX MARKING THOSE HEIGHTS BELOW SEA_LEVEL
6   C
7         MAP=' '
8         MAP(HEIGHT.LT.SEA_LEVEL)='X'
9         RETURN
10        END
```

The code takes a standard 64x64 matrix, MAP, and marks with an "X" where the corresponding integers in the provided matrix HEIGHT are less than the value given in SEA_LEVEL.

Algorithm Overview

The algorithm is approached as a sequence of parallel operations as seen in Figure 4.1. Firstly, every element of the matrix MAP is assigned, in parallel, the blank character, " ". Then a boolean mask is created by comparing, at the same time, every element of HEIGHT with SEA_LEVEL. This mask is then used to determine which elements of MAP are written with an "X".

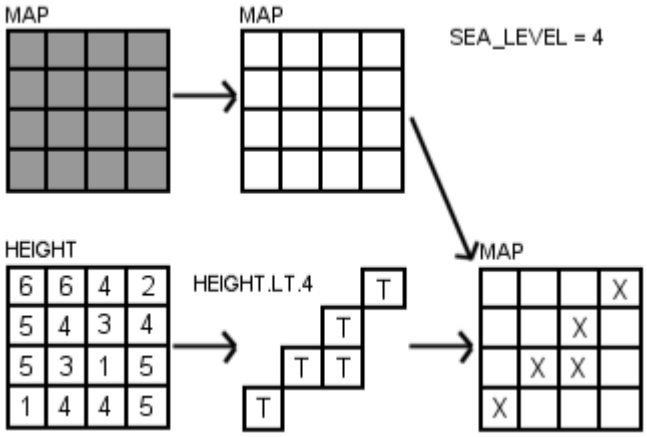

Figure 4.1: Example 1: Graphical representation of DAP-FORTRAN Algorithm

4.1.2 Version 1: Direct Conversion to Handel-C

In order to convert the DAP-Fortran Algorithm to Handel-C, we must first ensure the overall algorithm is compatible.

The CHARACTER MATRIX MAP can be emulated as a matrix of type unsigned int 8 and other DAP-Fortran INTEGERs as signed int 32. However, DAP-Fortran implements its mask system directly on the hardware. Operations such as the assignment seen here are dependent on the setting of a register bit at each PE in the processor array. Clearly this cannot be directly copied in Handel-C. Instead, we take the naïve approach of implementing the mask as a separate matrix, mask.

As Handel-C does not use boolean types, we default to the use of unsigned int 1 to symbolise true (1) and false (0). Similarly, we use 1 and 0 to represent "X" and the blank character in the map matrix.

The algorithm we must then follow is in a sequence of parallel operations in three stages. Firstly, we write the blank (0) to all entries of the matrix, map. We then populate the mask matrix with 1s where the corresponding value of height is less than sea_level. Finally, we write a 1 to the map matrix where a 1 is seen in corresponding positions of mask.

Following these decisions, we produce the code seen in Listing 4.2.

Listing 4.2: Version 1: A Direct Conversion of Example 1 to Handel-C

```
1 // Type definitions used to emulate the DAP–Fortran types
2 typedef signed int 32 int2;
```

```
3  typedef unsigned int 1 logic;
4  typedef unsigned int 8 charac;
5
6  // Types of structures used
7  int2 height[64][64];
8  int2 sea_level;
9  logic mask[64][64];
10 charac map[64][64];
11
12 // Body of the algorithm
13
14 // All elements in matrix map assigned in parallel
15 par(i=0; i<=63; i++)
16 { par(j=0; j<=63; j++)
17   { map[i][j] = 0; }
18 }
19
20 par(i=0; i<=63; i++)
21 { par(j=0; j<=63; j++)
22   { if(height[i][j]<sea_level) {
23             mask[i][j] = 1;
24       else mask[i][j] = 0; }}
25 }
26
27 par(i=0; i<=63; i++)
28 { par(j=0; j<=63; j++)
29   { if(mask[i][j]==1) map[i][j] = 1; }
30 }
```

For each step of the algorithm, the nested repeated par is used, allowing every element of the 64x64 matrix in use to be written to, tested or compared at the same time. The second and third steps—those of creating and using the mask—show sequential operations being followed in parallel.

Comparison to DAP-Fortran

At first sight, there is little similarity between the Handel-C code produced and the original code. The algorithm followed is in essence the same in Handel-C as it was in DAP-Fortran. However, whilst the use of a mask in DAP-Fortran was beneficial due to its strong ties to the hardware, in Handel-C the representation of the mask as a separate data structure, in this case a matrix, is redundant and only adds to the complexity of the code.

Even using the shorthand repeated par, it is clear to see that an algorithm which takes only two lines of DAP-Fortran's inherently

parallel code, can mushroom into a large Handel-C program. DAP-Fortran's high level shorthand—treating matrices the same as it would scalars—allows for much more succinct code.

Possible Future Improvements

An improvement to the code is thus the removal of the intermediate mask matrix.

Future Investigation Needed

A further investigation should also be made into the possible introduction of a set of functions or macros to enable a DAP-Fortran style shorthand in Handel-C.

Process Steps Identified

- All non-integer types should be mapped to integers.
- A mask can be represented in Handel-C as a matrix of bits, the components of which can then be used in a conditional.
- Nested repeated pars should be used to replace the parallel assignments in DAP-Fortran.

4.1.3 Version 2: Revised Conversion to Handel-C

An obvious improvement to the translated algorithm in version 1, was to remove the intermediate step of creating a mask and combining it with the conditional which involved it in step 3. By doing this, the code in Listing 4.3 was produced.

Listing 4.3: Version 2: A Revised Conversion of Example 1 to Handel-C

```
1 par(i=0; i<=63; i++)
2 { par(j=0; j<=63; j++)
3   { map[i][j] = 0; }
4 }
5
6 par(i=0; i<=63; i++)
7 { par(j=0; j<=63; j++)
8   { if(height[i][j]<sea_level) map[i][j] = 1; }
9 }
```

Comparison to DAP-Fortran

Here we can see that the three step algorithm from Version 1 has been reduced to two steps, as in the original algorithm. These steps comprise the wholly parallel stage of matrix assignment and a stage of sequences conducted in parallel. In terms of the individual elements of the matrix map we can see that they are each written to be "0" and then conditionally overwritten with a "1", just as in the original algorithm.

Possible Future Improvements

As was the case in version 1, we can see that the two steps of writing and rewriting to matrix map can easily be combined, exploiting Handel-C's "if" clause.

Process Steps Identified

- Where masks are used in the form

  ```
  A(conditional) = expression
  ```

  these can be converted to Handel-C in the form

  ```
  if (conditional) {
     A[i][j] = expression }
  ```

  where i and j index elements in the matrix where the conditional is true.

4.1.4 Version 3: Revised Conversion to Handel-C

Combining stages one and two from the code in Listing 4.3 produces the code in Listing 4.4.

Listing 4.4: A Simple Example in Handel-C: Version 3

```
1 par(i=0; i<=63; i++)
2 { par(j=0; j<=63; j++)
3   { if(height[i][j]<sea_level) {
4           map[i][j] = 1;
5       else map[i][j] = 0; }
6   }
7 }
```

Comparison to DAP-Fortran

Listing 4.4 is a much more concise translation of the algorithm. The algorithm has been converted from a sequence of parallel operations to a parallel block of sequential operations, reflecting the natures of the respective languages.

This was possible as the individual elements of the map matrix were dependent on the respective values in the height matrix, and were independent of the other values within map. In terms of the parallel operations, there was no need for the comparison of all values in height to complete before writing to map.

Possible Future Improvements

No further improvements can be identified at this stage.

Process Steps Identified

- Where the assignments to a matrix are of the form

  ```
  A = expression1
  A(conditonal) = expression2
  ```

  where the matrix as a whole is assigned one value and then components selectively reassigned to another values, the two steps may be combined into one statement.

