INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

CONCORDIA PARALLEL C: DESIGN AND IMPLEMENTATION

Ai Kong

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

September 2000

© Ai Kong, 2000

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-54335-8

Canada

// Abstract

Concordia Parallel C: Design and Implementation

Ai Kong

This thesis focuses on the design and implementation of Concordia Parallel C (CPC) and its compiler Concordia Parallel C Compiler (CPCC).

The Concordia Parallel Programming Environment (CPPE) is a novel parallel programing environment supporting virtual-architecture parallel programming paradigm, program and library development, simulated execution, syntax/semantics/performance debugging, and simulation of various multiprocessors and multicomputers. A parallel program written in CPC will be compiled by CPCC into a virtual code version, then interpreted by Concordia Parallel Systems Simulator (CPSS).

We extend the C language with features supporting parallel computing as well as selected features of C++ to facilitate parallel library writing. We adopt Abstract Syntax Trees (AST) as the internal code representation of our CPCC front-end, from which various program transformations can be performed, and code for various target parallel systems can be generated. The current back-end generates virtual code for CPSS.

Acknowledgments

I would like to take this opportunity to express my deepest gratitude to my supervisor, Dr. Lixin Tao, for not only his constant guidance and encouragement, but also his technical support to help me work from home.

Sincere thanks to the members of our CPPE research team, Dr. Lixin Tao, the team leader; Hosseine Hassan(CPCC), Hoang Uyen Trang Nguyen and Thien Bui(CPSS), and Jing Zhang (Visual performance debugger) who each contributed to the implementation of our programming environment.

I'm grateful to the professors and staffs in Computer Science department at Concordia for the wonderful courses and services. I would especially like to thank Ms. Halina Monkiewicz, who were always friendly and prompt in providing assistance.

My good friends and partners are difficult not to mention. I am thankful to Jing Zhang, Mao Zheng, Ji Lu, Fan Cheng, Lisha Kang, Lin Chen, Jaya Narain, Ximin Wang for giving me encouragement, advice and assistance.

Finally, I have to thank my family, for their understanding and continuous encouragement and support. My husband, Jiawei, was always there to provide various kinds of suppoprt, technical or non-technical. My newborn daughter, Rosa, was so cooperative. I would also like to dedicate this work to the memories of my father and to my mother, their expectation encouraged me never to give up.

Contents

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Motivation 4

1.2 Design Objectives 7

1.3 Thesis Contributions and Our Approach 8

1.4 Thesis Outline 10

2 Survey 12

2.1 Parallel Programming Paradigms 12

2.2 Parallel Programming Models 13

2.3 Parallel Programming Languages 16

3 The CPC Language Design 20

3.1 Virtual Architecture Approach 21

3.2 Differences Between the Sequential Part of CPC and ANSI C 24

3.2.1 Function Definition 25

3.2.2 Names 26

3.2.3 Call By Reference 26

3.2.4 Comments 26

3.3 Parallel Features of the CPC Language 26

3.3.1 Process Creation and Termination 27

3.3.2 Process Communication Via Channel Variables 34

3.3.3 Virtual Architecture Definition 39

3.3.4 Mapping Processes to Virtual Processors 40
3.3.5 CPC Builtin Functions 41

4 CPCC Front End **43**
4.1 Lexical Analysis 44
4.2 Syntax Analysis 44
4.3 Symbol Table 45
4.3.1 Symbol Table Entries 46
4.4 Concept of CPC AST 46
4.4.1 Nodes in CPC AST 47
4.4.2 Graphical Representation of Nodes 54
4.5 Data Declarations in CPC AST 55
4.5.1 Scalar Types and Pointers 55
4.5.2 Arrays 57
4.5.3 Aggregate Type – Structures and Unions 57
4.5.4 Enumerations 60
4.5.5 Typedefs 62
4.5.6 Physical vs. Virtual Architecture 62
4.5.7 Channels 62
4.6 Expressions in CPC AST 63
4.7 Statements in CPC AST 65
4.7.1 Compound Statement 66
4.7.2 Process Creation and Mapping Statements 68
4.8 Function Definitions in CPC AST 68
4.9 CPC AST Type Check—Semantic Analysis 70

5 Code Generation **76**
5.1 Tasks of CPCC Backend 76
5.2 Execution Environment 77
5.3 Virtual Codes 80
5.4 Storage Allocation 81
5.5 Statement Translation 83
5.5.1 Process Creation 84
5.5.2 Process Communication — Channel expression 87

5.5.3 Virtual Architecture 88
5.6 A Complete Example 88

6 CPC preprocessor 89

7 Conclusion 94

A CPC Builtin Functions 97

B CPC Yacc Grammer 101

C CPC language manual 110
C.1 Multiprocessors VS. Multicomputers 110
C.2 The CPC language features 111
C.2.1 Process and Process communication 111
C.2.2 Virtual Architecture and Two Level Mapping 112
C.3 *Notation* 113
C.4 Function definition 114
C.4.1 Syntax 114
C.4.2 Example 115
C.5 Names 115
C.6 Comments 116
C.6.1 Example 116
C.7 Call by reference 116
C.7.1 Example 117
C.8 Inclusion of additional CPC source files 117
C.8.1 Syntax 117
C.8.2 Examples 118
C.9 Run time library 118
C.10 Process creation 118
C.10.1 *fork* Statement 119
C.10.2 *join* statement 121
C.10.3 *forall* Statement 122
C.11 Process Communication via Channel Variables 127
C.11.1 Declarations of Channel Variables 128

C.11.2 Binding Channel Variables to New Processes 130
C.11.3 Read and Write on Channel Variables 133
C.12 Parallel Architecture Definition 136
C.12.1 Syntax 137
C.12.2 Examples 138
C.13 Mapping Processes to Virtual Processors 139
C.13.1 Syntax 139
C.13.2 Examples 139
C.14 A complex CPC example: Matrix Multiplication 140

D A Complete Example **142**
D.1 CPC source program 142
D.2 CPCC frontend generated file 143
D.3 CPC backend generated file 154

List of Figures

1 CPPE overview . . . 4
2 Mapping example . . . 24
3 Graphical representation of nodes and values . . . 54
4 Scalar declarations and pointers . . . 56
5 Array Declarations . . . 57
6 Structure and union Declarations . . . 59
7 Enumeration Declarations . . . 61
8 Constant Expressions . . . 64
9 Compound Statement . . . 67
10 Fork Statement . . . 69
11 Forall Statement . . . 75
12 A stack frame . . . 79
13 Traverse tree . . . 82
14 traverse symbol . . . 83
15 Trace type . . . 84
16 Storage allocation . . . 85
17 Nested Function Definition . . . 115
18 CPC comment . . . 116
19 Call By Reference . . . 117
20 Include Statement . . . 118
21 Fork Statement . . . 120
22 JOIN statement . . . 121
23 Another example on JOIN Statement . . . 122
24 FORALL Statement . . . 124
25 Nested FORALL Statement . . . 125
26 Forall Index . . . 125

27 Matrix Multiplication 128
28 Channel Declaration 129
29 Complex Channel Declaration 130
30 Array of Channel 131
31 Channel Binding 132
32 More examples on binding 132
33 Channel Write 133
34 Channel Read 134
35 Channel Variables 134
36 Channel Empty Test 135
37 InsertionSort 136
38 Architecture Declaration 138
39 Architecture Dimension 138
40 Mapping 140
41 Matrix Multiplication 141

List of Tables

1 Mapping table for process to physical processor. 23
2 Mapping table for physical processor to processes. 24
3 Identifiers and their returned tokens. 44
4 Storage Class 63

Chapter 1

Introduction

Parallel programming is in general difficult and error-prone.

The basic concept behind the parallel computer is to simply have more than one processor in the same computer. Parallel computers may have as few as 10 processors or as many as 50,000. The key feature that makes them *parallel* computers is that all the processors are capable of operating at the same time[1].

For many processors to be able to work together on the same computational problem, they must be able to share data and communicate with each other. There are currently two major types of parallel computers: shared memory multiprocessors and message-passing multicomputers.

In shared memory multiprocessors, all the individual processors have access to a common shared memory, allowing the shared use of various data values and data structures stored in the memory.

In message-passing multicomputers, each processor has its own local memory, and processors share data by passing messages to each other through some types of processor communication networks. A processor has direct access only to its own local memory module, and not to the memory modules attached to other processors. However, any processor can read data values from its own local memory and send that data to any other processor. Therefore, the data can be freely shared and exchanged between the processors when desired.

There is a wide variety of different types of communication network topologies that have been developed for multicomputers, such as hypercube, mesh, ring, and torus. The goal of these topologies is to try to reduce the cost and complexity of the

network, while supporting rapid communications between processors.

A **partitionable** parallel system is based on a fixed topology from which a partition is granted to the user upon request. Size of this partition depends on the system availability at the time of the request. Systems in which the topology may also change statically or dynamically are called **reconfigurable** systems.

The programming of multiprocessors is actually much easier because it more closely resembles the programming of uniprocessor systems, with which we are already familiar. Multicomputer programming is more complex than multiprocessor programming because the program must do explicit message-passing between processors. Partitionable and reconfigurable systems make programming on these systems even more difficult because the programmer is not aware of the system size and topology at programming time. In these systems a user's request for a partition is submitted after an executable image of the parallel program is built. The granted partition may differ between different runs of the same program depending on the requested size, system load, the maximum system size, and the scheduling policy.

Users verify the correctness and observe the performance of parallel programs through trial runs. However, the performance of a parallel program is affected by the system topology and system size. Each application may have many possible algorithms, and each algorithm may be more suitable for a particular parallel architecture than for others. Even within the same parallel platform, the performance of a parallel program may vary for different partitions. It may be high on small size systems, but degrade tremendously as system size increases. Therefore, users expect to evaluate parallel architectures and the performance of parallel programs on these architectures efficiently. In other words, they expect their programs to be *portable* not only across different parallel platforms and parallel architectures, but also across different partitions within the same parallel platform. Otherwise, it would be inefficient to optimize the parallel programs' performance.

Portability means that an application is portable across a class of environments to the degree that the effort required to transport and adapt it to a new environment in the class is less than the effort of redevelopment. In this thesis it has two levels of meanings:

- **Source code level portability**

  This kind of portability means that the source code of a program is portable

across different systems to the degree that the effort required to port the program to the new system and run it on the new system is less than the effort to redevelop the program and rerun it on the new system.

- **Executable code level portability**

  The executable code level portability is useful for partitionable and reconfigurable parallel system.

  This portability means that the executable code for a parallel application is portable across different partitions within the same platform to the degree that the effort required to transport and execute the executable code on a new partition is less than the effort of recompiling the parallel application, relinking, and reexecuting.

CPPE (Concordia Parallel Programming Environment) is being studied and developed in our group with the aim to provide users with simulated parallel systems for developing portable parallel programs, evaluate parallel architectures, and optimize performance of parallel programs on these architectures.

CPPE has three main components: CPC (Concordia Parallel C) language, CPCC (Concordia Parallel C Compiler), and the CPSS (Concordia Parallel Systems Simulator).

CPC is an extended version of ANSI C, which supports virtual architectures, process creation, and high-level channel abstractions of message passing. A **virtual architecture** in CPC is the architecture most natural and efficient to program domain. Its processors are called **virtual processors**. CPCC is the compiler for CPC language in CPPE. It is divided into the frontend and backend. CPCC frontend performs lexical analysis, syntax analysis, semantic analysis on the CPC programs, and transformation of the CPC source code to an intermediate representation — an Abstract Syntax Tree (CPC AST); CPCC backend traverses the CPC AST built by the frontend, allocate storage to all the variables and generate code for a target architecture or generate virtual code for our CPSS. CPSS performs instruction level simulation of various parallel systems and their communications subsystems. We illustrate the structure of CPPE in Figure 1.

The research effort of this thesis focuses on the design and implementation of CPC (Concordia Parallel C) language, aiming to provide a necessary parallel language in

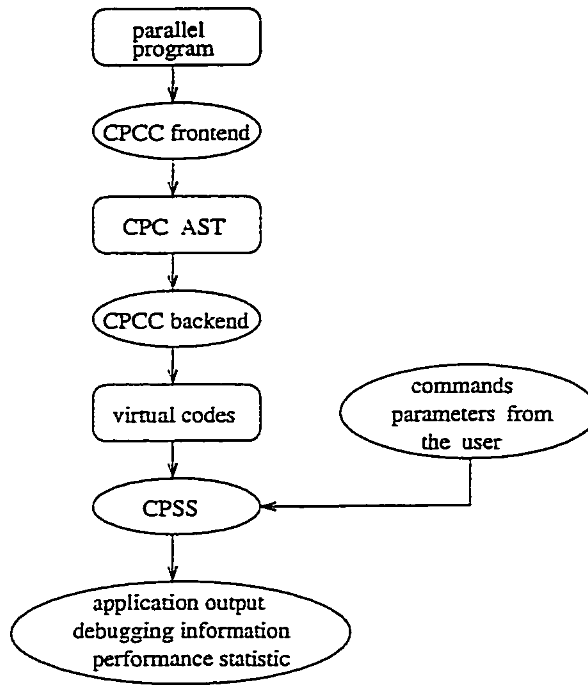

Figure 1: CPPE overview

which the CPPE users can develop portable parallel programs.

In this chapter, we will discuss the motivation of this research, our design objectives, our approach to meet the objectives and the thesis contributions. Finally, we will present the thesis outline.

1.1 Motivation

Program development is costly without the added difficulty of parallel programming.

There are many factors that affect the performance of a parallel program. In order to improve parallel programs' performance, reduce the cost in detecting and overcoming performance bottlenecks, users expect their parallel programs to be *portable*, so that they can evaluate the parallel program performance effectively. However, the parallel programs for the real parallel system may not be *portable* due to any of the following reasons:

1. Parallel programs lack *source code level portability*.

In sequential environments, the *source code level portability* is not difficult to achieve, especially for programs written in a popular high-level sequential language, such as C language. When a new sequential machine appears, if a compiler of the language can be implemented for the new machine to translate the programs written in the specific high-level language into the new machine instructions, all of the programs in the high-level language will be portable to the new machine. Therefore, if each sequential machine implements its own compiler for the specific high-level language, then all the software written in this language will be portable across these sequential machines.

However, *source code level portability* is not trivial for parallel environments.

In the development of many parallel systems, the effort has tended to focus on the hardware, leaving the software under-developed and primitive. The result is a tight coupling among the machine, the compiler, and the associated language. In other words, the machine is designed for a particular architecture, the language is designed to match the machine capability, and the compiler translates directly from the language syntax to the machine function. The system built by these components contains specific details that help make the system efficient. However a disadvantage is that the parallel program tuned to this system may lose the *source code level portability*.

An example is the Connection Machine CM-2[2] and its C* language[9]: the *where* construct is an excellent match for the SIMD control mechanism in the hardware, but implementing this language abstraction on an MIMD may involve more overhead.

Parallel systems differ from each other in CPU instruction sets, processor sizes, communication network topologies, communication subsystems, hardware synchronization techniques, and whether to support broadcasting. Unlike in sequential environments, it is impossible for a compiler to resolve these complex differences to support the source code level portability.

Since a parallel program is tuned to a specific parallel system and not portable to other parallel systems, to compare performance of a parallel program among several parallel systems, the user has to redevelop the parallel program for each of the systems.

2. Partitionable and reconfigurable parallel systems may not support *executable code level portability.*

   According to Amdahl's laws[1], there is a performance limitation to each parallel application, no matter how many processors are used. Usually, an application program uses only a partition in the partitionable or reconfigurable parallel system.

   Each of the partitionable or reconfigurable machines has a processor manager in its operating system. To run a parallel program on a partitionable or reconfigurable parallel system, the user should first compile and link his program for a partition with desired size and topology, and get the executable code for it. Then the user applies from the processor manager for the desired subsystem to run the executable code. The executable code can be executed only on this subsystem.

   However, the processor manager may grant a different subsystem to the user. The processor manager makes the decision whether to satisfy the user or modify the user's request according to a certain policy. Many factors should be taken into account for this policy, such as optimizing system utilization to reduce the number of idle processors, or optimizing the response time. For example, after a user gets the executable code for an 8*8 mesh, he may then apply for an 8*8 mesh subsystem to run his executable. But, he may be assigned a 5*6 mesh subsystem or an 8*8 mesh subsystem situated in a different position.

   When the granted subsystem mismatches with the subsystem the user requested, the user has to use the following two possible solutions to solve the problem.

   The first solution is to recompile and relink his program to get the executable code for the granted subsystem. However, the system is dynamic. After the user spends some time to recompile and relink his application to get the new executable code for the granted subsystem, the system status may have changed. The user has to reload the executable code and reapply for the granted subsystem. Unfortunately, this time he may be assigned a totally different subsystem. The executable code still cannot run. Therefore, this solution is infeasible.

   Another solution is to reserve the granted subsystem. Recompile and relink his program for this subsystem, get the executable code for this subsystem, run the

executable code on this subsystem, then release this subsystem. In this way, we can achieve the *executable code level portability.* The major disadvantage of this solution is the waste of the expensive parallel system resource. The execution time of the program will be much longer than expected, the performance of the parallel program will be reduced instead of improved, the cost to achieve the *executable code level portability* will overcome the benefit from it.

1.2 Design Objectives

To help users evaluate parallel programs performance efficiently, the most important objectives considered in the design and implementation of the CPC language are as follows.

1. Ease of use: CPC language should be easy to learn, to program; CPC programs should be easy to read, to maintain and to verify.

2. Essential parallel features: Like all the other parallel languages, CPC language must support specification of parallelism, starting and stopping parallel executions, and coordinating the parallel executions.

3. Source code level portability: To get the highest performance of a parallel program, the user has to compare performance of the parallel program for different parallel systems, therefore it is expected that CPC programs have *source code level portability*, so that users do not need to redevelop their programs for a different parallel system.

4. Executable code level portability: To help the user fine-tune his program and optimize the performance effectively, we hope the CPC executable code could run on different subsystems within the same platform, so that the user need not recompile his source code, and the executable code could be portable to other subsystems. In other words, we hope the CPC program to be able to run on a partition with different size and topology without recompilations.

5. Efficiency: Communication among processors is much more expensive than instruction execution. It is desirable that CPC communicating processors can be situated as close as possible to minimize the distance between these processors,

reduce communication overhead and communication latency, reduce the CPC program execution time, and make CPC programs more efficient.

1.3 Thesis Contributions and Our Approach

Contributions of this thesis include CPC language design and implementation.

We have based the CPC language design on the virtual architecture approach. The virtual architecture in CPC is the architecture most natural and efficient for a problem domain. The virtual architecture of an algorithm captures the communication pattern of the algorithm. CPC program is described on a virtual architecture with desired size and topology. At run time, the virtual architecture program will be mapped to the available physical architecture. The virtual architecture information in the CPC program will be used by the CPC compiler and run-time system to perform mapping of the computation to the physical processors. With the mapping, the virtual architecture will be independent from the physical architecture which helps make the CPC programs portable. Suitable mapping of the regular communications can reduce the communication overhead of the program. Virtual architecture algorithm is easy to implement since the programmer can describe the algorithm on the virtual architecture matching that of the algorithm and let the compiler and runtime system perform the necessary translation to account for the mapping of the computation and handle the communication among the virtual processors.

Now let's see how our approach meet our design objectives:

CPC language is based on the popular programming language C and enhanced with new features to support portable parallel programming. The CPC language preserves most existing sequential features of the C language. CPC language will be easy to learn since most of the users are familiar with C language. It will not be difficult for them to learn some essential parallel features in order to program in CPC.

New features of the CPC allow users to express parallelism, to create and terminate processes, to synchronize executions of processes, to communicate among parallel processors via channel variables. Therefore, the CPC language has the **essential parallel features**. The CPC language supports both shared-memory and message-passing programming paradigms.

CPC is a good candidate for explicit parallel programming that can be used to design, express, and implement efficient portable parallel algorithms.

New features of the CPC also allow users to declare virtual architectures and to map parallel processes to virtual processors which in turn will be mapped to physical processors at runtime. The user can choose a virtual architecture with the size and topology that is most natural and efficient to his algorithm. Since the virtual architecture of an algorithm is generally representative of the communication pattern of the algorithm, the CPC program will be easy to describe on the virtual architecture. The declaration is similar to that of a data type. The choice for a virtual architecture depends on the user's domain knowledge.

The virtual architecture approach provides the user with a high-level abstraction of the communication pattern, allowing the user to concentrate on the resolution of their problems without sacrificing performance. CPC virtual architecture algorithm is easy to implement since the virtual architecture allows the user to solve their specific problems in a way that best fits the model of his application. Therefore, the virtual architecture approach will make CPC program easy to write , easy to verify, easy to understand and easy to maintain. That is, CPC language is **easy to use**.

As we mentioned before, to support source code level portability of parallel programs is not trivial. To achieve the CPC programs source code level portability, we divide CPC compiler (CPCC) into frontend and backend. The frontend is architecture independent and is a shared frontend. It will first translate the CPC source code into an abstract syntax tree (CPC AST) which retains all the information, such as type and program structure, virtual architecture and process to virtual processor mapping scheme. It will not be attached to specific system configurations. The backend is architecture dependent. Different architectures should have different backend. In CPPE, we have various backends for different platforms. Since the CPC program is based on virtual architecture which is independent from any physical architectures, the CPC program will not be tuned to a specific physical architecture. Therefore, CPC programs have **source code level portability**.

When a user compiles the CPC source code for a platform, the compiler takes two inputs: virtual topology which the CPC source code is programmed on, physical topology which the executable code will be executed on. After compilation and linking, the executable code will be generated. When the system loads the executable code,

each physical processor will get a mapping table. Physical processors are processors that constitute the physical machine. The mapping table is created by the runtime system according to the mapping function in the mapping library or specified by the user. It contains the information about processor's physical ID and virtual processors which the physical processor should run. As a result, the virtual processors are dynamically mapped to physical processors to execute the code. Therefore, with the support of mapping schemes, virtual architecture become independent from physical architecture, and CPC executable code is portable. In other words, CPC programs have the **executable code level portability**.

Since the virtual topology algorithm is generally representative of the communication pattern of that algorithm, it is desirable to map the neighboring virtual processors as close as possible in the target physical topology. We can either store the optimized mapping functions in the mapping library or the user can design his own mapping functions. The runtime system will use the mapping function selected by the user or by default to map the process to the physical processors. The freedom in the choice of a mapping function allows the CPC user to minimize communication cost among communicating processes, to balance the workload among physical processors and to improve the **efficiency** of the CPC program.

To implement the CPC language, or to design the CPCC, the challenge is the design of the right data structures for intermediate representation which can efficiently hold all the necessary information in the source program and which can be easily transformed to machine code by the compiler backend, so we can easily relate runtime data to source code lines. As we described above, we choose abstract syntax tree as our intermediate representation.

CPC AST is a complete intermediate language which will make program transformation and systematic type checking easier.

1.4 Thesis Outline

This thesis is a description for the design and implementation of CPCC. This first chapter expresses motivation for CPC and CPCC design and provides an overview of the CPCC.

Chapter 2 will survey parallel languages and parallel language compilers.

Chapter 3 will describe the design of CPC, which is based on C, extended by machine-independent parallel constructs, such as `fork` and `forall`. CPCC is an explicit parallel language that can be used for both sequential and parallel programming.

Chapter 4 will describe the design of CPCC front-end and introduce how to create the complete abstract syntax tree, the CPC AST, to keep all the information in CPC source code. In this chapter, the data structures of the CPC AST will be described in more details. We describe the structure of four types of tree nodes, and how to build the ASTs for data declarations, expressions, statements, and functions in CPC.

Chapter 5 will describe the design of CPCC back end. It introduces the virtual codes, the storage allocations for CPC variables and the translations of the CPC statements to virtual codes.

Chapter 6 will describe the design of the CPCC preprocessor.

Finally, concluding remarks are given in Chapter 7.

Chapter 2

Survey

2.1 Parallel Programming Paradigms

A parallel programming paradigm describes the method in which processor nodes are coordinated in executing their tasks and in sharing data when running a parallel program. It usually performs the following tasks:

- Decompose the task into smaller tasks
- Assign the smaller tasks to processors to work on simultaneously
- Coordinate work and communicate when necessary

The parallel paradigm has the following components:

- Environment. Parallel environment often includes the processor, connection topology and memory architecture.

  The processors access data through memory, passing data back and forth between the processors through the connection topology.

  In a high level of abstraction, processors are classified into:

  - SISD - Single Instruction Stream, Single Data Stream
  - SIMD - Single Instruction Stream, Multiple Data Stream
  - MIMD - Multiple Instruction Stream, Multiple Data Stream

The popular connection topologies include hypercube, mesh torrus, ring, and line.

There are two primary memory architectures: shared memory and distributed memory.

- Program. This category includes the parallel language, the programming model, and algorithmic implementation.

  When writing code, the programming model underlies the syntax or language with which the algorithm is implemented.

2.2 Parallel Programming Models

Different programming paradigms trade off between ease of programming and efficiency of programs depending on the target architecture. An algorithm may be easy to describe in one language but more difficult in another. On the other hand, the efficiency of the algorithm depends on how well the programming language matches that of the underlying architecture. A closer match results in higher efficiency, at the expense of more programming effort. A programming paradigm with mismatched language and architecture allows the user to describe the algorithm in a more abstract level but may result in less inefficient execution. There is no simple way to calculate the relative weight of each factor contributing to the efficiency of programming. Effective programming models are expected for ease and efficiency of programming.

Parallel languages are designed to support the programming models. So far, numerous parallel languages [1, 3, 4, 7, 9, 15, 17] have been studied, designed and integrated into parallel environments.

Usually, parallel programming language do not offer direct access to the hardware but establish, as an abstract view of the hardware, a *programming model* which can more or less efficiently be emulated by the hardware. E.g., physical wires may be abstracted by logical channels, as in Occam [15, 16] and CSP [17], or a virtual shared memory may be simulated on top of a distributed memory message passing system. There are several models of parallel programming that can be used in parallel applications. For example:

- Shared Memory Parallel Programming Model

With this programming model, all tasks share a common address space, which they read and write asynchronously. Various mechanisms such as locks and semaphores may be used to control access to the shared memory. Although this programming model is more suitable for multiprocessors, it can be implemented on top of multicomputers. Implementation of this programming model on multicomputers requires analysis by the compiler as to the location of data so that appropriate *send* and *receive* can be inserted in the program for communication.

- Message-Passing Based Parallel Programming Model

  With this programming model, the programmer views the program as a collection of processes with private local variables. The communication among these processes is done using primitives such as send and receive. Message passing programs are more difficult to write, however they can be optimized and can run more efficiently on multicomputers.

  Over the last ten years, substantial progress has been made in casting significant parallel applications into message passing paradigm for distributed memory parallel computers. *CPC* also falls into this model.

- Data Parallel Programming Model

  The main feature of the data parallel programming is that many data items are subject to the same processing. Many data parallel languages have been developed to support this kind of parallelism. Data parallel imperative languages have been designed especially to program SIMD (Single Instruction Multiple Data) computers, e.g., pipelined vector processors or the CM2[2], since the synchronous execution of the instructions is supported by the hardware. Examples of such languages are *HPF,HPF1,Hig93*, *MODULA-2**[7] and *C** [9, 10, 11]. A more relaxed form of data parallel programming is one that doesn't require synchronization at each instruction. This form is called Single Program Multiple Data (SPMD) programming paradigm. In SPMD, synchronization is done only at communication points which are implicit in the algorithm. SPMD programs are suitable for multicomputers since the programming paradigm closely matches the underlying architecture. Examples of SPMD programming language include HPF and CPC.

Independently from the specific paradigms considered, in order to execute a program which exploits parallelism, the programming language must supply the means to:

- Identify parallelism, by recognizing the components of the program execution that will be (potentially) performed by different processors;

  It is custom to separate the approaches to parallel processing into explicit versus implicit parallelism.

  Explicit parallelism is characterized by the presence of explicit constructs in the programming language, aimed at describing (to a certain degree of detail) the way in which the parallel computation will take place. A wide range of solutions exists within this framework. CPC is one of them.

  Implicit Parallelism allows programmers to write their programs without concern about the exploitation of parallelism. Exploitation of parallelism is instead automatically performed by the compiler and/or the runtime system.

  Explicit parallelism allows to code a wide variety of patterns of execution, giving a considerable freedom in the choice of what should be run in parallel and how. Users can manage the parallelism, detect the components of the parallel execution, and guarantee a proper synchronization.

  For example, OCCAM describes parallelism explicitly, on the other hand, HPF, C* identify parallelism by their compilers implicitly.

- Start and stop parallel executions;

  For example, *fork* in Multipascal and CPC can be used to spawn a child process to start parallel execution.

- Coordinate the parallel executions (e.g., specify and implement interactions between concurrent components).

  For example, OCCAM uses physical channels to provide communications among processes.

CPC supports virtual architecture programming based on explicit message passing parallel programming model.

2.3 Parallel Programming Languages

Below, we review some of the existing parallel languages and systems.

- OCCAM

  OCCAM is the first language that is based upon the concept of parallel, in addition to sequential, execution, and to provide authentic communication and synchronization between concurrent processes. It's based on the CSP programming model[17].

