NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

SELSYN-C: A SELF-SYNCHRONIZING PARALLEL PROGRAMMING LANGUAGE

by

Weiren Ding

School of Computer Science
McGill University, Montreal

May 1992

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

ISBN 0-315-74497-9

Abstract

In this thesis we report the design and implementation of a new self-scheduling parallel programming language, SELSYN-C. As parallel processors become more accessible to a broad range of programmers, the development of simple to use and effective programming languages becomes increasingly important. Our approach to the challenge of parallel programming language design and implementation is two-fold: (1) the design of simple extensions to C that are both easy to use for the programmer, and useful for effective compilation, and (2) the design of efficient and effective scheduling strategies that can be automatically supported by a compiler and associated run-time environment.

We outline our approach by presenting: (1) our motivation, (2) an overview of the extensions to C that form the SELSYN-C programming language, and (3) the development of a new scheduling mechanism that can be used to effectively compile SELSYN-C programs for a real parallel processor, the BBN Butterfly GP-1000. Different scheduling strategies for this mechanism were studied via several experimental tests and the results of these experiments are reported.

A source-to-source compiler supporting the SELSYN-C language has been implemented. Included in this thesis is a description of both the compiler and associated run-time environment.

Résumé

Cette thèse présente la conception et la mise en oeuvre d'un nouveau langage de programmation parallèle auto-séquenceur, SELSYN-C. Les processeurs parallèles sont de plus en plus accessibles à un grand nombre de programmeurs et le développement de langages de programmation efficaces et simples d'utilisation devient très important. Notre approche au défi que représente la conception et l'implémentation de tels langages se scinde en deux volets : (1) la conception de simples extensions au langage C qui sont à la fois faciles d'utilisation et utiles pour une compilation efficace et (2) la conception de stratégies efficaces de séquencement qui peuvent être automatiquement supportées par un compilateur et son environnement d'exécution associé.

Nous résumons notre approche en présentant : (1) nos motivations, (2) une revue des extensions au langage C qui constituent le langage de programmation SELSYN-C et (3) le développement d'un nouveau mécanisme de séquencement qui permet de compiler efficacement des programmes en SELSYN-C pour le processeur parallèle GP-1000 du BBN Butterfly. Différentes stratégies de séquencement ont été expérimentées pour ce mécanisme. Nous présentons les résultats de ces tests.

Un compilateur source-à-source supportant le langage SELSYN-C a été mis en oeuvre. Le compilateur et l'environnement d'exécution associé sont décrits dans cette thèse.

Acknowledgments

There are many people I would like to thank for contributing either directly or indirectly to this thesis. First of all, I take this opportunity to express my indebtedness to Dr. Laurie J. Hendren, my thesis supervisor, under whose guidance this work was undertaken, and without whose help and encouragement its completion would have been impossible.

I would like to thank Dr. Guang R. Gao for introducing me to this exciting research field. Thanks are also extended to ACAPs group members, especially to Bhama who gave me much help that I should appreciate. Throughout my residency at McGill University, many outstanding, kind and good-humored people, with whom I have worked, studied and talked, made my academic study more enjoyable and unforgettable. To all of you, thanks.

Needless to say, my family played an important role in making it possible for me to attend McGill University in the first place. To my parents, my sister and my brother, I hope I have made you proud. To my grandparents, my uncles and my aunts, I appreciate your invaluable care and support.

Finally, I wish to express special thanks to Helen for her encouragement and her patience with me and, particularly, her loving support.

Contents

Abstract ii

Résumé iii

Acknowledgments iv

1 Introduction 1

1.1 Motivation 2

1.2 The SELSYN-C Approach 3

1.3 Thesis Organization 4

2 SELSYN-C Language Definition 5

2.1 An Introductory Example 6

2.2 Processor-Private vs. Globally-Shared Data 7

2.3 SELSYN-C Variable Declarations 9

2.3.1 Declaring Shared Variables in SELSYN-C 9

2.3.2 Dynamically-Allocated Shared Data 12

2.3.3 Overview 13

2.4 Parallel Function Calls: Issuing Parallel Tasks 13

2.5 Summary 15

3 SELSYN Synchronization Mechanism **17**

3.1 Motivation 18

3.2 Processor-Team model 18

3.3 A Weighted Team-Division Strategy 20

3.4 Team-Cooperation 22

3.4.1 Basic Principles 23

3.5 Summary 31

4 Experimental Results **33**

4.1 Butterfly Memory Organization 34

4.2 Experimental Approach 34

4.3 Example 1: Quicksort 35

4.4 Example 2: Binary Tree Evaluation 40

4.5 Example 3: Quadrature 49

4.6 Example 4: Parallel Search 52

4.7 Summary 56

5 Implementation **59**

5.1 Overview of the Compiler 59

5.2 Constructing the Internal Representation 60

5.3 Generating Code 62

5.3.1 Implementing Shared Data Storage 62

5.3.2 Implementation of the Cooperating-Team Model 65

5.3.3 Link To the Run-Time Environment 70

5.3.4 Initialization 73

5.4 Summary 74

6 Related Work **76**

6.1 Parallel Programming Languages 76

6.2 Scheduling Mechanism 78

6.3 Compiler Generated Self-Scheduling Programs 80

6.4 Summary 81

7 Conclusions and Further Work **82**

A A Case Study of the SELSYN Mechanism **85**

B SELSYN-C Code For Binary Tree Evaluation **95**

C Source Code of The Run-Time Library **98**

Bibliography **105**

List of Figures

2.1 An example SELSYN-C program 6

2.2 Architecture of the BBN parallel processor 8

2.3 An abstract view of Processor-Private and Shared memory 9

2.4 A overview of memory allocation 14

3.1 Team dividing 20

3.2 Divide-and-conquer call graph 26

3.3 Execution of example (Part 1) 28

3.4 Execution of example (Part 2) 29

3.5 A more general example 31

4.1 BBN memory access time 34

4.2 Quicksort: 8192 elements (10 cases) 37

4.3 Relative performance on six processors 38

4.4 Quicksort: 4096 elements (10 cases) 39

4.5 Quicksort: 4096 elements (100 cases) 40

4.6 Relative performance on four processors 40

4.7 A sample binary expression tree 41

4.8 Random binary tree evaluation: 20000 nodes 42

4.9 Balanced binary tree evaluation: 20000 nodes 43

4.10 Random binary tree evaluation with delay loop: 20000 nodes 45

4.11 Random binary tree evaluation: 20000 nodes 46

4.12 Balanced binary tree evaluation: 20000 nodes 47

4.13 Random binary tree evaluation: 1023 nodes 48

4.14 Random binary tree evaluation: 1023 nodes 49

4.15 Relative performance on six processors 49

4.16 Balanced binary tree evaluation: 1023 nodes 50

4.17 Quadrature 52

4.18 Search tree 53

4.19 Binary search tree 53

4.20 Searching: find one possible solution 55

4.21 Searching: find all possible solutions 56

5.1 Structure overview 60

5.2 Shared storage 63

5.3 An example SELSYN-C program 64

5.4 Task issuing algorithm for leading processor 67

5.5 Task hoping algorithm for child processor 68

5.6 Team-waiting algorithm 69

5.7 Pool-waiting algorithm 69

5.8 Revised task issuing algorithm 70

6.1 Principle of fuzzy barrier 79

A.1 Case example 1 86

A.2 Case example 2 87

A.3 Case example 3 88

A.4 Case example 4 89

A.5 Case example 5 90

A.6 Case example 6 91

A.7 Case example 7 92

A.8 Case example 8 93

Chapter 1

Introduction

Parallel processing and related parallel applications are capturing more and more attention. Parallel computing has proved to be essential in studying complex problems such as those related to image processing, pattern recognition, parallel searching, and other applications that are computation intensive.

The exploitation of parallel processing depends on three major areas: (1) the design of architectures that support parallelism, (2) the design of application programs that can utilize parallelism, and (3) a mechanism for effectively and efficiently mapping the application to parallel architectures. At the level of architecture design, a wide variety of models have been introduced [Ell86, HP90, HJ88, HB84, Kat85, Kog81, Sab88, Sto87, Tha87]. These architectures may exploit fine-grain parallelism at the instruction level, medium-grain parallelism at the loop or vector level, or coarse-grain parallelism at the process level. In addition, some architectures have been developed for particular kinds of applications. For instance, array-processors and vector-processors are suitable for matrix computation as required in the scientific area. On the side of the applications, much of the research concentrates on representing or exposing parallelism so that the problem can be solved in parallel. Although on both sides there has been remarkable progress, there is another problem which we believe is a general key problem in parallel processing, that is, how to map the parallel application to the architecture. The goal is to find mechanisms which can

be used to express and control parallelism. These mechanisms should be easy for the programmer to use and they should lead to portable programs.

