

9461041

Advances in Languages and Compilers for Parallel Processing



TP31
A244

9461041

RESEARCH MONOGRAPHS IN PARALLEL AND DISTRIBUTED COMPUTING

Edited by

Alexandru Nicolau, University of California, Irvine,

David Gelernter, Yale University,

Thomas Gross, Carnegie-Mellon University,

David Padua, University of Illinois at Urbana-Champaign

Advances in Languages and Compilers for Parallel Processing



E9461041

Pitman, London

The MIT Press, Cambridge, Massachusetts

PITMAN PUBLISHING
128 Long Acre, London WC2E 9AN

© A. Nicolau, D. Gelernter, T. Gross, D. Padua 1991

First published 1991

Available in the Western Hemisphere and Israel from
The MIT Press
Cambridge, Massachusetts (and London, England)

ISSN 0953-7767

British Library Cataloguing in Publication Data

Advances in languages and compilers for parallel
processing.—(Research monographs in parallel
and distributed computing)

I. Nicolau, Alexandru II. Padua, David

III. Series

004

ISBN 0-273-08841-6

Library of Congress Cataloging-in-Publication Data

Advances in languages and compilers for parallel processing / edited
by Alexandru Nicolau.

p. cm.—(Research monographs in parallel and distributed
computing)

Includes bibliographical references.

ISBN 0-262-64028-7

1. Programming languages (Electronic computers) 2. Compilers
(Computer programs) 3. Parallel processing (Electronic computers)

I. Nicolau, Alexandru. II. Series.

QA76.7.A38 1991

005.13—dc20

All rights reserved; no part of this publication may be reproduced,
stored in a retrieval system, or transmitted in any form or by any
means, electronic, mechanical, photocopying, recording or otherwise
without the prior written permission of the publishers or a licence
permitting restricted copying in the United Kingdom issued by the
Copyright Licensing Agency Ltd, 20 Tottenham Court Road, London
W1P 0BR. This book may not be lent, resold, hired out or otherwise
disposed of by way of trade in any form of binding or cover other than
that in which it is published, without the prior consent of the
publishers.

Reproduced and printed by photolithography
in Great Britain by Biddles Ltd, Guildford

TITLES IN THIS SERIES

Optimizing Supercompilers for Supercomputers, Michael Wolfe, 1989

Reconfigurable Processor-Array: A Bit-Sliced Parallel Computer,
Andrew Rushton, 1989

Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors,
Vivek Sarkar, 1989

Functional Programming for Loosely-coupled Multiprocessors, Paul H. J. Kelly,
1989

Algorithmic Skeletons: Structured Management of Parallel Computation,
Murray I. Cole, 1989

Execution Models of Prolog for Parallel Computers, Péter Kacsuk, 1990

Languages and Compilers for Parallel Computing, David Gelernter,
Alexandru Nicolau and David Padua (Editors), 1990

Massively Parallel Computing with the DAP, Dennis Parkinson and John Litt
(Editors), 1990

Implementation of a General-Purpose Dataflow Multiprocessor,
Gregory M. Papadopoulos, 1991

Implementation of Non-Strict Functional Programming Languages,
Kenneth R. Traub, 1991

Parallel Processing for Computer Graphics, Stuart Green, 1991

Lazy Functional Languages: Abstract Interpretation and Compilation,
Geoffrey Burn, 1991

Advances in Languages and Compilers for Parallel Processing,
Alexandru Nicolau, David Gelernter, Thomas Gross and David Padua (Editors),
1991

Foreword

This book contains selected refereed papers representing recent advances in Languages and Compilers for Parallel Computing. Early versions of these papers were presented at the Third Workshop on Languages and Compilers for Parallel Computing held during August 1–3 1990 in Irvine California, under the sponsorship of the Computer Systems Design Research Unit at the University of California at Irvine. The previous workshops in this series were held in Ithaca NY, August 1988, and in Urbana-Champaign, 1989.

The topics of the papers in the book are representative of the various aspects of research in this area and illustrate the great amount of interest parallel computing in general, and parallelizing compilers and languages in particular, are currently generating.