  OCCAM enables an application to be described as a collection of *processes*, where the processes execute concurrently, and communicate with each other through *channels*. OCCAM simplifies the writing of concurrent programs by taking most of the burden of synchronization away from the programmer. The *PAR* construction in OCCAM is used to start parallel processing. Communications between different parts of a program is built into the language itself – by physical channels which are one-way, point to point links among processors. Communication over a channel can only occur when both input and output processes are ready, so communication is synchronized. It is not possible to give a generally applicable recipe for terminating *PAR* processes, the detail will vary from case to case.

  OCCAM bears a special relationship with the INMOS Transputer, a high performance single-chip computer whose architecture facilitates the construction of parallel processing systems. The main implementations of the language are currently on INMOS Transputer which were designed with the CSP programming model. The Transputer reflects the OCCAM architectural model, and may be considered an OCCAM machine.

  OCCAM language uses physical channel, which is a one-way, point to point link from one processor to another, to communicate. Therefore, OCCAM users must know the low level detail of the underlying architecture, which makes the OCCAM language difficult to use.

  OCCAM programs are efficient for the Transputer, since the Occam programming model matches well the Transputer architecture model. But it is not easy to port the OCCAM programs to other architectures. Users have to rewrite the programs for different physical architectures.

- **HPF**

  High Performance Fortran is an extension of Fortran 90 to support the data-parallel programming model on MIMD and SIMD parallel computers using Fortran. HPF retains the usual programming paradigm: a parallel HPF program, like a sequential program, sees a global address space, and the control flow through the program is (conceptually) identical to the one of a normal Fortran program. The most important extensions are:

  - Logical architecture declaration directives

    In HPF, logical processor templates are declared using the **PROCESSOR** directive. This directive may only appear in the specification part of the program. The **NUMBER_OF_PROCESSORS** and **PROCESSORS_SHAPE** may be used to inquire about the total number and the shape of the actual physical processors used to execute the program.

  - Data distribution directives

    The distribution of data on the available processors can be explicitly specified, the user can declare several logical architectures of the same size in a single program. For each parallel phase, the user can then distribute the data onto a logical architecture using the **DISTRIBUTE** compiler directive. After completing a phase of the parallel program, the **REDISTRIBUTE** compiler directive may be used to reshuffle the data to the appropriate locations to be used in the next parallel phase of the program.

  - Parallel Constructs

    The FORALL construct, the INDEPENDENT directive on DO loops, start and stop processes; new Intrinsic and an HPF_LIBRARY module will co-ordinate the parallel processes. All of these extend the possibilities of Fortran 90 to express parallelism.

Existing compilers for HPF convert the user's program to the message passing form. The mapping of the logical processors to the physical processors is considered a compiler dependent issue in HPF and the communication libraries. The programmer does not have the freedom to specify a mapping which suits the parallel algorithm. The HPF language does not support directives which relay the communication pattern of the program to the compiler so that proper

mapping of the computation to the physical processors can be performed. User-defined mapping of the computation to the physical processors is recommended as an extension in the HPF language specification. Current implementations of the HPF language such as [6] do not support this feature. When the physical architecture is changed, the user must recompile and relink their program. Therefore, HPF is inconvenient for the user to port the program across physical architectures with different topology or size.

- **Multi-Pascal**

  Multi-Pascal is an extension of the programming language Pascal, with the addition of features for creation and interaction of parallel processes. It is designed to be machine independent, and can run on a wide variety of parallel computers.

  Multi-Pascal has features that allow the dynamic creation of parallel processes on the physical processors. Multi-Pascal uses channel variables to transmit data from one process to another.

  Multi-Pascal does not support the virtual architecture programming. The user needs to declare the physical architecture and specify the process-to-processor mapping in their application program. There is no run-time mapping. Moreover, the user is forced to organize the program to match the available physical architecture which may not be a natural structure to the application, hence the algorithm may be not easy to implement. This also limits the portability of the Multi-Pascal programs. It is impossible for the user to run the same Multi-Pascal application program on a different platform. The user has to rewrite the program to do that.

- **CPC**

  As we already know, CPC is an explicit parallel language based on C. It has the process creation/termination statements, such as fork/forall; processes communicate via channel variables. CPC users can declare the virtual architecture most suitable to the algorithms, the virtual architecture can be mapped to the available physical architecture at run time. Users have the freedom to specify the mapping functions as late as the program loading time.

The special feature of CPC is that CPC supports virtual architecture programming. With virtual architecture, CPC language is easy to learn, easy to use, and portable across different platforms and different partitions within the same platform.

Chapter 3

The CPC Language Design

CPC (Concordia Parallel C), which is an extension of ANSI C, supports explicit constructs that declare virtual architectures, forks processes, synchronizes executions of processes, and divides processes into groups and map processes to physical processors. It can also manage communications among processes by sending and receiving messages via channels. Channels and processes can be created dynamically. The CPC language supports both shared-memory and message-passing programming paradigms.

CPC language has the following principal features:

- **virtual architecture programming**

  The CPC user can write an application in the virtual architecture with the desired size and topology, which is most natural and efficient to the program performance. The user can select a mapping function from the mapping library or design his own mapping function to map parallel processes to physical processors at run time.

- **portable parallel programming**

  With the virtual architecture approach and the CPC AST, CPC programs are portable across different platforms and also across different partitions within the same platform.

- **ANSI C compatibility**

  CPC is based on the ANSI C, and preserves most of the existing sequential

features of the C language. Except for a few additions of C++ features and enhancements, CPC sequential sytax is compatible with ANSI C syntax.

- **explicit parallel programming**

  CPC is extended from the ANSI C, the extensions allow the user to express parallelism, to declare virtual architecture and to map process to virtual processors.

- **channel communication**

  In CPC, a new type of variable, the *channel* variable, is designed to help coordinate process interaction and communication.

- **C++ style comment and call-by-reference**

  The CPC user is allowed to use C++ style comment and call-by-reference in the CPC program.

- **no limit on the length of an identifier**

  There is no length limit on the CPC identifier.

- **nested function definition**

  The CPC user is allowed to nest a function definition inside a function. The normal scoping rule applies.

- **ease to use**

  The CPC language is easy to learn. CPC programs are easy to read, maintain, and verify.

3.1 Virtual Architecture Approach

Virtual architecture approach is the main feature of the CPC language.

The CPC application programs can be written using the virtual architecture approach. In this approach, the user writes the program in the virtual architecture, which has the most suitable topology and size for the problem in question, which captures the communication patterns among processes. At run time, the compiled program will be mapped to a physical topology specified by the user, which can be different from the virtual topology.

Virtual architecture approach provides the user with a high level of abstraction of the communication pattern, allows the user to concentrate on the resolution of the problems without sacrificing performance. Therefore the user need not worry about the architectural features of the target system. The virtual architecture is the most natural to the algorithm and most efficient to the algorithm performance, the program will be easy to implement, easy to read, easy to verify, and easy to maintain.

The total number of virtual processors needed by a CPC source program is defined by the user, not limited by the physically available processors in a real parallel machine. So the virtual architecture and virtual processors give CPC users more flexibility in programming.

For a particular algorithm, the execution delays resulting from communications will depend on the specific topology. The topology and size of the physical architecture may not match that of the virtual architecture. To solve the mismatch between virtual topology and physical topology, reduce the communication delay and program execution time, the virtual architecture approach allows the CPC user to select mapping functions available in the mapping library or design his own mapping functions, so that the virtual processors can be mapped to the available physical processors at run time according to the mapping function.

After mapping, the two communicating processes should be situated as close together as possible. The mapping can minimize communication cost among communicating processes, and balance the workload among physical processors.

There are two levels of program mapping. The first level is the mapping from processes to virtual processors. The second level is mapping from virtual processors to physical processors.

1. Process-to-virtual-architecture mapping. The mapping can be one-to-one or many-to-one. Often in the application program the user specifies the ID of the virtual processor on which a process will run. The virtual processor will be mapped to a physical processor at run time.

   If the user does not provide a virtual processor for a new process, at run time, the process is mapped directly to a physical processor, bypassing the virtual processor level since a virtual processor is not needed in this case. The physical processor allocated to the new process is determined by the processor manager. The default allocation criteria is to balance the workload among existing

process ID	physical processor ID
0	0
1	0
2	0
3	0
4	1
5	1

Table 1: Mapping table for process to physical processor.

physical processors.

2. Virtual-to-physical-architecture mapping. At run time, the user can specify the desired physical architecture for running the compiled virtual-architecture program. This mapping is accomplished by mapping the virtual processor ID to a physical processor ID. A virtual-to-physical mapping table is used to store the mapping pattern. The mapping library provides different mapping functions, which will maximize the resemblance between a virtual architecture and an available physical architecture in terms of their communication behavior, so that the advantage of a particular virtual architecture for an application program, such as the minimized communication cost, can be appreciated by the available physical architecture. Mapping functions include random mapping and user-defined mapping.

We can see, by means of an example, how parallel processes are mapped to physical processors. For instance, assume that the user declares a virtual architecture of ring topology with 3 processors (number 0,1,2), and he/she has 6 processes running in parallel. At runtime, he may get a subsystem of line architecture with 2 processors (number 0, 1). Then his mapping tables could be as shown in Table 1 and Table 2. Figure 2 indicates the two level mappings for this example.

With the mapping, the virtual architecture will be separated from the physical architecture; the CPC source code is portable to different parallel systems; and the CPC executable code of a virtual architecture is portable to different partitions within a platform. CPC programs have the executable code level portability.

With virtual architecture approach, CPC turns out to be a good candidate for explicit parallel programming that can be used to design, express, and implement

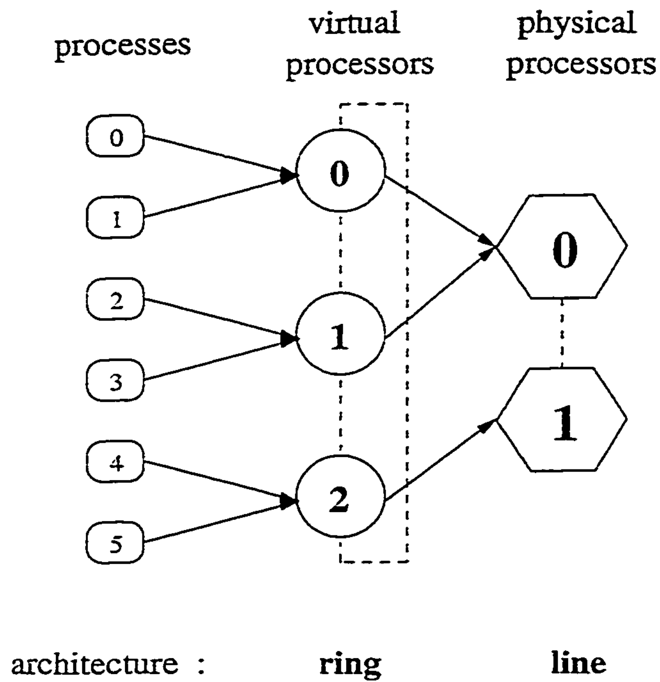

Figure 2: Mapping example

Physical processor ID	process ID
0	0,1,2,3
1	4,5

Table 2: Mapping table for physical processor to processes.

efficient portable parallel algorithms.

3.2 Differences Between the Sequential Part of CPC and ANSI C

Because the CPC language preserves most existing sequential features of the popular C language, we will only discuss the major differences between the sequential part of CPC and ANSI C in this section.

3.2.1 Function Definition

The syntax for function definition is different from ANSI C in three aspects:

- The function return type cannot be omitted.
- In a function call statement, the pair of parenthesis following the function name in ANSI C can be omitted in CPC if there is no argument for the function.
- Function definition can be nested. For example,

```
-----------------------------------------------------------------------
# include "cpc.h"

void main {                  //omit the "()" after main
  int x;

  int f(int y) {             //define a nested function
    int z;

    z = 2;
    return y+z;
  }

  x = 1; x = f(x) + 2; }
-----------------------------------------------------------------------
```

This feature allows the user to redefine the function which may exist in the CPC parallel library in a certain scope, thus avoiding naming conflicts of library programs with user programs. This will make the CPC parallel library reusable and portable.

The normal scoping rule applies. A user can declare a function inside a compound statement before the first statement of this compound statement. In this way, the user can nest a function within a function. The nested function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variable of the nested function. The nested function is unknown outside of the outer function.

3.2.2 Names

CPCC sets no limit to the length of a name of any identifier. Even though ANSI C standard sets no limit to its identifiers, most C/C++ compilers do. CPCC achieves this flexibility with even more efficient pointer-based comparisons among identifiers.

3.2.3 Call By Reference

The CPC supports the C++ style call-by-reference.

Arguments to a function are means of passing data to the function. Many programming languages pass arguments by reference, which means they pass a pointer to the argument. As a result, the callee function can change the value of the argument. When passing argument by value, the callee function can change the value of the parameter copy, but it cannot change the value of the argument of the caller.

In ANSI C, an array used as a parameter is always call-by-reference. In CPC, all parameters use call-by-value, unless there is an & before the parameter. For remote function calls, call-by-value parameters are used to pass initial data from the caller to the callee, and call-by-reference parameters are used to pass computing results back from the callee to the caller.

3.2.4 Comments

To insert a comment into a CPC program, the user, as with the ANSI C, can surround it with double-character symbols /* and */. But in CPC, comments in this style may extend over one or more lines, and the user is allowed to insert another comment of this style inside it. The C++ style comments are also supported.

3.3 Parallel Features of the CPC Language

Parallel features of the CPC support the creation of processes, the mapping of processes to virtual processors, and communications between processes through channel variables. Parallel features also synchronize executions of processors.

The CPC user can specify the virtual architecture on which the program will run. The user can declare it at the start of the program. The virtual architecture will then

be mapped to a physical architecture at run-time. The physical architecture can be the same as or different from the virtual architecture.

3.3.1 Process Creation and Termination

The most important building block of parallel program is the *process*. Computational activity takes place when a process is assigned to a processor in the underlying parallel computer. Ordinary sequential computer programs can be understood as a special case of parallel programs, in which there is just one single process and one single processor. When such a sequential program starts to run on a computer, the processor starts to execute the body of the main program, starting from the first statement. Thus, this main program can be considered as the process being executed by the processor.

In CPC programs, the program execution begins in exactly the same way: the *main* program becomes the first process and is assigned for execution to the first processor. The main program may contain any of the ordinary kind of statements that are found in sequential programs, such as assignments and loops. However, in CPC there is also a completely new kind of statement not found in sequential programs: a process creation statement. *fork* and *forall* are such statements, whose execution will cause new processes to be created and assigned to processors for execution. This is how parallel activity is initiated in the program: an existing process that is already running on a processor executes a process creation statement. The created process is sometimes called the child process, while the creator process is called the parent process.

When a CPC program begins its execution, the *main* program becomes the first process and is assigned for execution to the first virtual processor. Existing processes that are already running on a processor can execute a process creation statement such as *fork* or *forall* to create child processes .

fork statement

Fork statement is the most powerful statement in CPC to create a new process. It is able to turn an individual statement into a child process which is useful in many circumstances.

For example, in the following program segment

```
fork for (i=0; i<=10;i++) A[i] = i;
```

the `fork` statement will create a new child process, which will execute the enclosed `for` statement. The parent will continue execution immediately without waiting for the child in any way. Although the parent process does continue with its execution while its `fork` children are still running, the parent is not permitted to terminate until all its children have finished. If the parent reaches the end of its code while one or more of its children are still running, the parent will be suspended until all the children terminate. Only then will the parent be allowed to finish. This implementation prevents a premature termination by process 0 while some of its children are still running.

The substatement in `fork` statement, which will be executed by the new process, can be any valid statement in CPC, such as a single statement, a function call, a compound statement. Following are examples of how to use `fork` statement.

```
#include "cpc.h"

//The "<statement>"s in the following comments represent the right
//side <statement> in the first rule of the fork syntax.
channel int CI;
void mul(int i){
}

void main {
int i,j;

fork;                           //The <statement> contains no operation
fork  i = (i>10)? 1 : i+1;  //The <statement> is a single statement
fork [ ; CI] i = CI;
fork {                          //The <statement> is a compound statement
  if (i > 10) i = 1;
  else i++;
}
```

```
fork (i+j);                  //The <statement> is an expression
fork printf("Hello world"); //The <statement> is a function call
fork [i; CI] mul(i);
fork fork mul(i);            //The <staetment> is another fork statement
//omitting the first <opt_expr> in the second rule
fork[ ; CI] (i = CI, i+j);
//omitting the second <opt_expr> in the second rule
fork[i; ] sqrt(i);
}
```

The general syntax of `fork` is as follow:

```
<statement>  ::= FORK <opt_mapping> <statement>
<opt_mapping>::= [ <opt_expr> ; <opt_expr> ]
             ::= [ <opt_expr> ]
             ::= [ @ <opt_expr> ; <opt_expr> ]
             ::= [ @ <opt_expr> ]
             ::= [ ^ <opt_expr> ; <opt_expr> ]
             ::= [ ^ <opt_expr> ]
             ::= NIL
```

where `statement` on the right hand side in the first rule can be any CPC valid statement, such as compound statement, expression or even no operation. The first `opt_expr` in each rule of `opt_mapping` should always be any CPC valid integer-valued expression, it represents the processor number. If there is an `@` in front of it, it represents the virtual processor number, if there is an $\wedge$ in front of it, it represents the physical processor number. The second `opt_expr` in each rule of `opt_mapping`, if it exists, should be an expression with an valid left value in CPC. It's possible to omit either one of the `opt_exprs`. The fork statement creates a new child process, which will execute the `statement` on a virtual processor or physical processor specified by the user or by default.

We will discuss the semantics for `opt_mapping` in Section 3.3.4 and Section 3.3.2.

join statement

Sometimes, it may be desirable for a parent to wait at some point for the termination of one or all of its `fork` children. `join` statement in CPC is designed for this purpose. If the parent has only one `fork` child, then a `join` statement executed by the parent will force it to wait for the child to terminate. If the child has already terminated, then the execution of `join` will have no effect on the parent. One may think of the `join` as the opposite of a `fork`. `fork` separates a child process from its parent, and `join` brings the terminated child back together with its parent.

For example, in the following

```
fork mul(j);
for (i=0; i<10; i++)
  a[i] = i;
join;
...
```

---

the parent will execute the `for` loop after forking the child. After finishing this loop, the parent will suspend execution at the `join` to wait for the termination of its child, then continue to execute the statements right after `join`. If the child has already terminated, the parent will continue its the execution after `join` without wait.

The execution of each `join` by the parent will match one single `fork` child termination. If the parent has multiple `fork` children, it may execute multiple `join` statements to wait for them all to terminate. In the following program,

---

```
#include "cpc.h"

void main{
  int i;

  for (i=0; i<10; i++)
```

```
    fork sqrt(i);
  for (i=0; i<10; i++)
    join;
}
```

---

the first for loop creates 10 fork child processes. Each child calls function sqrt. Then in the following for loop, the parent executes the join statement 10 times, thus waiting for the termination of all 10 children. Without this second loop, the parent would just continue execution in parallel with all of its children. However, once the parent reached its end, it would not terminate until all children had terminated.

forall statement

The following is an example of forall statement,

```
forall (i from 0 to 9) c[i] = a[i] + b[i];
```

which will create 10 copies of the enclosed assignment statement $c[i] = a[i] + b[i]$ and make each one a separate parallel process with its own unique value of the index variable i.

Forall statement can create multiple child processes at the same time. It is a parallel form of a normal for loop in which all the loop iterations are executed in parallel rather then sequentially. Each iteration of a forall statement creates a child process which will run in parallel with other children created by the same forall. The program code for each process is the same, just a copy of the body of the forall loop. After finishing the creation of processes, the parent process suspends its execution, goes to sleep and waits until all of its children terminate. Only then will the parent continue its execution with the statement following the forall statement. This is one of the differences between forall statement and fork statement.

There is always an overhead associated with creating a process, mapping it to a particular processor where it will be executed, and terminating a process. If the process grain is too fine, the overheads may outweigh the speedup gained by parallel processing. For this overhead to be justified, the duration or execution of each process, sometimes called the **granularity** of the process must be much larger than

the creation overhead. Usually, with larger granularity processes, we can make good use of more processors to speedup the program. But for the following example,

```
forall (i from 0 to 99) a[i]=i;
```

assume a process creation time is 8 time units in the system, and the duration of each process is 8, then the total elapsed time since the start of the `forall` statement will be 8*100+8=808 time units. If we replace the `forall` statement with a `for` statement, then no child process will be created, therefore the total execution time will be 100*8=800 time units. Not only has the `forall` failed to speed up the execution, but actually lengthened the execution time due to the process creation time.

To help overcome this granularity problem with `forall` statement, grouping option is provided in CPC. It can be used in `forall` statement to group together a certain range of index values in the same process. The grouping size should be chosen so as to balance the program speedup and process creation/termination overheads. If the `grouping` index is omitted as the above example, then the default group size is 1. Now let's look at the following example:

```
forall ( i from 1 to 100 grouping 10 )
      add(a[i]);
```

the added notation `grouping 10` causes the index values to form groups of size 10 in each process. Thus, only 10 processes are created. The first process sequentially iterates through the index values 1 to 10, the second process iterates through 11 to 20, and so on.

A process terminates when it reaches the end of its code. Processes of the same parent may terminate at different time. This is due to slight variations in processor speeds, processor loads, or other environmental influences. In any case, the parent process executing the `forall` will always wait for all the child processes to terminate before executing the statement that follows the `forall`.

Following is the general syntax of the `forall` statement:

```
<statement>     ::= FORALL ( <non_comma_expr> FROM <expr> TO <expr>
                    <opt_grouping> ) <opt_mapping> <statement>
<opt_grouping>  ::= GROUPING <expr>
                ::= NIL
```

```
<opt_mapping>  ::= [ <opt_expr> ; <opt_expr> ]
               ::= [ <opt_expr> ]
               ::= [ @ <opt_expr> ; <opt_expr> ]
               ::= [ @ <opt_expr> ]
               ::= [ ^ <opt_expr> ; <opt_expr> ]
               ::= [ ^ <opt_expr> ]
               ::= NIL
```

where non_comma_expr must be a single expression, expr must be any valid integer-value expression, none of them can be omitted. expr in the second rule must be a valid CPC integer-valued expression, it can be omitted. The first opt_expr in each rule of opt_mapping should always be any CPC valid integer-valued expression, it represents the processor number. If there is an @ in front of it, it represents the virtual processor number, if there is an $\wedge$ in front of it, it represents the physical processor number. The second opt_expr in each rule of opt_mapping, if existing, should be valid left-value expression in CPC. It's possible to omit either one of the opt_expr. The statement on the right hand side can be any valid statement in CPC, such as expression, compound statement, function call, etc.

As stated in section 3.3.1, we will discuss the semantics for opt_mapping in Section 3.3.4 and Section 3.3.2.

Now let's look at some examples.

```
------------------------------------------------------------------

#include "cpc.h"

// The "statement"s in the following comment represent the second
// statement on the right side of the first rule.

int f(int j) {
  return j*10;
}
channel int c[200];
void main() {
```

```
  int i,j;
  int a[10][20];

  forall ( i from 1 to 10 )    //The statement is a single statement
    printf("Hello world!\n");
  //The statement is a function call
  forall (i from 1 to 10) [i; c[i]] f(i);
  forall ( i from 1 to 5) [i;] //The statement is another fork statement
     fork f(i);
  forall (i from i+1 to 2*i)    //The statement is a compound statement
    { int j;
      j = i*i;
    }
  forall ( i from 1 to 9)                  // nested forall loops
    forall ( j from 1 to 10 )  [; c[i*j]] sqrt(i*j);
  forall ( i from 1 to 10 grouping 5)   //with the <opt_grouping>
    printf("Hello !\n");
  //Omit the second <opt_expr> in the third rule
  forall(i from 0 to 9 grouping 1)
       [i;] f(i);
  //Omit the first <opt_expr> in the third rule
  forall (i from 1 to 10) [; c[i]] f(i);
  //A complete example of forall statement
  forall(i from 0 to 9 grouping 5) [i;c[i]] printf("Perfect!\n");
}
```

3.3.2 Process Communication Via Channel Variables

This section describes communication channels, the declaration of channel variables, reading and writing channel variables, binding channel variables to processes.

Communication channels provide an abstraction for communications of values between two processes. The channel variables abstract message sending and receiving

among processes. A process $p1$ can communicate with another process $p2$ by "sending" a message through a channel variable. Process $p2$ will "receive" the message from the same channel variable.

A message being sent is abstracted by a write to a channel variable. A message received is represented by a read from the channel variable. Conceptually, a channel acts like a first-in-first-out queue of values (messages) of the same data type. The capacity of the queue buffer is assumed to be unlimited.

Declarations of Channel Variables

A CPC user can declare a channel variable as follows:

```
channel int ci;
// "ci" is a single channel variable of type "int"
channel char cc;
// "cc" is a single channel variable of type "char"
```

The user can also define an *array of channel* and a *channel of array*, for instance:

```
channel int ArrOfChnl[50];
// "ArrOfChnl" is an array of 50 channels each of type "int"
typedef int partition[2][2]
channel partition ChnlOfArr;
// "ChnlOfArr" is a channel of an array which has int type element
```

Multidimensional arrays of channels are also permitted.

The only operations that can be performed with a *channel of array* is reading or writing a whole array from the channel. It is not permitted to read or write one element in the array. So the expression `ChnlOfArr[1][1]` is not a valid expression in the above example. To get at the 3rd element in the array, one must first read the whole array from the front of channel `ChnlOfArr` into an ordinary variable of the same array type such as a, and then use the expression $a[1][0]$.

Nested channels are not allowed: a channel may not contain any channels. A "channel of channel" is not permitted by the rule that the type must be a valid type in the C language because `channel` is not a valid type in C. For example, we cannot have a channel of structure where one field of the structure is another channel.

It is also permitted in CPC to have a channel whose component type is a pointer, as in the following example:

```
channel int x;
channel *dnchan;
...
dnchan = ChannAlloc(1,1);
```

In the CPC language, channel variables are declared in just the same way as variables are declared. Channel variables are usually declared as global variables. We may have a channel variable c that is local to a function. However this channel would be useless because no processes other than the original owner of c can access c (due to the lexical scope rule of the C language). Thus the original owner cannot use c to communicate with the other processes. That is why channel variables should be declared as global variables so that all processes can see these channel variables.

The general syntax for channel declaration is as follows:

```
<def>  ::= CHANNEL TYPE_QUAL <decl_list>;
       ::= CHANNEL TTYPE <decl_list>;
```

where TYPE_QUAL can be any valid type such as int, char, TTYPE can be typedef type for declaring a single channel variable or array of channel.

Binding Channel Variables to New Processes

Each channel variable has a unique owner process. Every process can write to a channel variable. But only the owner of the channel variable can read from it.

When any process is created with any of the usual CPC statements (**fork** or **forall**), a binding may be included to assign one or more of the channels to the created process. For example,

---

```
channel int c[200];
channel int CI;
main()
{ int i;
```

```
  fork [1; CI] ChildCode();      //assign channel CI to new process
    //assign channel c[i] to the ith new process
  forall(i from 0 to 9 grouping 5) [i;c[i]] printf("Perfect!\n");
}
```

Binding channel variables to a new process means to assign channel variables to a specific process. In this way the process can receive messages from other processes through the channels. However each process may read values only from its own assigned channels. A channel reference can be an array of channels designed to facilitate the binding of many channels (of the same component type) to a new process. For example, instead of binding 20 channels (of the same component type) to a process, we could declare an array of 20 channels, and then bind that array to the process. Any channel in the array may then be used by that process to receive messages. The array of channels can be an entire array, or one or more dimensions of a multi-dimensional array (e.g. one row of a 2D array, one plan of a 3D array), such as "MulArrOfChnl" in the following example.

```
typedef int partition[2][2]
typedef channel partition pchan;
pchan   MulArrOfChnl[3][3]
```

Recall the general sytax for *fork* or *forall* in section 3.3.1 and section 3.3.1 respectively, the syntax for binding channel variables is as follows:

```
<opt_mapping>  ::= [ <opt_expr> ; <opt_expr> ]
               ::= [ <opt_expr> ]
               ::= [ @ <opt_expr> ; <opt_expr> ]
               ::= [ @ <opt_expr> ]
               ::= [ ^ <opt_expr> ; <opt_expr> ]
               ::= [ ^ <opt_expr> ]
               ::= NIL
```

where the second opt_expr, if existing, in each rule of opt_mapping represents the channel which will be assigned to the new created process. It should be any CPC expression with a valid left value.

Reading from and Writing to Channel Variables

Channels are written by using their names on the left side of an assignment statement, and read by using their names on the right side of an assignment statement as in the following example:

```
C = j;   // C is written by process P1
i = C;   // C is read by process P2
```

In the first statement, P1 writes a value `j` to the end of queue in channel `C`. In the second statement, P2 reads a value from the front of the queue of values stored in channel `C` and writes this value into variable `i`. The type of variable `i` and `j` must match that of channel `C`. For example, if `C` is `CHANNEL int`, then both `i` and `j` must be of type `int`. Each `write` to a channel will add a new value to the internal queue, and each `read` will completely remove a single value from the internal queue.