  ```
  if (conditional) {
     a[i][j] = expression2
    else
     a[i][j] = expression1 }
  ```

- Where the algorithm is such that its application to the individual elements of the matrices in use are independent of the other elements, the process of sequential parallel actions may be converted to a parallel block of sequential processes, as seen in Figure 4.2.

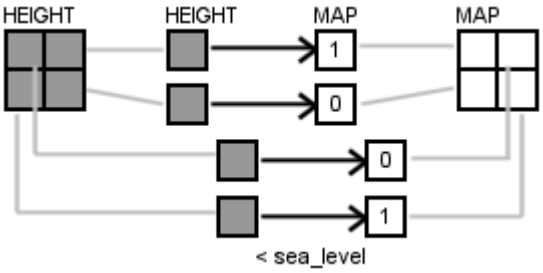

Figure 4.2: Example 1: Graphical representation of ported algorithm

4.1.5 Conclusion

Experiments so far have concluded that the following steps must be taken in order to successfully port a DAP-Fortran Algorithm to Handel-C.

- All non-integer types must be mapped to integers. Suggestions for four of the mappings can be found in Appendix D.
- The DAP-Fortran feature of masks used in the form

  ```
  A(conditional) = expression
  ```

  can be converted to Handel-C code of the form

  ```
  if (conditional) {
     a[i][j] = expression }
  ```

  where `i` and `j` index elements in the matrix where the conditional is true.
- Nested repeated pars should be used to replicate the parallel assignments performed in DAP-Fortran. For example, the DAP-Fortran Matrix assignment

  ```
  A = 1
  ```

  is replicated using the Handel-C code

```
par (i=0; i<=63; i++)
{ par (j=0; j<=63; j++)
   { a[i][j] = 1; }
}
```

- Where the assignments to a matrix are of the form

  ```
  A = expression1
  A(conditonal) = expression2
  ```

  where the matrix as a whole is assigned one value and then components selectively reassigned to another values, the two steps may be combined into one statement.

  ```
  if (conditional) {
     a[i][j] = expression2
    else
     a[i][j] = expression1 }
  ```

- Where the algorithm is such that its application to the individual elements of the matrices in use are independent of the other elements, the process of sequential parallel actions may be converted to a parallel block of sequential processes, as seen in Figure 2.7.

4.2 Example 2: Matrix Multiplication

4.2.1 Original DAP-Fortran Code

This second piece of code demonstrates the use of a DO loop, matrix indexing and two of DAP-Fortran's built-in matrix manipulation aggregate functions. The original code used, shown in Listing 4.5, is taken from (Parkinson, 1990).

Listing 4.5: Example 2: A DAP-Fortan Algorithm for Matrix Multiplication

```
1         INTEGER A(,), B(,), C(,)
2         C = 0
3         DO 10 K = 1,64
4      10 C = C + MATC(A(,K)) * MATR (B(K,))
```

The code takes two standard 64x64 matrices and performs a matrix multiplication on then, with intermediate and the final result stored in the matrix,C.

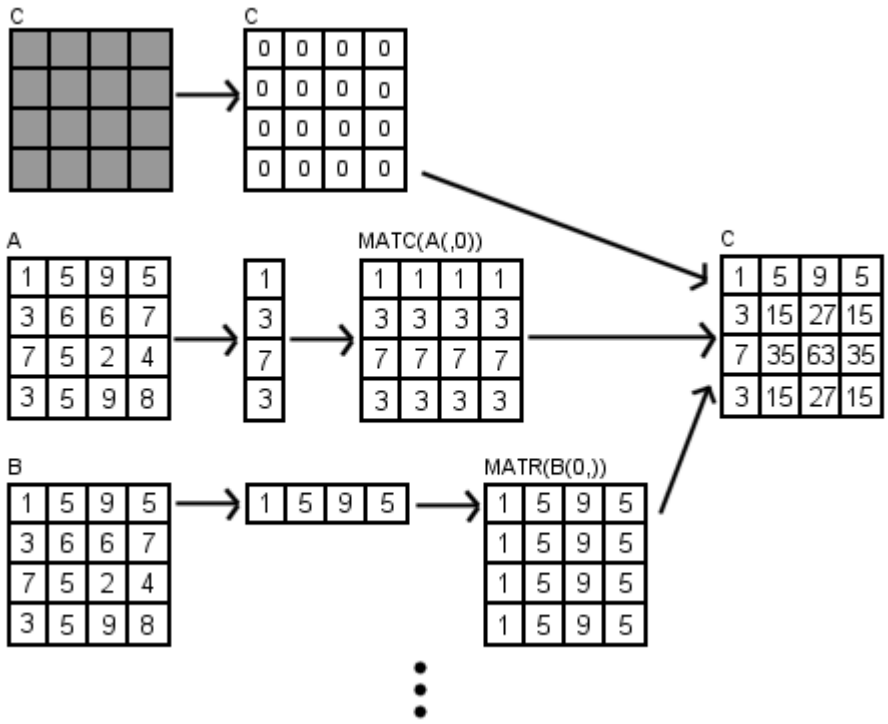

Figure 4.3: Example 2: Graphical representation of DAP-Fortran Matrix Multiplication Algorithm

Algorithm Overview

The algorithm uses three matrices. The values of A and B are given, and C is used to store the intermediate and final results. Firstly, the components of C are assigned 0. Then, the MATC aggregate function is applied to the first column of matrix A, and MATR applied to the first row of B, forming intermediate matrices of the repeated column and row respectively. On the DAP itself, these matrices are stored horizontally, where each element of the matrix is stored at a different PE and are stacked vertically so that each PE store has all the elements it needs to combine stored locally. The elements of the matrices are multiplied together and the result added to the values of C. This algorithm can be seen in Figure 4.3

This process of adding the multiplied intermediate matrices is repeated for each row/column pair in the matrices, ie. up to the 64th.

4.2.2 Version 1: Direct Conversion to Handel-C

The need to convert integer types to Handel-C format is covered in Section 4.1.2, therefore in order to ensure that the algorithm can be converted to Handel-C, we must focus on the use of DO loops and the aggregate functions MATC and MATR. The DO loop used in DAP-Fortran can be replaced by the Handel-C for loop with the body of the loop ending with the labeled line referenced by the original DO. In order to convert the MATC and MATR functions, we must look at how they are used.

These functions take a vector and return a matrix which is the vector repeated as each column and row respectively. These matrices are then used in the calculation of the multiplication. A naïve approach to this conversion is simply to create intermediate matrices to be used. This approach is followed in Listing 4.6.

Listing 4.6: Version 1: A Direct Conversion of Example 2 to Handel-C

```
1 int2 a[64][64], b[64][64], int2 c[64][64];
2 int2 d[64][64], e[64][64];
3
4 par (i=0; i<=63; i++)
5 { par (j=0; j<=63; j++)
6   { c[i][j] = 0 }
7 }
8
9 for (n=0; n<=63; n++)
```

```
10 {
11   par (i=0; i<=63; i++)
12   { par (j=0; j<=63; j++)
13     { d[i][j] = a[i][n]
14       e[i][j] = b[n][j] }
15   }
16
17   par (i=0; i<=63; i++)
18   { par (j=0; j<=63; j++)
19     { c[i][j] = c[i][j] + (d[i][j] * e[i][j]) }
20   }
21 }
```

Comparison to the DAP-Fortran

Here we see the use of nested repeated pars again as a method of parallel assignment of all elements of the matrix c to be 0. In order to follow the algorithm, the intermediate matrices of the DAP-Fortran are represented as the matrices d and e. These are established before the use of the calculation. In the original DAP-Fortran, the matrices are created separately within the calculation. This could be replicated in the Handel-C by using functions, but this method was not used so as to allow both matrices to be produced more efficiently in the same parallel action.

Possible Future Improvements

The use of intermediate matrices allows the exploitation of the whole processing array of the DAP, but is superfluous in the Handel-C as it is much more efficient to simply access the column or row vector directly in the parallel branches.

Future Investigation Needed

While the algorithm in use has identified a possible method of porting the use of MATC and MATR, the other built-in aggregate functions in DAP-Fortran should be investigated. Also, other uses of MATC and MATR should be investigated to ensure that the processes identified for translating these functions work in the general case and not just for this example.

Process Steps Identified

- The use of MATC and MATR can be replicated by the production of intermediate matrices before their use in calculations or assignments.
- The DO loop in DAP-Fortran can be replaced with a for loop in Handel-C where the body of the for loop is all parts of the algorithm represented by the code from the DO to the end of the referenced labelled line.

4.2.3 Version 2: Revised Conversion to Handel-C

Taking into consideration the need to streamline the algorithm by removing the intermediate matrices. The code in Listing 4.7 was developed.

Listing 4.7: Version 2: A Revised Conversion of Example 2 to Handel-C