1.1 Motivation

Dealing with the problem of parallel programming is becoming more important as rapid advances in architectures leads to the development of new and cheaper parallel processors. In general, programming a parallel processor demands much more knowledge and skills than traditional programming of sequential processors. Firstly, a programmer needs to exploit the parallelism of the application and express the parallelism in a particular way. Secondly, a programmer has to deal with more resource management, in particular the management of multiple processes or processors. This resource management is often specific to a particular parallel architecture. In addition to these burdens on the application programmer, a particular parallel programming system may require a supporting run-time system. This run-time system needs to support mechanisms to exploit parallelism, control parallelism, deal with memory contention, support synchronization and so on.

Thus, a great challenge is to provide software environments that make parallel processors usable by a wide variety of programmers, while at the same time achieving high efficiency of resource management and high utilization of resources. A critical step in this development is the design of parallel programming languages that are simple to use. Programs written in such a language should provide straight-forward mechanisms for expressing parallelism, and effective mechanisms for efficiently exploiting this parallelism on real parallel processors. That is, the language should allow the programmer to express parallelism at the application level, and not force the programmer to worry about the complicated details of resource management and synchronization of tasks for a particular parallel architecture. In addition, a good parallel programming language should support the development of portable programs.

1.2 The SELSYN-C Approach

One approach to handle the above challenge is to develop a language and associated parallelism exploitation and parallelism controlling mechanisms. In this thesis we present a new parallel programming language, SELSYN-C, an extension of the traditional C programming language that we have designed and implemented. Users of the system need only master a small set of new constructs in order to develop parallel C programs, or to convert their existing C programs to parallel C programs. The two major contributions of this work are: (1) the development of simple-to-use extensions to C, and (2) the development of a compiler and associated run-time system that supports a new kind of self-synchronizing mechanism.

In order to facilitate parallel programming, we propose a new extension of the imperative language C, called SELSYN-C. We have selected the programming language C as the basis of our work because it is familiar to a wide range of programmers, and thus it provides a large user base for our new language. The language extensions provide a simple notation for specifying parallel functions and is particularly suited to recursive divide-and-conquer parallelism.

To control parallelism and support load-balancing, a new run-time mechanism has been developed. The fundamental part of this mechanism is the processor-team model which can reduce scheduling overhead as compared to processor acquisition as is required for the traditional fork-join model. Different strategies for organizing the processor-team have been studied. A new team-cooperation scheme is adopted to further improve the processor-team model. This mechanism controls the parallelism and rebalances the work distribution between processor-teams so that high utilization of the processor resources can be achieved.

As demonstrated by some examples given later in this thesis, this approach provides a simple way to specify parallel programs, and an effective way to produce efficient programs that run on a shared memory parallel processor. Currently, the compiler and the run-time environment have been implemented on the shared memory machine, the BBN Butterfly GP-1000.

1.3 Thesis Organization

The thesis is organized as follows. First, we introduce an extension of the imperative language C, called SELSYN-C, and we illustrate how a programmer expresses parallelism and shared data storage. In Chapter 3, we illustrate the run-time mechanisms which can support the language extensions in an efficient manner. Several experiments have been carried out so that we could determine which strategy achieves the best performance. The experimental results are presented and compared in Chapter 4. Chapter 5 describes more implementation details about both the language extensions and the run-time mechanism. The comparison with the other related work is discussed in Chapter 6. We end with conclusions and a discussion of further work in Chapter 7.

Chapter 2

SELSYN-C Language Definition

In this chapter, we illustrate our parallel extension of the imperative language C, which we call SELSYN-C. As outlined in the introduction, one of our major goals was to design a language that is simple to use and familiar to a wide range of programmers. With this goal in mind, we decided to focus on designing simple extensions to the programming language C. These extensions were selected to be easy to use, but at the same time provide the necessary high-level information required for an effective translation to a real parallel processor. To keep these extensions portable for different target machines, our compiler performs a source-to-source transformation. Currently we have implemented the SELSYN system on the BBN Butterfly GP-1000. With this implementation the programmer specifies his or her parallel program using SELSYN-C, and our compiler produces an output program in a C dialect that is specific to the BBN Butterfly. This output program is then linked with our run-time environment to produce a program that can be executed directly on the BBN Butterfly GP-1000.

SELSYN-C supports two major extensions to C: (1) the distinction between processor-private and globally-shared data, and (2) the introduction of weighted parallel function calls. We introduce these two extensions by presenting a simple example. Following the example, a more detailed description of the extensions is presented.

2.1 An Introductory Example

Before we start to introduce our new language extension, we present an example which is written in SELSYN-C. A typical recursive function is given in Figure 2.1. The function, `sum()`, is a typical divide-and-conquer type problem. In this case the problem of summing all entries `a[l..r]` is divided into two smaller problems, summing the entries `a[l..midpoint]` and summing the entries `a[midpoint+1..r]`. Since the summing of the two halves are independent of each other, the two recursive calls to `sum()` are performed in parallel.

```
#define MAX 1000
shared int a[MAX];
main()
{ int  final_sum;
  ...
  sum(a, 0, MAX, &final_sum);
  ...
}

/* sum all entries a[l .. r], put in result */
sum(a,l,r,result)
int a[],l,r,*result;

{ shared int sum_left, sum_right;

  /* if only one entry left, then that is the sum */
  if (l == r)
    *result = a[l];
  else
    { /* sum left half and right half in parallel,
                        result is sum of left and right */
      int midpoint;
      midpoint = (l + r) / 2;
      sum(a,l,midpoint, &sum_left) // sum(a,midpoint+1,r,&sum_right);
      *result = sum_left + sum_right;
    }
}
```

Figure 2.1: An example SELSYN-C program

Two points need to be pointed out. A parallel function call has been introduced in this example,

```
sum(a,l,midpoint,&sum_left) // sum(a,midpoint+1,r,&sum_right).
```

This parallel function call indicates that the function `sum()` is called in parallel on two independent data sets. The other new concept introduced in this example is a new storage type. A new leading keyword, `shared`, is adopted to distinguish the shared data type and the ordinary data type. The definition of our language extension is outlined in the next sections. Along with the definition, more explanation on this example is given.

2.2 Processor-Private vs. Globally-Shared Data

As presented in the previous example, a new shared storage concept has been introduced to our parallel extension to imperative language C. In a parallel system, it is often important to distinguish between memory local to one processor and memory accessible to all processors. Our target machine, the BBN Butterfly GP-1000, is a typical parallel system of this model. As shown in Figure 2.2, it is a loosely coupled system in which each processor is paired with its own memory module. All processor cards are identical and independent. They are connected together by a network interconnection system, called the *Butterfly Switch*, which handles the data transfers from one memory module to another.

With an architecture like that of the BBN Butterfly, a processor may access a memory location associated with another processor, but at a penalty associated with the time for communication through the switch. Thus, we can abstractly define two levels of memory as follows:

Processor private memory: As the name suggests, data in processor private memory can be accessed only from the processor associated with that memory. In general, this is more efficient than accessing globally-shared memory.

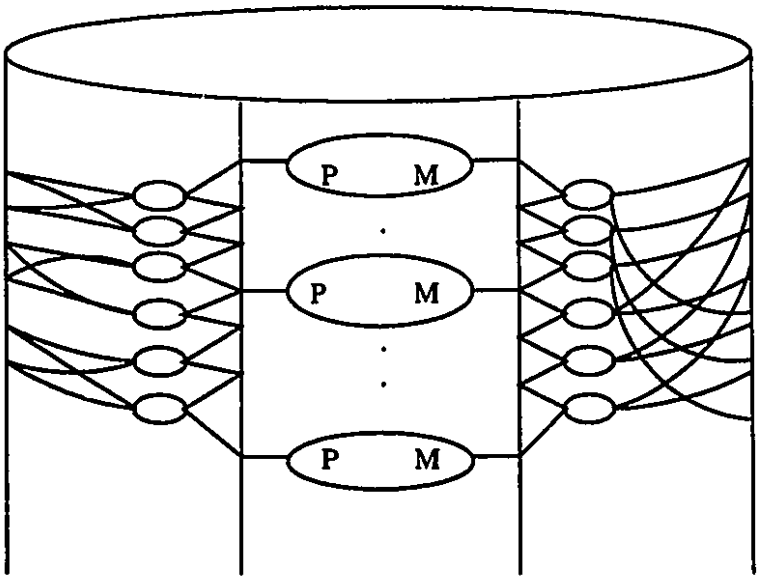

Figure 2.2: Architecture of the BBN parallel processor

Globally shared memory: Data in globally shared memory is accessible from all processors. Accesses to globally-shared memory are usually associated with a penalty for communication costs, and are therefore less efficient than accesses to processor-private memory.