Note that each time a channel is read, it produces a different value. This is because values are queued inside the channel during writing, and removed during reading. Let `CI` be a channel of integer. The assignment "`n = CI + CI`" is not equivalent to "`n = CI * 2`".

Any process may write values to any channel, provided that the channel variable is accessible by the process according to C lexical scope rules. However, each process may read values only from its own assigned channels.

Channel Empty Test

When a process executes a read operation to a channel that is currently empty, the execution of that process is automatically delayed until a value is written into the channel by another process. In the above example, if the P2 executes the assignment "i = C" while the channel `C` is empty, the execution of P2 will be automatically suspended. Later when some other processes, e.g. P1, finally writes a value into channel `C`, then the execution of P2 will be automatically resumed at this same assignment. Now that there is a value in that channel, the assignment will execute successfully, and P2 will continue to execute the next statement. However, the writer process will never be suspended; channels are supposed to have unlimited capacity and can hold any number of values. Channel writes are thus non-blocking.

To determine if a given channel currently contains any values, a Boolean-type expression may be created by using the name of the channel followed by a question mark, as in the following example for channel C:

```
if (C?)
   i = C;
else
   printf("Channel is empty");
```

The expression "C?" will evaluate to TRUE if the channel C currently contains any values and FALSE if the channel is empty. The owner process may evaluate this Boolean expression "C?" without fear of being delayed. The process executing the "C?" operation will not be suspended if the channel is empty.

3.3.3 Virtual Architecture Definition

As the CPC language supports both shared-memory and message-passing programming paradigms, an architecture declaration is needed for a message-passing program. The declaration is similar to that of a data type in C language. If the architecture declaration is absent in a program, the program is treated as a shared-memory program. The virtual architecture is specified with the keyword arch at the beginning of the program as in the following examples. The architecture of a multicomputer system is defined by the topology and the size of the system. For example, we can define the virtual architecture as follows:

```
-------------------------------------------------------------------

arch shared S[100];      //shared-memory with 100 processors
arch fullconnect F[25]; //fullconnect with 25 processors
arch line L[10];         //line with 10 processors
arch ring R[20];         //ring with 20 processors
arch hypercube H[5];     //hypercube with 2^5 = 32 processors
-------------------------------------------------------------------
```

The general syntax for the architecture declaration is as follows:

```
<def>      ::= ARCH ARCH_TYPE <var_decl> ;
<var_decl> ::= <new_name>
           ::= <var_decl> [ <const_expr> ]
<new_name> ::= NAME
```

where ARCH-TYPE is one of the following topologies: *shared*, *line*, *ring*, *mesh*, *torus*, *hypercube*, and *fullconnect*. *Shared* topology means that the program is intended for execution on a shared-memory multiprocessor. In a *fullconnect* topology, each processor is connected to every other processor. Name is the name of the architecture, const-expr must be any valid expression representing an integer constant. *shared*, *line* and *ring* should be declared as one dimentional arrays, const-expr represents the size of the architecture. *Mesh* and *torus* should be declared as multi-dimensional arrays. To multiply the values of each const-expr in the declaration will get the system size of this architecture. *Hypercube* should also be declared as an one dimentional array, the const-expr in this declaration represents the system dimension number.

3.3.4 Mapping Processes to Virtual Processors

In order to minimize communication cost among communicating processes, CPC users are allowed to map parallel processes to the processors of the virtual architecture. Therefore, communicating processes can be mapped to processors sitting close to each other.

The user can include his mapping in the process creation statement. In that statement, the user can specify the absolute ID of a virtual processor on which the newly created process will run, so that this child process can be mapped to the physical processor at run-time; or the user can specify the absolute ID of a physical processor to run the newly created process.

Following are two examples of the mapping used by *fork* and *forall* respectively.

```
for (i = 0; i < n; i++)
   fork [i; ] f(i);
forall (k from 1 to 10)
   [k-1; ] f(k);
```

The general syntax for *mapping* is as follows:

```
<opt_mapping>   ::= [ <opt_expr> ; <opt_expr> ]
                ::= [ <opt_expr> ]
                ::= [ @ <opt_expr> ; <opt_expr> ]
                ::= [ @ <opt_expr> ]
                ::= [ ^ <opt_expr> ; <opt_expr> ]
                ::= [ ^ <opt_expr> ]
                ::= NIL
```

Referring back to the general syntax of *forall* and *fork* statements in section 3.3.1 and section 3.3.1 respectively, we see that each primitive is ended by an statement which will be compiled into the parallel code to be executed by the new child.

The first opt_expr in each rule for opt_mapping represents the mapping of new process to processors. If there is an @ in front of the opt_expr, then the newly created child will be mapped to a particular virtual processor, the value of opt_expr is the virtual processor ID; if there is a symbol $\wedge$ in front of the opt_expr, then the newly created child will be mapped to a particular physical processor, and bypassing the virtual architecture, the value of opt_expr represents the physical processor ID. If @ or $\wedge$ is missing, opt_expr represents the virtual processor ID. So the opt_expr must be specified using any valid CPC integer-valued expression. The second opt_expr in each rule for opt_mapping, if existing, represents the binding of channel variable to the new created process, which we already discussed in section 3.3.2.

Process-to-virtual-processor mapping is optional. If the user does not specify the virtual processor ID for a new child, then at run time, the child process will be mapped to a default virtual processor. The mapping objective is to minimize the communication cost and balance the load among physical processors.

3.3.5 CPC Builtin Functions

CPC has a library of builtin functions to help users in programming, such as the basic arithmetic computation function, *sin* and *cos*; file open and close functions, *fopen* and *fclose*; and string operation functions, *strcpy*, *strcmp*. Message-passing functions are

also supported by this library, such as *vsend* and *vrecv*. Please refer to appendix A for prototypes of these library functions.

Chapter 4

CPCC Front End

CPCC is divided into two parts: a front end and a back end.

In order to provide particular fast compilation, CPCC frontend must generate efficient intermediate representation for its back end to manipulate. The CPCC frontend must be designed indepently of specific platforms so it can be reused to support code generation for any system architectures.

A CPC Abstract Syntax Tree (AST) structure is designed for this purpose. It is intended as an intermediate representation for the CPC language. CPC AST has a syntax matching well to the ANSI definition. All information in the source code are represented by four types of tree nodes and their interconnection. The biggest advantage should be that CPC AST allows us to build the CPCC frontend and backend independently, and keep all the information from the source code. CPCC frontend and CPC AST are designed to accept different backend modules to generate codes for different machines.

To use CPC AST, users must write a backend. The additional degree of freedom that CPCC frontend provides is that it allows integration of new backends to the CPCC which translate CPC code into specific target languages. One can easily take CPCC and redo its backend to produce a compiler for CPC language on a different machine.

identifier	token type
void	TYPE
main	NAME
{	LC
int	TYPE
i	NAME
;	SEMICOLON
i	NAME
=	EQUAL
10	ICON
;	SEMICOLON
}	RC

Table 3: Identifiers and their returned tokens.

4.1 Lexical Analysis

The CPCC lexical analyzer takes a stream of characters from CPC source file and produces a stream of names, keywords and punctuation marks; it discards white space and comments between the tokens. A lexical token is a sequence of characters that can be treated as an unit in the grammar of a programming language. CPC language classifies lexical tokens into a finite set of token types.

Given a program :

```
void main{
  int i;
  i = 10;
}
```

the lexical analyzer will return the stream as shown in table 3, where the token type of each token is reported with its semantic value (represented by character string) attached to it, giving auxiliary information in addition to the token type. We use **Lex** to implement our lexical analyzer for the CPC language.

4.2 Syntax Analysis

Yacc is a classical and widely used parser generator. A yacc specification is divided into three sections, separated by %% marks:

```
parser declarations
%%
grammar rules
%%
programs
```

The `parser declarations` include a list of terminal symbols, nonterminal symbols and so on. The `grammar rules` are productions of the forms like

```
expr: expr PLUS expr { semantic actions}
```

where expr is a nonterminal producing a right-hand side of expr+expr, and PLUS is a terminal symbol (token). The semantic action is written in ordinary C++ and will be executed whenever the parser reduces using the rule.

4.3 Symbol Table

To compile a C-like language, it is convenient to build a symbol table. The symbol table is the primary data structure used to maintain all the symbols (or identifiers) in a program, along with all their attributes. In CPCC, the symbol table is implemented by two independent hash tables—*Symbol_tab* and *Struct_tab*.

External chaining hash scheme is the best for symbol table implementation. Hash the identifier into a list head, and chain on collisions. Each collision chain forms a first-in-last-out stack matching the scope rules for nested language blocks.

The symbol table has the following properties:

- Each lexical identifier is stored once only in the symbol table.
- Each lexical identifier is associated with exactly one level, which records the scope level of the declaration of that identifier.
- A list of declarations in the same level is kept, with the usual simple algorithm: add symbols to table; remove symbol from table; search in reverse order; reverse the list; some pointers to these lists such as *Local_struct_list* are pushed into

stacks on block entry and popped off from stacks on block exit; the local symbol lists are removed from symbol tables when exiting from the corresponding block.

The reason for having the two tables is to keep the structure, union and enumeration tags of CPC in a different name space from the other identifiers. Both could easily be combined if the tags were marked as such and the supporting functions such as look up, add, and delete recognized the different entries, but the two tables provide a more distinct separation. The rest of this discussion refers to both as one entity, the symbol table.

4.3.1 Symbol Table Entries

Each symbol table contains a number of doublely-linked lists of elements. Symbols are identified by names.

Symbol_tab and *Struct_tab* will be made with indicated hash function and indicated size at the beginning of the CPCC parser. Both of them will be cleared at the end of the execution of the CPCC front end.

Routines for creating symbol table entries, adding entries to the sybmbol table, removing entries from the symbol table, and looking up the symbol table are provided to support manipulation and management of the symbol tables.

4.4 Concept of CPC AST

The abstract syntax of the CPC language is described using the commonly known BNF (Backus-Naur Form) notation. A complete grammar is provided in Appendix B.

CPCC will produce an abstract syntax tree — CPC AST as the intermediate representation of CPC source code according to its syntax. CPC AST is the central data structure in CPCC front end. Because a tree structure can be easily restructured, it is a suitable intermediate form for optimization compilers and from which code generation is performed. It represents the abstract structure of the particular input source program and is built by actions of the scanner and parser in CPCC frontend.

4.4.1 Nodes in CPC AST

CPC AST is constructed by nodes of the following four types : SYMBOL, LINK, VALUE, and STMT nodes. Each type of nodes represents a different structure component. Nodes of the same type share a common structure. The definitions of these nodes are included in the file `tree.h`. Each node represents an occurrence of either a nonterminal or a terminal symbol of the CPC abstract syntax. All the nodes and their interconnection in the CPC AST represent all the information in CPC source code. There is one *Root* pointer pointing to a SYMBOL node in each CPC AST.

SYMBOL nodes represent identifiers. LINK nodes represent type structures. VALUE nodes represent expressions made up of constants or other symbols in the statements of the code. STMT nodes represent the statements of the code.

In the following subsections, we will introduce these four types of nodes respectively.

SYMBOL nodes

Identifier symbols may be declared in any scope. Symbols are identified by names. Each symbol also has a set of flags to record various attributes. Symbol nodes, the primary important structure in the symbol table, contains the basic information about the identifier that it represents, along with pointers to other attributes. A variable or function symbol contains a pointer to the type of the variable. The type determines the amount of storage used to hold the variable as well as the interpretation of its contents. There are different kinds of symbols such as variables and functions. A SYMBOL node is defined as :

```
struct SYMBOL {
  char      *name;
  SYM_CODE  code;
  unsigned  level     : 8;
  unsigned  implicit  : 1;
  unsigned  duplicate : 1;
  unsigned  param_type: 1;
  unsigned  is_forall_index: 2;
  unsigned  funcall_mode: 2;
```

```
    LINK        *type;
    LINK        *etype;
    SYMBOL      *syms;
    SYMBOL      *return_args;
    VALUE       *val;
    STMT        *compound;
    SYMBOL      *next;
    char        *filename;
    int         lineno;
    int         size;
    int         offset;
    int         id_index;
    int         type_index;
};
```

SYMBOL nodes represent identifiers, which may be a variable, a function, a user-defined type, an enumeration element, or a bit field. Every SYMBOL node contains the same fields as the SYMBOL common structure illustrated above. Different SYMBOL node may use different fields, depending on the SYMBOL's type — variable, function, type definition, enumeration, enumeration element, structure, union, bit-field, architecture. We can distinguish the SYMBOL's type by its **code** field.

The SYMBOL node defines several fields that are used by all kinds of symbols. The most obvious of these is the symbol name. Each symbol has a **name** that should be unique within the same scope where it is defined. Because the name of a symbol alone is generally insufficient to uniquely identify it, the symbols are also given **level** to specify its declaration scope, **code** to differentiate different types of symbol nodes – such as a variable symbol or function symbol, **next** to point to the next symbol on the same level, **duplicate** to record if this symbol is redefined, **is_ref_param** to record if it's a call-by-reference or call-by-value parameter. Each item is also on a linked list with all other declarations on the same level. The level of a declaration is the same as its scope depth, so a global variable has a level of 0, and if a function name is at level **n**, then the parameters and its top-level local variables of the function are at level **n+1**. For a function symbol, there are pointers to its compound body.

SYMBOL nodes are also used as symbol table entries. SYMBOL nodes go to *Symbol-tab* if their code values are S_VAR, S_FUNCTION, S_TYPEDEF. Others go to *Struct-tab*. For example, in the declaration:

```
int f(int x);
```

f is an identifier representing a function, x is an identifier representing a variable, we will create a SYMBOL node named 'x' to represent variable x, the value of its **code** field is S_VAR, **type** field will be used to point a LINK node of type 'int'. We will also create a SYMBOL node named 'f' to represent function f, the value of the **code** field is S_FUNCTION, a **syms** field will be used to point to the SYMBOL node 'x'. Both SYMBOL node 'x' and SYMBOL node 'f' also go to *Symbol_tab*.

LINK Nodes

LINK nodes are used to represent type structures.

Every variable must be declared before it is used. A variable symbol contains a pointer to the type for the variable. The type determines the amount of storage used to hold the variable as well as the interpretation of its contents. A declaration provides the compiler with these information. For example, to declare *i* to be of integer type, we can write:

```
int i;
```

where the word int is a reserved word to specify a particular data type. There are dozens of reserved words to specify data types in CPC. Specifiers – int, float, char, double, long, short, signed, unsigned, void, enum, struct, union, typedef, const, volatile, fixed, extern, auto, register, static, channel — represent basic types, qualifiers and storage classes respectively. These specifiers can be expressed by LINK nodes.

Besides the basic types, there are some special types of variable — pointer, array and function. They are also represented by LINK nodes.

Like the SYMBOL nodes, all the LINK nodes share the common structure illustrated below:

```
struct LINK {
  unsigned linkType :1;
  union {
```

```
    SpecifierType  s;
    DeclaratorType d;
  } select ;
  LINK *next;
  int  type_index;
};
```

From above structure, we know LINK nodes are divided into two types: declarator type and specifier type. The type of a LINK node is identified by the value of the field **linkType** (DECLARATOR or SPECIFIER). When a LINK node is created, by default, the value of **linkType** is SPECIFIER. If it is a declarator, then the value of 'linkType' will be changed to DECLARATOR in the future. Each type of the LNIK node also has a field **next**, which is a pointer to another LINK node. The main difference between the structures of two type nodes exists in the **select** field. Now we will discuss this field for both types.

DECLARATOR type

If the LINK node represents a declarator, then it will contain the following two fields:
dcl_type: The type of the declarator. Its value can be DCL_POINTER, DCL_ARRAY, DCL_FUNCTION, or DCL_CHANNEL.
num_ele: If **dcl_type** has value DCL_ARRAY, this field specifies the number of elements in this array.

The names of the declarator types are self-explanatory. A declarator of type DCL_CHANNEL is the first link node for a type chain in which the specifier has CHANNEL as its storage class. This redundancy of information allows the user to specify channel as a storage class while internally the scope of a channel can be easily determined.

SPECIFIER type

If the LINK node represents a specifier, then it will contain the following six fields:
noun: Its value can be SP_INT, SP_CHAR, SP_FLOAT, SP_DOUBLE, SP_VOID, SP_STRUCTURE, SP_UNION, SP_ENUMERATION, SP_TYPEDEF, LINE, RING,

MESH, TORUS, HYPERCUBE, FULLCONNECT, or SHARED. The values with prefix SP (SPecifier) correspond to C's type specifiers. The other values specify the topology of virtual architectures.

sclass: It specifies the storage class of this object. Its value can be FIXED, EXTERN, AUTO, REGISTER, STATIC, or CHANNEL. All these values correspond to C's storage classes except CHANNEL, which denotes that this type is used to define a channel type.

type_qual: It specifies the extra type attributes. Its value can be NONE, CONST, or VOLATILE. NONE means no type qualifier.

_long: If this field is true, the type has attribute **long**.

_unsigned: If this field is true, the type has attribute **unsigned**.

sym: If the specifier **noun** is structure, union, or enumeration, **sym** will be a pointer to the symbol node which represents the specifier noun in the structure table. If the specifier **noun** is typedef, **sym** will be a pointer to the symbol node which represents the new type in the symbol table.

VALUE nodes

VALUE nodes are used to represent expressions in statements. A VALUE node is defined as follow:

```
struct VALUE {
  VALUE_CODE code;
  SYMBOL     *sym;
  char       *string;
  LINK       *type;
  VALUE      *str_next;
  VALUE      *expr1;
  VALUE      *expr2;
  VALUE      *expr3;
  VALUE      *next;
  char       *filename;
  int        lineno;
  int        number;
  int        const_index;
```

```
  VcType      vc_type;
  CAST_CODE   vc_cast;
  int         factor;
  int         size;
  int         is_channel;
  int         C_EXPR_top;
  int         is_parameter;
};
```

code field specifies the particular type of value node, for example V_INT_CST represents an integer constant, V_MULTI represents the operator *, V_IDENTIFIER represents an identifier with its name in **string** field. So we can distinguish VALUE nodes by **code** field together with the **string** field. **sym** is also in a VALUE node to point to a symbol node representing the identifier or function for this VALUE node.

An expression consists of one or more operands and zero or more operators linked together to compute a value. For instance:

```
x+1;
```

is a legal expression that results in the sum of x and 1. The variable x is also an expression, as is the constant 1.

The above expression will then be expressed by VALUE nodes. The type of this VALUE node is represented by the value of the field code — V_PLUS. It also has two other fields pointing to two other VALUE nodes named 'x' and '1' respectively.

STMT node

STMT nodes represent statements which may be an expression, a compound, a return, a goto, a if-then, a if-then-else, a while, a do, a for, a break, a continue, a switch, a case, a default, a fork, or a forall statement.

A STMT node is defined as follow:

```
struct STMT {
  STMT_CODE code;
  SYMBOL    *struct_list;
  SYMBOL    *symbol_list;
```

```
    SYMBOL    *undecl;
    STMT      *stmt;
    STMT      *else_stmt;
    VALUE     *expr1;
    VALUE     *test, *expr2;
    VALUE     *at_expr, *port_expr, *group_expr;
    char      *goto_label;
    char      *label;
    STMT      *next;
    char      *filename;
    int       lineno;
    int       code_index;
};
```

Every STMT node contains the same fields as the STMT common structure. Different STMT node may use different fields, depending on the STMT's type, such as expression, return, for, while. Each statement has a **code** field that indicates its type – C_NOP represents an empty statement, C_COMPOUND represents a compound statement, etc. For a compound STMT node, **struct_list**, **symbol_list**, **stmt** will be used to point to the compound statement's local struct symbols, local symbols, local statement list respectively. For a goto STMT node, **goto_label** will be used to represent label of the goto_target stmt, **stmt** will be used to point to goto_target stmt. For a fork STMT node, **at_expr**, **port_expr**, and **stmt** will be used to point to its at_expr, port_expr and statement respectively. For example:

```
x++;
```

x is an expression statement. A STMT node of type `expression` will be created to represent this statement, the value of the **code** field of this STMT is C_EXPR, another field **expr** will also be used to lead to a VALUE node — named 'post-inc'.

For the following statement,

```
forall [i from 1 to 10] j = i * 2;
```

we will create a STMT node of type 'forall', the code field of this node will be expressed as C_FORALL. Other fields **expr1**, **test**, **expr2**, **stmt** are also used to

point to VALUE nodes representing i, 1, 10, and STMT node representing expression statement j = i * 2 respectively.

4.4.2 Graphical Representation of Nodes

CPC AST provides CPCC with a complete intermediate representation of the input source CPC program, on which transformations can be done much more easily.

Now we will expain how the four types of nodes are represented graphically in CPC AST.

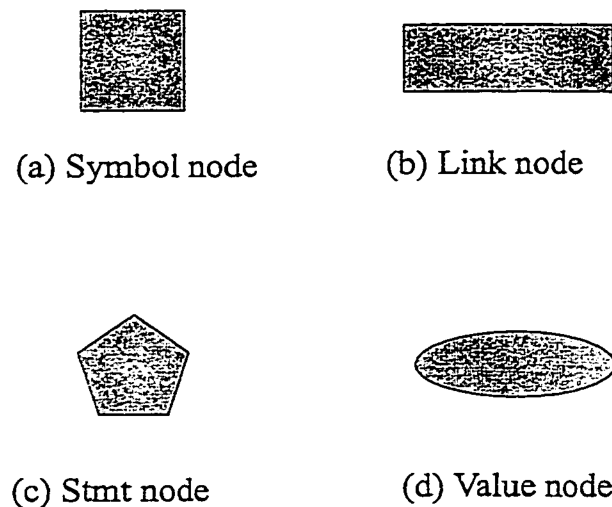

Figure 3: Graphical representation of nodes and values

We use the four types of icons with solid boundaries in Figure 3 to represent the four different types of tree nodes — SYMBOL, LINK, VALUE, STMT nodes. We will give each tree node a name written inside the icon. Usually it is the abbreviation of value of the **name** field or **code** field of this node.

We use other icons with dotted boundaries to represent values of certain fields in a tree node (not tree node themselves). The values are written inside these icons.

We use solid arrow line starting at the boundary of a tree node to represent a pointer field in the tree node pointing to another non-NULL tree node, with the field name attached to the arrow line, ending at the boundary of the pointed tree node.

We use a line attached by the field name to represent a field in this node with certain value, starting from the boundary of the tree node,ending at the boundary

of the icon for that value. In this way we can refer to all the fields necessary for the examples without cluttering the icons with compartments.

From next section to the end of this chapter, we will explain the way the major language constructs in CPC (including C) are represented in CPC AST and graphically illustrate with examples.

4.5 Data Declarations in CPC AST

4.5.1 Scalar Types and Pointers

Every variable in CPC language must be declared before it is used. A declaration provides the compiler with information about how many bytes should be allocated and how those bytes should be interpreted. The reserved words for scalar data types in CPC are: **char, const, int, float, double, short, long, signed, unsigned, void, volatile, arch.**

Char, **int** , **float** , **double**, **enum** are basic types. The others are qualifiers that modify a basic type in some way.

Figure 4 shows some scalar declarations and their corresponding node structures.

For example, for the scalar declaration

```
int x;
```

we will create a LINK node to represent the scalar type **int**, for the variable x; we will create a SYMBOL node and link it on the symbol_list in this scope, then LINK node for **int** will be linked to the type chain of SYMBOL node x.

It is usually a good idea to group declarations of the same type together for easy reference and for simplifying the tree structure. We will create a SYMBOL node for each variable in this group declaration, and link them one by one, but they will share the same LINK node to represent their types. For example

```
float y, z;
```

If there are more than one types in a declaration, for example :

```
long int i;
```

we will first create a LINK node A for **long**, then create a LINK node B for **int**, and then copy the initialized fields from LINK node B to LINK node A, and discard LINK node B. This procedure will be continued if more qualifying keywords for this type specifier `int` exist. So A will keep all the information for the combined types. In this way, we can save memory.

If there is an asterisk preceding the variable name in a declaration, e.g. variable p in Figure 4, then a pointer variable is declared. For a pointer variable, after creating the SYMBOL node for the variable, we will create a LINK node to represent the pointer declarator and add it to the end of its type chain, then link the specifier linked list to the end of the type chain.

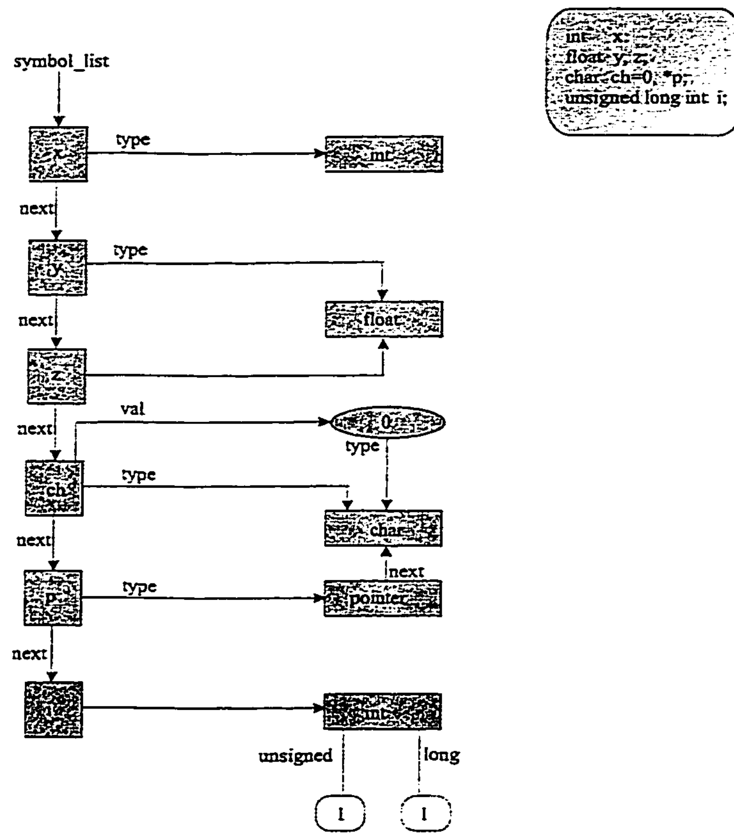

Figure 4: Scalar declarations and pointers

4.5.2 Arrays

Users declare an array by placing a pair of brackets after the array name. To specify the size of an array, enter the number of elements within the brackets.

Figure 5 shows some array declarations and their corresponding node structures. For an array variable, like a pointer variable, after creating a SYMBOL node for

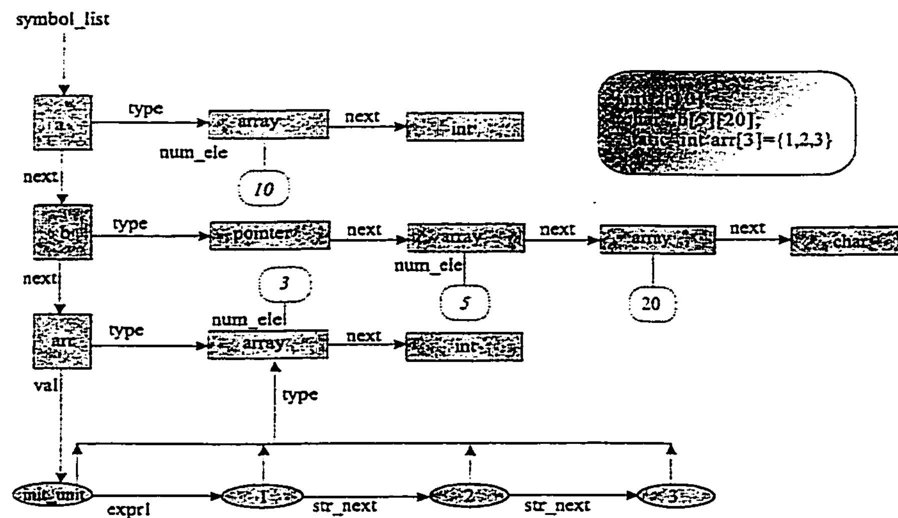

Figure 5: Array Declarations

this variable, we will create an array declarator LINK node and record the array size information in its *num_ele* filed. Link it to the array SYMBOL node's *type* chain, right in front of the specifier linked list.

If an array has been initialized at the declaration, we will create an **init_unit** VALUE node and one VALUE node for each initialized array element. Link all these VALUE nodes one by one and attach to the array SYMBOL node's *val* chain.

4.5.3 Aggregate Type – Structures and Unions

Aggregate type *struct* can serve as groups of mixed data and aggregate type *union* enables user to interpret the same memory locations in different ways. The syntax of a structure declaration can be fairly complex. The form of declaration we have used: declaring a tag name and then using the tag name to declare actual variables, is one of the most common ways. It is also possible to declare a struct without using a tag

name. Users can also declare a tag name together with variables.