```
1  int a[64][64], b[64][64], c[64][64];
2
3  par (i=0; i<=63; i++)
4  { par (j=0; j<=63; j++)
5    { c[i][j] = 0; }
6  }
7
8  for (k=0; k<=63; k++)
9  { par (i=0; i<=63; i++)
10   { par (j=0; j<=63; j++)
11     { c[i][j] = c[i][j] + ( a[i][k] * b[k][j] );}
12   }
13 }
```

Comparison to the DAP-Fortran

This new code now uses the same matrices as in the original DAP-Fortran code. It also looks similar to the original code. If we ignore the nested repeated par code required to implement the parallelism, we have

```
c[i][j]=0;
c[i][j] = c[i][j] + (a[i][k] * b[k][j]);
```

we compare this again to the DAP-Fortran code

```
C = 0
C = C + MATC(A(,K)) * MATR(B(K,))
```

and we can see the similarities. As we do not require the intermediate matrices produced by the MATC and MATR functions, we remove these function calls and we have almost the same code. The algorithm followed by the converted code is seen in Figure 4.4. Comparing the two algorithms, we can see that they are also very similar.

Possible Future Improvements

No further improvements can be identified at this stage.

Process Steps Identified

- Where the elements of a row or column of a matrix are to be accessed individually, the DAP-Fortran method of creating intermediate matrices through the use of MATC or MATR, this step may be bypassed by Handel-C and the column or row referred to directly.

4.2.4 Conclusion

In addition to the steps identified in Section 4.1.5, the following steps were found during the investigation of the matrix multiplication algorithm.

- The DO loop in DAP-Fortran may be replaced with a for loop in Handel-C. This conversion would change code of the form

  ```
        DO 10 K = 1,20
        A = 1
   10 B = 2
  ```

  to become of the form

  ```
  for (k=1; k<=20; k++)
  {  a = 1;
     b = 2; }
  ```

  where A and B are scalars.

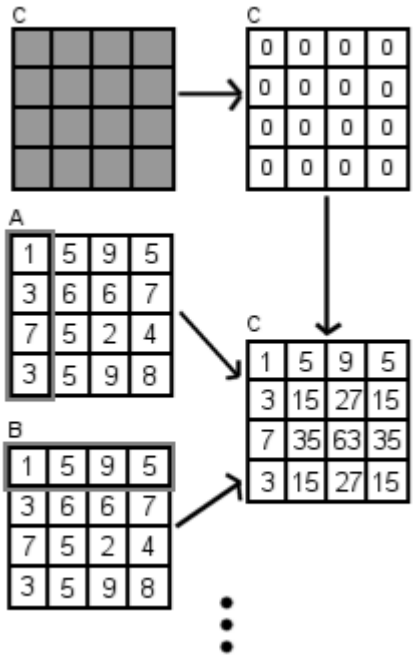

Figure 4.4: Example 2: Graphical representation of Handel-C Matrix Multiplication Algorithm

- Where the elements of a row of column of a matrix are the be accessed individually using MATR or MATC, the step of creating intermediate representations of these rows and columns as matrices may be bypassed and the rows and columns referred to directly.

4.3 Example 3: The Contour Problem

4.3.1 Original DAP-Fortran Code

This more complex example not only includes a combination of many of the features previously seen, but also demonstrates one of DAP-Fortran's more hardware-related abilities. Because of DAP-Fortran's strong ties to the hardware of the DAP itself, it is easily able to do such things as reference the adjacent cells of each component of a matrix, as this example shows.

The algorithm for this example is taken from (ICL, 1978). The main sections of code for the algorithm are shown in Listing 4.8. A full code listing is found in Appendix A.

Listing 4.8: Version 1a: Conversion of Example 3 Edge Detection to Handel-C

```
1          CHARACTER MATRIX FUNCTION
2         1          CONTOUR(HEIGHT,LEVEL,NUMBER_OF_LEVELS)
3
4          FREE=.TRUE.
5
6  C    Stage 1
7          DO 1 K=1,NUMBER_OF_LEVELS
8          SELECTED=FREE.AND.HEIGHT.LT.LEVEL(K)
9          REGION(SELECTED)=K
10         CONTOUR(SELECTED)=CODE(K)
11        1 FREE(SELECTED)=.FALSE.
12
13 C    Stage 2
14         EDGE=REGION.LT.REGION(,+).OR.REGION.LT.REGION(,-)
15        1.OR.REGION.LT.REGION(+,).REGION.LT.REGION(-,)
16         CONTOUR(.NOT.EDGE)=' '
17
18         RETURN
19         END
```

This code takes a matrix of INTEGER heights and creates three new

matrices. The first matrix, REGION shows the region which the height maps to. These regions are defined within the array LEVEL, where the position in the array is the region number and the entries in the array are the upper bounds of the heights allowed in that region. The second matrix produced, EDGE is an array of booleans, where each entry describes whether the corresponding entry in the array REGION is at a boundary between two different regions. The third matrix produced combines the EDGE matrix and the REGION matrix to produce a matrix whose only non-blank entries are at the edges between regions. The entries themselves are the character representations of the regions taken from the array CODE.

Algorithm Overview

The algorithm used has two main stages. The first is an iterative population of matrices with the appropriate data and the second uses the data found to create a contour map, as can be seen in Figure 4.5.

First Stage

In the first stage, the matrix FREE is set such that all its entries are assigned the value .TRUE. Next, the matrix SELECTED is produced which is populated with the values .TRUE. and .FALSE. dependent on whether the corresponding value in the matrix HEIGHT is less than the first entry in the array LEVEL. This indicates that the value is within region 1 and the cell has not already been assigned a region number, i.e. the corresponding entry in FREE is .TRUE.

Based on this array, the matrix REGION has the region number 1 entered where the corresponding value of SELECTED is .TRUE. The CONTOUR matrix is populated with the code corresponding to region 1 where the value of SELECTED is .TRUE. The SELECTED matrix is again used, this time to enter the value .FALSE. into FREE for those cells which were in region 1, to signify that these cells have now been classified.

The process of identifying cells regions is repeated NUMBER_OF_LEVELS times, i.e. up to the number of regions covered.

Second Stage

The second stage is used to select the upper edge of each region. It does this by checking each element of REGION against its four adjacent

elements. On the DAP, the ALU of each PE has a multiplexor which chooses what is input. This may be data from the PEs own store, or an output from on of its four neighbouring PEs. Thus, DAP-Fortran needs only specify which input the multiplexor should receive and the top multiplexor shifts this in while the bottom multiplexor sends the value out to the relevant neighbouring ALU. At the edges of the matrix, the value shifted in depends on the geometry in use. In this example, the default geometry is used, which is PLANE for both north-south and east-west. This means that a zero is shifted in at each edge. Where the component in REGION is less than any one of its neighbours, the value of EGDE is set to .TRUE.

Once this has been set, the EDGE matrix is used so that any place where the value is FALSE, i.e. the cell is not on an edge, the value of CONTOUR is set to the blank character. Thus, only the region code for each edge remains.

4.3.2 Version 1: Direct Conversion to Handel-C

For the first stage, the standard data type conversions must be made as seen in previous examples, along with the conversions seen for parallel assignments, masks and DO loops. The direct translation of the code for this stage is given in Listing 4.9

Listing 4.9: Version 1: A Direct Conversion of Example 3 to Handel-C