The book is divided into several sections, roughly corresponding to the major efforts in the field. The papers by Eigenmann *et al.*, Foster & Overbeek, and Jagannathan discuss languages and language extensions. Those by Gelernter *et al.* and Gannon *et al.* present two innovative environments for parallel programming. Miller & Netzer describe techniques for debugging parallel programs. Guzzi *et al.*, Eisenbeis *et al.*, Solworth, and Gao *et al.* deal with the very important issue of data organization and management during parallel processing. New compiler techniques for parallelizing loops are described by Banerjee, Ayguadé *et al.*, and Wolf & Lam. Important new results in code scheduling are given by Gross & Ward and Aiken & Nicolau. Innovative approaches to dependency analysis and representation are provided by Kallis & Klappholz, Haghighat & Polychronopoulos, and Pingali *et al.* An interesting insight into the measurement of parallelism implicit in ordinary programs is revealed by Larus. Finally, Mehrotra & Van Rosendale, Quinn *et al.*, Li & Chen, and Dietz *et al.* deal with programming and compiling for distributed and shared memory multiprocessors.

We, the Editors, are very pleased with the breadth and depth of the work presented in these papers. Taken together, these papers are an accurate reflection of the state of research in Languages and Compilers for Parallel Computing in 1990. We hope this book will be as interesting to the reader as it was for us to compile.

Alexandru Nicolau.

Contents

- 1 Cedar Fortran and its Restructuring Compiler 1**
R. Eigenmann
J. Hoeflinger
G. Jaxon
D. Padua
*Center for Supercomputing Research and Development,
University of Illinois at Urbana-Champaign*
- 2 Bilingual Parallel Programming 24**
Ian Foster
Ross Overbeek
Mathematics and Computer Science Division, Argonne National Laboratory
- 3 Optimizing Analysis for First-Class Tuple-Spaces 44**
Suresh Jagannathan
Department of Computer Science, Yale University
- 4 The Linda Program Builder 71**
Shakil Ahmed
Nicholas Carriero
David Gelernter
Department of Computer Science, Yale University
- 5 SIGMACS: a Programmable Programming Environment 88**
Bruce Shei
Dennis Gannon
Indiana University
- 6 Detecting Data Races in Parallel Program Executions 109**
Robert H. B. Netzer
Barton P. Miller
Computer Sciences Department, University of Wisconsin-Madison
- 7 A Strategy for Array Management in Local Memory 130**
Christine Eisenbeis
INRIA, Le Chesnay
William Jalby
Daniel Windheiser
François Bodin
IRISA, Campus Universitaire de Beaulieu, Rennes

- 8 On the Performance of Parallel Strips-Based Lists 152**
Jon A. Solworth
University of Illinois at Chicago
- 9 An Efficient Monolithic Array Constructor 172**
G. R. Gao
Advanced Computer Architecture and Program Structures Group, McGill University
Robert Kim Yates
Advanced Computer Architecture and Program Structures Group, McGill University
Jack B. Dennis
Laboratory for Computer Science, Massachusetts Institute of Technology
Lenore Restifo Mullin
Centre de Recherche Informatique de Montréal
- 10 Unimodular Transformations of Double Loops 192**
Utpal Banerjee
Intel Corporation, Santa Clara
- 11 Parallelism Evaluation and Partitioning of Nested Loops
for Shared Memory Multiprocessors 220**
E. Ayguadé
J. Labarta
J. Torres
J. M. Llaberia
M. Valero
*Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya,
Facultat d'Informàtica de Barcelona*
- 12 An Algorithmic Approach to Compound Loop Transformations 243**
Michael E. Wolf
Monica S. Lam
Computer Systems Laboratory, Stanford University
- 13 The Suppression of Compensation Code 260**
Thomas Gross
School of Computer Science, Carnegie Mellon University
Michael Ward
IBM T. J. Watson Research Center, Yorktown Heights, NY
- 14 A Realistic Resource-Constrained Software Pipelining Algorithm 274**
Alexander Aiken
IBM Almaden Research Center, San Jose
Alexandru Nicolau
Dept. Information and Computer Science, University of California, Irvine