We can abstractly view this memory hierarchy as shown in Figure 2.3. Note that each processor has an address space for its own processor-private storage, while the shared storage provides an address space accessible to all processors.

In order to model the distinction between processor-private and globally-shared memory in our language, we have included the concept of **shared** variables and **shared** dynamically-allocated memory. We discuss these two concepts in the following sections.

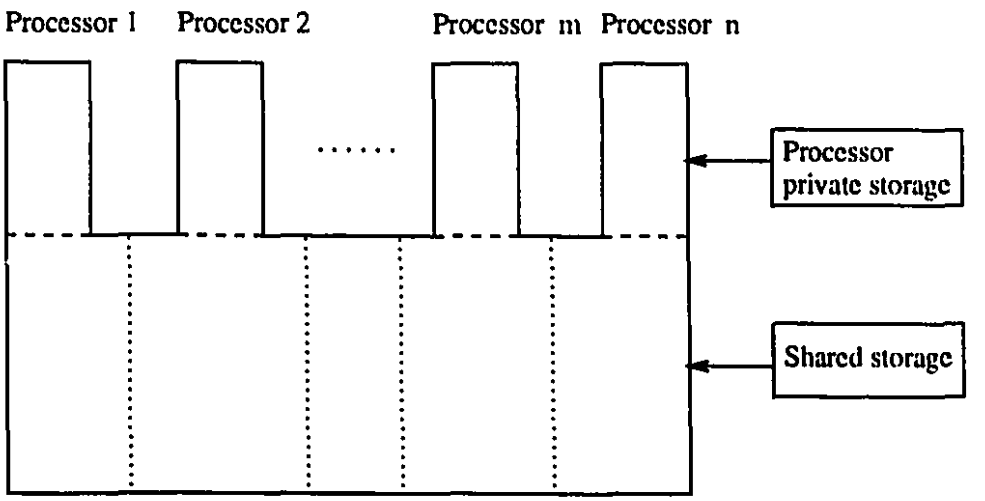

Figure 2.3: An abstract view of Processor-Private and Shared memory

2.3 SELSYN-C Variable Declarations

One main difference between conventional C and our SELSYN-C is that SELSYN-C distinguishes the original variables into to two classes, **shared variables** and **private variables**. In SELSYN-C, all of the conventional variable declarations in C fall into the private variable class. All private variables are permitted to be used as in ordinary C programs, except for the restriction that their addresses (pointers to the variables) should not be passed as arguments to parallel function calls.

2.3.1 Declaring Shared Variables in SELSYN-C

In SELSYN-C, shared variables are declared by adding a new leading keyword, **'shared'**, before the conventional declaration. This keyword indicates that the variables will be stored in shared storage address space as presented in the abstract model

in Figure 2.3. These variables are accessible to any processor, and thus their addresses may be accessed by any function, including parallel functions.

Some declarations of shared variables are,

```
shared int i;
shared float j[10];
shared int *pointer;
```

which define one integer `i`, an array of 10 floats and an integer pointer can be accessed by any processor within the cluster[1] during their lifetime. The `shared` specifier fills in the same syntactic class as other storage class specifier like `extern`. The `shared` specifier may be used for global and local variable declarations, but not with parameter declarations.

Within the two memory management classes, processor-private memory and globally shared memory, several different types of storage are available to SELSYN-C programs. In conventional C, according to the storage types, the variables can be classified into local variables, global variables and dynamic storage variables. For each of these classes, SELSYN-C provides a processor-private type and a globally-shared type. Thus, the storage types of SELSYN are:

Private local variables: Local private variables are processor private and are stored on the stack. A private local variable is visible only within the scope of the function or block that declares it. There is one instance of the variable for every function call. Hence, the variable is private to the function call, and hidden from every other call. Since private local variables are private to one processor, the addresses of these variables should not be used as an argument to any parallel function call[2] in SELSYN-C. In the example, Figure 2.1, `int final_sum` of the function `main()` and `int midpoint` of the function `sum()` belong to this variable class.

[1] A collection of processors [Sen88]. Here it means the set of processors which are assigned to execute the program.

[2] We describe the parallel function call in section 2.4. The general idea behind this restriction is that a parallel function call may lead to the function being executed on a different processor. Thus, all addresses that are passed to parallel functions should be accessible by all processors.

Shared local variables: Shared local variables are accessible from all processors and stored in the shared memory part. However, note that a shared local variable is visible only within the scope of the function or block that declares it, and it has a lifetime associated with the function or block where it is declared. In the SELSYN system, addresses of the shared local variables can be used as arguments in any function calls. Two shared local variables, `shared int sum_left, sum_right`, can be found in the example, Figure 2.1.

Private global variables: Private globals are processor private. There is one instance of each variable per processor. These variables are visible to any function called on the same processor, but are not accessible from any other processor. The addresses of the private global variables should not used as arguments in parallel function calls.

Shared global variables: Shared globals are globally visible by all functions on all processors and are stored in the shared memory part. The addresses of shared global variables can be used as arguments to any function call. The variable, `shared int a[MAX]`, is a typical shared global variable in the example, Figure 2.1.

We now give an example to explain private global variables. All the private global variables will only be visible on its own processor. However, when a program starts executing, SELSYN propagates their initial values to all of the processors. For instance, if a integer `j` declared as,

```
int j = 3;
main()
{
   ...
}
```

when the program enters the `main()`, every processor within the cluster will have the same value of `j`. After this initial phase, changing the value of a private global variable will not affect the value of the copies on the other processors. In the example

above, this means that changing the value of j will only be effective on the processor where the assignment has taken place. Since the updates are only done on the local copy of the variables, the effect of the updates will depend on how the program is mapped to different processors at run-time. This can lead to non-determinism programs and therefore we recommand that private global variables be used as read-only variables, and that shared global variables should be used for update variables.

2.3.2 Dynamically-Allocated Shared Data

In addition to statically declared variables, one needs to make a distinction between dynamically-allocated memory that is processor-private and dynamically-allocated memory that is globally-shared and thus accessible to all processors. We have supported this distinction by providing two families of allocation functions. The ordinary functions such as `malloc()` allocates processor-private memory, while the function `shared_malloc()` allocates globally-shared memory. Just as in the case of declared variables, pointers to processor-private memory may not be used as arguments to parallel function calls, while pointers to globally-shared memory may be freely used.

All variables accessed by different tasks executing in parallel must be identified as shared variables. All private variables are only visible on their own processor, therefore they cannot be accessed by different tasks executing in parallel.

Private dynamic storage: Storage of this type, obtained by *malloc()* and related routines, is processor private. These variables can be accessed by functions within that processor (provided the necessary pointers have been made available), but are hidden from all other processors. In particular, while you can pass a pointer from one processor to another, if you try to use it within another processor you will get a hardware fault or (worse) access a random chunk of memory in that process.

Shared dynamic storage: Storage of this type is obtained by using the SELSYN System allocator *shared_malloc()*, and is globally shared. Since this storage is

globally shared, the address of shared dynamic storage is valid on all processors and such address can be passed freely among them, i.e., can be used as an argument in any function call.

2.3.3 Overview

We end this section with an overview figure that illustrates the distinction between processor-private and globally-shared data. Figure 2.4 presents this overview that indicates the memory allocation of the different data types. Referring to this Figure 2.4 and Figure 2.3, we can see clearly where the variables will be located. All the private variables are located in the individual processor private storage part. All the shared variables are located in the shared storage part. There are two ways to make shared variables visible to other processors. One is to declare a shared variable as a global variable so that it is always visible to all processors during the whole execution time. The other way is passing the address of a shared variable via a parallel function call. Passing an address of processor private storage to other processors will cause a segmentation fault if it is used on any other processor.

2.4 Parallel Function Calls: Issuing Parallel Tasks

In the SELSYN system, we support fork/join parallelism by introducing a parallel operation which can be used by the programmer to specify that two function calls (tasks) may be executed in parallel. Each task can be associated with a weight which is used by the SELSYN mechanism when it assigns processors to the tasks.