```
1  int2    region[64][64], height[64][64];
2  int2    numoflevels, level[64];
3  logic   selected[64][64], edge[64][64];
4  logic   free[64][64];
5  charac code[1,2,3,4,5,6,7,8,9,10];
6
7  \\ Sets the free matrix to be filled with true
8  par (i=0; i<=63; i++)
9  { par (j=0; j<=63; j++)
10   { free[i][j] = 1; }
11 }
12
13 \\ Parallel assignments performed in sequence
14 for (k=1; k<=numoflevels; k++)
15 {
16   \\ Select cells to be used in subsequent stages
17   par (i=0; i<=63; i++)
18   { par (j=0; j<=63; j++)
19     { selected[i][j] = (free[i][j] && (height[i][j] < level[k]));}
```

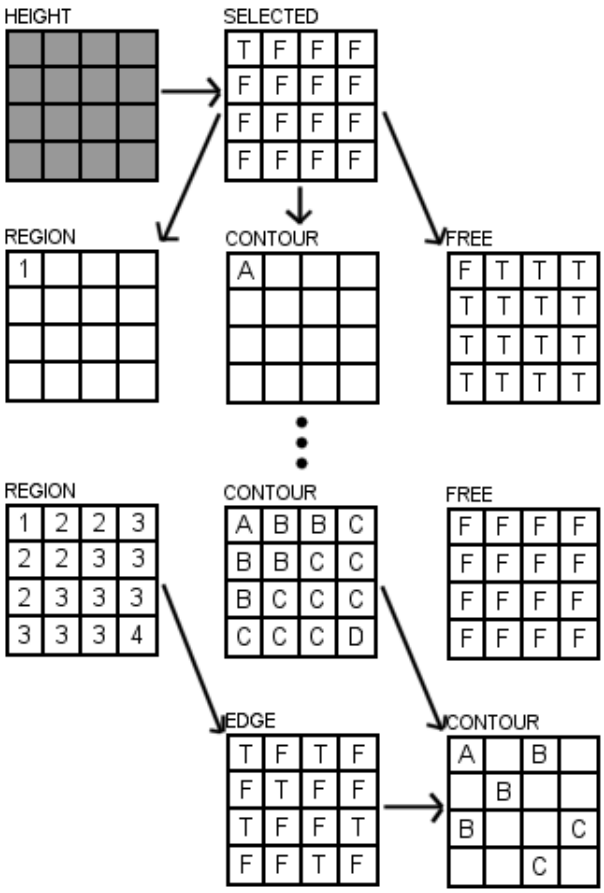

Figure 4.5: Graphical representation of algorithm used in DAP-FORTRAN method

```
20    }
21
22    \\ Assigns region numbers to selected cells
23    par (i=0; i<=63; i++)
24    { par (j=0; j<=63; j++)
25      { if (selected[i][j]) region[i][j] = k;}
26    }
27
28    \\ Assign code to contour matrix
29    par (i=0; i<=63; i++)
30    { par (j=0; j<=63; j++)
31      { if (selected[i][j]) contour[i][j] = code[k];}
32    }
33
34    \\ Set free to show used cells
35    par (i=0; i<=63; i++)
36    { par (j=0; j<=63; j++)
37      { if (selected[i][j]) free[i][j] = 0;}
38    }
39  }
```

For the second stage, however, we find that a direct translation of the shifting of cells is not a simple task. In Handel-C, the accessing of variables by more than one parallel branch should be avoided. If we took the approach of simply having each cell access those around them, each cell of the matrix would be accessed by up to four parallel branches. Therefore, this must be approached in a different manner.

There are two immediately obvious ways of solving this. The first method for this would be using four copies of the region matrix which can be accessed for the values of the north, south, east and west neighbour. The second would be to use the Handel-C feature of channels. The first method requires each of the copied matrices to be 66x66, with the edges filled by zeroes. This will allow the cells at the edges of the region to have surrounding cells to shift in if needed and so allow them to be treated the same as other cells. This approach can be seen in Listing 4.10

Listing 4.10: Version 1a: A Direct Conversion of Stage 2 of Example 3 to Handel-C, using copied matrices

```
1  int2 north[66][66], south[66][66], east[66][66], west[66][66];
2
3  par (i=0; i<=65; i++)
4  { par (j=0; j<=65; j++)
5    { north[i][j] = 0;
6      south[i][j] = 0;
7      east[i][j] = 0;
8      west[i][j] = 0; }
```

```
 9  }
10
11  par (i=1; i<=64; i++)
12  { par (j=1; j<=64; j++)
13    { north[i][j] = region[i][j]; }
14  }
15
16  par (i=1; i<=64; i++)
17  { par (j=1; j<=64; j++)
18    { south[i][j] = region[i][j]; }
19  }
20
21  par (i=1; i<=64; i++)
22  { par (j=1; j<=64; j++)
23    { east[i][j] = region[i][j]; }
24  }
25
26  par (i=1; i<=64; i++)
27  { par (j=1; j<=64; j++)
28    { west[i][j] = region[i][j]; }
29  }
30
31  par (i=0; i<=63; i++)
32  { par (j=0; j<=63; j++)
33    { if (region<north[i+1][j] &&
34          region<south[i+1][j+2] &&
35          region<east[i][j+1] &&
36          region<west[i+2][j+1])
37      {       edge[i][j] = 1;
38        else edge[i][j] = 0; }
39    }
40  }
```

This method of copying matrices can be seen to be crude and inefficient and does not reflect the simple nature of comparing two elements of the same matrix.

Using the channels method, each cell would output its value on four channels, which can then be read by the element's neighbours.

For the majority of the elements, this can be implemented in a regular fashion. However, exterior cells have fewer neighbours and so outputs to superfluous channels will not be received. Similarly, cells at the boundaries attempting to read from non-existent channels will fail. Special cases must be made for these exterior cells.

Listing B.1 in Appendix B shows an implementation of stage 2 using channels.

Comparison to the DAP-Fortran

First Stage

The code for the first stage well reflects the nature of the DAP-Fortran algorithm. The four chunks of parallel code match to the equivalent four lines of DAP-Fortran. However, we can see that the code produced, while reflecting the nature of the DAP-Fortran algorithm, is unnecessarily inefficient. New parallel hardware is created for each of the nested par's created for the four steps in the first stage. This hardware could be reduced and a greater degree or parallelism used by reducing the code to a parallel block of sequences.

Second Stage

The use of copied matrices to enable the parallel comparison of elements within a matrix reflects neither the DAP-Fortran code nor the original algorithm. However, the use of these matrices allows for the simple conversion of the code for the checking of the north, east, south and west neighbours.

The use of channels to enable the parallel comparison of elements within a matrix reflects the algorithm used by the DAP-Fortran and the eventual hardware implementation of this code is likely to match the method used by the DAP hardware itself, with outputs routed from PE to neighbouring PE. However, the code itself is very complex and the writer is forced to think of the elements of the grid separately, consider their physical location and describe low level courses of action for each. This is far from the philosophy of the original algorithm.

Possible Future Improvements

As seen previously the sequential parallel processed of stage one may be combined to form a block of sequential processes running in parallel, where race conditions do not apply.

The matrix copying code for stage 2 is an unpleasant work-around but requires no alterations.

The code for the use of channels is necessarily complex, as it describes lower level actions and special cases.

Future Investigation Needed

Investigation is required into the relative merits of using the copied matrices method and the channel method. These two methods both produce the same results, but neither is ideal for this situation in terms of efficiency and algorithm followed.

Further investigation should also be made into the implementation of other geometries used by the DAP, for example the case where the planes wrap around.

It is apparent that this simple DAP-Fortran code which uses hardware specific features translates to complex Handel-C code. This code must either attempt to describe the operation of the DAP or use "hacks" which do not relate to the original algorithm, but produce the same result. Investigations are required into other DAP-Fortran functions which rely heavily on the DAP hardware.

Process Steps Identified

- Where offset referencing is used, the requirement of the programmer may dictate the translation.
  - If readability and simplicity of code is preferred, copies of the target matrix should be made and used for the comparisons.
  - If hardware efficiency and links to the original algorithm are required, channels should be used to communicate data between array elements.

4.3.3 Version 2: Revised Conversion to Handel-C

The revised code for stage 1 of the algorithm can be seen in Listing 4.11.

Listing 4.11: More Intuitive Version