- 15 Handling Unresolvable Array-Access Aliases in Refined C** 291
Apostolos D. Kallis
David Klappholz
*Department of Electrical Engineering and Computer Science,
Stevens Institute of Technology*
- 16 Symbolic Dependence Analysis for High-Performance Parallelizing Compilers** 310
Mohammad Reza Haghighat
Constantine D. Polychronopoulos
*Center for Supercomputing Research and Development,
University of Illinois at Urbana-Champaign*
- 17 Parallelism in Numeric and Symbolic Programs** 331
J. R. Larus
Computer Sciences Department, University of Wisconsin-Madison
- 18 An Efficient Implementation of Thread-Specific Data** 350
Mark D. Guzzi
Rich Simpson
Don Parce
Encore Computer Corporation, Marlborough MA
- 19 Programming Distributed Memory Architectures Using Kali** 364
Piyush Mehrotra
John Van Rosendale
*Institute for Computer Applications in Science and Engineering,
NASA Langley Research Center, Hampton Va*
- 20 Implementing a Data Parallel Language on a Tightly Coupled Multiprocessor** 385
Michael J. Quinn
Oregon State University
Philip J. Hatcher
University of New Hampshire
Bradley K. SeEVERS
Oregon State University
- 21 Automating the Coordination of Interprocessor Communication** 402
Jingke Li
Dept. Computer Science, Portland State University
Marina Chen
Dept. Computer Science, Yale University
- 22 An Introduction to Static Scheduling for MIMD Architectures** 425
Henry G. Dietz
Matthew T. O'Keefe
Abderrazek Zaafrani
School of Electrical Engineering, Purdue University

23 Dependence Flow Graphs: an Algebraic Approach to Program Dependencies 445

Keshav Pingali

Micah Beck

Richard Johnson

Mayan Moudgill

Paul Stodghill

Cornell University

1 Cedar Fortran and its Restructuring Compiler

R. Eigenmann, J. Hoeflinger, G Jaxon, D. Padua

Abstract

The Cedar architecture integrates shared memory into a distributed system of Alliant mini-supercomputers. Nested parallel loops and a hierarchical memory model allow Cedar Fortran to offer a wide range of implementation possibilities for an algorithm, which makes automatic parallelization easy to do, but hard to do well. Techniques from both shared memory and distributed memory programming paradigms are applied to this problem. Early results show that restructuring can speed up kernels and algorithms. We identify improved techniques that may extend these results to full applications.

1 Cedar Fortran within the Cedar System

The hardware and software structure of Cedar was motivated by a desire to achieve a low cost/performance ratio for a wide variety of applications, and to make that performance easily available to the user [28]. The hardware offers a hierarchical memory structure as well as the means to control fine-, medium-, and coarse-grain parallelism. The Cedar Fortran compiler, with support from its runtime library and the Xylem operating system, makes those architectural features available through the direct expression of parallelism within a Fortran context. The compiler includes a restructurer which can automatically translate serial programs into a parallel form.

Cedar combines two complementary approaches to parallel processing. On one hand, it can be viewed as a distributed system with high-bandwidth communication channels. On the other hand, it may be seen as a shared memory machine. A given program may