A new operator, parallel-function-call is introduced. It is represented as two slashes, '//'. A typical parallel function call has been presented in the example, Figure 2.1 in section 2.1. In the example, the

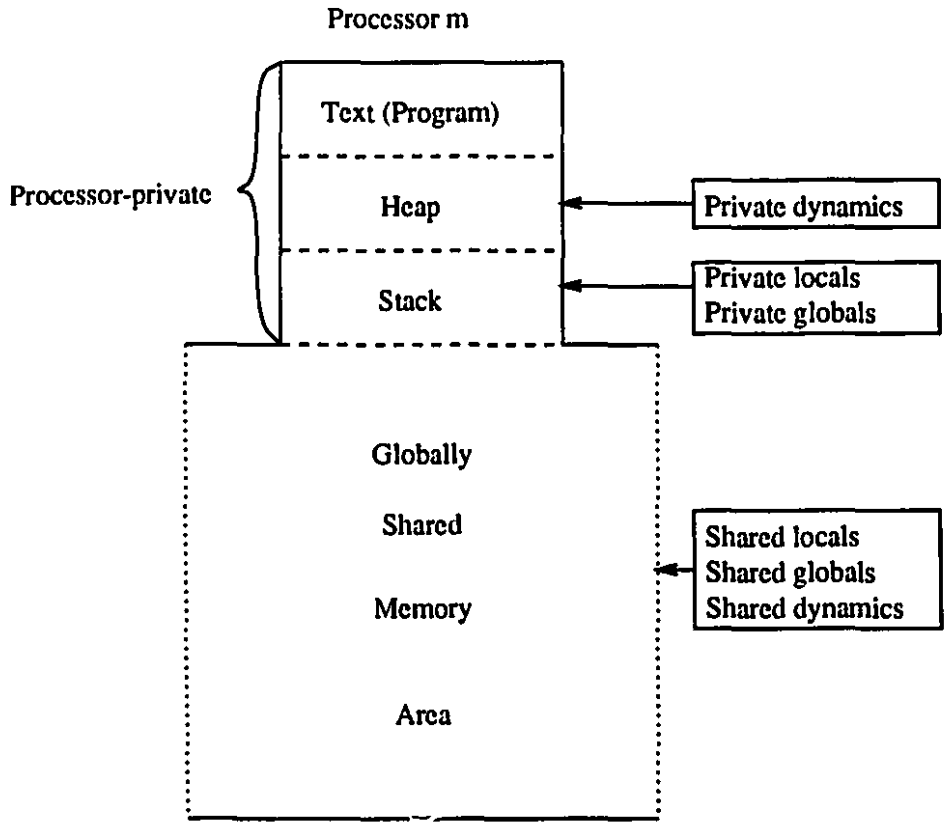

Figure 2.4: A overview of memory allocation

```
sum(a,1,midpoint, &sum_left) // sum(a,midpoint+1,r,&sum_right)
```

is a parallel-function-call. The operator '//' indicates the two function calls

```
sum(a,1,midpoint, &sum_left)
```

and

```
sum(a,midpoint+1,r,&sum_right)
```

can be executed in parallel.

In general, there are the following two forms for parallel function calls:

$$p1(arg_1, arg_2, \ldots, arg_n)//p2(arg_1, arg_2, \ldots, arg_m)$$

$$p1(arg_1, arg_2, \ldots, arg_n)@weight1//p2(arg_1, arg_2, \ldots, arg_m)@weight2$$

The second type of parallel function call allows the programmer to add some information about the relative weights of the two procedure calls. These weights can be fixed constants, or they can be other expressions that are evaluated at run-time to estimate the relative importance(weight) of the two function calls $p1(arg_1, arg_2, \ldots, arg_n)$ and $p2(arg_1, arg_2, \ldots, arg_m)$. The programmer may supply a wide variety of such weight estimates. For example, the weight might be a function of the size of the input to each call, or the weight might be a heuristic that is guiding a search. In both forms of parallel function call, the return type is restricted to void. We give a more concrete example of using weights in Chapter 3.

In term of the C syntax, a parallel function call can occur anywhere that an ordinary procedure call of the form $p1(arg_1, arg_2, \ldots, arg_n)$ may occur as a statement. The weight expression is any primary expression allowable in C.

We can also use the `sum()` example, Figure 2.1, to illustrate the various variable classifications. Note that it is only those variables whose addresses are used in parallel function calls that must be declared as shared, all others are declared as in ordinary C programs. In our example the variables `sum_left` and `sum_right` were declared as shared variables local to the function `sum`. These two variables have a lifetime associated with the lifetime of a particular call to `sum`, and during that lifetime they can be accessed by any processor to which the parallel recursive sub-tasks may be assigned. Also note that the parameter for array `a[]` corresponds to an address and therefore the corresponding argument in a function calling `sum` must be declared as **shared**.

2.5 Summary

SELSYN-C is an extension of the conventional imperative programming language, C. It provides users a simple way to declare the parallelism explicitly. New keywords, constructs and idioms are minimized for this purpose. In the next chapter, the SELSYN mechanism, the run-time environment is introduced. This mechanism acts as

the support backbone of this language extension so that the parallel tasks can be scheduled on target machine efficiently and transparently for the users.

Chapter 3

SELSYN Synchronization Mechanism

In the previous chapter we have outlined our SELSYN-C parallel programming language. However, as we discussed in the introduction, the definition of a parallel language is not enough - we also need a way of effectively mapping the high-level parallel programs onto real parallel machines. In order to solve this problem we have developed the SELSYN scheduling mechanism that is based on processor-team model and a new idea of processor team cooperation. Our SELSYN-C compiler automatically inserts all of the necessary synchronization needed to support this scheduling mechanism, and thus the name of our language, **SELf-SYN**chronizing **C**.

In this chapter we outline the motivation for our solution, introduce the notion of processor teams and a dynamic strategy for dividing teams based on programmer-specified weights, and then we discuss our solution for supporting processor team cooperation. We end this chapter with a summary.

3.1 Motivation

One possible implementation of parallel function calls is to use the traditional fork-join model in which a parallel function call of the form `f1() // f2()` is handled by forking the call `f1()` to a new processor and then joining again after the parallel function call. This traditional fork-join model, however, faces two problems on most shared memory multiprocessors, such as the BBN Butterfly. One is that the acquisition of a processor is a very expensive proposition [BAD87]. If a heavy cost must be paid to acquire a processor, the granularity of the work in a fork-join block must be very large if the code is to run efficiently. The other is that there is no performance advantage in having more runnable tasks than available processors. Instead, this situation represents a performance liability, since the additional tasks imply increased scheduler overhead.

In order to overcome these problems, we introduce a Processor-Team mechanism which has the ability to control the parallelism and the assignment of processors to runnable tasks.

3.2 Processor-Team model

Instead of incurring the large penalty of acquiring processors as the program executes, we have designed a mechanism that reduces processor acquisition costs through the use of processor-teams. When a program starts to execute, it is assigned a team of processors that will be available to handle new tasks as they become available through the execution of parallel function calls. When a parallel function call such as `f1() // f2()` is encountered, the team is divided into two independent sub-teams, one sub-team will execute `f1()` and the other sub-team will execute `f2()`. This dividing procedure can be applied repeatedly as the problem is recursively decomposed until there is only one processor in each team.

Here is a trivial example to illustrate how the Processor-Team model works for our language extension.

```
shared int a, b;
int x, y;
main()
{
  shared int c, g;
  ...
  func_A(&g) // func_B(&c);
  a = g + c;
}
```

When the execution encounters the parallel function call, `func_A() // func_B()`, the current team, which we assume has N processors on hand, is divided into two subteams, each with N/2 processors. The first subteam will execute unction `func_B` and the second team will execute function `func_A`. These two subteams will join together once both of their tasks have been done, and then the statement `a = g + c` will be executed.

If there are parallel function calls within the function `func_A()` or `func_B()`, the same Processor-Team dividing procedure will be applied until there is only one processor left in the current team.

This simple team-dividing strategy[1] can be illustrated as Figure 3.1. In this figure, the numbers enclosed in the symbols "[ ]" represent the team members. At the root node (corresponding to initiating the main program), there is only one team consisting of all processors which will take part in this computation (in this case processors 0 through 7). The main program begins executing on processor 0 and when a parallel function call is encountered, the team is divided into two equal size subteams, one subteam for each parallel function call. This dividing procedure continues for each parallel function call encountered until there is only one processor per team. When there is only one processor in a team there can be no further subdividing and subsequent parallel function calls are executed sequentially. In this manner, the overhead required to start up a new parallel task execution is only incurred when there are processors available for the task.

[1]Similar strategies have been suggested by others including Brooks [Bro89] and Hendren [Hen90]. We give a comparison of our techniques and others in chapter 6.