```
1 for (k=1; k<=numoflevels; k++)
2 { par (i=0; i<=63; i++)
3   { par (j=0; j<=63; j++)
4     { if (free[i][j] && (height[i][j] < level[k]))
5       { region[i][j] = k;
6         contour[i][j] = code[k];
7         free[i][j] = 0;} }
8   }
9 }
```

Comparison to the DAP-Fortran

This code does not follow the same algorithm as the DAP-Fortran, but provides a more efficient Handel-C implementation. It shows the same structure as the DAP-Fortran as it now describes the operations on each of the elements of the matrix individually.

Possible Future Improvements

No areas have been identified for improvement in additional iterations.

Future Investigation Needed

No further investigations are prompted by this iteration.

Process Steps Identified

No new steps are identified by this iteration.

4.3.4 Conclusion

It is clear to see that as the DAP-Fortran algorithms are translated to Handel-C, the code produced is highly inefficient. Optimisation processes followed have all changed the algorithm from parallel step algorithms to sequences of operations occurring in parallel. In addition to this, DAP-Fortran code which exploits features of the DAP must be analysed in terms of how each matrix element is treated dependent on its location in the matrix and that treatment translated to a low level description in Handel-C. Both of these issues contradict the reasons for producing a method for porting the algorithms to Handel-C.

Because of these two discoveries and shortness of time, no further prototypes will be attempted.

The final step identified whilst attempting to translate the contour algorithm is:

- Where offset referencing is used, the requirement of the programmer may dictate the translation.
  - If readability and simplicity of code is preferred, copies of the target matrix should be made and used for the comparisons.

- If hardware efficiency and links to the original algorithm are required, channels should be used to communicate data between array elements.

A full set of process steps from all experiments is available in Appendix C.

Part III

Conclusion

5 Conclusion

The original idea behind the project was sparked by the apparent similarities between the architectures of the FPGA and the DAP. Superficially, the processing elements and highways appear very similar to the logic blocks and interconnections of an FPGA, which can be arranged such that each of these blocks is able to operate on a separate element of data. When lifted to the language level, however, we see that while Handel-C is happy to think in these terms, DAP-Fortran does not. DAP-Fortran thinks in terms of the arrays, with each operation applied across the whole of it. Handel-C works on the element level, with each independently receiving a stream of operations.

This difference in the software architectures can be simply summarised. DAP-Fortran uses parallel operations in sequence and Handel-C uses sequential operations in parallel. While DAP-Fortran allows the user to concentrate on the structures used in the algorithms, Handel-C's need to involve channels requires the programmer to think in terms of the data itself and how this must be treated. In essence, the use of channels defines the connections required on the FPGA itself and so claims that Handel-C is not a hardware description language are overstated.

These mismatches between software architectures mean that any direct translation of algorithms between the two lead to forced and clumsy results. In order for a DAP-Fortran Algorithm to be represented in Handel-C, work arounds are needed and simple operations in the original code become laborious tasks in Handel-C. This can be seen in Section 4.1.2, where two lines of DAP-Fortran code require 14 lines of descriptive Handel-C to form a direct conversion of the algorithm. Process steps taken later in that section lead eventually to a more efficient Handel-C representation in Section 4.1.4 of only 6 lines. However, by this point, the algorithm itself has been necessarily changed to match the 'sequences in parallel' approach of Handel-C. "Often the optimum DAP algorithm differs from the optimum algorithm on other machines and this complicates comparisons" (Reddaway, 1979).

The effect is amplified when DAP-Fortran algorithms which exploit

features of the DAP hardware are ported. A direct translation of a single operation in Section 4.3.1, which includes an exploit of the DAP's use of inter-PE connections, results in a 143 line description of how each element in the Handel-C equivalent matrix should interact with those around them (Appendex B).

It is clear to see from the results of Section 4.3 that DAP-Fortran's different approach to parallelism and its dependence on the underlying hardware architecture leads to complex and unmanageable code in Handel-C. Any attempts to simplify and improve upon this code lead to drastic changes in the nature of the algorithms.

As it was the intention of the project to preserve the algorithms themselves, we may conclude that this is possible by direct translation. However, we also conclude that the problems encountered by the translation may outweigh the benefits of this act. The simplicity of the DAP-Fortran code, and therefore the ease with which it is able to convey parallelism, is lost in translation to Handel-C. As this was one of DAP-Fortran's most celebrated assets, its loss is too great.

The use of the processes developed has, however, uncovered an unexpected benefit. The translation of the highly parallel DAP-Fortran algorithms may have resulted in a different kind of algorithm, but these new algorithms naturally retain their high degree of parallelism. The use of DAP-Fortran forces the writer to use parallelism, sometimes without them realising it. It is in the process of the conversion to Handel-C, and most likely most other languages, that the parallelism comes to light. We therefore recommend the use of DAP-Fortran as a parallel specification language.

6 Evaluation

This project is evaluated in terms of four factors. These are: the background research done; the conclusions of the investigation; the method for porting the algorithms developed during the investigation; and, the way in which the project as a whole was approached.

6.1 Research

The background research done around the DAP touched on many parts of the lifecycle of the DAP. This included research on the context in which the DAP was conceived, pre-prototype discussions, the findings of academic and commercial users and the uses of the DAP throughout the 1980s and 1990s. Research was also done on other parallel architectures of the time, parallel languages, and the needs of the users of these.

Much of the research done was eventually omitted from the final project. This was due to the spectrum of uses found for the DAP and the large number of different papers which addressed the same issues. From the point of view of time management, this may be considered wasted time, but from a research point of view all information uncovered potentially contained important information. For example, data on the exact implementation of some algorithms on the DAP was difficult to find, but in some cases was mentioned in passing in a paper whose overall theme lent little to the project.

With hindsight, and an aim to stick to a more rigid plan, time and effort could have been saved by developing a more detailed skeleton for the context section and researching only material relevant to that structure. However, I do not believe that such an approach would have allowed the writer to obtain as rich a picture of the era of the DAP as was eventually achieved.

Many different avenues were explored when attempting to find material on the DAP. Attempts were made to contact the authors of papers and those who had worked with the DAP, but in all cases

no replies were received. Another interesting source which failed to produce results was the Internet Archive (Archive, 2007) which contains saved copies of old internet sites. This uncovered a 2004 AMT webpage advertising their latest DAP model, but unfortunately no relevant information could be found as all links were dead.

6.2 Conclusions

It was expected that the project would result in a set of processes to be followed in order to port DAP-Fortran algorithms to Handel-C. While some process steps were developed, the conclusion that the project came to was that the two languages involved were sufficiently different in architecture that the reasons for porting between the two were not satisfied by this process.

This conclusion is based on three measures:

- the paradigm of parallelism employed;
- the simplicity of code produced; and
- the ease with which the parallelism is portrayed

These three measures are all properties of the DAP-Fortran code and algorithms that were the focus of much of the praise and research on the DAP. The paradigm of parallelism closely matched the way in which the DAP operated, the simplicity of the code allowed for succinct descriptions of complex functions and the inherently parallel structures allowed programmers to naturally write parallel programs. As the code produced failed to match any of these criteria, the expected advantages of porting the code would not be met.

6.3 Method for Porting Algorithms

The investigation into the porting of the algorithms produced a list of conversion steps from one structure to another. These steps were not, however, concrete and must be followed within the overall context of the algorithm. They give general advice and direction for dealing with issues such as conversion of data types and syntax.

The method for porting algorithms is also incomplete. This is due to two factors. Firstly, time constraints meant that many aspects and

features of DAP-Fortran could not be explored. Secondly, and more importantly, it became apparent through the investigation itself that the method in development was no longer the focus of the project, and the porting of code from DAP-Fortran to Handel-C was in fact destroying the important properties of the algorithms.

If the project were repeated with the strict necessity to produce a porting methodology, regardless of the effect, a complete syntax listing for DAP-Fortran would need to be found or reverse engineered. At that point, the structures, or a limited subset thereof, could be converted one by one to Handel-C. Within this context, a set of algorithms would need to be found which covered the full set of potential data structures and the prototyping process followed to exhaustion. I do not believe that such a process would have been beneficial in the light of conclusions made in this project.

6.4 Project Review

The majority of time dedicated to this project was taken up by research into the DAP and DAP-Fortran. It was known from the outset that material for these would be difficult to obtain, but once material started coming to light, research led to the discovery of more and more studies on the uses of the DAP. The amount of time dedicated to this part of the project resulted in a relatively short period of time left for investigation. However, as the relative importance of these two areas changed throughout the duration of the project, the flexible methodology employed allowed for priorities to be shifted.

While this reduced period of investigation into the porting itself meant that only three prototype attempts were made, I believe that sacrifice was a positive one. The goal to provide a way of retaining important knowledge gained by research on the DAP would be empty if the original research was lost. By finding and analysing dozens of research papers and articles on the DAP, I was able to develop a rich picture of the reasons for the DAP's development, the benefits it provided to its users, and its diverse scope of application. I was also able to form an understanding of why, despite its glowing reviews, satisfied customers, and cost effectiveness, it never became a widely used architecture. With hindsight, I would not have reduced the effort spent on this research. Instead, based on the research done, I would have changed the focus of the project to be a study of how parallel programming is done today

and whether the DAP and DAP-Fortran could be used as a study tool, to enable students to ”think parallel”.

In terms of what was done for the investigation, I believe the method employed was a sound one. Through the porting of each algorithm to Handel-C, new steps were identified and new avenues of investigation were uncovered. While it would have been preferred that more algorithms with different DAP-Fortran features be studied, it was already apparent that the process of porting the algorithms was producing results counter to the intentions of the project. Little benefit is expected to have resulted from further attempts.

6.5 Future Work

The investigations conducted during this project have lead to the need for future work. The conclusion that Handel-C is not an appropriate language to emulate DAP-Fortran algorithms does not discount the use of other languages for the FPGA. A future study should be conducted into the possibility of porting algorithms for the DAP to one of the FPGA's other high level programming languages.

In addition to this, we may also consider that the way in which DAP-Fortran expresses parallelism is worth preserving. If so, we may decide to cut out the step of porting the algorithm and work could be carried out to produce a method for compiling the DAP-Fortran code directly onto an FPGA.

It may also be useful to investigate whether DAP-Fortran's shorthand method of describing parallelism could be of use in other programming languages. This could be expressed by the introduction of macro libraries.

Finally, the original research on the DAP highlighted DAP-Fortran's use as an example of how to use parallelism. The research also showed that it was used this way not only at the beginning of its life, but on into the 1990s. This raises the question as to how parallelism is approached today and whether the DAP and DAP-Fortran would be effective as tools for introducing students to parallel thinking.

Bibliography

Web Archive. Internet archive wayback machine, 2007. URL http://web.archive.org/. Last accessed: 14 April, 2007.

N.R. Arnot, G.G. Wilkinson, and R.E. Burge. Applications of the ICL–DAP for two–dimentional image processing. *Computer Physics Communications*, 26:455–457, 1982.

Neil Audsley. Embedded systems lecture notes, 2006. URL http://www-course.cs.york.ac.uk/csw/MyLectures/. Last accessed: 28 May, 2007.

K.E. Batcher. The STARAN computer. In C.R. Jesshope and R.W. Hockney, editors, *Infotech State of the Art Report: Supercomputers*, volume 2, pages 33–50, Maidenhead, 1979. Infotech International Ltd.

D. Bhagavathi, W.M. Denny, C. Grosch, P.J. Looges, and S. Olariu. Sorting and merging on the DAP. In *ACM-SE 30: Proceedings of the 30th annual Southeast regional conference*, pages 93–99, New York, NY, USA, 1992. ACM Press.

Matthew Bowen. *Handel-C Language Reference Manual*. Embedded Solutions Limited, 2.1 edition, 1998.

James M. Boyle, Maurice Clint, Stephen Fitzpatrick, and Terence J. Harmer. Deriving DAP implementations of numerical mathematical software through automated program transformation. Technical Report JMB.MC.SFTJH, The Queen's University of Belfast, Jul 1992.

Graham R. Brookes and Andrew J. Stewart. *Introduction to occam 2 on the Transputer*. Macmillan, 1989.

Stephen Brown and Jonathan Rose. Architecture of FPGAs and CPLDs: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996.

Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field Programmable Gate Arrays*. Kluwer Academic Publishers, 1st edition, 1992.

Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. Wiley-Interscience, 1st edition, 1995.

Celoxica. *Handel-C Advanced Optimization*. Celoxica Limited, 1 edition, 2003.

M. Clint, C. Holt, R.H. Perrott, and A. Stewart. A DAP fortran subroutine for the eigensolution of real symmetric matrices. *The Computer Journal: Algorithms Supplement*, 28(3):340–342, 1985.

G. Feierbach and D. Stevenson. The ILLIAC IV. In C.R. Jesshope and R.W. Hockney, editors, *Infotech State of the Art Report: Supercomputers*, volume 2, pages 77–92, Maidenhead, 1979. Infotech International Ltd.

S.I. Feldman, D.M. Gay, M.W. Maimone, and N.L. Schryer. Fortran to C converter, 1990. URL http://netlib.sandia.gov/f2c/. Last accessed: 01 May 2007.

Stephen Fitzpatrick, A. Stewart, Maurice Clint, and James M. Boyle. An algebra for deriving efficient implementations for an array processor parallel computer from functional specifications. Technical Report JF.AS.MC.JMB, The Queen's University of Belfast, Jun 1995.

Stephen Fitzpatrick, Maurice Clint, Terence J. Harmer, and P. Kilpatrick. The tailoring of abstract functional specifications of numerical algorithms for sparse data structures through automated program derivation and transformation. *The Computer Journal*, 39(2):145–168, 1996.

Stephen Fitzpatrick, Terence J. Harmer, Alan Stewart, Maurice Clint, and James M. Boyle. The automated transformation of abstract specifications of numerical algorithms into efficient array processor implementations. *Science of Computer Programming*, 28(1):1–41, 1997.

P.M. Flanders. Fortran extensions for a highly parallel processor. In C.R. Jesshope and R.W. Hockney, editors, *Infotech State of the Art Report: Supercomputers*, volume 2, pages 117–134, Maidenhead, 1979. Infotech International Ltd.

P.M. Flanders, D.J. Hunt, S.F. Reddaway, and D. Parkinson. Efficient high speed computing with the distributed array processor. In *High–*

Speed Computer Algorithm Organisation, pages 113–128, London, 1977. Academic Press.

R.W. Gostick. Supercomputer diagnostics. In C.R. Jesshope and R.W. Hockney, editors, *Infotech State of the Art Report: Supercomputers*, volume 2, pages 135–146, Maidenhead, 1979. Infotech International Ltd.

Richard L. Hellier. DAP implementation of the WZ algorithm. *Computer Physics Communications*, 26:321–323, 1982.

C.A.R. Hoare. *Inmos Limited occam 2 Reference Manual*. Prentice Hall International Series in Computer Science, 1988.

R.W. Hockney. Characterization of parallel computers and algorithms. *Computer Physics Communications*, 26:285–291, 1982.

D.J. Hunt. Application techniques for parallel hardware. In C.R. Jesshope and R.W. Hockney, editors, *Infotech State of the Art Report: Supercomputers*, volume 2, pages 205–220, Maidenhead, 1979. Infotech International Ltd.

Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing*. McGraw Hill, international edition, 1985.

ICL. *DAP : Introduction to Fortran Programming*. International Computers Limited, 1st edition, December 1978. Technical Publication 6755.

G.A.P. Kindervater and H.W.J.M. Trienekens. Experiments with parallel algorithms for combinatorial problems. *European Journal of Operational Research*, 33:65–81, 1988.

E.J. Kontoghiorghes. Ordinary linear model estimation on a massively parallel SIMD computer. *Concurrency: Practice and Experience*, 11(7): 323–341, 1999.

A.J Krzeczkowski, E.A. Smith, and T. Gethin. Seismic migration using the ICL distributed array processor. *Computer Physics Communications*, 26:447–453, 1982.

Heather M. Liddell and G.S.J. Bowgen. The DAP subroutine library. *Computer Physics Communications*, 26:311–315, 1982.

J. Martin. *Rapid Application Development*. Macmillan, New York, 1991.

D. Parkinson. Using the ICL DAP. *Computer Physics Communications*, 26: 227–232, 1982.

D. Parkinson. The distributed array processor (DAP). *Computer Physics Communications*, 28:325–336, 1983.

D. Parkinson. *Massively Parallel Computing with the DAP*. MIT Press, 1990.

Karen Parnell and Nick Mehta. *Programmable Logic Design Quick Start Handbook*. Xilinx, January 2002.

Parallel Computer Centre PCC. DAP course info, 2007. URL http://www.pcc.qub.ac.uk/tec/courses/dap/dap.html. Last accessed: 28 May, 2007.

R.H. Perrott and D.K. Stevenson. Considerations for the design of array processing languages. *Software – Practice and Experience*, 11:683–688, 1981.

R.H. Perrott and A. Zarea-Aliabadi. Supercomputer languages. *ACM Computer Survey*, 18(1):5–22, 1986.

Dick Pountain and David May. *A Tutorial Introduction to occam Programming*. BSP Professional Books, 2nd edition, March 1988.

Edie M. Rasmussen and Peter Willett. Non-hierarchic document clustering using the ICL distributed array processor. In *SIGIR 1987: Proceedings of the 10th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 132–139, New York, NY, USA, 1987. ACM Press.

S.F. Reddaway. DAP–a distributed array processor. In *ISCA 1973: Proceedings of the 1st annual symposium on Computer architecture*, pages 61–65, New York, NY, USA, 1973. ACM Press.

S.F. Reddaway. The DAP approach. In C.R. Jesshope and R.W. Hockney, editors, *Infotech State of the Art Report: Supercomputers*, volume 2, pages 309–330, Maidenhead, 1979. Infotech International Ltd.

A. Stewart, C. Holt, M. Clint, and R.H. Perrott. A DAP Fortran subroutine for thinning digitised pictures. *The Computer Journal: Algorithms Supplement*, 34(4):377–379, 1991.

R. Toral. Dynamical properties of a new fast algorithm for the simulation of the Ising model. *Journal of Physics A: Mathematical and General*, 21: 315–320, 1988.

D.J. Wallace. Scientific computation on SIMD and MIMD machines. *Scientific applications of multiprocessors*, pages 125–142, 1989.

S.J. Webb, J.J. McKeown, and D.J. Hunt. The solution of linear equations on a SIMD computer using a parallel iterative algorithm. *Computer Physics Communications*, 26:325–329, 1982.

Charles Wetherell. Design considerations for array processing languages. *Software – Practice and Experience*, 10:265–271, 1980.

G.G. Wilkinson and R.E. Burge. Using the ICL-DAP for satellite climatology. *Computer Physics Communications*, 26:469–471, 1982.

M.C. Wunderlich. Implementing the continued fraction factoring algorithm on parallel machines. *Mathematics of Computation*, 44(169): 251–260, January 1985.

Part IV

Appendices

A DAP-Fortran Code: The Contour Problem

Taken from (ICL, 1978).