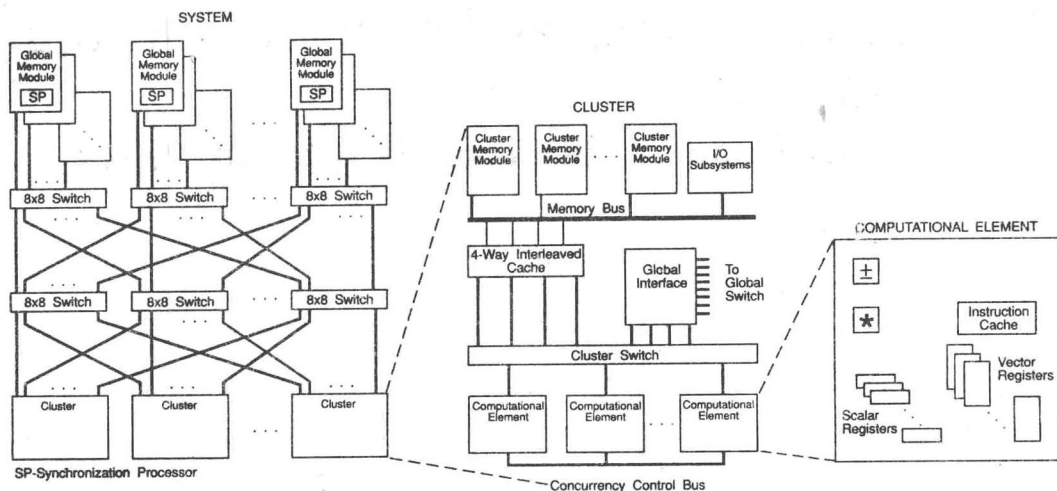


Figure 1: The Cedar Architecture

make use of both programming paradigms as the situation dictates.

The layers of concurrency in Cedar and the various ways parallel activity can be expressed in the Cedar Fortran language present many opportunities to speed up parallel programs. But, this becomes a complexity issue for the Fortran restructurer, which must choose one good translation for a given serial program from among many alternatives.

In this paper, we describe the Cedar architecture, the software environment in which Cedar Fortran runs, and the Cedar Fortran language. We then discuss the issues that this hardware and software environment presents to the Fortran restructurer. Finally, we mention some results and future directions for our research.

1.1 The Cedar Hardware Architecture

The Cedar machine (Figure 1) consists of several processor *clusters* (currently four) connected through a network to a *global memory*.

Each cluster is a modified Alliant FX/8 multiprocessor, containing up to eight vector-pipelined *computational elements* (CEs) that share a *cluster memory* [3]. Access to each cluster's local memory is accelerated by a large data cache shared by all the CEs of that cluster. On each cycle, it can respond to four memory references from different CEs in the cluster. In addition, each CE has a private instruction cache and a virtual-to-physical address translation buffer.

The physical addresses generated by Cedar's CEs have been extended. Addresses beyond the original range are routed to the CE's *global interface* instead of to the cluster's memory cache. The global interface links each CE to global memory through two communication networks. One sends data, the other receives it. Each direction in the network crosses two stages, each stage is made of 8×8 crossbar switches. The switches interconnect in the "perfect shuffle" pattern that forms an Ω -network. The resulting communication bandwidth is close to that of two complete crossbars at a much lower cost [13].

The latency to access a single word in the global memory can still be large, so the global interface includes a pipelined vector-prefetch capability. Using this, a CE may request a block of data from global memory and continue its computations while the prefetch buffer fills up. Once vector data resides in the prefetch buffer it can be accessed at the CE's peak rate. The Cedar Fortran compiler places a prefetch request in the code stream before each use of a global vector; sometimes overlapping other computation. The following table compares the incremental cost of each Cedar memory access mode.

		RELATIVE SPEEDS OF CEDAR MEMORY			
		LOCALE		ACCESS MODE	
FAST	Register	R	W		
	Global	R		vector read already prefetched	
	Cluster	R	W	cache hit	
	Global	R		vector read while prefetching	
	Cluster	R	W	vector read/write	
	Global		W	write	
	Cluster	R	W	cache miss	
	Cluster		W	test & set	
	Global	R		vector read no prefetching	
	Global	R		scalar read	
	Global		W	test & set	
LESS FAST	Global	R	W	synchronization op	

Synchronization processors on the global memory boards provide a large set of atomic operations on data stored there. The Cedar synchronization hardware [45, 38] operates on up to two consecutive words of global memory. Many common critical sections are available as indivisible instructions, including the *fetch-next-iteration* step for a multi-cluster parallel loop. This hardware supports fine-grain synchronization among all CEs in every Cedar cluster.

The CEs within each cluster also connect to a *concurrency control unit* (CCU) by a special purpose bus. The CCU can form a group of CEs that will cooperate to execute parallel codes. This has been implemented in software on other machines such as the IBM RP3 [41], Sequent [36], and the Cray X-MP [9], where it is known as *microtasking*. The Alliant FX series hardware support for microtasking includes

self-scheduling of the iterations of a 1-cluster parallel loop: as each CE finishes an iteration, the CCU gives it the next available iteration number,

AWAIT and ADVANCE instructions to serialize the order of some memory references made from different iterations of a parallel loop, and

barrier synchronization after the last iteration, to ensure that all iterations are done before the program continues.