For example, to declare a structure to hold one's vital statistics, we can write:

```
struct vitalStat
{
 char name[20], SIN[9];
 int month, day, year;
};
struct vitalStat vs;
vitalStat vs2;   // C++ style: struct tab is also type name
```

OR

```
struct
{
 char name[20], SIN[9];
 int month, day, year;
} vs;
```

OR

```
struct vitalStat
{
  char name[20], SIN[9];
  int month, day, year;
} vs;
```

The rule of using tag name is the same for *union* and *enum* type declarations.

Figure 6 shows some struct declarations and their corresponding node structures. Unions are represented in the same way.

The algorithm for adding *structure* and *union* declaration to CPC AST is as follow:

```
Search the Struct_tab symbol table;
If there is no symbol named the same as this tag
     in the same nesting level as this declaration
   Create a new SYMBOL node for this tag, and set its type to
   be 'struct' type or 'union' type according to the declaration.
```

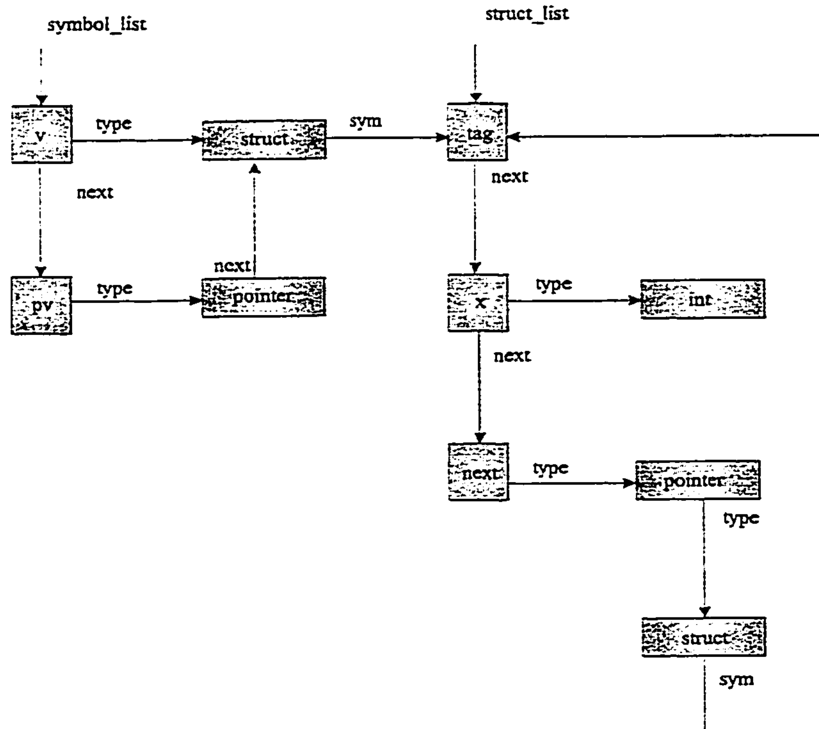

Figure 6: Structure and union Declarations

```
   If there is no tag name in this declaration, create a unique
   name for it, such as tag00, tag01, ...
   Insert this SYMBOL node to Struct_tab table;
   If this is a global declaration
        link the SYMBOL node to the head of Global_struct_list;
   Else link the SYMBOL node to the head of Local_struct_list;
If this is a redefinition
    Give error information
    Discard the structure or union's
       elements symbol chain for this new declaration
Else
    Link the structure or union's elements symbol chain for the
```

```
    new declaration to the tag SYMBOL node
   If it is an illegal structure definition, give error
   information.
Create a new 'struct' or 'union' LINK node for specifier
  -- struct or union
Let the 'sym' field of this LINK node point to
  the corresponding tag SYMBOL node in Struct_tab
```

4.5.4 Enumerations

In addition to integer, floating-point, and pointer types, the scalar types also include *enumeration types* which enable users to declare variables and the set of named constants that can be legally stored in the variable. Starting with the **enum** keyword followed by a tag name (can be ignored), followed by the list of constant names enclosed in braces, followed by the names the enum variables, we can declare an *enum* type variable. For example:

```
enum colorType {red, blue, green, yellow} color;
colorType houseColor = blue;  // C++ style
```

Figure 7 shows how to represent an enumeration declaration in CPC AST.
The algorithm for adding this *enumeration* specifier type to CPC AST is as follow:

```
For each constant element in the enumeration declaration
  Create a SYMBOL node for it.
  Set its type to be enumerator element type,
Link these SYMBOL nodes one by one.
Search the Struct_tab table
 If there is no symbol named the same as this tag
     in the same nesting level as this declaration
   Create a new SYMBOL node for this tag, and set its type to
   be ENUM type. If there is no tag name in this declaration,
   create a unique name for it, such as $enum00, $enum01, ...
   Insert this SYMBOL node to Struct_tab table;
   If this is a global declaration
```

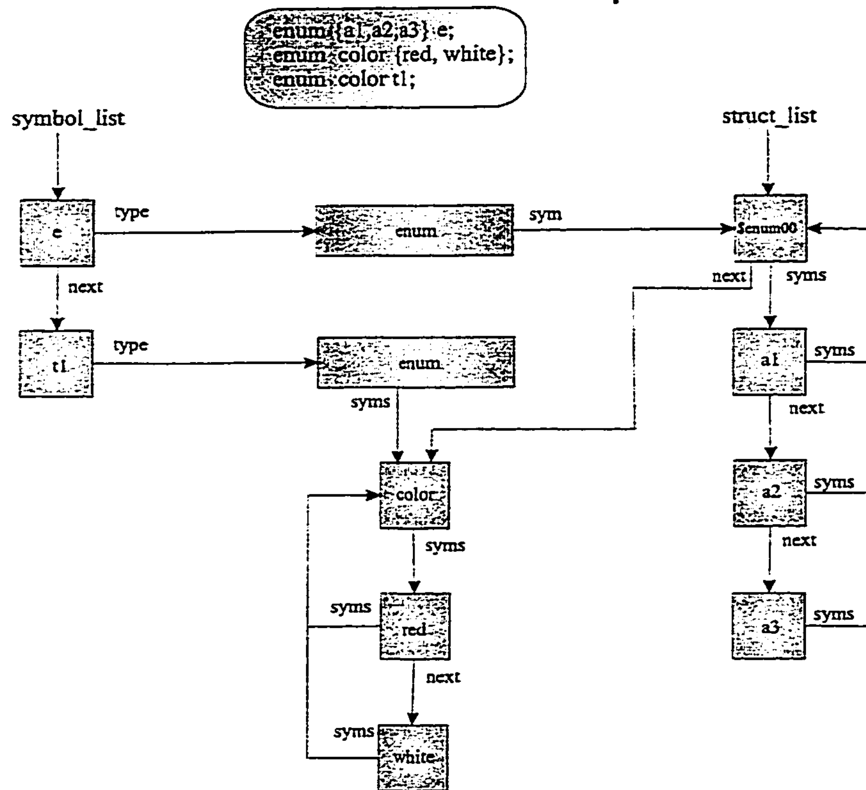

Figure 7: Enumeration Declarations

```
        Link the node to the head of Global_struct_list;
   Else link the node to the head of Local_struct_list;
If this is a redefinition
    Give error information
    Discard the enumeration element symbol chain of the
      new declaration
Else
    Link the enumeration element symbol chain of the new
    declaration to the tag SYMBOL node
    Add these SYMBOLs to Symbol_tab table;
Create a new 'enum' LINK node for specifier -- enum
Let the 'sym' field of this LINK node point to the corresponding
  tag SYMBOL node in Symbol_tab
```

4.5.5 Typedefs

Like C language, CPC allows users to create their own names for data types with the **typedef** keyword. Syntactically, a typedef is exactly like a variable declaration except that the declaration is preceded by the **typedef** keyword. For example:

```
typedef short int USHORT;
```

makes the name USHORT synonymous with **short int** rather than a variable that has memory allocated for it.

As before we'll create a LINK node to represent specifier **short int** and a SYMBOL node to represent symbol USHORT. We will initialize the **code** field of the Link node to S_TYPEDEF, and link it to the end of the symbol's type chain. Search the Symbol_tab table. If it's a multiple typedef or redefinition, discard this SYMBOL node; otherwise, add this SYMBOL node to the CPC AST and the related table.

4.5.6 Physical vs. Virtual Architecture

CPC users can declare physical and virtual architecture :

```
phyArch line l[5];
arch    mesh m[2][2];
```

The virtual architectures are used to support our virtual-architecture programming paradigm. The physical architecture declarations are only for the convenience of using the CPSS simulator. If a physical architecture is declared in source code, the CPSS will configure itself to simulate the specified physical architecture during simulated execution, and users do not need to manually configure CPSS for the same purpose.

The CPC AST for these declarations is similar to that of an array, except we use **line** or **mesh** specifier LINK node instead of **int** or **char** specifier LINK node.

4.5.7 Channels

The scope and duration of a variable is called its *storage class*. The keywords of storage class are presented in Table 4.

From above table, we can see that CPC has one more storage class—*channel* than the C language.

auto	channel	extern	register	static	volatile

Table 4: Storage Class

In CPC, interprocess communication channels are abstracted by channel variables, and message send and receive are abstracted by assigning values to the channel variables and reading values out of channel variables. A channel variable has its basic type plus its channel attribute. The basic type can be any type or user defined type. A channel variable can be a component of a normal C compound variable. Typedef can also be used to define channel types. But along any type chain there can be at most one channel link.

Usually, storage class is processed just like the type specifier, but *channel* will be handled as a special case. Consider the following two declarations:

```
static int i;
```

and

```
channel int i;
```

For the first declaration, after creating two LINK nodes for type specifier **int** and storage class **static**, the value of the initialized field in **int** LINK node will be copied to the **static** LINK node, and original **int** LINK node will be discarded. So only one LINK node can describe a specifier together with its storage class. But for the second declaration, its storage class is a **channel**. In this case, we should create two LINK nodes to represent **int** and **channel** respectively, none of them will be discarded. The **channel** LINK node will be added at the beginning of type chain, but after array declarators if there exists.

4.6 Expressions in CPC AST

An *expression* in CPC consists of one or more operands and zero or more operators linked together to compute a value. A constant expression, a variable, a function call, an operator can all be represented by a VALUE node. Each VALUE node has a *type* field pointing to a LINK node (this Link node can be shared by other VALUE

node) to represent its type in CPC AST. The smallest expression units are constants, identifiers, and function calls. Operators can be used to build compound expressions.

Figure 8 shows some constant expressions and their node structures.

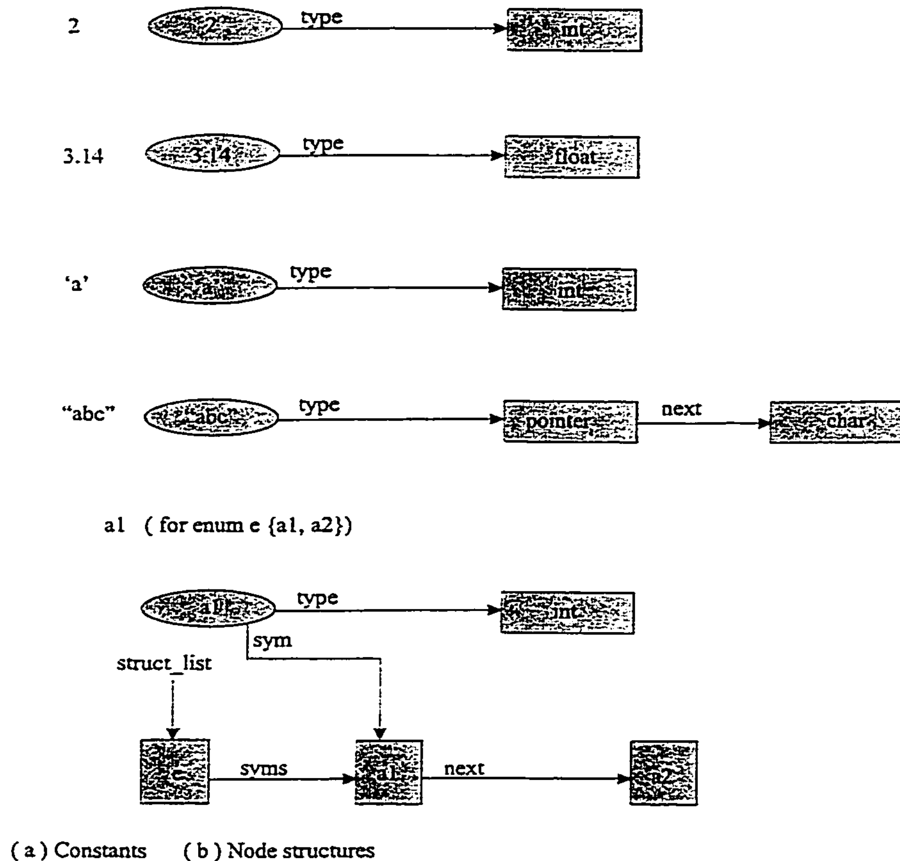

Figure 8: Constant Expressions

Now we will describe how to build the CPC AST for an expression in CPC.

```
When the expression is an unary expression:
=========================================


If it is a constant
   Create a VALUE node for it, set its type to be a constant type
   (such as int, float, or char.)
If it is a variable
```

```
   Create a VALUE node for it, set its type to be an identifier type
   Search the Symbol_tab
   If there is no symbol named the same as this variable
      Create a new SYMBOL node to represent it, mark it as an implicit
      variable and create an int LINK node to represent its type
      Link the SYMBOL node to the head of Local_undecl list
      Add this SYMBOL node to Symbol_tab
      Give error information: Identifier is not defined
   If this variable symbol represents enumeration element,
     Change the type of the VALUE node to be a enumeration constant
   If this variable symbol represents a function
      Create a new function call VALUE node, let its 'expr' field
      point to the variable VALUE node
If it is a unary operator followed by an expression
   Create a new operator VALUE node
   Build a VALUE node for the expression
   Let the 'expr' field of the operator VALUE node point to
      the expression VALUE node

When the expression is a binary expression:
===========================================

If it is a binary operator plus two expressions
   Create an operator VALUE node
   For each of the two expressions, build a VALUE node
     to represent it.
   Let the 'expr1' and 'expr2' fields of operator VALUE node point
   to the two expression VALUE nodes respectively.
```

4.7 Statements in CPC AST

Statement in CPC can be expressed by an expression, compound statement, control-flow statement, and parallel statements. Statements in one block are linked to the

statement list **stmt** for this block.

When a new statement is generalized, we first create a STMT node with a specific type (for example, switch, return, if then else) for it. If there are expressions in this statement, the STMT node will use its fields **expr** or **expr1** or **expr2** to point to the corresponding VALUE nodes of the expressions; if there exists s substatement in this statement, the STMT node will use the field **stmt** to point to the STMT node for the substatement.

4.7.1 Compound Statement

We have three linked lists —*stmt*, *symbol_list*, *struct_list* for each compound statement in CPC AST to hold information in the compound statement. For our convenience and efficiency, we use two stacks to store the *Local_struct_list*, and *Local_undecl* for a block, when we enter a new block, we just push the *Local_struct_list*, and *Local_undecl* of the old block into the stacks, and then empty the two lists so they can be reused in current block. When it is the time to exit the current block, we pop the stacks to get old *Local_struct_list*, and *Local_undecl* to restore the context for the old block. Figure 9 shows an example compound statement containing some simple statements. Now we will describe how to build CPC AST for a compound statement.

```
Increase the nesting level by 1
Push the current two lists --- Local_struct_list, Local_undecl, into
struct_stack and undecl_stack respectively
Initialize them to NULL (so that they can be reused in this compound
statement).
Create a new compound statement STMT node for this compound
statement.
Link the local symbols dynamically to corresponding local symbol lists
  such as 'Local_struct_list', 'Local_undecl' while building CPC AST
  for the list of local declarations in this compound statement.
Link the local statements dynamically to corresponding local statement
  list while building CPC AST for the list of local statements in this
  compound statement.

Before exiting a compound statement
```

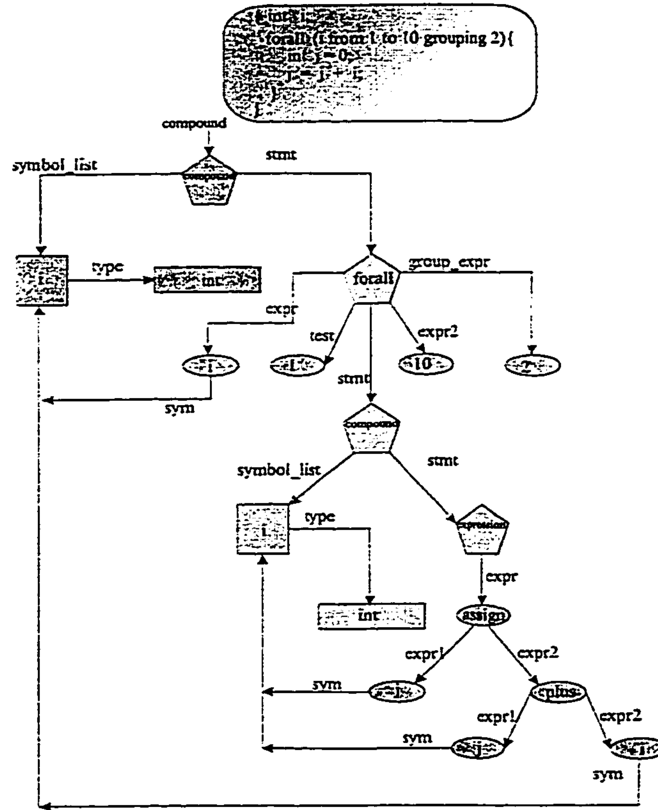

Figure 9: Compound Statement

```
====================================

  Decrease the nesting level by 1
  Remove all the local symbols in this compound statement from both the
    symbol_tab and the struct_tab
  Let the 'struct_list' field in the compound STMT node point to the
    reversed 'Local_struct_list',
  Let the 'undecl' field in the compound STMT node point to the
    reversed 'Local_undecl',
  Let 'symbol_list' field in the compound STMT node point to the local
    declarations' CPC AST
  Let 'stmt_list' field in the compound STMT node point to the local
    statements' CPC AST
```

```
Pop'Local_struct_list', 'Local_undecl' from struct_stack and
  undecl_stack respectively to restore the context before entering
  this compound statement
```

4.7.2 Process Creation and Mapping Statements

fork and *forall* statements are process creation statements in CPC. They have some special features in their syntax. They may have a mapping expression and *forall* may have a grouping expression.

Fields **at_expr** and **port_expr** in *fork* or *forall* STMT node will point to VALUE nodes for its mapping expression and channel variable assignment expression respectively. Field **mapType** will record the information of this mapping type as the 'phyMap' or 'LocalVirMap'. Field **group_expr** in *forall* STMT node will point to the VALUE node for its grouping expression.

When the mapping expression in the **fork** or **forall** statement is not empty, we will create a STMT node for this mapping expression, set the **at_expr**, **port_expr** and **mapType** fields in this node, then copy these fields to the related fields in the **fork** or **forall** STMT node, then discard the mapping STMT node.

Figure 10 shows how to represent a fork in CPC AST.

Figure 11 shows how to represent a forall in CPC AST. Also refer to Figure 9.

4.8 Function Definitions in CPC AST

We create a *function* SYMBOL node to represent the identifier of a function name in a function declaration, and a *function* LINK node will be added to the head of its type chain. If there is a parameter list for this function, let the **sym** field of this function SYMBOL node point to the linked parameter SYMBOL list. Now we will describe the algorithm for the function definition:

```
Create a function SYMBOL node and a function LINK node.
Add the new specifiers LINK node for this function to the function SYMBOL
node type chain
Search the Symbol_tab
If there is a symbol named the same as this function name
```

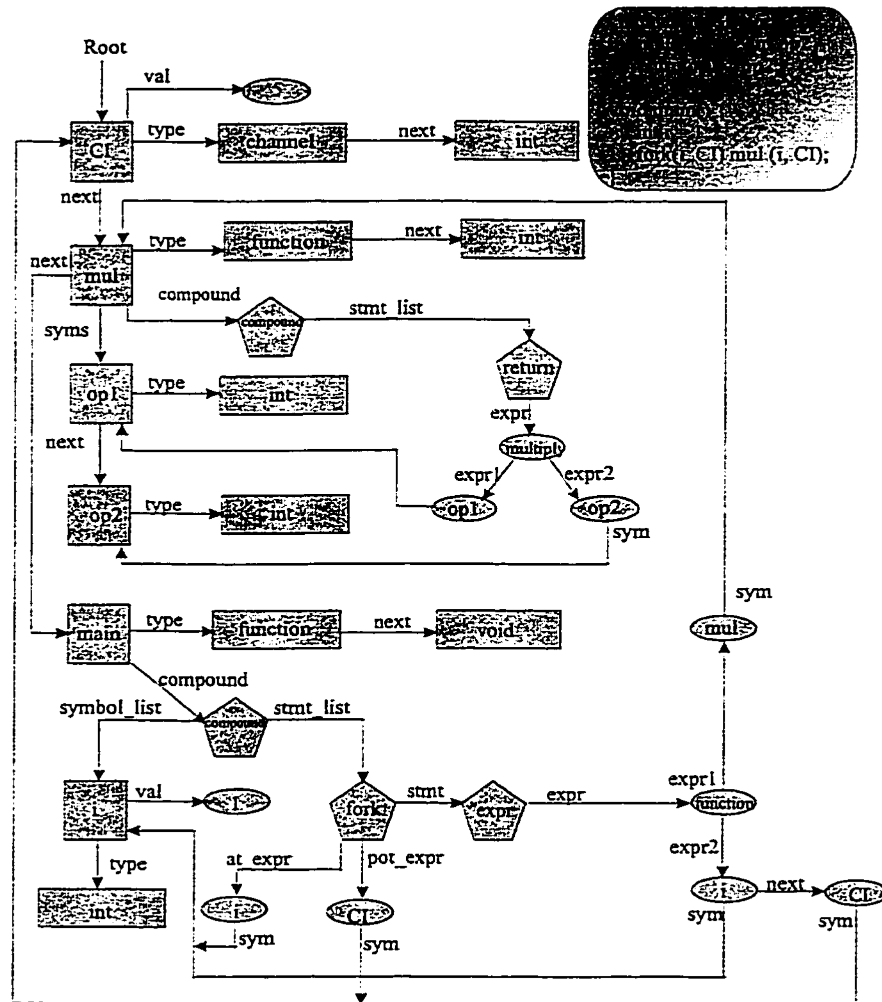

Figure 10: Fork Statement

```
        in the same nesting level as this function declaration
    If the two function declarations are different
      Give "Inconsistent decl and def for function" error information
    Discard the symbol chain and link chain in the existing SYMBOL node
    Copy the new symbol chain and link chain from the new function
      SYMBOL node to the existing SYMBOL node
    Discard the new function SYMBOL node
Else add the symbol of the function identifier to the Symbol_tab
Increase the nesting level by 1
Add symbols of this function's arguments to the Symbol_tab
```

After processing the parameter list for this function declaration,

```
Decrease the nesting level by 1
```

After processing the compound statement for this function,

```
Remove symbols of the function's arguments from the Symbol_tab
Let the stmt field in the function SYMBOL node point to the
  compound STMT node
```

Nested function definition are also supported in CPC.

4.9 CPC AST Type Check—Semantic Analysis

At this point, we know how to build a CPC AST for a CPC program. There is still one more thing to do to complete the CPC AST—Semantic Analysis. We have to go through the existing CPC AST to check if the tree is semantically right; to add more information to the tree to make it complete; to reduce the tree when it's possible; to make the CPC AST simple and convenient for the code generation.

We will traverse the tree recursively to check all the SYMBOL nodes, VALUE nodes, and STMT nodes.

First, we will check for semantic errors in all the symbols in a source CPC program, such as whether each variable has a type and type specifier, whether each function has a return type, whether each struct or union or enumeration has one or more field. A SYMBOL node in CPC AST represents an identifier in the CPC source program. To check all the symbols, we have to go through the SYMBOL nodes in the CPC AST one by one. The algorithm for checking a SYMBOL node is as following:

```
If the SYMBOL node represents a variable
    If the variable doesn't have a type and type specifier
        Give error information
    If the variable has a value, check its value

If it represents a function
    If the function doesn't have a type, type specifier or return type
        Give error information
```

```
    If the function has parameters
        Check the function's parameter symbols
        Set the offset of each parameter
    If the function return type is a function
        Check the argument symbols of the return function
    If the function body is not empty, check the compound statement

If it represents a typedef
    If the typedef doesn't have  a type and a type specifier
        Give error information.

If it represents an enumeration
    If the enumeration has no fields
        Give error information.
    For each element in this enumeration
        Set the offset of the element


If it represents a structure or union
    If the structure or union has no fields
        Give error information.
    else
        Check all the field symbols in the struct or union
        For each element in this struct or union
            Set the offset of the element.

If it represents a bitfield
    If the bitfield has no width
        Give error information.

If it represents an architecture
    If the architecture doesn't have a type or is not defined as an array
        Give error information.
```

```
    If it is defined as SHARED, FULLCONNECT, LINE, RING, HYPERCUBE
        but not a 1-D array,
        Give error information.
```

From above algorithm, we can see that in order to check the semantic errors in a symbol, sometimes we should also check the related value and statement.

To check a value, for example, when there is a function call, we will check whether the function has been defined, whether the argument types match the parameter types in the function definition. We also need to check whether there is access to the component of a channel variable, and whether the types of expressions in a compare or conditional operation match with each other. A value in a CPC source program is represented by a VALUE node in CPC AST.

The algorithm for checking value is as follow:

```
If the value is a constant or for an identifier, we don't need to
check it anymore.

If the value is a function call
    If the identifier of the value is not defined as a function
        Give error information.


If there is arguments in a function call
    and no parameter in the function definition
        Give error information.

If types of arguments in the function call don't match
     the parameter types in the function definition
   or the number of arguments is not equal to the number of
     parameters in the function definition
        Give error information

If it is one of the following operators : pre-increase, pre-decrease,
  post-increase, post-decrease, address of
```

```
    If the expression for this operation does not have a left-value
        Give error information.

If it is an operator of indirect addressing
    If the expression does not have a pointer type
        Give error information

If it is an operator of '->' or '.'
    If the expression is a channel type
        Give error information:
   "Access to component of a channel variable is forbidden in CPC language"
    If the left value of -> is a pointer to a struct/union
            or the left value of '.' is a struct/union
        If the right values are not valid fields of a struct/union
            Give error information

If it is an operator of array access
    If the expression has a channel type
        Give error information:
   "Access to component of a channel variable is forbidden in CPC language"

If it is a conditional or compare operator
    Check whether the expression types match with each other

If it is an assignment operator
    Check whether the expression types match with each other
    If this assignment does not combine channel read
      and channel write, check whether the expression on the left of
      the assignment has a left-value.

...
```

To check the statements in CPC source program is to guarantee that the statements are legal statements in CPC language. For example, the expression in *switch*

and *case* statements must be integer_valued expression, *fork* and *forall* at expression should also be integer_valued expression, port expression should be valid channel expression.

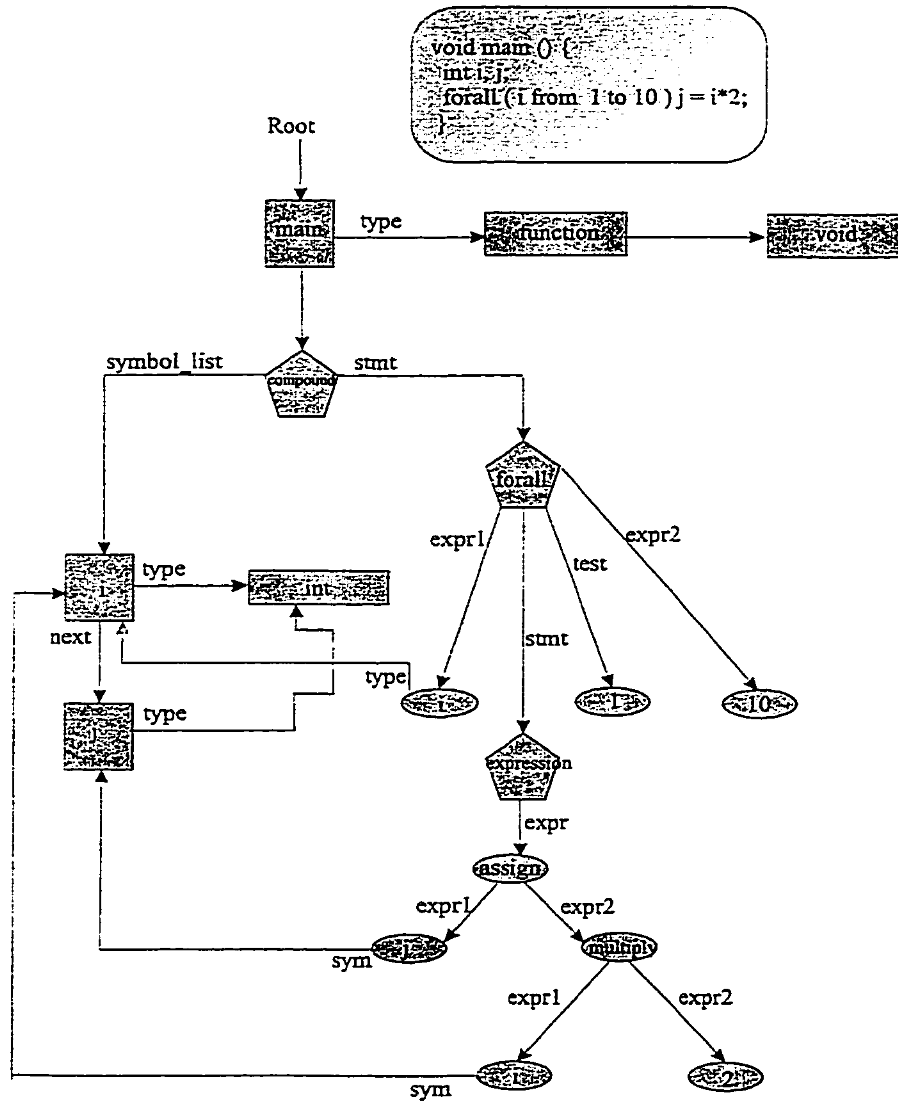

Figure 11: Forall Statement

Chapter 5

Code Generation

The frontend of CPCC produces an abstract syntax tree CPC AST on which the static semantic analysis has already been performed. This analysis includes type checking and scope resolution. The abstract syntax tree can be traversed by the backend, and subsequently manipulated. In this chapter, we describe the backend of CPCC, including the data structures and execution environments, virtual codes and the algorithm employed by CPCC backend.