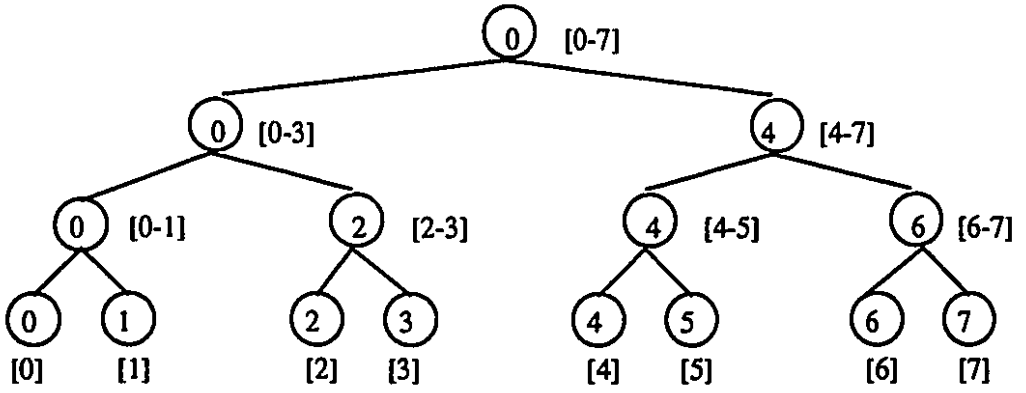

Figure 3.1: Team dividing

Note that in the processor-team model, the parallelism is exploited by rearranging the processors. This is different than the traditional fork-join model where processors are acquired as they are needed. In addition, since the sub-teams are independent and controlled by different processors, the amount of contention and synchronization is reduced. As we discuss in subsequent sections, this team-dividing strategy is a crucial key in achieving the goals of high utilization and efficiency.

3.3 A Weighted Team-Division Strategy

As shown in Figure 3.1, a simple team-division strategy is to divide a team into two equal size subteams. Although this strategy can be efficiently implemented and is applicable to some types of programs, it has an obvious drawback. That is, if the work load of two tasks is uneven, processor resources will be wasted. Thus, as outlined in this subsection, we have developed the dynamic weighted team-division strategy.

Let us illustrate the problem of even team-division and our weighted team-division solution with an example. Consider the following recursive divide-and-conquer function, `foo()`.

```
foo(argn)
{  if  (base_case(argn))
     /* process base case */
     ...
   else
     { /* divide input into 2 components arg1 and arg2 */
       ...
       /* solve the two sub-problems in parallel */
       foo(arg1) // foo(arg2);
     }
   ...
}
```

If the procedure `foo()` always splits the input problem into equal sized pieces such that the time to solve `foo(arg1)` and `foo(arg2)` is approximately the same, then the even-team dividing mechanism is a good choice. However, if the size of `arg1` and `arg2` may vary widely, then the even-team mechanism will result in wasted processor resources. Now recall that our parallel function call mechanism provides the programmer with a way of specifying a weight associated with each function call. Let us assume that for this particular application, we know that the size of `arg1` and the size of `arg2` may be quite different, and we know that the time complexity for `foo()` can be approximated as $O(n^2)$ where n is the size of the input to `foo`.

Clearly, we would like to be able to express this application-specific knowledge in our parallel program, and we would like our team-dividing mechanism to make use of this information. We can encode our knowledge using weights in the parallel function call. For example, we could change the line "`foo(arg1) // foo(arg2)`" in our initial program to the following:

```
{ int s1 = arg_size(arg1),  s2 = arg_size(arg2);
  int weight1 = s1 * s1,    weight2 = s2 * s2;
  foo(arg1)@weight1 // foo(arg2)@weight2;
}
```

Using this weight information available from the programmer, our weighted team-division mechanism calculates the sizes of the sub-teams based on: (1) the number of processors in the original team, and (2) the ratio of weight1 to weight2.[2] As illustrated

[2]If no weights are given, then the division defaults to the simple strategy of dividing the team equally.

by the example above, these weights may be dynamically evaluated at run-time and are provided by the programmer. The weights can be used to express any sort of application specific information. In our example we illustrate one sort of information - the expected time complexity with respect to the size of the input data. Other sorts of weights may be constants, or even complex heuristics.

Comparing the simple evenly-divided team strategy with the dynamic weight-divided team strategy, we can see that the evenly-divided strategy has less overhead (no dynamic computation of weights), while the dynamic strategy may get higher processor utilization and efficiency. We would also like to emphasize that this is an example of how very simple language features can encode application specific information that is very useful to the compiler. In sequential programs there is no processor allocation to be done, and therefore no need to encode information about relative weights of function calls. However, in parallel programs we can make good use of this information which is often easily expressed by the programmer.

3.4 Team-Cooperation

Although the dynamic strategy leads to improved processor utilization, we still need to deal with the problem of unbalanced work load distribution. This situation may occur when the programmer has given no weight information, or when the weight information does not always accurately predict the execution time. Thus, at run-time we may find the situation whereby one team has only one processor and has parallel tasks to be executed, while its sibling team may have one or more processors that are idle with nothing to work on. This situation clearly causes wasted processor resources, and we would like an inexpensive mechanism to help rebalancing the work load when such a situation exists.

3.4.1 Basic Principles

In this section, we introduce a new concept of team-cooperation as a mechanism that further improves the processor-team model. This team-cooperation is used in the situation when one sub-team has processors idle and at the same time the other sub-team has run out of processors and has more parallelism that can be exploited.

In order to explain our team-cooperation mechanism, we introduce the following concepts:

Team: A collection of processors. If there is more than one processor, the team can be divided into two sub-teams that work on independent parallel tasks. For example, in Figure 3.1, at the beginning there is only one team, team [0 - 7], and then this team is subdivided into two sub-teams, team [0 - 3] and team [4 - 7].

Sibling Team: When a team is divided into two sub-teams, each sub-team is the sibling team of the other. For example, in Figure 3.1, the team [0 - 3] and the team [4 - 7] are the sibling teams of each other.

Leading processor: The processor which has the lowest processor number in a team. This is the only processor which can issue the parallel tasks. In Figure 3.1, the number inside the nodes represent the leading processors. If the team has only one processor, the single processor is the leading processor. As shown at the bottom of the Figure 3.1, such 1-processor teams will always be at the leaves of the team-tree.

Task Stack: A stack which keeps the information of the team-dividing and task status. Each team, which has more than one processor, has a task stack.

Task Pool: A pool may contain waiting tasks. It keeps the status of the tasks. Each processor team has a task pool.

As described previously, the basic program execution model is that of team-dividing. This can be summarized as follows. A procedure starts to run sequentially

on the leading processor of its allocated team. When the execution of the procedure encounters a parallel function call, the team is divided into two sub-teams with the leading processors of these two sub-teams continuing on with the individual tasks as specified in the parallel function call. When both tasks have completed, the two sub-teams are joined and execution continues. In the case that a team has only one processor, instead of team-dividing, the parallel function calls will be executed sequentially on the current processor.

By introducing the team-cooperation, we change this basic execution model for the case in which a team has only one processor. In the new model, team-cooperation is carried out between the leading processors of the sibling teams. This cooperation is done for the situation in which a team has only one processor, but has encountered a parallel function call that provides a new parallel task. In the previous simple model, a 1-processor team running on leading processor P simply executes the two parallel tasks sequentially. However, in the team-cooperation model, processor P takes one task to perform itself, and puts a concise description of the other task on its own task pool (if the pool is not full). This task is now available to processor P's sibling team. If the leading processor of P's sibling team, call it P_{sib}, finds itself idle waiting for P to complete, then it will look in P's task pool to see if there is a task to steal. If such a task exists then P_{sib} steals it from P's pool and executes the task. Note that the stolen task may execute new parallel function calls, and if the team associated with P_{sib} has more than 1 processor, then the stolen task will be further sub-divided and executed on sub-teams associated with P_{sib}.

When P finishes its first task, it checks to see if the other task has been "stolen" by its sibling team's leading processor. If the task remains (the sibling team was always busy with its own tasks), P will take back the task and execute it. Otherwise, P can deduce that P_{sib} has stolen the task, and perhaps has generated more subtasks. Thus, P checks the task pool of P_{sib} to see if it can steal a sub-task back.

Thus, we can summarize the team-cooperation mechanism as follows:

Entering a parallel function call `f1()@w1 // f2()@w2`:

CASE 1 - The current team has more than one processor: Use the ratio of the weights to sub-divide the current team into two sub-teams, T_1 and T_2. The leading processor for T_1 executes `f2()` and the leading processor for T_2 executes `f1()`.

CASE 2 - The current team has only one processor, call it P: If there is no room on the task pool, execute functions `f1()` and `f2()` sequentially. Otherwise, put a concise descriptor of the call `f1()` on P's task pool, and execute `f2()`. After `f2()` has completed, check to see if the task for `f1()` has been stolen. If it has not been stolen, then execute `f1()`. If the task has been stolen, then its sibling team may still be working on the task, and P will look in P_{sib}'s task pool to try and steal back a sub-task.

Exiting a parallel function call `f1()@w1 // f2()@w2`:

When the leading processor that was given the task for `f1()` finishes, then it must wait for the team with `f2()` to finish before control can move to the statement following the call `f1()@w1 // f2()@w2`. However, rather than just waiting idly, it performs the Team-Cooperation to reduce the processor stall time. This waiting scheme also has two different cases.