1.2 The Xylem Operating System

The software that coordinates the clusters is the Xylem Operating System [20], an extension of Unix. Under Xylem, a program executes as a *Xylem process*. A process is made up of one or more independently scheduled *tasks*. Each task is assigned to a particular Cedar cluster for its lifetime. In a task's address space, each page of memory has its own attributes of *locale* (**cluster** or **global** memory) and *visibility* (**private** or **shared** by all tasks of this process). All four combinations of locale and visibility are possible [33].

Pages that are **shared cluster** are handled in a special way. If the page is *read-only*, it is simply copied into the cluster memory of each task that refers to it. If the page is *writable*, the first task using it gets a cluster memory copy; Xylem traps subsequent uses in other tasks so that the program's runtime system can mediate the communications needed to actually share the data.

The **shared** memory of a process is owned by a master address translation table which Xylem keeps in global memory. The individual tasks take registered copies of entries from this table to initialize or restore invalid entries in their private memory maps. Xylem's virtual memory system periodically invalidates a few random entries in each task's memory map. Thus, unused pages gradually lose their registered users. A shared page with no registered referents may be moved to secondary storage: either to the cluster memory of the last referent, or to a disk connected to that cluster.

This design for shared virtual memory adds a hidden cost to programs that make sustained use of shared memory: keeping the private memory maps valid. When amortized over all memory references, this cost is quite low. But, revalidating any single entry takes several hundred microseconds, comparable to entire loops in many applications. This is a source of load imbalance between processors. Self-scheduling of parallel work can compensate for some of this imbalance. But, scheduling across clusters usually increases the map revalidation costs since it increases the number of memory maps involved.

1.3 The Cedar Fortran Runtime Library

Various intrinsic functions, for which it is not practical to generate inline code, are supplied by a runtime library. The library includes a set of Cray-style tasking and synchronization routines [16] that exercise Xylem system services. In particular these support the CTSK and MTSK family of intrinsics to be described in section 1.4.3.

Chief among the services supplied by the runtime library is a software version of microtasking that can quickly spread work across clusters. In order to use several clusters in parallel, the library runs the program in a Xylem process that contains several extra *helper* tasks ("implicit tasks" in IBM terminology[26]). The helpers run for the duration of the process, waiting for microtask work to be posted. The most important unit of microtask work is a multicluster parallel loop, which we call a *spread loop*. When some task reaches a spread loop, it posts one microtask describing the loop into global memory. We say this task is the *parent* of the microtask. This signals the helper tasks to wake up and join the parent in competing for loop iterations to execute. We implemented three variations of this simple idea in separate libraries, which we call *Queued*, *Simple*, and *Static*.

Interfaces	Queued	Simple	Static
	MTSKSTART SDOALL XDOALL QUIT QQUIT	SDOALL XDOALL	SDOALL ID=0,n
SDOALL helped by	$0 \rightarrow \#helpers$	$0 \rightarrow \#helpers$	$n \equiv \#helpers$
Max# microtasks	<i>queue size</i>	1 at a time	1 at a time
Max# parents	<i>process size</i>	<i>#nonhelpers</i>	1
Sync Used	Test&Set Lock	Test&Set Lock	Barrier
Wait Used	spin/dawdle/block	spin	spin
Avg. Latency ¹	9 μ /100 μ /100m sec	9 μ sec	9 μ sec
4-way Fork & Join ²	320 μ sec	180 μ sec	40 μ sec
Scheduling	self-scheduled	self-scheduled	static schedule
Wait Used	spin/dawdle	spin	none
Get Next Iteration ²	25 μ sec	<10 μ sec	0

¹no other jobs active

²minimum times

Figure 2: Comparison of Three Microtasking Libraries

Queued: Like a traditional scheduler, this library queues up microtasking work as it is generated by several parent tasks, or by the nesting of spread loops. In theory, the helpers should stay busy constantly emptying this queue; they do this by *self-dispatching*. This scheme does not guarantee that a particular helper will participate in the execution of any particular loop. The flexible control mechanism needed for this design invited the inclusion of additional features: Loops spread by this library version can be terminated prematurely by a QUIT statement; and parallel subroutine calls can be queued alongside parallel loops by MTSKSTART.