5.1 Tasks of CPCC Backend

The CPCC code generator has two main tasks: it has to allocate storage for all static data such as constants and initialized data, and for all variables other than those which will be allocated dynamically during program execution. The second task is the generation of the virtual codes for CPSS.

The intermediate representation for CPC source code — CPC AST, is an input tree to CPCC backend. If there is a subtree in the input tree that matches a rewriting rule, the subtree is replaced by a sequence of matched instructions. The virtual code is generated by a process in which the input tree is reduced by recursively finding subtrees in the input tree.

5.2 Execution Environment

CPCC backend will create a file, which has the same name as the source code file, but with the suffix "**.cod**". This file contains a vcode table, an identifier table, a block table, a channel/pointer table, a real constant table, a string table, an array table, a breakability table, a source code table, a table of virtual architecture specification and also some other information like `DEFINED LEVEL 0 FUNCTIONS`, `DEFINED LEVEL 0 VARIABLES`. These tables will be used by CPSS to support the mapping of the virtual processors to the physical processors and the debugging environment.

- The vcode table will hold the information about the line number of the vcode, the related source code line number for this vcode, the function number of this vcode, the mnemonic of this vcode, and the two auguments for the vCode.
- The table of virtual architecture specification will hold the information about the virtual topology, dimension, and number of virtual processors.
- Identifier table is designed to hold the information about the identifier name, size, static level, object category, and also parameter type (call-by-reference or call-by-value) if it is a parameter.
- Block table is designed to hold the information about the size of all parameters in this function, the size of all parameters and local variables in this function, the identifier table indices for the last identifier and last parameter in this function.
- Array table is designed to hold information about the upper bound and lower bound of the index, element type, array size, etc. Channel/pointer table is designed to hold information about the channel/pointer variable name, size, type, etc.
- String table is designed to store all the C string constants.
- Source code table is designed to hold the CPC source code and put a line number in front of each line.
- Breakability table is designed to hold for each source code line number the first vCode line number of its generated vCode program, so that when the user set a

breakpoint, the debugger can follow the pointer to the vCode table and retrieve the starting and ending vCode line numbers for a breakable source line.

In CPC language, local variables are created on function entry and destroyed on function exit, so we can use stack to hold them. Stack in CPCC backend is a data structure that supports two operations, push and pop. A complete activation record of a function will be pushed on the stack upon invocation, and popped off upon exiting from the function. We treat the stack as a big array S[]. Some special pointers are kept with respect to the stack, one is current stack top pointer T, one is the activation record bottom pointer B, and we also have a current $frameTop$ pointer, all stack accesses are relative to these pointers. The area on the stack devoted to the local variables, parameters, return address, and other temporaries for a function is called the function's activation record or stack frame. The positive difference between the pointer B and the current $frameTop$ is the frame size of the current function. Figure 12 shows a typical stack frame layout.

Suppose a function g() calls a function f(), we say g is the caller and f is the callee. On entry to f, the stack pointer points to the first argument that g passes to f. On entry, f is allocated a frame by simply subtracting the frame size from the stack pointer B.

CPC language allows nested function declarations, so the inner functions may use variables declared in outer functions.

To make this work, the inner function must have access not only to its own frame but also to the frames of its outer function. We use static link to accomplish this in CPCC.

Whenever a function f is called, it can be passed a pointer to the frame of the function statically enclosing f; this pointer is the static link. Thus a global array $display$[] should be maintained for static links. This array contains, in position i, a pointer to the frame of the most recently entered function whose static nesting depth is i.

For each function call, a stack frame is allocated in S[]. B holds the base address for the stack frame, pointing to the first word of Link.

Link contains 9 words. Link words link the related stack frames together. The meaning of each of the 9 link words are defined as follows:

- B+0: Return value of the current function call.

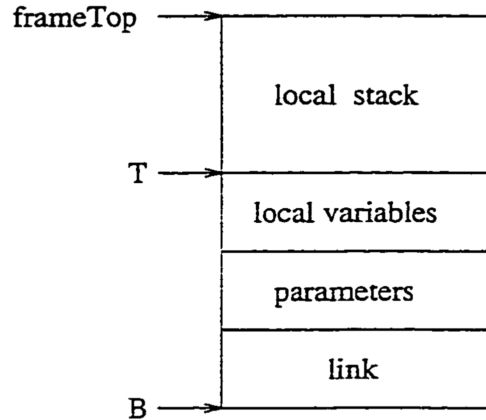

Figure 12: A stack frame

- B+1: Return address of the calling function (value of PC of caller).
- B+2: Static link – the value of B for the stack frame the function of which contains directly the declaration of the function for the current stack frame. It is also the display value for the previous lexical level.
- B+3: Dynamic link – the value of B for the caller's stack frame.
- B+4: Identifier table index for the function.
- B+5: Reference counter for the current frame (initialized to 1). It equals to number of stack frames that are referring to this frame.
- B+6: Size of this stack frame.
- B+7: Value of T for the caller's stack frame.
- B+8: Value of stackTop for the caller's stack frame.

The parameters are allocated space above the 9 link words. The local variables are allocated space above the parameters. Each parameter or local variable V is represented by a relative address which is a pair of integers (level(V), offset(V)), where level(V) denotes the lexical level in which V is declared, and offset(V) denotes the offset of V relative to B, the starting address of the current stack frame.

The active stack frame at a lexical level is the stack frame for the function declared at this level and called the latest. An array display[ ] is used to speed up the address

calculation for variables from their offset format. For any integer i>=0, display[i] holds the value of B of the active stack frame at lexical level i.

Given variable V with relative address (level(V), offset(V)), its absolute address in S[ ] is display[level(V)] + offset(V).

The collection of the 9 link words, the words for parameters, and the words for local parameters is called the activation record for the function.

Above the activation record is a local stack for evaluation of expressions within the current function body. Variable T points to the stack top. T is initialized to point to the word immediately below the local stack area. frameTop holds the index for the last word of the stack frame, which bounds the growth of the local stack.

Each process has a unique local stack for the evaluation of expression in case the body of the process is an expression. Variable *base* holds the starting address of this local stack and never changes value during the life of the process.

The variables B, T, frameTop, base, and PC (program counter) are all defined in a process descriptor.

Some constants:

`LINK_SIZE`: number of words in a link (=9)

`STACK_SIZE`: size of a local stack (=30)

S[T] refers to the stack top. Given variable V, we use level(V) to denote the nesting level of the block in which V is declared, and offset(V) the displacement of V in its stack frame from the beginning of the stack frame (B). Given the level number P of the currently active block, the starting address of the stack frame (B) for this block is display[P]. The address of a variable V on stack is display[level(V)] + offset(V).

5.3 Virtual Codes

The CPC AST generated by the CPCC front end must be translated into assembly language or machine language. The CPC AST does not correspond exactly to machine languages, so we must translate by specific translation rules.

Finding the appropriate virtual instructions for CPSS to implement CPC AST is the job of the instruction selection phase of CPCC.

Virtual code is the target language chosen for CPSS. It has the following features:

- low level

- easy to generate
- can be written in an architecture-independent manner

Now we explain some of them:

`LDAddr` means loading address of variable V onto stack.
`Dereference` will replace the pointer by a value pointed to by the pointer.
`NewForkChild` will begin a new fork process.
`ForkJump` is a special jump with *Fork*. After execution of this instruction, the accumulation of sequential execution time for this process will be resumed.
`NewFrame` will set up a new stack frame for a function on the stack.
`LDCHwOffset` will load value onto stack from channel with offset.
`STChannel` will store value from top of stack into channel.
`BeginParallel` will begin a parallel execution.
`NewForallChild` will begin new forall body process.
`EndParallel` will end the parallel execution.
`BeginForallLoop` will begin FORALL loop.
`EndForallLoop` will end FORALL loop.
`DadLDForallIndexVal` will load forall index value onto the stack from index id.
`SonLDForallIndexVal` will load value of forall index onto stack from child process' base.
`JOIN` V-operation on join semaphore.
`MVChannVar` will assign the owner of a channel var to a new processor.
`SeqOn` will resume the accumulation of sequential time.
`SeqOff` will stop accumulating sequential time.

5.4 Storage Allocation

In CPCC backend, we will first allocate storages to all the external variables and functions. For each function, we have to allocate space to its parameter and nested functions recursively, then allocate space for its local variables and generate codes for all the statements in its compound statement. If the scope level is 0, we will generate initialization codes for starting execution. Starting at the root of a CPC

AST, the entire tree will be traversed and the virtual codes for CPSS will be generated recursively.

There is one entry for each identifier in the identifier table, one entry for each function in the block table, one entry for each 1-D array in the array table, and one entry for each channel or pointer variable in channel table.

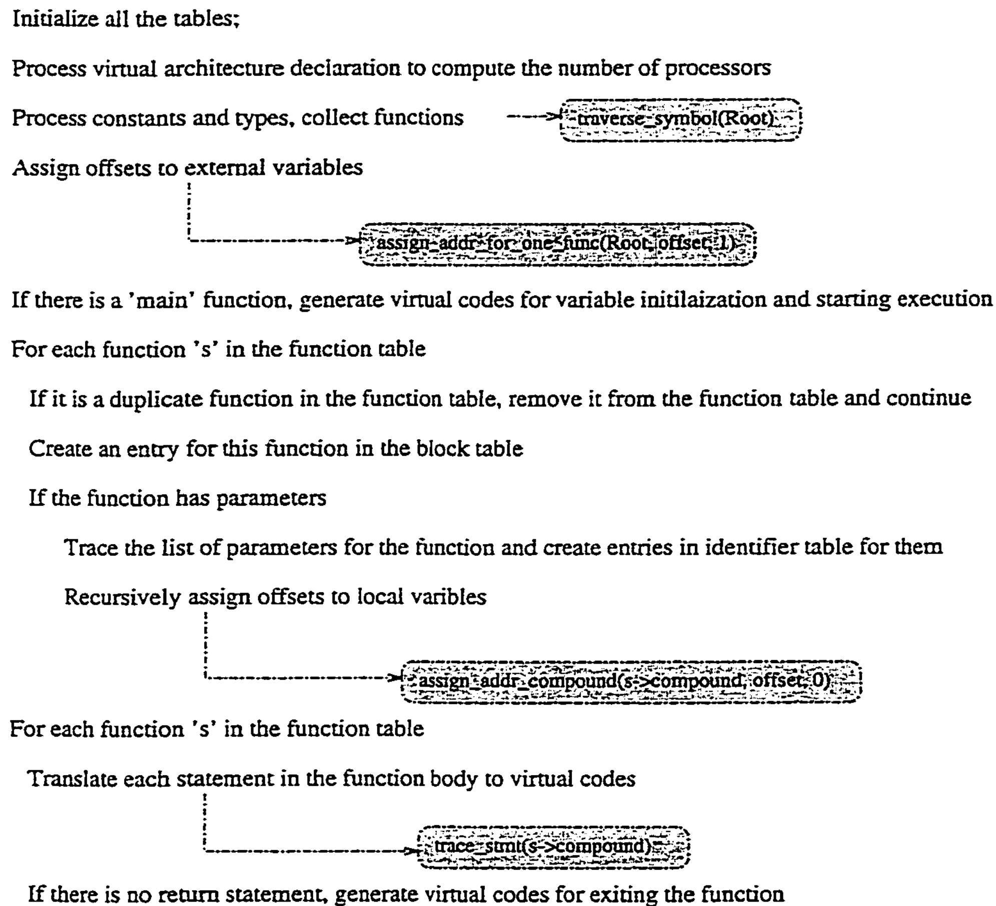

Figure 13: Traverse tree

Figure 13 is the algorithm for traverse and AST tree. With this algorithm, the whole CPC AST will be traversed and storage will be allocated to varibles, and virtual code for each CPC statement will be generated accordingly. This algorithm is composed by some other algorithms like `travers_symbol`, `trace_type`, `assignAddr`. These algorithms are deiscribed in Figure 14, Figure 15, Figure 16 respectively. `Trace_stmt` is used to translate statement to virtual code, we will give some examples

[illegible]

—— makes depth-first search to process constants and types

for (; s; s=->next){

If this SYMBOL node is an ARCH type, continue

If this SYMBOL node is a FUNCTION type

If it has a compound or it represents an external function

Create an entry for this function in the function table

If the SYMBOL represents 'main' function

Copy its index in the function table to 'mainFuncTabIndex'

If it has no compound, continue

trace_type(s->type)

traverse_value(s->value) → Process constants

traverse_symbol(s->syms)

traverse_stmt(s->compound) → Traverse symbols, statements and values in this block

}

Figure 14: traverse symbol

to see what vcodes are generated for a CPC statement in the next section.

5.5 Statement Translation

CPCC uses standard groups of instructions, often called a translation rule, which can be simply incorporated into the code generator whenever a particular STMT node is encountered in CPC AST.

Now let's see what virtual codes are generated through the statement translation rules by some examples.

```
void trace_type([illegible])
If node t is nil or has been processed, return
If node t represents  a  declarator
    If it is a pointer or channel type LINK node
        Create an entry for it in channel table
        Return
    If it is an array type LINK node
        Create an entry for it in array table
        [ trace_type(t->next) ]
        Return
    If it is a function type LINK node
        [ trace_type(t->next) ]
If node t represents a specifier
    If it is a scarlar type, return
    If t is a union or struct type
        Create an entry for the tag in identifier table
        for (s = t->SYM->syms;  s;  s = s->next){
            Create an entry for s in identifier table
        }
        Return
    If t is a typedef type
        [ trace_type(t->SYM->type) ]
        Return
Else  give error information("wrong type")
```

Figure 15: Trace type

5.5.1 Process Creation

Fork

For a fork statement

```
fork [10; c2] f2(a1);
```

the following virtual codes will be generated:

`SequentialTimeOff`—Stop accumulating sequential time

`LoadIntegerLiteral`— load integer 10

`LoadAddress`—load address of `c2`

```
[illegible]

If (!s) return;
For (; s; s = s->next) {
    If s is an external function
        assign_addr_compound(s->compound, offset, 1)
    If s is not a variable, continue;
    If s has been defined as an external variable and the current scope level is higher than 0
        Create an entry for s in identifier table
}

assign_addr_compound(STMT c, int offset, int forExternal)

If the c STMT node is empty or is not a compound type, continue
for (st = c->stmt; st; st = st->next) {
    If STMT node -- st  is for a compound
        assign_addr_compound(st, offset, forExternal)
    If STMT node -- st->stmt is not empty and is for a compound
        assign_addr_compound(st->stmt, offset, forExternal)
    If STMT node -- st->stmt is not empty and is for a compound
        assign_addr_compound(st->else_stmt, offset, forExternal)
}
```

Figure 16: Storage allocation

`MoveChannelVarToNewProc`—assign the owner of a channel variable `c2` to a new processor

`CreateNewForkChild`— Begin new fork process

`ForkJump`—resume the sequential execution time for this process

`NewFrame`—set up a new stack frame for `f2`

`LoadAddress`—load address of `a1`

`LoadBlock`—load block of `f2`

`WakeupProcess`—Child process wakes up parent after argument evaluation

`Call`—call function `f2(a1)`

`UpdateDisplay`—Update display table

ForkChildEnd—End forked process

Forall

For a forall statement

```
forall(i from 0 to 9 grouping 1)
  [i;c3[i]] f(i, a1[i]);
```

the following virtual codes will be generated:
LoadAddress—load address of variable i
LoadIntegerLiteral—load integer 0 on stack top
LoadIntegerLiteral—load integer 9 on stack top
LoadIntegerLiteral—load integer 1 on stack top
BeginParallel—begin parallel processing
BeginForallLoop—Begin FORALL loop
DadLoaDForallIndexVal—Load forall index value i onto the stack from index id
LoadAddress—load base address for array c3
DadLoaDForallIndexVal—Load forall index value i onto the stack from index id
ArrayIndexing—pop i and base address for array c3, push c3[i] on stack
MoveChannelVarToNewProc—assign the owner of a channel variable c3[i] to a new processor
CreateNewForallChild—Begin new forall process
Jump—jump
NewFrame—create a new stack frame for this child
SonLoadForallIndexVal—Load forall index value i onto the stack from index id
LoadAddress—load base address for array a1
SonLoadForallIndexVal—Load value of forall index i onto stack from child process' base
ArrayIndexing—pop i and base address for array a1, push a1[i] on stack
WakeupProcess—Child process wakes up parent after argument evaluation
Call—call function f(i, a1[i])
UpdateDisplay—update display table
TestGroupIncForallIndex—Test FORALL grouping and increment index at the end of FORALL process

`ForallChildEnd`—End forall child process
`EndForallLoop`—End FORALL loop
`EndParallel`—stop parallel processing

5.5.2 Process Communication — Channel expression

Read Channel

For a channel read statement

```
y = c3[x];
```

the following virtual codes will be generated:
`LoadAddress`– load base address for array c3 on stack top
`LoadValue`—-load value x on the stack top
`ArrayIndexing`—-pop x and base address for array c3, push c3[x] on stack
`LoadCHwAdrOnStack`— load absolute address of the channel variable on stack
`Dereference`— replace stack top pointer by value of c3[x]
`LoadAddress`— load address for y on stack
`Store`– assign value of c3[x] to y and pop off 2 arguments from top

Write Channel

For a channel read statement

```
c2 = a;
```

the following virtual codes will be generated:
`LoadAddress`—load address of variable a on stack top
`CopyToNewBlock`— Allocate a new stack block and copy the existing block of a into it
`LoadAddress`—load address of c2 on stack top
`StoreChannel`— Store value a into channel

Binding Channel Variable To Processes

Please refer to the examples in section 5.5.1 and 5.5.1.

5.5.3 Virtual Architecture

Virtual Architecture Declaration

We don't generate vcode for the virtual architecture declaration, instead this information is stored in the **virtual arch spec** table. At run time, CPSS will map the virtual processors to physical processors based on this table and the mapping function.

Process-to-virtual-processor Mapping

Refer to the examples in Section 5.5.1.

5.6 A Complete Example

Please refer to appendix D.3.

Chapter 6

CPC preprocessor

CPC preprocessor is a separate program that runs before the compiler, with its own simple, line-oriented grammar and syntax. It's designed to suit our own needs, to make CPCC a complete and portable. Briefly, the preprocessor has the following functions:

- Macro processing: CPC macros allow a user to define shorthands for longer constructs.
- Inclusion of additional C source files: CPC users can include header files into their program texts.
- Conditional compilation: CPC users are able to conditionally compile sections of CPC source code contingent on the value of an arithmetic expression.

All preprocessor directives begin with a pound sign (#) which must be the first non-space character on the line. They may appear anywhere in the source file : before, after, or intermingled with regular CPC language statements. However, the pound sign (#), which denotes the beginning of a preprocessor directive, must be the first non-space character on the line.

Unlike CPC statements, a macro command ends with a newline, not a semicolon. To span a macro over more than one line, enter a backslash immediately before the newline, as in :

```
#define stack_clear(stack) ( (p_##stack)  = (stack + \
                                        sizeof(stack)/sizeof(*stack)) )
```

The following preprocessor directives are supported : #define, #include , #undef, #if, #ifdef, #ifndef, #endif

Macro substitution

A macro is a name that has an associated text string, called the macro body. In the following declaration,

```
# define max 100
```

max is the macro name, and *100* is the macro body.

When a macro name appears outside of its definition (referred to as its invocation), it is replaced with its macro body. The act of replacement is referred to as macro expansion. For example, having defined above, you might write:

```
int line[max];
```

During the preprocessing stage, this line of code would be translated into:

```
int line[100];
```

We also support the operator(##) that pastes two tokens. For example, if there is a macro

```
#define pop(stack) ( *p_##stack++ )
```

then the sequence pop(ident) will be expanded to (*p_ident++).

Conditional compilation

The preprocessor enables user to screen out portions of source code that they don't want to compile. This is done through a set of preprocessor directives that are similar to the `if` and `else` statements in the CPC language. The preprocessor versions are #if , #else, #elseif, #endif.

Comment removal

Like ANSI C, to insert a comment into a CPC program, you can surround it with double-character symbols /* and */. Comments in this style may extend over one or more lines, and you can also insert another comment of this style inside it. C++ style comments are also supported.

Algorithm for preprocessor

For each source file to the preprocessor, we will create a new file after preprocess, which will be the input file to the CPCC front end. We have a definition table to store each macro in the user progarm. If there is a new definition, we will create a new entry for it. If it is a redefinition, we replace the old definition with the new one. If there are parameters in the definition, they will be recorded in the entry for this definition. Before we describe the algorithm for CPC preprocessor, we explain the major data structure first.

The data structure of the entry for definition table is defined as follow:

```
struct Definition {
  char name[MAX_TOKEN_LENGTH],      // macro name
       def[MAX_DEF_LENGTH];         // macro body
  int  map[MAX_PLACE_HOLDER_NUM];
       // the i-th position in def is for the map[i]-th arg
  int  numParam;                    // number of parameters
  int  numPlaceHolder;              // number of place holders
};
```

For example, if the macro is

```
#define max(x,y) if (x >= y) m = x;  else m = y;
```

then the fields in the entry for this macro have the following values:

```
name : max
def  : if (  >=  ) m = ;  else m = ;
```

The jth position in *def* where the parameter should be inserted in the future is recorded as the index j of the array map, and map[j] records the index of the parameter which should be substituted. In this example, map[5] has the value 1 (parameter x), map[10]is equal to 2 (parameter y) and numPlaceHolder = 4 (number of parameters appeared in the definition).

The algorithm for CPC preprocessor is as follow:

```
void process_file(char fileName){
  FILE  *fdIn;       // file descriptor of the current file
  int lineNum = 0;  //line number
  char  line[MAX_LINE_LENGTH];  // buffer for the current line

  while the next line in the input file is not empty
     lineNum++;
     process_line(fdIn, fileName, lineNum, line);
     output this new line to temperary file
}

process_line{
  If the line doesn't begin with a #
      getToken;
     while not at the end of the line{
        Look up the definition table
           If there is a match
              If this definition need parameters
                 If the followed tokens in this line doesn't match these
                 parameters, process_error
              Replace these tokens with its definition
        getToken;
      }

  If the line begins with a #include
     Get the name of the file which the user wants to include
     process_file(included file)

  If the line begins with a #define
     Create an entry for this macro
     Look up the definition table
     If it has been defined, cover the old definition by the new one
     Else enter the new one
```

```
  If the line begins with a #undef
     Get the symbol to be undefined
     Look up the definition table
     If the symbol has been defined before, delete it from the
     definition table

  If the line begins with a #ifdef
     If the symbol in the sentence had been defined
                  process_ifdef(TRUE)
     Else         process_ifdef(FALSE)

  If the line begins with a #ifndef
     If the symbol in the sentence has not been defined
                  process_ifdef(TRUE)
     Else         process_ifdef(FALSE)
}
```

Chapter 7

Conclusion

Performance levels for single processor systems are reaching their limits as it becomes increasingly difficult and expensive to shorten clock cycles. To go beyond such limits, designers are turning to parallel architectures as a cost effective way to significantly increase processing power by several orders of magnitude. But, impeded by the *portability* of parallel programming and some other factors, the parallel systems do not grow rapidly as expected. The parallel language and its compiler are the main aspects that affect the *portability*. Our research effort focuses on providing ease-to-use and portable parallel programming language CPC and its compiler CPCC.

In this thesis, we have presented the design and implementation of the CPC language. The CPC language is designed as an explicit parallel programming language. It is based on C, and it provides new parallel features to create and terminate processes, and to communicate between processes via channels. CPC is based on the virtual architecture aproach. The virtual architecture of an algorithm captures the comunication pattern of the algorithm. With virtual architecture approach, the CPC user can declare the most natural and efficient architecture in the CPC program, so that the communication pattern best fits the algorithm, therefore improve the program performance. CPC program is described on a virtual architecture with desired size and topology. At run time, the virtual architecture program will be mapped to the available physical architecture. The virtual architecture information in the CPC program will be used by the CPC compiler and run-time system to perform mapping of the computation to the physical processors. With the mapping, the virtual architecture will be independent from the physical architecture which helps make the

CPC programs portable. Suitable mapping of the regular communications can reduce the communication overhead of the program. CPC users have the freedom to specify the mapping functions.Virtual architecture algorithm is easy to implement since the programmer can describe the algorithm on the virtual architecture matching that of the algorithm and let the compiler and runtime system perform the necessary translation to account for the mapping of the computation and handle the communication among the virtual processors.

The CPC language makes CPPE an excellent environment for developing and fine-tune parallel programs, because

- The CPC language is easy to learn.

  CPC language is an explicit message passing language based on C, it won't be difficult for the user to learn some essential extensions regarding how to declare a virtual architecture, how to create and terminate a process, how to map processes to virtual and then to physical processors, and how to communicate among processes. Therefore, CPC language is easy to learn, especially for C programmers.

- The CPC language is ease to use.

  The virtual architecture approach let the user describe the algorithm on the virtual topology matching well with the algorithm. Users can solve their specific needs in a way that best fits the model of their application. This makes CPC programs most natural to the communication pattern. Therefore CPC program is easy to write, and CPC language is easy to use.

- CPC programs are efficient.

  The virtual architecture captures the communication pattern of the CPC program, and the mapping scheme can minimize the communication cost, balance the workload among physical processors. This can improve CPC program performance greatly, and make CPC programs efficient.

- CPC programs are portable across different platforms.

  CPC compiler has been divided into frontend and backend. Intermediate representation, CPC AST, is generated by the frontend to keep all the information of the source program, such as data declarations, expressions, and statements.

The backend is a code generator which will traverse the CPC AST recursively to allocate storage and generates codes for target architecture. The frontend is shared and reusable. With CPC AST and a new backend, it is convenient to translate CPC code into real machine languages, for all the desired target machines. Therefore, CPC program is portable across different platforms.

- CPC program is portable across different partitions of a parallel system.

  When the system loads the executable code, each physical processor will get a mapping table. With the two level mapping, a mapping table will be created at run time when the system loads an executable image. It contains the information about processor's physical ID and IDs for the virtual processors that the physical processor should run. As a result, the virtual processors are dynamically mapped to physical processors to execute the code. Therefore, with the support of a mapping table, a virtual architecture becomes independent of physical architectures, CPC executable code is portable among subsystems, and CPC source code is portable across different platforms.