```
      CHARACTER MATRIX FUNCTION CONTOUR(HEIGHT,LEVEL,NUMBER_OF_LEVELS)
C
C       GIVE A 64X64 MATRIX OF HEIGHTS, THE PROBLEM IS TO
C       DIVIDE IT INTO REGIONS BETWEEN THE GIVEN LEVELS.
C       THE MAXIMUM NUMBER OF LEVELS CATERED FOR IS 25.
C
C       THE SOLUTION IS EFFECTED BY SHADING EACH REGION,
C       THEN BLANKING OUT ALL BUT THE UPPER EDGE OF EACH
C       REGION, LEAVING ROUGH CONTOUR LINES.
C
      INTEGER REGION(,),NUMBER_OF_LEVELS,HEIGHT(,),LEVEL()
      LOGICAL SELECTED(,),EDGE(,),FREE(,)
      CHARACTER CODE() /*ABCDEFGHIJKLMNOPQRSTUVWXYZ*/
C
C       TEST FOR VALID NUMBER OF LEVELS
C
      IF(NUMBER_OF_LEVELS.LT.1.OR.NUMBER_OF_LEVELS.GT.25) ERROR 1
C
C       STAGE 1: SHADE OUT EACH REGION WITH A DIFFERENT
C       CHARACTER FROM CODE, USING A LOGICAL ASSIGNMENT
C       MASK TO SELECT THE HEIGHTS AT EACH LEVEL. THE
C       LEVELS ARE IN ASCENDING ORDER.
C
      FREE=.TRUE.
      DO 1 K=1,NUMBER_OF_LEVELS
C
C       SET UP A MASK FOR FREE SQUARES AT LEVEL(K)
C       (OR LOWER)
C
      SELECTED=FREE.AND.HEIGHT.LT.LEVEL(K)
C
C       USE THIS MASK TO SHADE OUT THE RELEVANT AREAS.
C
      REGION(SELECTED)=K
      CONTOUR(SELECTED)=CODE(K)
    1 FREE(SELECTED)=.FALSE.
C
```