CASE 1 - Team-Waiting: For a team which has more than one processor, when a leading processor has finished a task, it must synchronize with its sibling team before control can be given to the statement following the parallel function call. It checks the task pool of the leading processor of its sibling team for a task that can be stolen. If such a task exists, it steals and executes the task.

CASE 2 - Pool-Waiting: For a team which has only one processor, when the leading processor has finished a task, it must synchronize with the team which stole the task (`f1()` in this example) from its task pool, so that the control can be given to the statement following the parallel function call. It checks the task pool of the leading processor of the team that stole the task. If such a task exists, then it must be a sub-task of the task it is waiting for, so it steals the sub-task back and executes the task.

We can express our divide-and-conquer strategy as illustrated in Figure 3.2. Each triangle represents a task. The `triangle 1` can be divided into two triangles, `triangle 1.1` and `triangle 1.2`. The same division can be applied to `triangle 1.1` and `triangle 1.2` and so on. A stolen task must be a sub-triangle of a larger enclosing triangle. For example, if task `1` is running on a 1-processor team P, then the task `1.1` would be put on P's task pool. P's sibling team may steal task `1.1` and in turn put task `1.1.1` on its task pool. This means that P may steal back task `1.1.1` and so on.

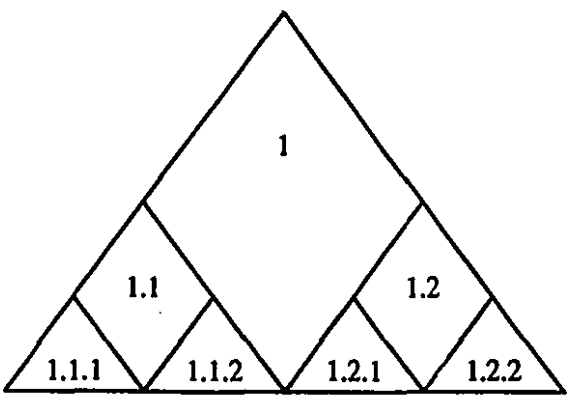

Figure 3.2: Divide-and-conquer call graph

Let's take a trivial example again,

```
shared int a, b;
int x, y;
main()
{
  shared int c, g;
  ...
  func_A(&g)@x // func_B(&c)@y;
  a = g + c;
}
func_B(j)
int *j
{
  ...
  func_C() // func_D();
  ...
```

```
}
...
```

In this example, there is a parallel function call `func_C() // func_D()` in the `func_B()`. Assume the program has a total of 2 processors and the execution time of `func_A()` is less than that of `func_B()`, and the execution time of `func_C()` is less than that of `func_D()`. Figure 3.3 and Figure 3.4 illustrate the execution of the example. In these two figures, `d` indicates that the task is finished, `e` indicates that the task is being executed, and `w` indicates that the task is waiting on task pool.

At first Processor 0 and Processor 1 are assigned as one team, team [0 - 1], to the program. The program runs initially on the leading processor, Processor 0, of this team. The following description summaries the execution of the program starting at the first parallel function call in `main()`.

Step(1) (Figure 3.3): When execution of the program reaches the parallel function call,

```
func_A(&g)@x // func_B(&c)@y,
```

it put these two tasks on the task stack of team [0 - 1]. The team is divided into two sub-teams which are led by Processor 0 and Processor 1 respectively. These two sub-teams run `func_B()` and `func_A()` independently.

Step(2) (Figure 3.3): While team [1] is running `func_A()`, team [0] encounters another parallel function call `func_C() // func_D()`. At this time, there are no more processors to be divided into two sub-teams, and one task (`func_C()`) has to be put on the task pool of team [0] for waiting while the other task `func_D()` is left running on team [0].

Step (3) (Figure 3.3): At this point, team [1] has done its own task and is waiting for joining with its sibling team which still has a task on hand to run.

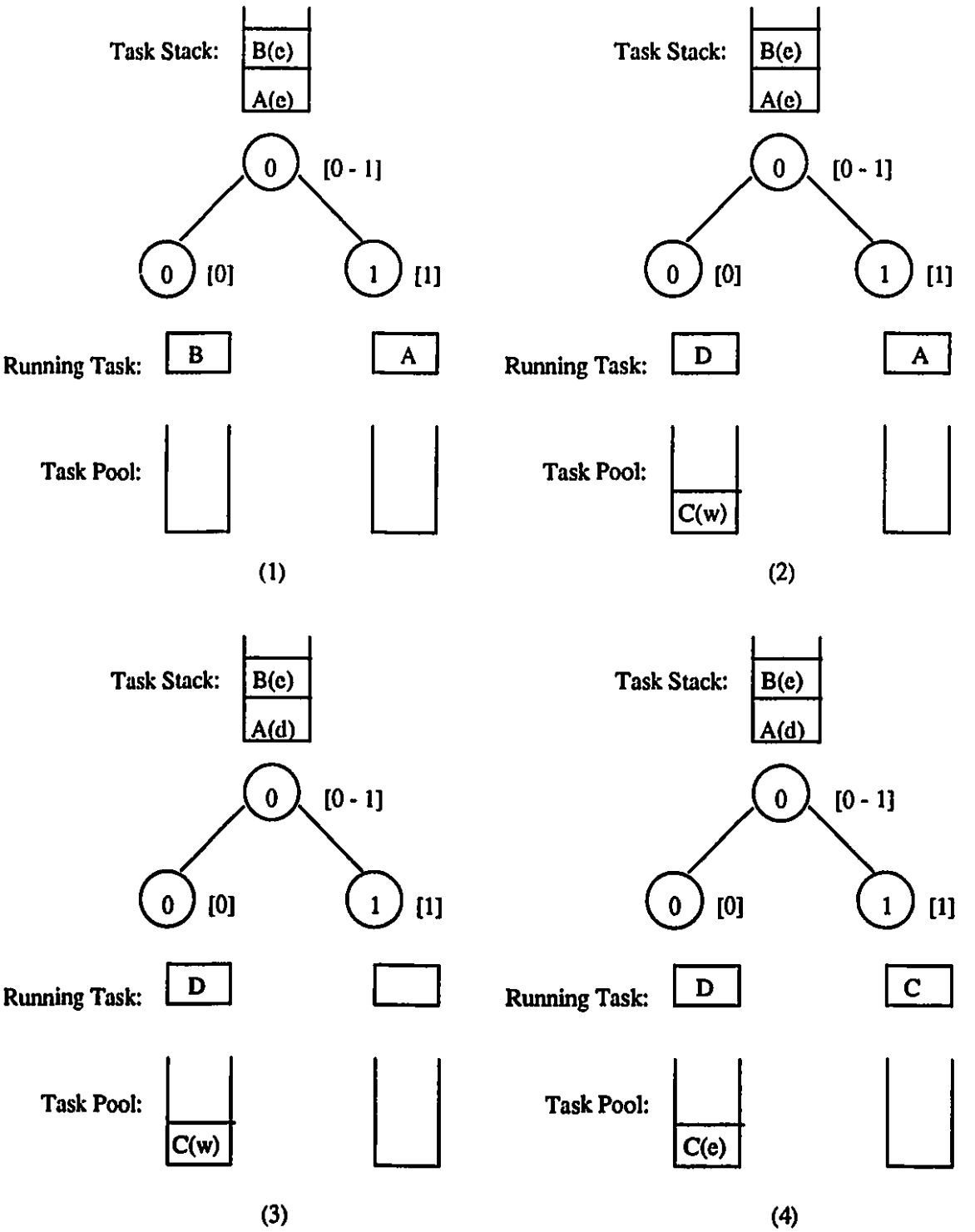

Figure 3.3: Execution of example (Part 1)