Appendix A

CPC Builtin Functions

Protypes for CPC builtin functions are as follows:

```
int  printf(char format, ...);
int  fprintf(int fp, char *format, ...);
void  scanf(char *format, ...);
void  fscanf(int fp, char *format, ...);
int   fopen(char *filename, char *mode);
void  fclose(int fd);
int   getchar(void);
int   fgetc(int fp);
int   abs_id(void);
void  cart_id(int &array[]);
int   min(int, ...);
float fmin(float, ...);
int   max(int, ...);
float fmax(float, ...);
int   abs(int);
float fabs(float);
float sin(float);
float cos(float);
float tan(float);
float sqrt(float);
int   odd(int);
```

```
int   even(int)
int   floor(float);
int   ceil(float);
float exp(float);
float pow(float, float);
float log(float);
float log10(float);
float atof(const char *);
int   *malloc(int num_of_words);
void  free(int start_add);
void  delay(int);
void  join(void);
void  lock(int *);
void  unlock(int *);


float rand(void);
void  srand(int seed);


void  vsend(int vpid, char *buf, int size);
void  vrecv(char *buf, int size, int vpid);
void  strcmp(char *str1, char *str2);
void  strcat(char *to_str, char *from_str);
void  strcpy(char *to_str, char *from_str);
int   strlen(char *str);
void  barrier(int nbrProcesses);
void  globalMaxInt(int src, int *dest);
void  globalMaxFloat(float src, float *dest);
void  globalMinInt(int src, int *dest);
void  globalMinFloat(float src, float *dest);
void  globalAnd(int src, int *dest);
void  globalOr(int src, int *dest);
```

```
void  globalPrefixInt(int src, int *dest, int sequenceNum);
void  globalPrefixFloat(float src, float *dest, int sequenceNum);
void  send(char *msg, int size, int tag, int dest);
void  receive(char *msg, int size, int tag);
int   probe(int tag, int size);
void  bcast(char *msg, int size, int tag, int root);
void  who(int *numProc, int *me, int *host);


typedef channel int CHANNEL[1];  /* [1] : trick to force generating
                                    CopyBlock when writing to channel*/

CHANNEL *ChannAlloc(int num_chann, int chann_size)
typedef struct
int v_topo
int num_dim
int dim_size[20]
} *ARCHITECTURE


ARCHITECTURE VtopoCreate(int topo, int num_dim, ...);


#define LINE
#define RING 1
#define MESH 1
#define TORUS 1
#define HYPERCUBE 1
#define FULLCONNECT 1
#define SHARED 1


typedef int spinlock;
```

```
void virtual_processor(int &array[]);
void physical_processor(int &array[]);
int clock(void);
int seqtime(void);
void seqon(void);
void seqoff(void);
void timeOn(void);
void timeOff(void);


#define stdin
#define stdout
#define stderr
```

Appendix B

CPC Yacc Grammer

```
program        : def_list
               || def_list def
specifiers
               : type_class_qual
               || specifiers type_class_qual
               ;
type           : type_specifier
               || type type_specifier
               ;
type_class_qual
               : type_specifier
               || CLASS
               || TYPE_QUAL
               ;
type_specifier
               : TYPE
               || enum_specifier
               || struct_specifier
               || TTYPE                                    ;
var_decl
               : new_name
               || var_decl LP RP
```

```
                || var_decl LP var_list RP
                || var_decl LB RB
                || var_decl LB const_expr RB
                || STAR var_decl                          %prec SIZEOF
                || LP var_decl RP
                ;
new_name        : NAME
                ;
decl_list
                : decl
                || decl_list COMMA decl
                ;
decl            : var_decl
                || var_decl EQUAL initializer
                || funct_decl
                ;
funct_decl
                : STAR funct_decl
                || funct_decl LB RB
                || funct_decl LB const_expr RB
                || LP funct_decl RP
                || funct_decl LP RP
                || funct_decl LP var_list RP
                || new_name LP RP
                || new_name LP name_list RP
                || new_name LP var_list RP
                ;
name_list
                : new_name
                || name_list COMMA new_name
                || name_list COMMA ELLIPSIS
                ;
var_list
```

```
                : param_declaration
                || var_list COMMA param_declaration
                || var_list COMMA ELLIPSIS
                ;
param_declaration
                : type var_decl
                || type AND var_decl
                || abstract_decl
                ;
abstract_decl
                : type abs_decl
                ;
abs_decl
                : ϵ
                || LP abs_decl RP LP RP
                || LP abs_decl RP LP var_list RP
                || STAR abs_decl
                || abs_decl LB RB
                || abs_decl LB const_expr RB
                || LP abs_decl RP
                ;
struct_specifier
                : STRUCT opt_tag LC struct_def_list RC
                || STRUCT tag
                ;
opt_tag         : tag
                ||
                ;
tag             : NAME
                || TTYPE
                ;
struct_def_list
                : struct_def_list struct_def
```

```
            || ε
            ;
struct_def
            : specifiers struct_decl_list SEMI
            ;
struct_decl_list
            : struct_decl
            || struct_decl_list COMMA struct_decl
            ;
struct_decl
            : var_decl
            || var_decl COLON const_expr                %prec COMMA
            || COLON const_expr                         %prec COMMA
            ;
enum_specifier
            : ENUM tag LC enumerator_list RC
            || ENUM tag
            || ENUM LC enumerator_list RC
            ;
enumerator_list
            : enumerator
            || enumerator_list COMMA enumerator
            ;
enumerator
            : new_name
            || new_name EQUAL const_expr
            ;
compound_stmt
            : LC def_list stmt_list RC
            ;
def         : TYPEDEF specifiers var_decl SEMI
            || specifiers decl_list SEMI
            || specifiers SEMI
```

```
            || specifiers funct_decl c_style_param_def_list compound_stmt
            || specifiers new_name compound_stmt
            || ARCH ARCH_TYPE var_decl SEMI
            ;
c_style_param_def_list
            : c_style_param_def_list c_style_param_def
            || /* epsilon */
            ;
c_style_param_def
            : specifiers c_style_param_decl_list SEMI
            ;
c_style_param_decl_list
            : var_decl
            || c_style_param_decl_list COMMA var_decl
            ;
stmt_list
            : stmt_list statement
            || ε
            ;
opt_mapping
            : LB opt_expr SEMI opt_expr RB
            || LB opt_expr RB
            || ε
            ;
opt_grouping
            : GROUPING expr
            || ε
            ;
statement
            : SEMI
            || compound_stmt
            || expr SEMI
            || RETURN SEMI
```

```
            || RETURN expr SEMI
            || GOTO target SEMI
            || target COLON statement
            || IF LP expr RP statement
            || IF LP expr RP statement ELSE statement
            || WHILE
            || DO
            || FOR LP opt_expr SEMI opt_expr SEMI opt_expr RP statement
            || BREAK SEMI
            || CONTINUE SEMI
            || SWITCH LP expr RP compound_stmt
            || CASE const_expr COLON
            || DEFAULT COLON
            || FORK opt_mapping statement
            || FORALL LP non_comma_expr FROM expr TO expr
            opt_grouping RP opt_mapping statement
            ;
target      : NAME
            ;
unary       : LP expr RP
            || ICON
            || FCON
            || CHARCON
            || NAME
            || string_const                           %prec COMMA
            || SIZEOF LP string_const RP              %prec SIZEOF
            || SIZEOF LP expr RP                      %prec SIZEOF
            || SIZEOF LP abstract_decl RP             %prec SIZEOF
            || LP abstract_decl RP unary              %prec UNOP
            || MINUS unary %prec UNOP
            || UNOP unary
            || QUEST unary
            || unary INCOP
```

```
                || INCOP unary
                || AND unary                                          %prec UNOP
                || STAR unary                                         %prec UNOP
                || unary LB expr RB                                   %prec UNOP
                || unary STRUCTOP NAME                                %prec STRUCTOP
                || unary LP expr RP
                || unary LP RP
expr            : expr_reverse
                ;
expr_reverse
                : non_comma_expr
                || expr_reverse COMMA non_comma_expr
                ;
non_comma_expr
                : non_comma_expr QUEST non_comma_expr
                COLON non_comma_expr
                || non_comma_expr ASSIGNOP non_comma_expr             %prec EQUAL
                || non_comma_expr EQUAL non_comma_expr
                || or_expr
                ;
or_expr         : or_list
                ;
or_list         : or_list OROR and_expr
                || and_expr
                ;
and_expr        : and_list
                ;
and_list        : and_list ANDAND binary
                || binary
                ;
binary
                : binary RELOP binary
                || binary EQUOP binary
```

```
		|| binary STAR binary
		|| binary DIVOP binary
		|| binary SHIFTOP binary
		|| binary AND binary
		|| binary XOR binary
		|| binary OR binary
		|| binary PLUS binary
		|| binary MINUS binary
		|| unary
		;
opt_expr
		: expr
		|| ε
		;
const_expr
		: expr_reverse				%prec COMMA
		;
initializer
		: non_comma_expr			%prec COMMA
		|| LC initializer_list RC
		;
initializer_list
		: initializer
		|| initializer_list COMMA initializer
		;
string_const
		: STRING
		|| string_const STRING
		;
```

The abbreviations of operators in CPC abstract syntax are defined as follows:

```
STRUCTOP : ->   .
INCOP    : ++   --
UNOP     : ~    !
DIVOP    : /    %
SHIFTOP  : <<   >>
RELOP    : <    <=   >   >=
EQUOP    : ==   !=
ASSIGNOP : *=   /=   %=   +=   -=   &=   != ^=   <<=   >>=
```

Appendix C

CPC language manual

C.1 Multiprocessors VS. Multicomputers

The basic concept behind the parallel computer is to simply have more than one processor in the same computer. The use of multiple parallel processors in the same computer system introduces some additional requirements on the architecture of the computer. For many processors to be able to work together on the same computational problem, they must be able to share data and communicate with each other. There are currently two major architectural approaches to fulfilling this requirement: shared memory and message passing.

In shared memory computers, usually called *multiprocessors*, all the individual processors have access to a common shared memory, allowing the shared use of various data values and data structures stored in the memory. All of the processors can compute in parallel, and each is able to access the central shared memory. Each processor continues to read data from the shared memory, compute new values, and write them back to the shared memory. This computational activity is performed by all the processors in *parallel.*

In message passing computer architectures, usually called *multicomputers*, each processor has its own local memory, and processors share data by passing messages to each other through some type of processor communication network. Multicomputer can provide a large local memory for each processor and a communication network for processor interaction via message-passing. A processor has direct access only to its own local memory module, and not to the memory modules attached to other

processors. However, any processor can read data values from its own local memory, and send data to any other processor. Therefore, the data can be freely shared and exchanged among the processors when desired.

C.2 The CPC language features

CPC (Concordia Parallel C) language is based on the popular programming language C and enhanced with new features to support parallel programming. The CPC language preserves most existing sequential features of the C language. Parallel features of the CPC support the creation of parallel processes, the definition of parallel architectures, process communications through channel variables and mapping of parallel processes to physical processors. The CPC language supports both shared-memory and message-passing programming paradigms. This document will describe the differences between the CPC and ANSI C, and the parallel features supported in the CPC.

CPC is a good candidate for explicit parallel programming, and can be used to design, express, and implement efficient portable parallel algorithms. It offers special features, such as the virtual architecture approach, which provide the user with a high level of abstraction of the communication pattern, allowing the user to concentrate on the resolution of their problems without sacrificing performance. Therefore the user need not worry about the architectural features of the target system.

Because the CPC language was not designed with a particular architecture in mind, it allows users to solve their specific needs in a way that best fits the model of their application.

C.2.1 Process and Process communication

CPC is designed to be machine-independent and can run on a wide variety of parallel computers, including multiprocessors with shared memory and multicomputers based on message-passing between processors with local memory.

To create programs for parallel computers, a useful conceptual tool is the notion of a *process*, which is essentially a sequence of operations that can be performed by a single processor. The *process* can be used as the basic building block of parallel programs: each processor executes a particular process at any given time.

CPC has the features that allow the dynamic creation of parallel processes to run on the physical processors.

Since a multiprocessor has a shared memory that is accessible to all the processors. CPC allows data to be shared by parallel processes through the use of *shared variables* (or *global variables*), which are a software abstraction of the shared memory found in multiprocessor computer hardware.

In case of multicomputers, each processor has a local memory, but there is no shared memory. The processors in a multicomputer use a network of communication channels to send messages to each other during computation. Intermediate data values produced during the course of a computation can be transmitted to other processors through these communication channels. As a software abstraction of these communication channels, CPC has *channel variables.* A channel variable can be used to transmit data from one process to another. A process can write a data value into a channel variable, from which it can be read by another process running in parallel. Channel variable in CPC is a conceptual software entity that allows communication between parallel processes via "messages" that are transmitted through the channels.

Because multicomputer have no shared memory, they require a different style of parallel programming. Instead of simply placing data values into the shared memory to be later retrieved by other processors, data values must be explicitly sent to other processors by using messages. This style of parallel programming is often called "messages-passing" style. Since CPC has channel variables as a built in feature, the language can be used for writing message passing style programs for multicomputers.

Actually, channel variable can be also implemented on multiprocessors by using the shared memory. In these shared memory multiprocessors, channel variables also serve an important function to help processes synchronize with each other and exchange data values.

The CPC language is adaptable to both the shared-memory parallel programming paradigm and the message-passing parallel programming.

C.2.2 Virtual Architecture and Two Level Mapping

The overall pattern of the direct processor connections is usually called the multicomputer *topology.* Line, ring, mesh, torus and hypercube are some of the most important multicomputer topologies. For a particular algorithm, the execution delays resulting

from communications will depend on the specific *topology*. In order to improve program performance, reduce the communication delay and program execution time, we introduce *virtual architecture* to CPC language. *Virtual architecture* is the main feature of the CPC language.

In the CPPE (Concordia Parallel Programming Environment), we identify two kinds of processors: *virtual processors* and *physical processors*. The user writes an application using the architecture most natural and efficient to program performance. This architecture is referred to as virtual architecture, and its processors are called virtual processors.

In the application programs, users can declare virtual architecture, which captures the communication patterns among processes. The topology and size of the physical machine may not match that of the virtual architecture. Processors constituting the physical system are physical processors. At run time, the virtual processors are mapped to the available physical processors.

The mapping objectives are to minimize communication cost among communicating processes, and to balance the workload among physical processors. After mapping, the two communicating processes should be situated as close to each other as possible.

There are two levels of program mapping. The first level is the mapping from processes to virtual processors. The second level is mapping from virtual processors to physical processors.

1. Process-to-virtual-architecture mapping. The mapping can be one-to-one and many-to-one. Often in the application program the user specifies the ID of the virtual processor on which a process will run. The virtual processor will be mapped to a physical processor at run time.

2. Virtual-to-physical-architecture mapping. At run time, the user can specify the desired physical architecture for running the compiled virtual-architecture program. The user is asked to select a mapping function provided by the CPSS mapping library, or specify a mapping function himself.

C.3 *Notation*

Throughout this manual, the syntax of the CPC language is described using the commonly known BNF (Backus-Naur Form) notation.

Any symbol or string of characters not otherwise covered below is called a terminal symbol in the sense that it represents itself. A terminal symbol is considered to be a single indivisible symbol.

A sequence of one or more words enclosed in a pair of angular brackets is called a non-terminal symbol and is used to name a syntactic construct of the language. Each non-terminal symbol is defined in terms of terminal and non-terminal symbols.

The symbol `::=` is used in the definition of non-terminal symbols. The non-terminal symbol appearing on its left is defined as consisting of any of the sequences of terminal and non-terminal symbols appearing on its right. Note that a sequence may be empty, thus supporting an optional syntactic component.

Some conventions that improve readability of the grammar have been adopted:

- All keywords and special symbols are in uppercase.
- A prefix *opt* designates a non-terminal which is optional when used on the right side of a definition.
- The suffix *list* denotes a list of objects separated by commas.
- The non-terminal `name` refers to a CPC identifier.
- The non-terminal `statement` refers to a CPC executable statement.
- The non-terminal `expr` refers to a CPC expression.

The grammar displayed in this section has been simplified to more easily explain the subject at hand. Thus, a concatenation of these pieces of grammar will not necessarily provide a sensible grammar for the entire language. However, a complete grammar is provided in Appendix B.

We will describe the differences between the CPC and ANSI C first, then describe the new features in CPC starting from section C.10.

C.4 Function definition

C.4.1 Syntax

The syntax for function definition is different from ANSI C in three aspects:

- Function type can't be omitted
- Functions can be defined inside a compound statement before the first statement of this compound statement.
- If there is no argument for a function, the pair of parenthesis followed by the function name in ANSI C can be omitted in CPC.

CPC supports nested function definition, which will make CPC much more powerful in supporting parallel libraries.

C.4.2 Example

The example was in Figure 17.

```
# include "cpc.h"

void main {          //omit the "()" after main
        int x;

        int f(int y) {  //define a function inside a compound statement
                int z;

                z = 2;
                return y+z;
        }

        x = 1;
        x = f(x) + 2;
}
```

Figure 17: Nested Function Definition

C.5 Names

CPCC sets no limit to the length of any identifier.

C.6 Comments

Like ANSI C, to insert a comment into a CPC program, you can surround it with double-character symbols /* and */. Comments in this style may extend over one or more lines, and you can also insert another comment of this style inside it. C++ style comments are also supported.

C.6.1 Example

Examples is in Figure 18

```
// This is an example of C++ style comment-----//
/* This is an example of C style commnet ---/* nested comment */---- */
  void f() {
          int x;

          x = 1;
}
```

Figure 18: CPC comment

C.7 Call by reference

Parameters to a function are means of passing data to the function. Many programming languages pass arguments to parameters by reference, which means they pass a pointer to the argument. As a result, the called function can change the value of the argument. When passing argument by value, the called function can change the value of the copy, but can't change the value of the argument in the caller routine.

In ANSI C, array parameters always use call-by-reference. CPC supports C++ style call-by-reference. In CPC, all parameters use call-by-value, unless there is an '&' before the parameter name. Array parameters can use either call-by-reference or call-by-value. The former is mainly used for a parallel process to return an array of values. The latter is mainly used to pass initial values to a new process.

C.7.1 Example

Example is in Figure 19.

```
#include "cpc.h"

  void f( int &a[]) {
        printf("%d\n", a[99]);
}

int main() {
        int a[100];

        a[99] = 100;
        f(a);
}
```

Figure 19: Call By Reference

C.8 Inclusion of additional CPC source files

C.8.1 Syntax

The general syntax for `#include` is:

```
#include "filename"
```

OR

```
#include "pathname/filename"
```

The file can be any valid CPC source file. The preprocessor will look for the file in the specified directory. If the pathname has not been specified in the command, the

preprocessor will search the file in the current directory. An environment variable "CPPE" can be used to set the include search path.

C.8.2 Examples

```
#include "cpc.h"
# include "include/include_file_1.c"
# include "include_file_2.c"
```

Figure 20: Include Statement

Figure 20 is an example of CPC include statement.

C.9 Run time library

The following runtime library functions are available: printf(), fprintf(), scanf(), fscanf(), fopen(), fclose(), getchar(), fgetc( ), abs_id(), cart_id(), min(), fmin(), max(), fmax(), abs(), fabs(), sin(), cos( ), tan(), sqrt(), odd(), even(), floor(), ceil(), exp(), pow(), log(), atof(), malloc(), free(),delay(), join(), lock(), unlock(), vsend(), vrecv(), strcmp(), strcat(), strcpy(), strlen(), barrier(), globalMaxInt(), globalMaxFloat(), global MinInt(), globalMinfloat(), globalAnd(), globalOr(), globalPrefixInt(), globalPrefixFloat(), send(), receive(), probe(), bcast(), who()

C.10 Process creation

The most important building block of a parallel programs is the process. Computational activities take place when a process is assigned to a processor in the underlying parallel computer.

In CPC, there are process creation statements whose execution will cause completely new processes to be created and assigned to processors for execution. The "main" program in CPC becomes the first *process* and is assigned for execution to

the first processor. The main program may contain any of the ordinary kind of statements that are found in sequential programs, such as assignments, loops, conditionals, and I/O statements. However, in CPC there is also the possibility of a completely new kind of statement not found in sequential programs: a *process creation* statement. There are certain statements in CPC whose execution will cause completely new processes to be created and assigned to processors for execution. This is how parallel activities are initiated in the program: an existing process that is already running on a processor executes a process creation statement. The created process is sometimes called the child process, while the creator process is called the parent process.

A process can create child processes using either *fork* statement or *forall* statement in a CPC program.

C.10.1 *fork* Statement

Syntax

In CPC, *fork* is useful to turn an individual statement into a child process. By preceding any statement with the *fork* operator, it becomes a child process running in parallel with its parent.

The general syntax of *fork* is as follow:

```
<statement>   ::= FORK <opt_mapping> <statement>
<opt_mapping>::= [ <opt_expr> ; <opt_expr> ]
              ::= [ <opt_expr> ]
              ::= [ @ <opt_expr> ; <opt_expr> ]
              ::= [ @ <opt_expr> ]
              ::= [ ^ <opt_expr> ; <opt_expr> ]
              ::= [ ^ <opt_expr> ]
              ::= NIL
```

where `statement` on the right hand side in the first rule can be any CPC valid statement, such as compound statement, expression or even no operation. The first `opt_expr` in each rule of `opt_mapping` should always be any CPC valid integer-valued expression, it represents a processor number. If there is an `@` in front of it, it represents a virtual processor number, if there is an $\wedge$ in front of it, it represents the physical

processor number. The second `opt_expr` in each rule of `opt_mapping`, if existing, should have valid left value. It is possible to omit either one of the `opt_exprs`.

Example

```
#include "cpc.h"

//The "<statement>"s in the following comments represent the right
//side <statement> in the first rule.
  channel int CI;
void mul(int i){
}

void main {
      int i,j;

      fork;               //The <statement> contains no operation
      fork  i = (i>10)? 1 : i+1;  //The <statement> is a single statement
      fork [ ; CI] i = CI;
      fork {              //The <statement> is a compound statement
            if (i > 10) i = 1;
            else i++;
      }
      fork (i+j);              //The <statement> is an expression
      fork printf("Hello world"); //The <statement> is a function call
      fork [i; CI] mul(i);
      fork fork mul(i);         //The <staetment> is another fork statement
//omitting the first <opt_expr> in the second rule
      fork[ ; CI] (i = CI, i+j);
//omitting the second <opt_expr> in the second rule
      fork[i; ] sqrt(i);
}
```

Figure 21: Fork Statement

Figure 21 is an example with various *fork* statements to illustrate how to use *fork* statements.

Semantics

Each `fork` statement creates a new child process, which will execute the `statement` after the "fork" operator. The parent will continue execution immediately without waiting for the child in any way. Although the parent process does continue with its execution while its *fork* children are still running, the parent is not permitted to terminate until all its children have finished. If the parent reaches the end of its

code while one or more of its children are still running, the parent will be suspended until all the children terminate. Only then will the parent be allowed to finish. This implementation prevents a premature termination by process 0 while some of its children are still running.

We will discuss the semantics for `opt_mapping`, that is how to map a process to a processor and how to bind channel variables to a process, in section C.13 and section C.11.2 respectively.

C.10.2 *join* statement

Sometimes, it may be desirable for a parent to wait at some point for the termination of one or all of its `fork` children. *join* statement in CPC is designed for this purpose. If the parent has only one *fork* child, then a *join* statement executed by the parent will force it to wait for the child to terminate. If the child has already terminated, then the execution of *join* will have no effect on the parent. One may think of the *join* as the opposite of a *fork*. *Fork* separates a child process from its parent, and *join* brings the terminated child back to its parent. In the example in Figure 22,

```
fork mul(j);
for (i=0; i<10; i++)
        a[i] = i;
        join;
        ...
```

Figure 22: JOIN statement

the parent will execute the `for` loop after forking the child. After finishing this loop, the parent will suspend execution at the `join` to wait for the termination of its child, then continue to execute the statements right after `join`. If the child has already terminated, the parent will continue its the execution after `join` without wait.

The execution of each `join` by the parent will match one single `fork` child termination, if the parent has multiple `fork` children, it may execute multiple `join` statements to wait for them all to terminate. In the example in Figure 23, the first

```
#include "cpc.h"

  void main{
        int i;

        for (i=0; i<10; i++)
                fork sqrt(i);
        for (i=0; i<10; i++)
                join;
}
```

Figure 23: Another example on JOIN Statement

for loop creates 10 fork child processes. Each child calls function sqrt. Then in the following for loop, the parent executes the join statement 10 times, thus waiting for the termination of all 10 children. Without this second loop, the parent would just continue execution in parallel with all of its children. However, once the parent reached its end, it would not terminate until all children had terminated.

C.10.3 *forall* Statement

Syntax

Following is the general syntax of the *forall* statement.

```
<statement>     ::= FORALL ( <non_comma_expr> FROM <expr> TO <expr>
                    <opt_grouping> ) <opt_mapping> <statement>
<opt_grouping> ::= GROUPING <expr>
               ::= NIL
<opt_mapping>   ::= [ <opt_expr> ; <opt_expr> ]
               ::= [ <opt_expr> ]
               ::= [ @ <opt_expr> ; <opt_expr> ]
               ::= [ @ <opt_expr> ]
               ::= [ ^ <opt_expr> ; <opt_expr> ]
               ::= [ ^ <opt_expr> ]
```

```
::= NIL
```

where non_comma_expr must be a single expression, expr must be any valid integer-value expression, none of them can be omitted. expr in the second rule must be a valid integer-valued expression, it can be omitted. The first opt_expr in each rule of opt_mapping should always be valid integer-valued expression, it represents the processor number. If there is an @ in front of it, it represents a virtual processor number, if there is an $\wedge$ in front of it, it represents a physical processor number. The second opt_expr in each rule of opt_mapping, if existing, should have valid left value. It's possible to omit either one of the opt_exprs. The statement on the right hand side of the first rule can be any valid statement in CPC, such as expression, compound statement, function call, etc.

As stated in section C.10.1, we will discuss the semantics for opt_mapping in section C.13 and section C.11.2.

Examples

Figure 24 is an example with various it forall statements to illustrate how to use *forall* statement.

Nested *forall* Loops

forall statements may be nested to offer greater parallelism as shown in the following example.

In the example in Figure 25, the outer *forall* incurs the creation of 10 processes, one for each value of i. Each of these child processes will consist of an instance of the inner *forall* loop, with the appropriate value of i. When each member of this first generation is executed, it will then spawn 5 more processes, one for each value of index j. Thus a total of 50 processes will be created in the second generation. The parent process will then have 10 children and 50 grandchildren.

Scope of *forall* Indices

Although children of a process do not have access to variables belonging to the the parent process, the *forall* index is an exception. The child processes can reference the

```
#include "cpc.h"

//The "statement"s in the following comment represent the second
// statement on the right side of the first rule.

  int f(int j) {
          return j*10;
}
channel int c[200];
void main() {
          int i,j;
          int a[10][20];

          forall ( i from 1 to 10 )  //The statement is a single statement
                    printf("Hello world!\n");
 //The statement is a function call
          forall (i from 1 to 10) [i; c[i]] f(i);
          forall ( i from 1 to 5) [i;] //The staetment is another fork statement
                    fork f(i);
          forall (i from i+1 to 2*i)  //The statement is a compound statement

                    { int j;
          j = i*i;
                              }
          forall ( i from 1 to 9)           // nested forall loops
                    forall ( j from 1 to 10 )  [; c[i*j]] sqrt(i*j);
          forall ( i from 1 to 10 grouping 5)  //with the <opt_grouping>
                    printf("Hello !\n");
 //Omit the second <opt_expr> in the third rule
          forall(i from 0 to 9 grouping 1)
                    [i;] f(i);
 //Omit the first <opt_expr> in the third rule
          forall (i from 1 to 10) [; c[i]] f(i);
 //A complete example of forall statement
          forall(i from 0 to 9 grouping 5) [i;c[i]] printf("Perfect!\n");;
}
```

Figure 24: FORALL Statement

forall index as if the loop were a normal *for* loop.

In the example in Figure 26, the *forall* index i is defined at the start of the main program. Outside of the *forall* statement, the variable i behaves like any ordinary integer variable. However, within the *forall* body, the index i behaves as if it were defined as a local variable within the *forall* . Since the loop iterations are to be executed in parallel, one single index variable i is insufficient–all the values of i from 1 to 10 must be available simultaneously. This is automatically handled in CPC implementation by providing each processor with its own local copy of the index i. That is, processor 1 has a local i with value 1, processor 2 has a local i with value 2,

```
#include "cpc.h"
  void main() {
      int i,j;
      int a[10][5];

      forall ( i from 1 to 10)              // nested forall loops
              forall ( j from 1 to 5 ) a[i-1][j-1] = i*j;
}
```

Figure 25: Nested FORALL Statement

```
   void main() {
     int i, k;

     forall (i from 1 to 10)
     {
       printf("Process %d\n", i);          // this access to i is allowed
       printf("Value of k = %d\n", k);  // access to k is not allowed
        // will generate run-time error: reference to a non-local
     }
}
```

Figure 26: Forall Index

and so on. The body of the *forall* loop is the code of every child process. Although each child process will execute the same code, the result will be different due to the differing values of the local index value i. The child processes are allowed to read variable i in their code (the first *printf* statement).

Although the child processes are allowed to access the *forall* index, there is a restriction: references to the *forall* index must be "read" only. In other words, each child process can see only a unique value of the index. Once a process is created and assigned its unique value of the *forall* index, this value cannot be changed within the process. Any attempt to alter the value of this index in an assignment statement will result in a compiler error. Similarly, inside the *forall* body, the *forall* index cannot be passed by reference to a function, or used as the target of an I/O read operation (e.g.

scanf). The *forall* index may, however, be used in any context that does not change its value. It can be used, for example, in a CPC expression, to index an array, or be passed by value to a function.

Semantics

forall statement is a parallel form of the normal *for* loop. Each iteration of a *forall* statement creates a child process which will run in parallel with other children created by the same *forall*. The program code for each process is the same, just a copy of the body of the *forall* loop.

A process terminates when it reaches the end of its code. Processes of the same parent may not terminate at the same time. This is due to slight variations in processor speeds, processor loads, or other environmental influences.

After finishing the creation of processes, the parent process suspends its execution, goes to sleep and waits until all of its children terminate. Only then will the parent continue its execution with the statement following the *forall* statement. This is one of the differences between *forall* statement and *fork* statement.