```
38  C          SET ANY REMAINING POSITIONS TO A DEFAULT CODE.
39  C
40            REGION(FREE)=NUMBER_OF_LEVELS+1
41            CONTOUR(FREE)=CODE(NUMBER_OF_LEVELS+1)
42  C
43  C          STAGE 2: SELECT THE UPPER EDGE OF EACH REGION.
44  C
45            EDGE=REGION.LT.REGION(,+).OR.REGION.LT.REGION(,-)
46           1.OR.REGION.LT.REGION(+,).REGION.LT.REGION(-,)
47  C
48  C          NOTE THAT THE DEFAULT GEOMETRY PLAN APPLIES TO
49  C          THESE SHIFTS, ENSURING THAT SPURIOUS CONTOURS ARE
50  C          NOT INTRODUCED AT THE EDGE, WHERE ZERO IS SHIFTED
51  C          IN.
52  C
53  C          STAGE 3: BLANK OUT THOSE PARTS OF CONTOUR THAT
54  C          ARE NOT ON AN UPPER EDGE.
55  C
56            CONTOUR(.NOT.EDGE)=' '
57            RETURN
58            END
```

Listing A.1: The Contour Problem

B Handel-C Code: The Contour Problem

```
1  channel identity[64][64][4] \\ values to be read by neighbours
2  int     neighbours[64][64][4] \\ east, west, north, south neighbours
3
4  // All writes to channels and reads from channels can be done in
5  // parallel.
6  par {
7
8    // This par writes the number in the cell to its four output
9    // channels to be used by the cells to their north, south,
10   // east and west.
11   // In the case of the edges and corners of the matrix, these
12   // will not be read, so a process will be needed which reads
13   // from these unused channels to prevent deadlock.
14   par (i=0; i<=63; i++)
15   { par (j=0; j<=63; j++)
16     { par (k=0; k<=3; k++)
17         { identity[i][j][k] ! region[i][j]; } }
18   }
19
20   // Read the identity of the four neighbours of each cell that
21   // that is not an edge or a corner
22   par (i=1; i<=62; i++)
23   { par (j=1; j<=62; j++)
24     { par (k=0; k<=3; k++)
25       { identity[i-1][j][0] ? neighbours[i][j][0];
26         identity[i+1][j][1] ? neighbours[i][j][1];
27         identity[i][j-1][2] ? neighbours[i][j][2];
28         identity[i][j+1][3] ? neighbours[i][j][3]; }
29     }
30   }
31
32   // Read the identity of the three neighbours of those cells
33   // that are edges of the matrix, but not corners.
34   par (j=1; j<=62; j++)
35     { identity[1][j][1] ? neighbours[0][j][1]
36       identity[0][j-1][2] ? neighbours[0][j][2];
37       identity[0][j+1][3] ? neighbours[0][j][3];
38     }
```