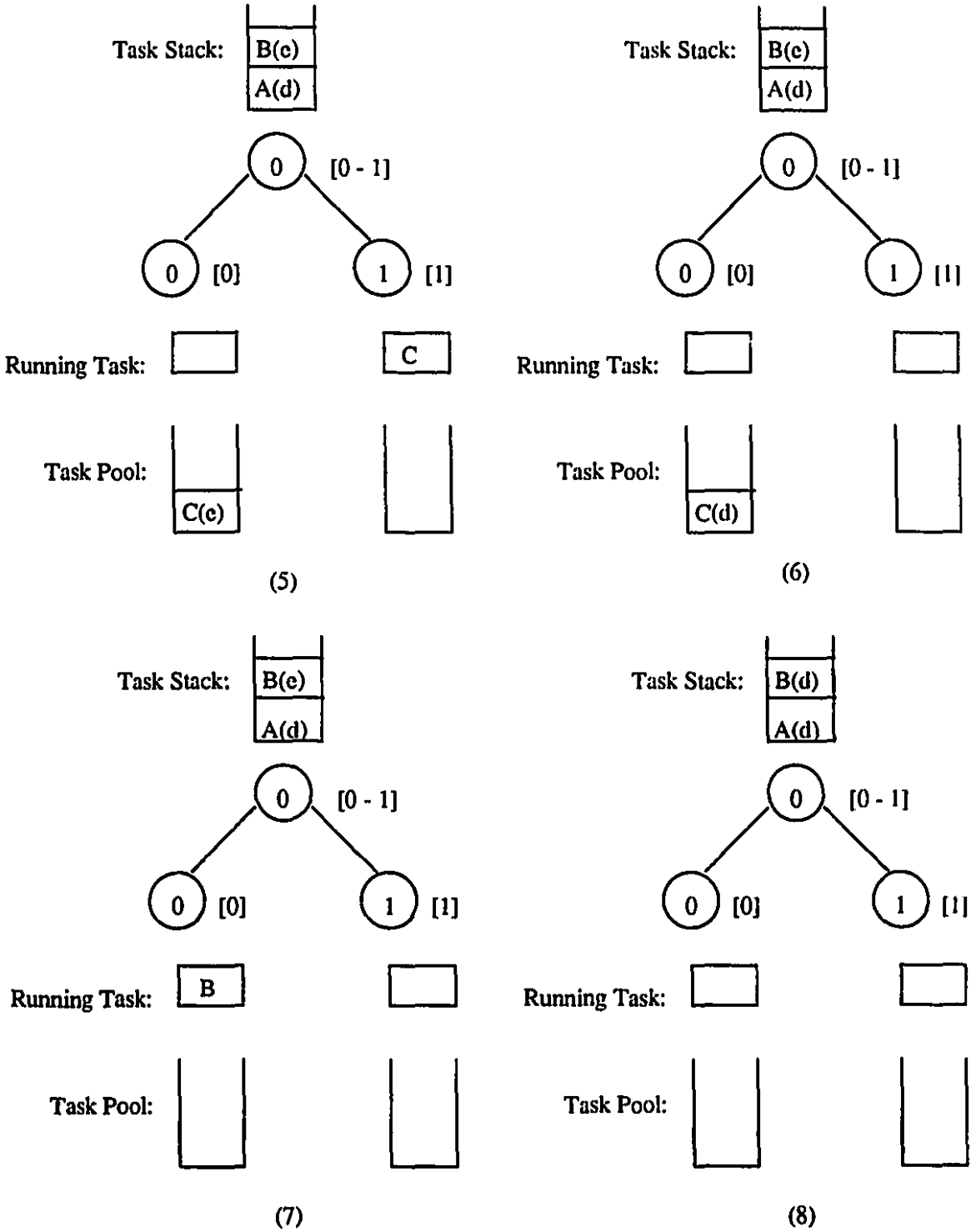

Figure 3.4: Execution of example (Part 2)

Step (4) (Figure 3.3): This figure shows the team-cooperation between these two sibling teams. Team [1] steals the task which is waiting on the task pool of team [0].

Step (5) (Figure 3.4): At this point, team [0] has finished its own task `func_D()` and is waiting for joining with team [1]. There are no waiting tasks on the task pool of team [1].

Step (6) (Figure 3.4): At this point, `func_C()` has been completed by team [1]. Team [1] is waiting for team [0] to join into the original team, team [0 - 1].

Step (7) (Figure 3.4): After finishing the parallel function call

```
func_C() // func_D(),
```

team [0] continues on the execution remaining in `func_B()`. Team [1] is waiting for joining.

Stem (8) (Figure 3.4): Finally both team [0] and team [1] have encountered the join point. A team-join will take place. And after this team-join action, the statement `a = g + c` left in `main()` will be executed on the leading processor of team [0 - 1].

From this example we can see the ability of the team-cooperation to rebalance work loads between teams so that high utilization of processors can be achieved. The following figure, Figure 3.5, presents the situation when the original team has three processors. We can see that the task stack is used when there is a team-dividing in a team, and the task pool is used only for the one processor team. Similar figures can be developed for the situation when a team has more processors. The implementation of the task stack and task pool is discussed in chapter 5. More different cases can be found in Appendix A. The reader can refer to them for detailed case studies on how team-dividing and team-cooperation can work together on these cases.

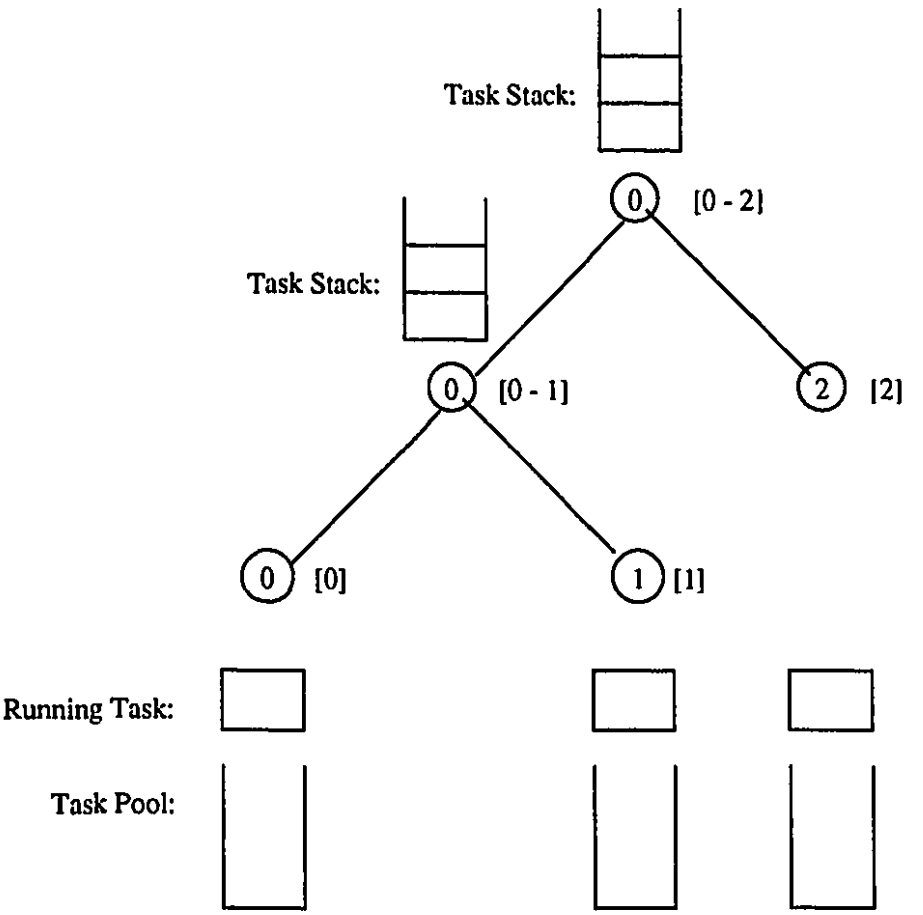

Figure 3.5: A more general example

3.5 Summary

In this chapter we have addressed our solution to the problem of scheduling tasks. In developing our approach we considered the following desirable requirements:

1. processor allocation should be inexpensive,
2. parallelism should be controlled so that scheduling overhead is reduced,
3. processors should work relatively independently in order to reduce contention for locks and minimize synchronization, and
4. a limited form of task redistribution should be incorporated to allow load balancing.

To achieve the first three requirements we introduced the notion of teams of processors and team-dividing strategies that can make use of user-supplied weights. To achieve the fourth requirement we introduced the idea of team cooperation.

We have chosen the Processor-Team model to reduce the overhead of the acquisition of a processor. Different team-dividing strategies have been presented. A team-cooperation has been introduced to enhance the performance of the Processor-Team model. For our SELSYN system, the dynamic team-dividing strategy combining with the team-cooperation was chosen. We call this mechanism the Cooperating-Team mechanism. Accordingly, the static team-dividing is called Even-Team mechanism, and the dynamic team-dividing is called Weighted-Team mechanism. In order to further study these mechanisms, we have done several test experiments using these mechanisms on the BBN Butterfly GP-1000. The test results proved again that Cooperating-Team mechanism has the best performance among the three mechanisms. In the next chapter we present the details of these test results.

Chapter 4

Experimental Results

In this chapter, we present the performance figures for the three scheduling strategies which have been described in the previous chapter. For the purposes of this chapter we consider the following scheduling strategies:

Even-Team: Divide each team into two equal sized sub-teams. When there is only 1 processor in a team, execute parallel function calls sequentially.

Weighted-Team: Divide each team according to the ratio of weights as indicated by the programmer. When there is only 1 processor in a team, execute parallel function calls sequentially.

Cooperating-Team: Divide each team according to weights, assume the weights are equal if no weights are provided. When there is only 1 processor in a team, use the Team-Cooperation mechanism outlined in section 3.4.

Performance figures reported in this chapter were achieved on the BBN Butterfly GP-1000 parallel processor. The test programs include small illustrative programs and larger programs that are more similar to real applications.