An important issue that arises with the *forall* statement is the duration or execution time of each process, sometimes called the *granularity* of the process. In any computer system, there is always an overhead associated with creating a parallel process, dispatching it to a particular processor where it will be executed, and terminating the process. For this overhead to be justified, the duration of the process must be much larger than the creation overhead. If the process grain is too fine, the overheads may outweigh the speedup gained by parallel processing.

Assume a process creation time is 8 time units in the system, and the duration of each process is 8, then the total elapsed time since the start of the *forall* statement will be 8*100+8=808 time units. If we replace the *forall* statement with a *for* statement, then no child process will be created, therefore the total execution time will be 100*8=800 time units. Not only has the *forall* failed to speed up the execution, but actually lengthened the execution time due to the process creation time.

To help overcome this granularity problem with *forall* statement, *grouping* option is provided in CPC. It can be used in *forall* statement to group together a certain range of index values in the same process. The grouping size should be chosen so as to balance the program speedup and process creation/termination overheads. If the

grouping index is omitted as in the above example, then the default group size is 1. Now let's look at the following example:

```
forall ( i from 1 to 100 grouping 10 )
        f(i);
```

the added notation *grouping* 10 causes the index values to be formed into groups of size 10 in each process. Thus, only 10 processes are created. The first process sequentially iterates through the index values 1 to 10, the second process iterates through 11 to 20, and so on.

We will discuss the semantics of opt_mapping, that is how to map a process to a processor and how to bind a channel variable to a process in section C.13 and section C.11.2 respectively.

Example: Matrix Multiplication

Now let's look at a complete CPC program for Matrix multiplication.

Matrix multiplication is used frequently in many of the numerical techniques common in scientific and engineering computing. A matrix in this context is simply a two-dimensional array of real numbers. Multiplying two matrices involves a complex pattern of multiplying and adding numbers from the two matrices.

In the example in Figure 27, function *seqMultiply* is the sequential matrix multiplication, and function *ParallelMatrixMultiply_1* is a simple parallel version of *seqMultiply*.

C.11 Process Communication via Channel Variables

In this section, we begin to consider the issue of process communication and interaction. We describe a special feature of the CPC programming language called *channel variable*, which is used for process communication and synchronization.

A channel variable, as its name implies, collects data values from writer processes and "channels" them into reader processes. The message forwarding and routing are done by the underlying network simulator, and completely transparent to the programmer.

```
#include "cpc.h"
#define N 8

  void printMatrix(float x[N][N]) {
       int i, j;

       for (i=0; i<N; i++) {
            for (j=0; j<N; j++)
                 printf("%.2f\t ", x[i][j]);
            printf("\n");
       }
       printf("\n");
}
void ParallelMatrixMultiply_1
  (float &a[N][N], float &b[N][N], float &c[N][N]) {
       int i, j;

       void VectorProduct(int i, int j ) {
            int   k;
            float sum;

            sum = 0.0;
            for (k=0; k<N; k++)
                 sum += a[i][k]*b[k][j];
            c[i][j] = sum;
       }
       forall (i from 0 to N1)
                 forall (j from 0 to N1)
                 VectorProduct(i,j);
}
void main() {
       float a[N][N], b[N][N], c[N][N];
       int i, j;

       for (i=0; i<N; i++)
            for (j=0; j<N; j++)
                 a[i][j] = rand()*5;
       for (i=0; i<N; i++)
            for (j=0; j<N; j++)
                 b[i][j] = rand()*5;
       ParallelMatrixMultiply_1(a, b, c);
       printMatrix(c);
}
```

Figure 27: Matrix Multiplication

Conceptually, a channel acts like a first-in-first-out queues of values (messages) of the same data type. As values are written to the channel, they are saved in a queue until they are read by some other process. The capacity of the queue buffer is assumed to be unlimited. Channel variables allow parallel processes to exchange and share data values in a controlled and abstract way.

C.11.1 Declarations of Channel Variables

Syntax

The general syntax for channel declaration is as follows:

```
<def>  ::= CHANNEL TYPE_QUAL <decl_list>;
```

::= CHANNEL TTYPE <decl_list>;

where TYPE_QUAL can be any valid type such as *int*, *float*, *char*, *enum*, array, and structure, TTYPE can be typedef type for declaring a single channel variable or array of channel. The type is the data type of messages written to or read from the channel. For instance, if the type is *int*, every message stored in the channel is of type integer.

Examples

```
typedef enum {Red, Green, Blue} Colors;
typedef char arrayChar[10];
typedef struct
  { arrayChar name;
float mark;
  } structStudent;

// The channel type is a scalar type in CPC
channel int ci;
channel float cf;
channel char cc;
channel enum { positive, negtive } ce;

// The channel type is a typedef type in CPC
channel Colors CE;
channel arrayChar CA;
channel structStudent CS;

// Using typedef to create a new name for channel type
typedef channel int chanInt; //channel of integer
typedef char arrChar[20];
typedef channel arrChar chanArrChar;
            //channel of array; the array is 20 characters long
chanInt myChanInt;        //variable of defined type chanInt
chanArrChar myChanArrChar; //variable of defined type chanArrChar
```

Figure 28: Channel Declaration

Channel types may appear at any level of a structured type.

In the example in figure 28, chanArrChar is *channel of array*; and arrayChan is *array of channel*. Multidimensional arrays of channels are also permitted.

The only operations that can be performed with a *channel of array* is reading or writing a whole array from the channel. It is not permitted to read or write one element in the array. So the expression chanArrChar[5] is not valid CPC syntax. To

```
channel int arrayChan[10]; //array of 10 channels
                           //each channel is a list of integer values
typedef struct
        { int processID;
        channel int mailbox; //structure field is of type channel
        } structProcess;
```

Figure 29: Complex Channel Declaration

get the fifth element in the array, one must first read the whole array from the front of channel `chanArrChar` into an ordinary variable such as a; and then use the expression $a[5]$.

Nested channels are not allowed: a channel may not contain any channels. A "channel of channel" is not permitted by the rule that the type must be a valid type in the C language because `channel` is not a valid type in C. For example, we cannot have a channel of structure where one field of the structure is another channel.

In the CPC language, channel variables are usually declared as global variables. We may have a channel variable c that is local to a function. However this channel would be useless because no processes other than the original owner of c can access c (due to the lexical scope rule of the CPC language). Thus the original owner cannot use c to communicate with the other processes. That is why channel variables should be declared as global variables so that all processes can see these channel variables.

C.11.2 Binding Channel Variables to New Processes

The channel declarations in Figure 30 will be used for examples in this section, assuming that the channels are global variables.

To bind channels to a new process means that one or more channels can be assigned to a new process by preceding the new child's code (the `statement`) with the names of the channels to be assigned.

```
channel int   CI, CJ, CK;        //channel of integer
channel float arrayCF[10];        //array of 10 channels
channel int   arrayCI[20];       //array of 20 channels
channel char  arr3Dchan[5][4][8]; //3D array of (5x4x8=)160 channels
typedef struct
  { int ID;
    float mark;
  } StudentStruct
channel StudentStruct myRecord;
```

Figure 30: Array of Channel

Syntax

Binding channel variable will happen only when new processes are created, so it is related only with `fork` and `forall` statements. It is used to assign a channel variable to a specific process, so that the process can receive messages from other processes through the channel. Any process may write values to any channel, but each process may read values only from its own assigned channels. When any process is created with one of the CPC *fork* or *forall* statement, the binding may be specified to assign one or more of the channel variables to the newly created process.

Recall the general sytax for *fork* or *forall* in section C.10.1 and section C.10.3 respectively, the syntax for binding channel variables is as follows:

```
<opt_mapping>  ::= [ <opt_expr> ; <opt_expr> ]
               ::= [ <opt_expr> ]
               ::= [ @ <opt_expr> ; <opt_expr> ]
               ::= [ @ <opt_expr> ]
               ::= [ ^ <opt_expr> ; <opt_expr> ]
               ::= [ ^ <opt_expr> ]
               ::= NIL
```

where the second `opt_expr`, if existing, in each rule of `opt_mapping` represents the channel variables that will be assigned to the newly created process. It should be one or more comma-separated channel variables.

Example

```
channel int c[200];
channel int CI;
main()
{int i;

//assign channel CI to new process
fork [1; CI] ChildCode();

//assign channel of array c[i] to the ith process
forall(i from 0 to 9 grouping 5) [i;c[i]] printf("Perfect!\n");
}
```

Figure 31: Channel Binding

A channel reference can be an array of channels to facilitate the binding of many channels of the same component type to a new process. For example, instead of binding 20 channels (of the same component type) to a process, we could declare an array of 20 channels, and then bind that array to the process. Any channel in the array may then be used by that process to receive messages. The array of channels can be an entire array, or one or more dimensions of a multi-dimensional array (e.g. one row of a 2D array, one plane of a 3D array). The examples are in Figure 32:

```
void f{};
void main()
{ int i;

fork [ ; arrayCI] f();        //assign 20 channels
fork [ ; arr3Dchan[1]] f();   //assign 32 channels (1 plane)
forall(i from 1 to 4) [i;arr3Dchan[1][0]] f();//assign 8 channels (1 row)
}
```

Figure 32: More examples on binding

C.11.3 Read and Write on Channel Variables

Channels are written by using their name on the left side of an assignment statement, and read by using their name on the right side of an assignment statement. A channel may be written by many writers but read only by one reader, namely the owner of the channel.

Any process may write values to any channel, provided that the channel variable is accessible by the process according to C lexical scope rules. However, each process may read values only from its own assigned channels.

If the channel is empty, the reader is suspended until some other process writes a value into the channel. However, a writer process will never be suspended; channels are supposed to have unlimited capacity and can hold any number of values. Channel writes are thus non-blocking.

Channel Write

```
writer()
{
        int i;
        StudentStruct tempRecord;

    //Write to channels whose component type is a basic type
        CI = 0;
        CI = i + 1;
        arrayCF[1] = 3.1416;
    //Write to a channel whose component type is a composite type
        tempRecord.ID = 12345;
        myRecord = tempRecord; //channel write
}
```

Figure 33: Channel Write

Examples of channel writes are shown in Figure 33. Note that it is not permitted to use a subscript with a *channel of array*. The only operations that can be performed with a *channel of array* is reading or writing a whole array from the channel. It is not allowed to read or write one element in the array. The rules are similar for a *channel of structure*: one may not read or write a single field of the structure.

Channel Read

```
reader()
{ int i;
float f;
StudentStruct bufferRecord;
  //Assume that this reader owns the channels used below
  // Read from channels whose component type is a basic type
i = CJ;
f = 2 * arrayCF[1] / 5;
  // Read from a channel whose component type is a composite type
bufferRecord = myRecord; //channel read
bufferRecord.mark++;
}
```

Figure 34: Channel Read

Any channel variable name can be part of an expression on the right side of an assignment statement, as shown in figure 34:

As stated earlier, for a *channel of structure*, it is not permitted to read a single field of the structure. In the above example, to read field `mark`, one must first read the whole structure from the front of channel `myRecord` into an ordinary structure variable such as `bufferRecord`, and then use the expression `bufferRecord.mark`. The same rules apply to a *channel of array*. Channel variables may be used in any

```
anotherReader()
{ int i, j;
        //Assume that this reader owns the channels used below
if (CK) i++;
else   i = 0;
if (arrayCF[5] > arrayCF[6] * 2) i = j;
}
```

Figure 35: Channel Variables

expression in the program, provided that the component type matches the context in the expression. The general rule is that wherever an ordinary variable of a given type may be used in an expression, a channel variable with that same component

type may be used. For example, channel variables may be used as array indices or in `boolean` expressions, as examples in Figure 35:

An exception to the above rule is that channel variables are not allowed in I/O statements such as *printf* or *scanf*.

Note that each time a channel is read, it produces a different value. This is because values are queued inside the channel during writing, and removed during reading. Let `CI` be a channel of integer. The assignment (`n = CI + CI`) is not equivalent to (`n = CI * 2`).

Channel Empty Test

It occurs in many parallel programs that one process is computing some values that are to be used by other parallel processes. To handle this situation, a *channel variable* in CPC has the property that it can be "empty". If the channel is empty, the reader is suspended until some other process writes a value into the channel. So the channel has the ability to delay a reader process until the necessary values are supplied by a writer process. Each channel has unlimited capacity for storing any number of values. Therefore, writer processes are never delayed.

```
if (ch?)
        n = ch; //read the channel
  else
          printf("Channel currently empty");
```

Figure 36: Channel Empty Test

To avoid this suspension when the channel is empty, the reader can test to determine whether the channel currently contains any values. This is achieved by using a boolean-valued expression containing the name of the channel variable followed by a question mark, as in the example in Figure 36:

The expression "ch?" will evaluate to 1 (true) if channel *ch* currently contains any values and 0 (false) if the channel is empty.

Example: InsertionSort

```
#include "cpc.h"
#define N 100
  arch fullconnect F[101];
int list[N], sorted[N];
channel int pipechan[N];
int j,k;

void Pipeprocess(int me, int &sorted[]){
        int internal, newitem, i;

        internal = pipechan[me];        // read first item from the left
        for (i=0; i<N-me-1; I++){
                newitem = pipechan[me];        // read new item from the left
                if (newitem < internal)
                {
                        pipechan[me+1] = internal; // send internal to the right
                        internal = newitem;
                }
                else pipechan[me+1] = newitem; // send newitem to the right
        }
        sorted[me] = internal;          // return my final sorted list item
        printf("sorteditem= %d \n", sorted[me]); }
void main {
        for (j=0; j<N; j++){
                list[j] = N-j;
                sorted[j] = 0;
        }
        for (j=0; j<N; j++)
                fork[ ;pipechan[j]] Pipeprocess(j, sorted);
        for (k=0; k<N; k++)
                pipechan[0] = list[k];
        join;
        for (j=0; j<N; j++)
                printf("%d %d %d\n", j, sorted[j], list[j]);
}
```

Figure 37: InsertionSort

Now let's see a complete example program `InsertionSort` in Figure 37, in which precesses use channel variable to communicate with each other.

C.12 Parallel Architecture Definition

The CPC language allows users to specify the virtual architecture in the CPC program. The virtual architecture will then be mapped to a physical architecture at

run-time. The physical architecture can be the same as or different from the virtual architecture.

As the CPC language supports both shared-memory and message-passing programming paradigms, an architecture declaration is needed to identify a message-passing program. If the architecture declaration is absent in a program, the program is treated as a shared-memory program. The virtual architecture is specified with the keyword `arch` at the beginning of the program as in the following examples. The architecture of a multicomputer system is defined by the topology and the size of the system.

C.12.1 Syntax

The general syntax for the architecture declaration is as follows:

```
<def>        ::= ARCH ARCH_TYPE <var_decl>
             ::= phyARCH ARCH_TYPE <var_decl>;
<var_decl> ::= <new_name>
             ::= <var_decl> [ <const_expr> ]
<new_name> ::= NAME
```

where `ARCH-TYPE` is one of the following topologies: *shared*, *line*, *ring*, *mesh*, *torus*, *hypercube*, and *fullconnect*. *Shared* topology means that the program is intended for execution on a shared-memory multiprocessor. In a *fullconnect* topology, each processor is connected to every other processor. `Name` is the name of the architecture, `const-expr` must be any valid expression representing an integer constant. it shared,*line* and *ring* should be declared as one dimensional array, `const-expr` represents the size of the architecture. *mesh* and *torus* should be declared as multidimensional array. Multiplying the values of all the `const-exprs` in the declaration will get the system size of this architecture. *Hypercube* should also be declared as one dimensional array, the `const-expr` in this declaration represents the system dimension numbers.

In the case when CPC users want to bypass the virtual architecture for their programming convenience, they have the freedom to specify a physical architecture in the CPC program. In that way virtual architecture will be bypassed, processes will be mapped to physical processor directly. The second rule of `def` is designed for this

purpose.

C.12.2 Examples

```
arch shared S[100];  //shared-memory with 100 processors
arch fullconnect F[i]; //if i == 25, fullconnect with 25 processors
arch line L[10];     //line with 10 processors
arch ring R[20];     //ring with 20 processors
arch hypercube H[5]; //hypercube with 2^5 = 32 processors
```

Figure 38: Architecture Declaration

Examples of virtual architecture declaration are in figure 38. We can define a physical architecture as following:

```
phyArch line l[10];
```

A physical architecture is declared only to make the CPSS simulation of a physical architecture more convenient.

If the topology is *shared*, *line*, *ring* or *fullconnect*, the size of the architecture is the total number of processors. If the topology is *hypercube* and the number of dimensions of the hypercube is 5, then the size of the architecture is 2*2*2*2*2=32.

For a mesh (or torus), the size of the architecture is to multiply the size of each dimension of the mesh (or torus). The minimum number of dimensions of a mesh or torus is 2. Examples are shown in Figure 39

```
arch mesh twoDmesh[5][7];          // 5x7 mesh arch mesh
mymesh[3][5][10][8][12];       // 3x5x10x8x12 mesh arch torus
threeDtorus[10][10][10];     // square 10x10x10 torus
```

Figure 39: Architecture Dimension

C.13 Mapping Processes to Virtual Processors

C.13.1 Syntax

The general syntax for *mapping* is as follows:

```
<opt_mapping>  ::= [ <opt_expr> ; <opt_expr> ]
               ::= [ <opt_expr> ]
               ::= [ @ <opt_expr> ; <opt_expr> ]
               ::= [ @ <opt_expr> ]
               ::= [ ^ <opt_expr> ; <opt_expr> ]
               ::= [ ^ <opt_expr> ]
               ::= NIL
```

Users are allowed to map parallel processes to the processors of the virtual architecture to optimize program performance by reducing communication latency. Communicating processes should be mapped to processors sitting close to each other.

Referring back to the general syntax of *forall* and *fork* statements in section C.10.1 and section C.10.3 respectively, we see that each forall or fork ends by a `statement` which will be compiled into the parallel code to be executed by the new child. The first `opt_expr` in each rule of `opt_mapping` represents the mapping of new process to processors. If there is an `@` in front of the `opt_expr`, then the newly created child will be mapped to a particular virtual processor, the value of `opt_expr` is the virtual processor ID; if there is a $\wedge$ in front of the `opt_expr`, then the newly created child will be mapped to a particular physical processor, bypassing the virtual architecture, and the value of `opt_expr` represents the physical processor ID. If both `@` and $\wedge$ is missing, `opt_expr` represents the virtual processor ID. So the `opt_expr` must be specified using any valid CPC integer-valued expressions. The second `opt_expr` in each rule of `opt_mapping`, if existing, represents the binding of channel variables to the newly created process, which we already discussed in section 3.3.2.

C.13.2 Examples

Two examples of the mapping in figure 40 are used by *fork* and *forall* respectively.

As indicated in section C.10.1 and section C.10.3, the mapping is optional. If the user does not specify the virtual processor ID for a new child, at run time, the

```
for (i = 0; i < n; i++)
        fork [i; ] f(i);
        forall (k from 1 to 10)
                        [k-1; ] f(k);
```

Figure 40: Mapping

CPSS will map the child to a default virtual processor. The mapping objective is to minimize communication cost and balance the load among physical processors.

C.14 A complex CPC example: Matrix Multiplication

Figure 41 is a complex example on matrix multiplication, it will cover most of the features of the CPC language.

```
#include "cpc.h"
#define N 8
   arch torus T[N][N];
phyArch mesh L[N][N];
channel float Achan[N][N], Bchan[N][N];
void seqMultify(float &a[N][N], float &b[N][N], float &c[N][N]) {
       int i, j, k;
       float sum;

       for (i=0; i<N; i++)
             for (j=0; j<N; j++) {
                    sum = 0.0;
                    for (k=0; k<N; k++)
                          sum += a[i][k]*b[k][j];
                    c[i][j] = sum;
             }
}
void printMatrix(float x[N][N]) {
       int i, j;

       for (i=0; i<N; i++) {
             for (j=0; j<N; j++)
                    printf("%.2f\t ", x[i][j]);
             printf("\n");
       }
}
void multiply(int row, int col, float myA, float myB, float &mainC) {
       int iter, above, left;
       float myC;

       if (row>0) above = row 1;  // up neighbor
       else       above = N 1;
       if (col>0) left  = col 1;  // Left neighbor
       else       left  = N 1;
       myC = 0;
       for (iter=0; iter<N; iter++) {
             Achan[row][left] = myA;     // send myA in leftward rotation
             Bchan[above][col] = myB;    // send myB in upward rotation
             myC += myA * myB;
             myA = Achan[row][col];      // receive new myA
             myB = Bchan[row][col];      // receive new myB
       }
       mainC = myC; // Send final value to main process
}
void pMultify(float &a[N][N], float &b[N][N], float &c[N][N]) {
       int i, j;
       forall (i from 0 to N1)
                    forall (j from 0 to N1)
                    fork [i*N+j; Achan[i][j], Bchan[i][j]]
                    multiply(i, j, a[i][(j+i) % N], b[(i+j) % N][j], c[i][j]);
}
void main() {
       float a[N][N], b[N][N], c[N][N];
       int i, j;

       timeOff();
       for (i=0; i<N; i++)
             for (j=0; j<N; j++)
                    a[i][j] = rand()*5;
       for (i=0; i<N; i++)
             for (j=0; j<N; j++)
                    b[i][j] = rand()*5;
       timeOn();
       pMultify(a, b, c);
       timeOff();
       printMatrix(c);

}
```

Figure 41: Matrix Multiplication

Appendix D

A Complete Example

The source program is named example.c, the CPCC fronteend will generate the file example.lst and the CPC backend will generate example.cod.

D.1 CPC source program

```
#include "cpc.h"

arch ring m[11];

channel int c1;

typedef int int10[10];
channel int10 c2;

channel int c3[10];

void f(int x, int& y)
{
  if (x==9) c3[0] = x;
  else      c3[x+1] = x;
  y = c3[x];
}
```

```
void f2(int a[10])
{
  c2 = a;
  printf("Last fork process finished!");
}

void main()
{
  int i, j, a1[10];

  for (i=0; i<10; i++) a1[i] = i*i;
  for (i=0; i<10; i++) fork [i+1;c3[i]] f(i, a1[i]);
  for (i=0; i<10; i++) join();
  for (i=0; i<10; i++) printf("a1[%d] = %d\n", i, a1[i]);

  forall(i from 0 to 9 grouping 1)
    [i;c3[i]] f(i, a1[i]);

  for (i=0; i<10; i++) printf("a1[%d] = %d\n", i, a1[i]);

 fork [10; c2] f2(a1);
}
```