The "queued" library assumes that there is a vast variety of parallel work in progress, so it allows other tasks to take over the cluster when one task must wait. For a short time the library code repeatedly tests the wakeup condition; but it soon surrenders the task's timeslice (via the dawdle system call). Finally, if the wakeup condition is still not satisfied, further execution of the task is blocked until a wakeup signal is posted (Xylem system calls wait_ and clear_xlock). Synchronization delays can be quite high because the task scheduler on each cluster and the microtask schedulers in each process work independently. But since the clusters can stay busy during these delays, overall throughput is usually acceptable.

Under a simple load (e.g., one spread loop running in single-user mode) the "queued"

library can avoid much of the overhead and unpredictability introduced by the system calls. But it cannot avoid all costs of the extra features, nor the cost of queueing and dequeuing. The available parallelism is also reduced by the latency time it takes to reactivate the helpers and return them from the `dawdle` or `wait_xlock` system calls that they made while waiting for work.

Simple: So, in the "simple" library version, we restricted or removed features to reduce costs: tasks which must wait never surrender their processor; QUITs and MTSKs are not supported; and only one loop may be spread at any one time (others are forced into serial execution). We retained the idea that loop iterations are self-scheduled and that helpers are self-dispatched (i.e., helpers are only volunteers). In exchange we experienced lower, and more reproducible, startup and shutdown costs, and improved turnaround time for single applications under light load conditions.

Self-scheduling and self-dispatching distribute a loop's iteration space across the clusters in a fortuitous pattern, that tends to balance the workload. These techniques must be controlled by short *critical* sections of code during which one processor has exclusive use of the microtask description. In both the "simple" and "queued" libraries, the critical sections can be provided by the Cedar synchronization hardware. But as of this report, we have not yet timed library versions using that hardware. Instead we have used traditional locks, supported by an atomic "Test & Set" instruction to protect critical regions. This software method is three times slower than the Cedar hardware, so further improvement is expected.

Static: In some cases, load balancing is not as important as low overhead and absolute repeatability. So we implemented a *statically* dispatched and scheduled SDOALL, without critical regions. The "static" library guarantees that every helper task participates in every parallel section. Each helper is assigned a unique index for the life of the process, and also knows the total count of helpers. With these parameters, a compiler can distribute any iteration space or index set in a repeatable pattern. In section 2.7 we describe how repeatability will allow the restructurer to use this form of SDOALL for data distribution.

The high performance of the "static" library is vulnerable to scheduling disruptions that arise on one cluster, but require that all helpers be delayed. The average performance of static microtasks deteriorates quickly as other loads are added to the system. However, this library has let us explore Cedar's peak performance.

4 The Cedar Fortran Language

The Cedar Fortran language [24] is an extension of Alliant's FX/Fortran [4], which is FORTRAN77 augmented with vector constructs like those in the Fortran 90 standard [44]. The additions made to Alliant Fortran to form Cedar Fortran are mainly those that express various types of parallel loops, and make use of Cedar's memory system and runtime library facilities.

1.4.1 Cedar Fortran data declarations

Cedar Fortran includes three new statement types that declare the memory attributes (*locale* and *visibility*) of the storage needed for program variables.

```
GLOBAL variable [ , variable ] ...  
CLUSTER variable [ , variable ] ...  
PROCESS COMMON / name / variable [ , variable ] ...
```

Figure 3: Cedar Fortran data declaration statements

In Cedar Fortran, an ordinary Fortran **COMMON** declaration statically allocates private storage in the cluster memory of each task, not visible to any other task. A **COMMON** block may be declared as **PROCESS COMMON**, which marks its content as shared by all tasks of this Xylem process. By default a **PROCESS COMMON** block is placed in global memory.