```
39
40    par (j=1; j<=62; j++)
41      { identity[62][j][0] ? neighbours[63][j][0]
42        identity[63][j-1][2] ? neighbours[63][j][2];
43        identity[63][j+1][3] ? neighbours[63][j][3];
44      }
45
46    par (i=1; i<=62; i++)
47      { identity[i-1][0][0] ? neighbours[i][0][0]
48        identity[i+1][0][1] ? neighbours[i][0][1];
49        identity[i][1][3] ? neighbours[i][0][3];
50      }
51
52    par (i=1; i<=62; i++)
53      { identity[i-1][63][0] ? neighbours[i][63][0]
54        identity[i+1][63][1] ? neighbours[i][63][1];
55        identity[i][62][2] ? neighbours[i][63][2];
56      }
57
58    // Read the identities of the two neighbours of the four
59    // corners of the matrix
60    identity[1][0][1] ? neighbours[0][0][1]
61    identity[0][1][3] ? neighbours[0][0][3]
62
63    identity[62][0][0] ? neighbours[63][0][0];
64    identity[63][1][3] ? neighbours[63][0][3];
65
66    identity[1][63][1] ? neighbours[0][63][1];
67    identity[0][62][2] ? neighbours[0][63][2];
68
69    identity[62][63][0] ? neighbours[63][63][0];
70    identity[63][62][2] ? neighbours[63][63][2];
71
72    // Finding the cells which are edges between regions, writing
73    // 1 (true) to cells which are to be edges
74    par (i=1; i<=62; i++)
75    { par (j=1; j<=62; j++)
76      { if ( (region[i][j]<neighbour[i][j][0]) ||
77             (region[i][j]<neighbour[i][j][1]) ||
78             (region[i][j]<neighbour[i][j][2]) ||
79             (region[i][j]<neighbour[i][j][3]) )
80        { edge[i][j] = 1;
81          else edge[i][j] = 0; } }
82    }
83
84
85    // For each of the cells that are neither an edge or corner
86    // of the matrix, find whether it is at the upper boundary
87    // between regions
```

```
88    par (j=1; j<=62; j++)
89    { if ( (region[0][j]<neighbour[0][j][0]) ||
90           (region[0][j]<neighbour[0][j][1]) ||
91           (region[0][j]<neighbour[0][j][3]) )
92         { edge[0][j] = 1;
93           else edge[0][j] = 0; }
94    }
95
96    // For each of the cells that are at the edge of the matrix
97    // find whether it is at the upper boundary between regions
98    par (j=1; j<=62; j++)
99    { if ( (region[63][j]<neighbour[0][j][0]) ||
100          (region[63][j]<neighbour[0][j][1]) ||
101          (region[63][j]<neighbour[0][j][2]) )
102       { edge[63][j] = 1;
103         else edge[63][j] = 0; }
104   }
105
106   par (i=1; i<=62; i++)
107   { if ( (region[i][0]<neighbour[i][0][1]) ||
108          (region[i][0]<neighbour[i][0][2]) ||
109          (region[i][0]<neighbour[i][0][3]) )
110       { edge[i][0] = 1;
111         else edge[i][0] = 0; }
112   }
113
114   par (i=1; i<=62; i++)
115   { if ( (region[i][63]<neighbour[i][63][0]) ||
116          (region[i][63]<neighbour[i][63][2]) ||
117          (region[i][63]<neighbour[i][63][3]) )
118       { edge[i][63] = 1;
119         else edge[i][63] = 0; }
120   }
121
122   // For each of the corners of the matrix, find whether it is
123   // an upper boundary between regions.
124   if ( (region[0][0]<neighbour[0][0][1]) ||
125        (region[0][0]<neighbour[0][0][3]) )
126     { edge[0][0] = 1;
127       else edge[0][0] = 0; }
128
129   if ( (region[0][63]<neighbour[0][63][0]) ||
130        (region[0][63]<neighbour[0][63][3]) )
131     { edge[0][63] = 1;
132       else edge[0][63] = 0; }
133
134   if ( (region[63][0]<neighbour[63][0][1]) ||
135        (region[63][0]<neighbour[63][0][2]) )
136     { edge[63][0] = 1;
```

```
137         else edge[63][0] = 0; }
138
139     if ( (region[63][63]<neighbour[63][63][0]) ||
140          (region[63][63]<neighbour[63][63][2]) )
141        { edge[63][63] = 1;
142          else edge[63][63] = 0; }
143 }
144
145 // Based on the edge matrix, writing 0 to the cells which are
146 // not edges
147 par (i=1; i<=62; i++)
148 { par (j=1; j<=62; j++)
149     { if !(edge[i][j]) contour[i][j] = 0; }
150 }
```

Listing B.1: Version 1b: A Direct Conversion of Stage 2 of Example 3 to Handel-C, using channels

C Process Steps for Porting DAP-Fortran to Handel-C

- All non-integer types must be mapped to integers. Suggestions for four of the mappings can be found in Appendix D.
- The DAP-Fortran feature of masks used in the form

  ```
  A(conditional) = expression
  ```

  can be converted to Handel-C code of the form

  ```
  if (conditional) {
     a[i][j] = expression }
  ```

  where `i` and `j` index elements in the matrix where the conditional is true.
- Nested repeated pars should be used to replicate the parallel assignments performed in DAP-Fortran. For example, the DAP-Fortran Matrix assignment

  ```
  A = 1
  ```

  is replicated using the Handel-C code

  ```
  par (i=0; i<=63; i++)
  { par (j=0; j<=63; j++)
    { a[i][j] = 1; }
  }
  ```

- Where the assignments to a matrix are of the form

  ```
  A = expression1
  A(conditonal) = expression2
  ```

where the matrix as a whole is assigned one value and then components selectively reassigned to another values, the two steps may be combined into one statement.

```
if (conditional) {
   a[i][j] = expression2
  else
   a[i][j] = expression1 }
```

- Where the algorithm is such that its application to the individual elements of the matrices in use are independent of the other elements, the process of sequential parallel actions may be converted to a parallel block of sequential processes, as seen in Figure 2.7.
- The DO loop in DAP-Fortran may be replaced with a for loop in Handel-C. This conversion would change code of the form

```
      DO 10 K = 1,20
      A = 1
   10 B = 2
```

to become of the form

```
for (k=1; k<=20; k++)
{  a = 1;
   b = 2; }
```

where A and B are scalars.

- Where the elements of a row of column of a matrix are the be accessed individually using MATR or MATC, the step of creating intermediate representations of these rows and columns as matrices may be bypassed and the rows and columns referred to directly.
- Where offset referencing is used, the requirement of the programmer may dictate the translation.
  - If readability and simplicity of code is preferred, copies of the target matrix should be made and used for the comparisons.
  - If hardware efficiency and links to the original algorithm are required, channels should be used to communicate data between array elements.

D Mapping

DAP-Fortran	Handel-C
.TRUE. .FALSE.	1 0
.NOT. .AND. .OR. .EQ. .NE. .LT. .GT. .LTE. .GTE.	! && || == != < > <= >=
INTEGER REAL CHARACTER LOGICAL	signed int 32; unsigned int 32; unsigned int 8; unsigned int 1;
INTEGER AGE/25/	signed int 32 age = 25;
A( )	a[64]
B(,)	b[64][64]
DO 10 K = 1, 15 10 *statement*	for(n=1; n<=15; n++) { *statement* }
C *comment*	\\ *comment*

Table D.1: Summary of some simple mappings from DAP-Fortran to Handel-C