D.2 CPCC frontend generated file

This fileis named as exampl.lst, its content is shown as following.

```
(*----------------Global symbol list-----------------------*)

 Function[0]  printf     : ()int FIX      (size=2) (line 4 in "cpc.h")
 Parameter[1] format     : ^char AUT      (size=1, offset=0) [val]
 Parameter[1] ...        : int AUT      (size=0, offset=1) [val]
```

```
Function[0]  fprintf     : ()int FIX      (size=3) (line 5 in "cpc.h")
Parameter[1] fp          : int AUT       (size=1, offset=0) [val]
Parameter[1] format      : ^char AUT      (size=1, offset=1) [val]
Parameter[1] ...         : int AUT       (size=0, offset=2) [val]

Function[0]  scanf       : ()void FIX      (size=2) (line 6 in "cpc.h")
Parameter[1] format      : ^char AUT      (size=1, offset=0) [val]
Parameter[1] ...         : int AUT       (size=0, offset=1) [val]

Function[0]  fscanf      : ()void FIX      (size=3) (line 7 in "cpc.h")
Parameter[1] fp          : int AUT       (size=1, offset=0) [val]
Parameter[1] format      : ^char AUT      (size=1, offset=1) [val]
Parameter[1] ...         : int AUT       (size=0, offset=2) [val]

Function[0]  fopen       : ()int FIX      (size=2) (line 8 in "cpc.h")
Parameter[1] filename    : ^char AUT      (size=1, offset=0) [val]
Parameter[1] mode        : ^char AUT      (size=1, offset=1) [val]

Function[0]  fclose      : ()void FIX      (size=1) (line 9 in "cpc.h")
Parameter[1] fd          : int AUT       (size=1, offset=0) [val]

Function[0]  getchar     : ()int FIX      (size=0) (line 10 in "cpc.h")
Parameter[1]             : void AUT       (size=0, offset=0) [val]

Function[0]  fgetc       : ()int FIX      (size=1) (line 11 in "cpc.h")
Parameter[1] fp          : int AUT       (size=1, offset=0) [val]

Function[0]  absVirProsor : ()int FIX      (size=0) (line 12 in "cpc.h")
Parameter[1]             : void AUT       (size=0, offset=0) [val]

Function[0]  cartVirProsor : ()void FIX      (size=0) (line 13 in "cpc.h")
Parameter[1] array       : []int AUT       (size=0, offset=0) [ref]
```

```
Function[0]  absPhyProsor : ()int FIX      (size=0) (line 14 in "cpc.h")
Parameter[1]              : void AUT     (size=0, offset=0) [val]

Function[0]  cartPhyProsor : ()void FIX      (size=0) (line 15 in "cpc.h")
Parameter[1] array        : []int AUT     (size=0, offset=0) [ref]

Function[0]  phyTopo      : ()int FIX      (size=0) (line 16 in "cpc.h")

Function[0]  virTopo      : ()int FIX      (size=0) (line 17 in "cpc.h")

Function[0]  nbrPhyProsor : ()int FIX      (size=0) (line 18 in "cpc.h")

Function[0]  nbrVirProsor : ()int FIX      (size=0) (line 19 in "cpc.h")

Function[0]  phyDimNbr    : ()int FIX      (size=0) (line 20 in "cpc.h")

Function[0]  virDimNbr    : ()int FIX      (size=0) (line 21 in "cpc.h")

Function[0]  phyDimSizes  : ()void FIX      (size=0) (line 22 in "cpc.h")
Parameter[1] array        : []int AUT     (size=0, offset=0) [ref]

Function[0]  virDimSizes  : ()void FIX      (size=0) (line 23 in "cpc.h")
Parameter[1] array        : []int AUT     (size=0, offset=0) [ref]

Function[0]  localProcessID : ()int FIX      (size=0) (line 24 in "cpc.h")

Function[0]  processID    : ()int FIX      (size=0) (line 26 in "cpc.h")

Function[0]  min          : ()int FIX      (size=2) (line 27 in "cpc.h")
Parameter[1]              : int AUT      (size=1, offset=0) [val]
Parameter[1] ...          : int AUT      (size=0, offset=1) [val]

Function[0]  fmin         : ()float FIX      (size=2) (line 28 in "cpc.h")
```

```
Parameter[1]            : float AUT      (size=1, offset=0) [val]
Parameter[1] ...        : int AUT        (size=0, offset=1) [val]

Function[0]  max        : ()int FIX      (size=2) (line 29 in "cpc.h")
Parameter[1]            : int AUT        (size=1, offset=0) [val]
Parameter[1] ...        : int AUT        (size=0, offset=1) [val]

Function[0]  fmax       : ()float FIX    (size=2) (line 30 in "cpc.h")
Parameter[1]            : float AUT      (size=1, offset=0) [val]
Parameter[1] ...        : int AUT        (size=0, offset=1) [val]

Function[0]  abs        : ()int FIX      (size=1) (line 31 in "cpc.h")
Parameter[1]            : int AUT        (size=1, offset=0) [val]

Function[0]  fabs       : ()float FIX    (size=1) (line 32 in "cpc.h")
Parameter[1]            : float AUT      (size=1, offset=0) [val]

Function[0]  sin        : ()float FIX    (size=1) (line 33 in "cpc.h")
Parameter[1]            : float AUT      (size=1, offset=0) [val]

Function[0]  cos        : ()float FIX    (size=1) (line 34 in "cpc.h")
Parameter[1]            : float AUT      (size=1, offset=0) [val]

Function[0]  tan        : ()float FIX    (size=1) (line 35 in "cpc.h")
Parameter[1]            : float AUT      (size=1, offset=0) [val]

Function[0]  sqrt       : ()float FIX    (size=1) (line 36 in "cpc.h")
Parameter[1]            : float AUT      (size=1, offset=0) [val]

Function[0]  odd        : ()int FIX      (size=1) (line 37 in "cpc.h")
Parameter[1]            : int AUT        (size=1, offset=0) [val]

Function[0]  even       : ()int FIX      (size=1) (line 38 in "cpc.h")
```

```
Parameter[1]                : int AUT       (size=1, offset=0) [val]

Function[0]   floor         : ()int FIX       (size=1) (line 39 in "cpc.h")
Parameter[1]                : float AUT       (size=1, offset=0) [val]

Function[0]   ceil          : ()int FIX       (size=1) (line 40 in "cpc.h")
Parameter[1]                : float AUT       (size=1, offset=0) [val]

Function[0]   exp           : ()float FIX       (size=1) (line 41 in "cpc.h")
Parameter[1]                : float AUT       (size=1, offset=0) [val]

Function[0]   pow           : ()float FIX       (size=2) (line 42 in "cpc.h")
Parameter[1]                : float AUT       (size=1, offset=0) [val]
Parameter[1]                : float AUT       (size=1, offset=1) [val]

Function[0]   log           : ()float FIX       (size=1) (line 43 in "cpc.h")
Parameter[1]                : float AUT       (size=1, offset=0) [val]

Function[0]   log10         : ()float FIX       (size=1) (line 44 in "cpc.h")
Parameter[1]                : float AUT       (size=1, offset=0) [val]

Function[0]   atof          : ()float FIX       (size=1) (line 45 in "cpc.h")
Parameter[1]                : ^char AUT CONST (size=1, offset=0) [val]

Function[0]   malloc        : ()^int FIX       (size=1) (line 46 in "cpc.h")
Parameter[1] num_of_words : int AUT       (size=1, offset=0) [val]

Function[0]   free          : ()void FIX       (size=1) (line 47 in "cpc.h")
Parameter[1] start_add     : int AUT       (size=1, offset=0) [val]

Function[0]   delay         : ()void FIX       (size=1) (line 48 in "cpc.h")
Parameter[1]                : int AUT       (size=1, offset=0) [val]
```

```
Function[0]  join        : ()void FIX      (size=0) (line 49 in "cpc.h")
Parameter[1]             : void AUT      (size=0, offset=0) [val]

Function[0]  lock        : ()void FIX      (size=1) (line 50 in "cpc.h")
Parameter[1]             : ^int AUT      (size=1, offset=0) [val]

Function[0]  unlock      : ()void FIX      (size=1) (line 51 in "cpc.h")
Parameter[1]             : ^int AUT      (size=1, offset=0) [val]

Function[0]  rand        : ()float FIX      (size=0) (line 53 in "cpc.h")
Parameter[1]             : void AUT      (size=0, offset=0) [val]

Function[0]  srand       : ()void FIX      (size=1) (line 54 in "cpc.h")
Parameter[1] seed        : int AUT      (size=1, offset=0) [val]

Function[0]  strcmp      : ()void FIX      (size=2) (line 56 in "cpc.h")
Parameter[1] str1        : ^char AUT      (size=1, offset=0) [val]
Parameter[1] str2        : ^char AUT      (size=1, offset=1) [val]

Function[0]  strcat      : ()void FIX      (size=2) (line 57 in "cpc.h")
Parameter[1] to_str      : ^char AUT      (size=1, offset=0) [val]
Parameter[1] from_str    : ^char AUT      (size=1, offset=1) [val]

Function[0]  strcpy      : ()void FIX      (size=2) (line 58 in "cpc.h")
Parameter[1] to_str      : ^char AUT      (size=1, offset=0) [val]
Parameter[1] from_str    : ^char AUT      (size=1, offset=1) [val]

Function[0]  strlen      : ()int FIX      (size=1) (line 59 in "cpc.h")
Parameter[1] str         : ^char AUT      (size=1, offset=0) [val]

Function[0]  barrier     : ()void FIX      (size=1) (line 60 in "cpc.h")
Parameter[1] nbrProcesses : int AUT      (size=1, offset=0) [val]
```

```
Function[0]   globalMaxInt : ()void FIX       (size=2) (line 61 in "cpc.h")
Parameter[1] src          : int AUT       (size=1, offset=0) [val]
Parameter[1] dest         : ^int AUT       (size=1, offset=1) [val]

Function[0]   globalMaxFloat : ()void FIX
                                        (size=2) (line 62 in "cpc.h")
Parameter[1] src          : float AUT       (size=1, offset=0) [val]
Parameter[1] dest         : ^float AUT       (size=1, offset=1) [val]

Function[0]   globalMinInt : ()void FIX       (size=2) (line 63 in "cpc.h")
Parameter[1] src          : int AUT       (size=1, offset=0) [val]
Parameter[1] dest         : ^int AUT       (size=1, offset=1) [val]

Function[0]   globalMinFloat : ()void FIX
                                            (size=2) (line 64 in "cpc.h")
Parameter[1] src          : float AUT       (size=1, offset=0) [val]
Parameter[1] dest         : ^float AUT       (size=1, offset=1) [val]

Function[0]   globalAnd   : ()void FIX       (size=2) (line 65 in "cpc.h")
Parameter[1] src          : int AUT       (size=1, offset=0) [val]
Parameter[1] dest         : ^int AUT       (size=1, offset=1) [val]

Function[0]   globalOr    : ()void FIX       (size=2) (line 66 in "cpc.h")
Parameter[1] src          : int AUT       (size=1, offset=0) [val]
Parameter[1] dest         : ^int AUT       (size=1, offset=1) [val]

Function[0]   globalPrefixInt : ()void FIX
                                        (size=3) (line 67 in "cpc.h")
Parameter[1] src          : int AUT       (size=1, offset=0) [val]
Parameter[1] dest         : ^int AUT       (size=1, offset=1) [val]
Parameter[1] sequenceNum : int AUT       (size=1, offset=2) [val]

Function[0]   globalPrefixFloat : ()void FIX
```

```
                                             (size=3) (line 68 in "cpc.h")
Parameter[1] src         : float AUT      (size=1, offset=0) [val]
Parameter[1] dest        : ^float AUT      (size=1, offset=1) [val]
Parameter[1] sequenceNum : int AUT       (size=1, offset=2) [val]

Function[0]  send        : ()void FIX      (size=4) (line 69 in "cpc.h")
Parameter[1] msg         : ^char AUT      (size=1, offset=0) [val]
Parameter[1] size        : int AUT      (size=1, offset=1) [val]
Parameter[1] tag         : int AUT      (size=1, offset=2) [val]
Parameter[1] dest        : int AUT      (size=1, offset=3) [val]

Function[0]  receive     : ()void FIX      (size=3) (line 70 in "cpc.h")
Parameter[1] msg         : ^char AUT      (size=1, offset=0) [val]
Parameter[1] size        : int AUT      (size=1, offset=1) [val]
Parameter[1] tag         : int AUT      (size=1, offset=2) [val]

Function[0]  probe       : ()int FIX      (size=2) (line 71 in "cpc.h")
Parameter[1] tag         : int AUT      (size=1, offset=0) [val]
Parameter[1] size        : int AUT      (size=1, offset=1) [val]

Function[0]  bcast       : ()void FIX      (size=4) (line 72 in "cpc.h")
Parameter[1] msg         : ^char AUT      (size=1, offset=0) [val]
Parameter[1] size        : int AUT      (size=1, offset=1) [val]
Parameter[1] tag         : int AUT      (size=1, offset=2) [val]
Parameter[1] root        : int AUT      (size=1, offset=3) [val]

Function[0]  virSend     : ()void FIX      (size=4) (line 73 in "cpc.h")
Parameter[1] msg         : ^char AUT      (size=1, offset=0) [val]
Parameter[1] size        : int AUT      (size=1, offset=1) [val]
Parameter[1] tag         : int AUT      (size=1, offset=2) [val]
Parameter[1] dest        : int AUT      (size=1, offset=3) [val]

Function[0]  virReceive  : ()void FIX      (size=3) (line 74 in "cpc.h")
```

```
Parameter[1] msg         : ^char AUT      (size=1, offset=0) [val]
Parameter[1] size        : int AUT       (size=1, offset=1) [val]
Parameter[1] tag         : int AUT       (size=1, offset=2) [val]

Function[0]  virProbe    : ()int FIX      (size=2) (line 75 in "cpc.h")
Parameter[1] tag         : int AUT       (size=1, offset=0) [val]
Parameter[1] size        : int AUT       (size=1, offset=1) [val]

Function[0]  virBcast    : ()void FIX      (size=4) (line 76 in "cpc.h")
Parameter[1] msg         : ^char AUT      (size=1, offset=0) [val]
Parameter[1] size        : int AUT       (size=1, offset=1) [val]
Parameter[1] tag         : int AUT       (size=1, offset=2) [val]
Parameter[1] root        : int AUT       (size=1, offset=3) [val]

Typedef [0]  CHANNEL     : <>[1]int CHANNEL      (size=1)

Function[0]  ChannAlloc  : ()^[CHANNEL]typedef FIX
                                    (size=2) (line 81 in "cpc.h")
Parameter[1] num_chann   : int AUT       (size=1, offset=0) [val]
Parameter[1] chann_size  : int AUT       (size=1, offset=1) [val]

Typedef [0]  ARCHITECTURE : ^[$tag00]struct FIX      (size=1)

Function[0]  VtopoCreate : ()[ARCHITECTURE]typedef FIX
                                         (size=3) (line 88 in "cpc.h")
Parameter[1] topo        : int AUT       (size=1, offset=0) [val]
Parameter[1] num_dim     : int AUT       (size=1, offset=1) [val]
Parameter[1] ...         : int AUT       (size=0, offset=2) [val]

Typedef [0]  spinlock    : int FIX       (size=1)

Function[0]  virtual_processor : ()void FIX
                                           (size=0) (line 99 in "cpc.h")
```

```
Parameter[1] array       : []int AUT       (size=0, offset=0) [ref]

Function[0]  physical_processor : ()void FIX
                                        (size=0) (line 100 in "cpc.h")
Parameter[1] array       : []int AUT       (size=0, offset=0) [ref]

Function[0]  clock       : ()int FIX       (size=0) (line 101 in "cpc.h")
Parameter[1]             : void AUT       (size=0, offset=0) [val]

Function[0]  seqtime     : ()int FIX       (size=0) (line 102 in "cpc.h")
Parameter[1]             : void AUT       (size=0, offset=0) [val]

Function[0]  seqon       : ()void FIX       (size=0) (line 103 in "cpc.h")
Parameter[1]             : void AUT       (size=0, offset=0) [val]

Function[0]  seqoff      : ()void FIX       (size=0) (line 104 in "cpc.h")
Parameter[1]             : void AUT       (size=0, offset=0) [val]

Function[0]  timeOn      : ()void FIX       (size=0) (line 105 in "cpc.h")
Parameter[1]             : void AUT       (size=0, offset=0) [val]

Function[0]  timeOff     : ()void FIX       (size=0) (line 106 in "cpc.h")
Parameter[1]             : void AUT       (size=0, offset=0) [val]

Architec[0]  m           : [11]ring
Variable[0]  c1          : <>int CHANNEL        (line 5 in "learn3.c")
Typedef  [0] int10       : [10]int FIX      (size=10)
Variable[0]  c2          : <>[int10]typedef CHANNEL
                                              (line 8 in "learn3.c")
Variable[0]  c3          : [10]<>int CHANNEL
                                             (line 10 in "learn3.c")

Function[0]  f           : ()void FIX    (size=2) (line 12 in "learn3.c")
```

```
Parameter[1] x          : int AUT       (size=1, offset=0) [val]
Parameter[1] y          : int AUT       (size=1, offset=1) [ref]
{   (*  Start of compound statment  *)

    (*----Statement list--*)
    if (x == 9)
        c3[0] = x;
    else
        c3[(x + 1)] = x;
    y = c3[x];
}   (*  End of compound statment    *)

Function[0]  f2         : ()void FIX    (size=10) (line 19 in "learn3.c")
Parameter[1] a          : [10]int AUT    (size=10, offset=0)     [val]
{   (*  Start of compound statment  *)

    (*----Statement list--*)
    c2 = a;
    printf("Last fork process finished!");
}   (*  End of compound statment    *)

Function[0]  main       : ()void FIX
                                      (size=0) (line 25 in "learn3.c")
{   (*  Start of compound statment  *)
    (*----Symbol list-----*)
    Variable[1]  i          : int AUT
                            (size=1, offset=0) (line 27 in "learn3.c")
    Variable[1]  j          : int AUT
                            (size=1, offset=1) (line 27 in "learn3.c")
    Variable[1]  a1         : [10]int AUT
                           (size=10, offset=2) (line 27 in "learn3.c")

    (*----Statement list--*)
```

```
    for (i = 0; i < 10; i++)
        a1[i] = (i * i);
    for (i = 0; i < 10; i++)
        fork [(i + 1); port c3[i]]
            f((i, a1[i]));
    for (i = 0; i < 10; i++)
        join();
    for (i = 0; i < 10; i++)
        printf(("a1[%d] = %d\n", i, a1[i]));
    forall i from 0 to 9
        [i; port c3[i]; group 1]
        f((i, a1[i]));
    for (i = 0; i < 10; i++)
        printf(("a1[%d] = %d\n", i, a1[i]));
    fork [10; port c2]
        f2(a1);
}   (* End of compound statment    *)

(*---------------Global structure list-------------------*)
 struct[0] $tag00 (size=22)
    v_topo              :  int (size=1, offset=0)
    num_dim             :  int (size=1, offset=1)
    dim_size            :  [20]int (size=20, offset=2)
```

D.3 CPC backend generated file

This file is named as example.cod, its content is asfollowing.

```
                    DEFINED LEVEL 0 FUNCTIONS
 Name               Signature                          ID Table Index
=======================================================================
 f                  f(int,int&)void                         8
 f2                 f2([10]int)void                         11
 main               main()void                              13
```

<<>>

DEFINED LEVEL 0 VARIABLES

Name	Type	ID Table Index
c1	<>int	5
c2	<>[int10]typedef	6
c3	[10]<>int	7

<<>>

EXTERNAL FUNCTIONS

Name	Signature	ID Table Index

<<>>

EXTERNAL VARIABLES

Name	Type	ID Table Index

<<>>

0 0 0

VCODE Table

198

Line	SourceLine	F	X	Y	Mnemonic
0	-1	19	0	13	NewFrame
1	-1	20	0	8	Call
2	-1	21	0	0	Halt
3	14	4	1	9	LoadValue
4	14	6	0	9	LoadIntegerLiteral
5	14	34	0	0	Equal
6	14	16	0	13	Jumpz
7	14	4	1	9	LoadValue
8	14	3	0	20	LoadAddress
9	14	6	0	0	LoadIntegerLiteral
10	14	24	0	4	ArrayIndexing
11	14	53	2	1	StoreChannel

12	14	15	0	20	Jump
13	15	4	1	9	LoadValue
14	15	3	0	20	LoadAddress
15	15	4	1	9	LoadValue
16	15	6	0	1	LoadIntegerLiteral
17	15	46	0	0	Add
18	15	24	0	4	ArrayIndexing
19	15	53	2	1	StoreChannel
20	16	3	0	20	LoadAddress
21	16	4	1	9	LoadValue
22	16	24	0	4	ArrayIndexing
23	16	52	0	0	LoadCHwAdrOnStack
24	16	3	1	10	LoadAddress
25	16	8	0	0	Dereference
26	16	9	2	0	Store
27	-1	22	0	0	ExitProcedure
28	21	3	1	9	LoadAddress
29	21	27	0	10	CopyToNewBlock
30	21	3	0	10	LoadAddress
31	21	53	2	10	StoreChannel
32	22	2	0	2	LoadIntString
33	22	14	1	0	BuiltinFunc(printf)
34	22	70	0	0	PopStack
35	-1	22	0	0	ExitProcedure
36	29	6	0	0	LoadIntegerLiteral
37	29	3	1	9	LoadAddress
38	29	9	1	0	Store
39	29	70	0	0	PopStack
40	29	4	1	9	LoadValue
41	29	6	0	10	LoadIntegerLiteral
42	29	36	0	0	LessThan
43	29	16	0	59	Jumpz
44	29	4	1	9	LoadValue

45	29	4	1	9	LoadValue
46	29	48	0	0	Multiply
47	29	3	1	11	LoadAddress
48	29	4	1	9	LoadValue
49	29	24	0	6	ArrayIndexing
50	29	9	2	0	Store
51	29	4	1	9	LoadValue
52	29	68	0	0	DuplicateStackTop
53	29	6	0	1	LoadIntegerLiteral
54	29	46	0	0	Add
55	29	3	1	9	LoadAddress
56	29	9	2	0	Store
57	29	70	0	0	PopStack
58	29	15	0	40	Jump
59	30	6	0	0	LoadIntegerLiteral
60	30	3	1	9	LoadAddress
61	30	9	1	0	Store
62	30	70	0	0	PopStack
63	30	4	1	9	LoadValue
64	30	6	0	10	LoadIntegerLiteral
65	30	36	0	0	LessThan
66	30	16	0	94	Jumpz
67	30	69	0	0	SequentialTimeOff
68	30	4	1	9	LoadValue
69	30	6	0	1	LoadIntegerLiteral
70	30	46	0	0	Add
71	30	3	0	20	LoadAddress
72	30	4	1	9	LoadValue
73	30	24	0	4	ArrayIndexing
74	30	67	1	0	MoveChannelVarToNewProc
75	30	57	0	1	CreateNewForkChild
76	30	13	0	86	ForkJump
77	30	19	0	8	NewFrame

78	30	4	1	9	LoadValue
79	30	3	1	11	LoadAddress
80	30	4	1	9	LoadValue
81	30	24	0	6	ArrayIndexing
82	30	65	0	3	WakeupProcess
83	30	20	0	10	Call
84	30	71	0	1	UpdateDisplay
85	30	58	0	0	ForkChildEnd
86	30	4	1	9	LoadValue
87	30	68	0	0	DuplicateStackTop
88	30	6	0	1	LoadIntegerLiteral
89	30	46	0	0	Add
90	30	3	1	9	LoadAddress
91	30	9	2	0	Store
92	30	70	0	0	PopStack
93	30	15	0	63	Jump
94	31	6	0	0	LoadIntegerLiteral
95	31	3	1	9	LoadAddress
96	31	9	1	0	Store
97	31	70	0	0	PopStack
98	31	4	1	9	LoadValue
99	31	6	0	10	LoadIntegerLiteral
100	31	36	0	0	LessThan
101	31	16	0	111	Jumpz
102	31	14	0	21	BuiltinFunc(join)
103	31	4	1	9	LoadValue
104	31	68	0	0	DuplicateStackTop
105	31	6	0	1	LoadIntegerLiteral
106	31	46	0	0	Add
107	31	3	1	9	LoadAddress
108	31	9	2	0	Store
109	31	70	0	0	PopStack
110	31	15	0	98	Jump

111	32	6	0	0	LoadIntegerLiteral
112	32	3	1	9	LoadAddress
113	32	9	1	0	Store
114	32	70	0	0	PopStack
115	32	4	1	9	LoadValue
116	32	6	0	10	LoadIntegerLiteral
117	32	36	0	0	LessThan
118	32	16	0	135	Jumpz
119	32	2	0	30	LoadIntString
120	32	4	1	9	LoadValue
121	32	3	1	11	LoadAddress
122	32	4	1	9	LoadValue
123	32	24	0	6	ArrayIndexing
124	32	8	0	0	Dereference
125	32	14	3	0	BuiltinFunc(printf)
126	32	70	0	0	PopStack
127	32	4	1	9	LoadValue
128	32	68	0	0	DuplicateStackTop
129	32	6	0	1	LoadIntegerLiteral
130	32	46	0	0	Add
131	32	3	1	9	LoadAddress
132	32	9	2	0	Store
133	32	70	0	0	PopStack
134	32	15	0	115	Jump
135	34	3	1	9	LoadAddress
136	34	6	0	0	LoadIntegerLiteral
137	34	6	0	9	LoadIntegerLiteral
138	34	6	0	1	LoadIntegerLiteral
139	34	55	0	0	BeginParallel
140	34	61	0	159	BeginForallLoop
141	35	79	1	9	DadLoaDForallIndexVal
142	35	3	0	20	LoadAddress
143	35	79	1	9	DadLoaDForallIndexVal

144	35	24	0	4	ArrayIndexing
145	35	67	1	0	MoveChannelVarToNewProc
146	35	59	0	1	CreateNewForallChild
147	35	15	0	158	Jump
148	35	19	0	8	NewFrame
149	35	63	1	9	SonLoadForallIndexVal
150	35	3	1	11	LoadAddress
151	35	63	1	9	SonLoadForallIndexVal
152	35	24	0	6	ArrayIndexing
153	35	65	0	3	WakeupProcess
154	35	20	0	10	Call
155	35	71	0	1	UpdateDisplay
156	35	64	148	1	TestGroupIncForallIndex
157	35	60	0	0	ForallChildEnd
158	35	62	0	141	EndForallLoop
159	35	56	0	0	EndParallel
160	37	6	0	0	LoadIntegerLiteral
161	37	3	1	9	LoadAddress
162	37	9	1	0	Store
163	37	70	0	0	PopStack
164	37	4	1	9	LoadValue
165	37	6	0	10	LoadIntegerLiteral
166	37	36	0	0	LessThan
167	37	16	0	184	Jumpz
168	37	2	0	44	LoadIntString
169	37	4	1	9	LoadValue
170	37	3	1	11	LoadAddress
171	37	4	1	9	LoadValue
172	37	24	0	6	ArrayIndexing
173	37	8	0	0	Dereference
174	37	14	3	0	BuiltinFunc(printf)
175	37	70	0	0	PopStack
176	37	4	1	9	LoadValue

177	37	68	0	0	DuplicateStackTop
178	37	6	0	1	LoadIntegerLiteral
179	37	46	0	0	Add
180	37	3	1	9	LoadAddress
181	37	9	2	0	Store
182	37	70	0	0	PopStack
183	37	15	0	164	Jump
184	39	69	0	0	SequentialTimeOff
185	39	6	0	10	LoadIntegerLiteral
186	39	3	0	10	LoadAddress
187	39	67	10	0	MoveChannelVarToNewProc
188	39	57	0	1	CreateNewForkChild
189	39	13	0	197	ForkJump
190	39	19	0	11	NewFrame
191	39	3	1	11	LoadAddress
192	39	25	0	10	LoadBlock
193	39	65	0	10	WakeupProcess
194	39	20	0	12	Call
195	39	71	0	1	UpdateDisplay
196	39	58	0	0	ForkChildEnd
197	-1	22	0	0	ExitProcedure

Physical Arch Spec (topo, dim, #_of_proc)

-1

Virtual Arch Spec (topo, dim, #_of_proc)

10 1 11 ring

11

Identifier Table

16

	name	obj		ref	lev	siz	forl	lnk	adr	typ	tnam	cref	chflg
1	$tag00	1	vari	4	0	22	0	0	0	7	stru	0	0

2	v_topo	5	comp	-1	0	1	0	0	0	2	int	0	0
3	num_dim	5	comp	-1	0	1	0	2	1	2	int	0	0
4	dim_size	5	comp	2	0	20	0	3	2	5	arra	0	0
5	c1	1	vari	3	0	1	0	1	9	6	chan	0	1
6	c2	1	vari	4	0	10	0	5	10	6	chan	0	1
7	c3	1	vari	4	0	10	0	6	20	5	arra	0	0
8	f	4	func	1	0	0	0	7	3	0	void	0	0
9	x	1	vari	-1	1	1	0	0	9	2	int	0	0
10	y	1	vari	-1	1	1	0	9	10	2	int	1	0
11	f2	4	func	2	0	0	0	8	28	0	void	0	0
12	a	1	vari	5	1	10	0	0	9	5	arra	0	0
13	main	4	func	3	0	0	0	11	36	0	void	0	0
14	i	1	vari	-1	1	1	0	13	9	2	int	0	0
15	j	1	vari	-1	1	1	0	14	10	2	int	0	0
16	a1	1	vari	6	1	10	0	15	11	5	arra	0	0

Array Table

6

	elref	elsize	size	low	high	inxtyp	eltyp	elchflag	eltypnam
1	-1	1	1	0	0	2	2	0	int
2	-1	1	20	0	19	2	2	0	int
3	-1	1	10	0	9	2	2	0	int
4	5	1	10	0	9	2	6	0	chan
5	-1	1	10	0	9	2	2	0	int
6	-1	1	10	0	9	2	2	0	int

Block Table

4

	last	lastpar	psize	vsize
0	8	-1	-1	30
1	11	10	11	11

```
  2   13        12       19      19
  3   16        -1        9      21

          Channel/Pointer Table
5
    eltyp  eltypname  elref  elsize
===================================
    1   5     arra       1       1
    2   7     stru       1      22
    3   2     int       -1       1
    4   5     arra       3      10
    5   2     int       -1       1

          Real Constant Table
0
 Index    Constant
=====================


             String Table
57
*Last fork process finished!
           Source code
40
Line  Contents
=====================
  1 #include "cpc.h"
  2
  3 arch ring m[11];
  4
  5 channel int c1;
  6
  7 typedef int int10[10];
  8 channel int10 c2;
```

```
 9
10 channel int c3[10];
11
12 void f(int x, int& y)
13 {
14   if (x==9) c3[0] = x;
15   else      c3[x+1] = x;
16   y = c3[x];
17 }
18
19 void f2(int a[10])
20 {
21   c2 = a;
22   printf("Last fork process finished!");
23 }
24
25 void main()
26 {
27   int i, j, a1[10];
28
29   for (i=0; i<10; i++) a1[i] = i*i;
30   for (i=0; i<10; i++) fork [i+1;c3[i]] f(i, a1[i]);
31   for (i=0; i<10; i++) join();
32   for (i=0; i<10; i++) printf("a1[%d] = %d\n", i, a1[i]);
33
34   forall(i from 0 to 9 grouping 1)
35     [i;c3[i]] f(i, a1[i]);
36
37   for (i=0; i<10; i++) printf("a1[%d] = %d\n", i, a1[i]);
38
39  fork [10; c2] f2(a1);
40 }
```

Breakability Table

41

Line	VC_index_value
1	3
2	3
3	3
4	3
5	3
6	3
7	3
8	3
9	3
10	3
11	3
12	3
13	3
14	7
15	3
16	20
17	28
18	28
19	28
20	28
21	28
22	32
23	36
24	36
25	36
26	36
27	36
28	36
29	36

30	59
31	94
32	111
33	135
34	135
35	135
36	160
37	160
38	184
39	184
40	197
41	198

Bibliography

[1] Bruce P. Lester. *The Art of Parallel Programming*

[2] Hillis, W.D. *The Connection Machine.* Cambridge, MASS. The MIT Press, 1986.

[3] C. Koelbel et al. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[4] K.Reeuwijk, W. Denissen, H. Sips, and E. Paalvaast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897-914, Sept 1996.

[5] High Performance Fortran Forum. High Performance Fortran Language Specification, May 1993.

[6] *FORGE High Performance Fortran.* Applied Parallel Research, Sacramento, California. October 1995.

[7] Michael Philippsen and Markus U. Mock. Data and Process Alignment in Modula-2*. In C.W. Kebler(Ed.): *Automatic Parallelization – New Approaches to Code Generation, Data Distribution and Performance Prediction,* pages 177-191. Wiesbaden: Vieweg, 1994.

[8] K.-C Li and H. Schwetman. Vector C: A Vector Processing language. *Journal of Parallel and Distributed Computing,* 132-169, 1985.

[9] J. R. Rose and G. L. Steele Jr. C*: An Extended C Language for Data Parallel Programming. In *Proceedings Second International Conference on Supercomputing, Vol. 2*, pages 2-16, San Francisco, CA, May 1987.

[10] Michael J. Quinn and Philip J. Hatcher. *Data-Parallel Programming on Multicomputers.* IEEE Software, 7(5):69-76, September 1990.

[11] Philip J. Hatcher and Michael J.Quinn. *Data-Parallel Programming in MIMD Computers.* MIT Press, 1991.

[12] Judith Schlesinger and Maya Gokhale. DBS Reference Manual. Technical Report TR-92-068, Supercomputing Research Center,1992.

[13] ANSI American National Standard Institute, Inc., New York. American National Standard for Information Systems, *Programming Language C.* ANSI X3.159-1989,1990.

[14] Peter A. Darnell Philip E. Margolis. *Software Engineering in C.*

[15] *The Occam Programming Manual.* Englewood Cliffs, N.J. Prentice Hall. Inmos, 1985.

[16] G.Jones and M. Goldsmith. *Programming in Occam 2.* Prentice-Hall, 1988.

[17] C.A.R. Hoare. *Communicating Sequencial Processes.* Prentice-Hall International Series in Computer Science, 1985.

[18] Karp, A., and Babb, R. G. *A comparison of 12 parallel Fortran dialects.* IEEE Software (September), pp. 52-67. 1988

[19] Arthur B.Pyster. *Compiler Design and Construction.*

[20] Jean-Paul Tremblay, Paul G. Sorenson. *The theory and practice of compiler writing.*

[21] Sangyeun Cho, Jenn-Yuan Tsai. *High-Level Information - An Approach for Integrating Front-End and Back-End Compilers.*

[22] David A. Padua. *Outline of a Roadmap for Compiler Technology.*

[23] Christopher W. Fraser. *A Retargetable Compiler for ANSI C*, SIGPLAN Notices 26, 10(Oct. 1991), 29-43.

[24] Christoph W. Kebler, Helmut Seidl. "The Fork95 Parallel Programming Language: Design, Implementation, Application", *Int. Journal of Parallel Programming 25(1), Feb. 1997, pages 17-49*

[25] Mohammad R. Haghighat and Constantine D. Polychronopoulos. *Symbolic analysis for parallelizing compilers.* Volume 18, No. 4 (July 1996). ACM Transactions on Programming Languages and Systems. Pages 477-518

[26] Designing and Building Parallel Programs, by Ian Foster.

http://www-unix.mcs.anl.gov/dbpp/

[27] Parallel Functional Programming: An Introduction Next: Introduction Parallel Functional Programming: An Introduction Kevin Hammond Department of Computing Science, University of Glasgow, Gl. http://www.dcs.st-and.ac.uk/ kh/papers/pasco94/pasco94.