Variables outside common blocks may be declared **GLOBAL** or **CLUSTER**. By default, a subprogram's variables are dynamically allocated on a stack. This yields separate storage for each recursive or parallel call. Static allocation (**SAVE**) is also available, but it must be properly synchronized in the case of parallel calls to the subprogram.

A **CLUSTER** declaration specifies that the variable will be (stacked) in private cluster memory, inaccessible to other tasks. Declaring a variable **GLOBAL** means it will be allocated on a stack in shared global memory so that it can be used by every task in the program. By itself, the **GLOBAL** statement does not enlarge the *scope* of a variable; other subprogram calls only learn the variable's address by ordinary parameter passing mechanisms.

COMMON and **PROCESS COMMON** block names may also appear in **CLUSTER** or **GLOBAL** statements to change their default locale (but not their visibility). For example, a **PROCESS COMMON** block named in a **CLUSTER** statement is placed in shared cluster memory as described in section 1.2. Figure 3 summarizes the syntax of these statements.

1.4.2 Concurrent loops

The syntax of a concurrent loop in Cedar Fortran is an extension of the syntax for a FORTRAN77 **DO** loop, with "**DO**" changed to a new keyword that selects a concurrent execution rule. The keywords are: **CDOALL**, **CDOACROSS**, **SDOALL**, **SDOACROSS**, **XDOALL**, and **XDOACROSS**. Figure 4 gives a synopsis of this syntax. Several optional sections have been added to the loop body that are not available in ordinary **DO** loops:

Local Declarations define variables that are private to each iteration of the loop and inaccessible outside the loop. Several iterations may execute concurrently and refer to their local variables without interfering with each other, since they are really referring to different storage cells.

The **Preamble** is executed once by each participating entity before starting its first iteration of the loop.

SDO and **XDO** loops offer one further optional part:

$$\begin{array}{c}
 \left\{ \begin{array}{c} C \\ S \\ X \end{array} \right\} \left\{ \begin{array}{c} \text{DOALL} \\ \text{DOACROSS} \end{array} \right\} [label] \text{ index} = \text{start} , \text{ end} [, \text{incr}] \\
 [local\ declarations] \\
 \left[\begin{array}{c} \text{preamble} \\ \text{LOOP} \end{array} \right] \\
 \text{body} \\
 \left[\begin{array}{c} \text{ENDLOOP} \\ \text{postamble} \end{array} \right] \quad (\text{only for SDO or XDO loops}) \\
 \left\{ \begin{array}{c} \text{labeled statement} \\ \text{END} \left\{ \begin{array}{c} C \\ S \\ X \end{array} \right\} \left\{ \begin{array}{c} \text{DOALL} \\ \text{DOACROSS} \end{array} \right\} \end{array} \right\}
 \end{array}$$

Figure 4: Concurrent loop syntax

A **Postamble** is executed once by each participant when it finds no more iterations left to execute.

CEs participate independently in CDO and XDO loops, whereas in an SDO loop, the participants are whole tasks (parents or helpers). Every participant takes at least one iteration. Every iteration is executed by exactly one participant. In each of these types of parallel loop, once all the iterations and postambles have finished, one CE continues executing the code that follows the loop, and the others go idle or go to work for other tasks.

DOALL loops may perform their iterations in any order whatsoever and should not contain any synchronization between iterations. In contrast to this, the iterations of a DOACROSS loop are guaranteed to start in the same order as they would if the loop were serial. This makes it possible to pass synchronization signals from early iterations to later ones, an interaction called *cascade synchronization*, which can be implemented without deadlock.

CDOALL and CDOACROSS loops are executed totally within a single cluster. They use the cluster concurrency bus to activate and coordinate all CEs assigned to the task.

In CDOACROSS loops, the cluster's concurrency control unit (CCU) may be used for cascade synchronization. Cedar Fortran provides intrinsic subroutines **AWAIT** and **ADVANCE** that use the CCU to synchronize between iterations.

SDOALL and SDOACROSS loops are examples of *spread loops*. They use global memory and facilities provided by the runtime library to bring the program's *helper tasks* into parallel execution of the loop.

Idle helper tasks always leave one CE awake to watch for microtasking work. When the program's execution reaches a spread loop, a description of the loop is posted