4.1 Butterfly Memory Organization

The architecture of the BBN Butterfly parallel processor is a Shared-Memory / Omega-Switch model. As illustrated in Chapter 2 in Figure 2.2, the BBN Butterfly is a loosely coupled system in which each processor is paired with its own memory module. This memory organization makes all memory local but globally accessible through the switch at some penalty in access time. All processor cards are identical and independent and are connected together by a network interconnection system, called the *Butterfly Switch*, which handles the data transfers from one memory module to another. The timing information for the memory access is given in Figure 4.1 [BBN89]. Note that the time for a global access is considerably more than for a local access. Also because of switch contention, this penalty access time may be considerably worse at run time depending on the data traffic between the processors. We will show this effect with some of our results presented in the next sections.

	Global (microseconds)	Local (microseconds)	Global/ Local
Read	8.12	1.22	6.7
Write	3.52	0.81	4.3

Figure 4.1: BBN memory access time

4.2 Experimental Approach

We have selected four different programs as our benchmarks. They vary from larger programs to small programs that were designed to test specific characteristics of the scheduling mechanisms. They include,

1. Parallel quick sort.

2. An illustrative program to do computation on the nodes of a binary tree.

3. An example that computes an approximation for the integral of a function.

4. A 4x4 puzzle search program.

We applied our three mechanisms, which are described in previous sections, to each benchmark. Each mechanism was tested on varying numbers of processors, and each mechanism was tested ten times on every processor configuration to get the average performance. In order to compare the performance of the three mechanisms, we generated the same test cases for each test benchmark. From the experimental results, which are presented in following sections, we can see the different ability of the three mechanisms to deal with a variety of work load distributions.

4.3 Example 1: Quicksort

The first experiment we performed was with parallel quicksort. Quicksort manipulates a list. It first divides the input list into three parts, where the first part contains elements less than the pivot value (called P), the second part contains the elements equal to P, while the elements greater than P fall into the third part. Note that the first part and the third part of this list can be quicksorted independently. Thus, we can simply specify that these two executions may execute in parallel. The basic sequential algorithm can be shown in pseudo-code as follows,

```
quick-sort(head)
{
  if (length(head) > 1) {
     partition(head, &part1, &part2, &part3);
           /* using first element, part3, as pivot */
     quick-sort(part1);
     quick-sort(part3);
     combine-lists(part1, part2, part3);
  }
}
```

Our parallel version will be

```
quick-sort(head)
{
  if (length(head) > 1) {
     partition(head, &part1, &part2, &part3);
          /* using first element, part3, as pivot */
     quick-sort(part1)@(length(part1))
       // quick-sort(part3)@(length(part3));
          /* sort part 1 and part 3 in parallel */

     combine-lists(part1, part2, part3);
  }
}
```

In terms of testing our scheduling mechanisms, the important characteristic of this parallel quicksort is that the work load of these two executions may be quite different. For example, consider the following input list:

```
3 2 1 100 23 13 35 8 9 10 3 12 7 9 30
```

After `partition()`, it will be divided into the following three parts,

Part 1: 2 1

Part 2: 3 3

Part 3: 100 23 13 35 8 9 10 12 7 9 30

Now we can see that part 1 and part 3 have 2 elements and 11 elements that need to be sorted respectively. This unevenness of the partitioning step tests the ability of our mechanisms to deal with an unbalanced work load distribution which is unpredictable at compile-time.

In Figure 4.2, we present the result of sorting 8192 elements. The x-axis represents the number of processors, while the y-axis represents execution time in machine ticks (one tick equals to 62.5 microseconds[BBN89].). The dotted line represents the performance of the Even-Team mechanism, the dashed line represents the Weighted-Team mechanism and the stippled line represents the Cooperating-Team mechanism. As we have described in previous sections, the Weighted-Team and Cooperating-Team mechanisms have weights associated with each parallel task. In this experiment, for the Weighted-Team and Cooperating-Team mechanisms, the weight of the task is chosen as the number of the elements it will sort.

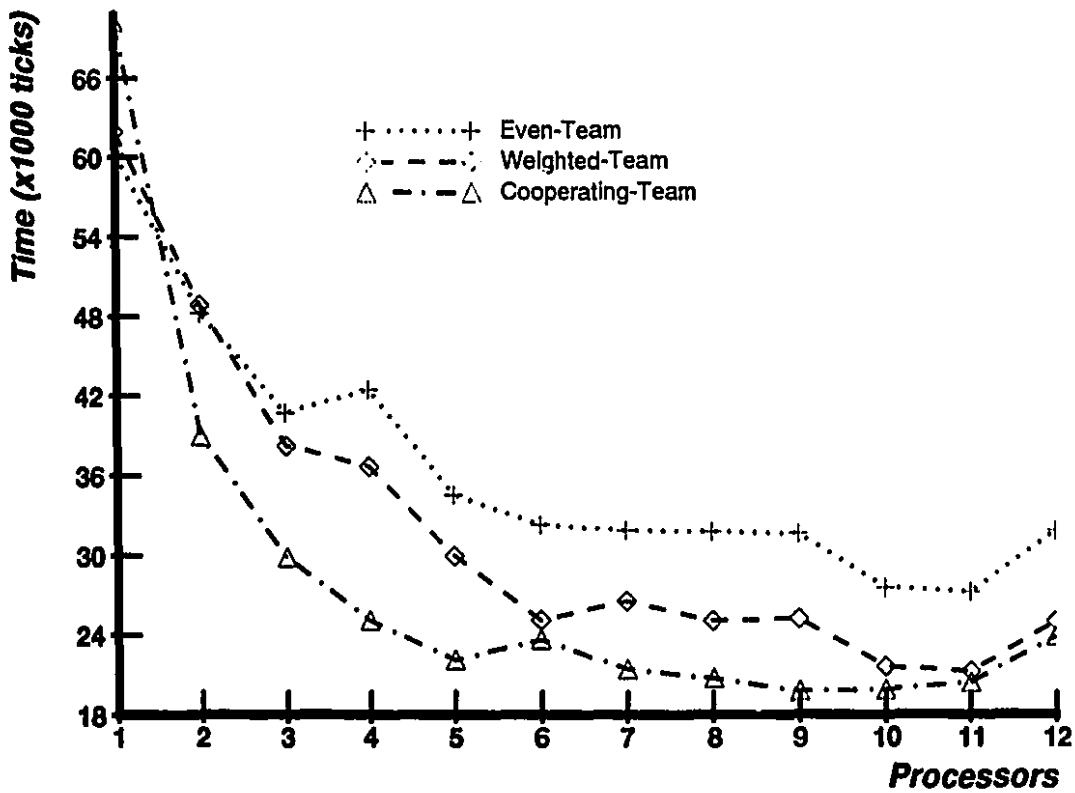

Figure 4.2: Quicksort: 8192 elements (10 cases)

Comparing the three performance lines, we can see that if the number of processors is greater or equal to 2, the Cooperating-Team gets the best performance. When there are more than 2 processors, the Weighted-Team is in the second position and the Even-Team is the worst. Figure 4.3 summarizes the relative performance among the three mechanisms running on six processors.

	Time (x10^3ticks)	Speedup (compared to Even-Team)
Even-Team	32	1.00
Weighted-Team	25	1.28
Cooperating-Team	23	1.39

Figure 4.3: Relative performance on six processors

Running the Even-Team on one processor is equivalent to running the sequential program. That is, the Even-Team mechanism immediately resorts to the sequential computation when only one processor is available. However, the Weighted-Team and the Cooperating-Team mechanisms have some amount of overhead, even if only one processor is available. From the three performances for the one processor case, we can determine the overhead of the Weighted-Team and the Cooperating-Team mechanisms by comparing them with the Even-Team case. Even though the overhead is significant, we can see the benefits of the more expensive mechanism for all cases with more than one processor. Even the worst case, on 11 processors, the Cooperating-Team still can get a 1.3 speedup when compared to the Even-Team.

In the previous quicksort experiment, we used the number of elements to be sorted to predict the work load. In order to test the effectiveness of the Cooperating-Team mechanism for the cases where the workload cannot be predicted, we experimented with setting the weights of the two parallel calls to be equal. This experiment is shown as bold line in Figure 4.4. Let us first compare the Cooperating-Team mechanism with predicted workloads, the stippled line, and the Cooperating-Team mechanism with workloads artificially set to be equal, the bold line. As expected, the stippled line shows better performance than the bold line. However, we can see that the Cooperating-Team mechanism adapts to wrongly predicted workloads and outperforms the Even-Team mechanism in all cases by comparing the bold line with the dotted line. For the two to six processors cases it even outperforms the Weighted-Team mechanism that has correctly predicted workloads.

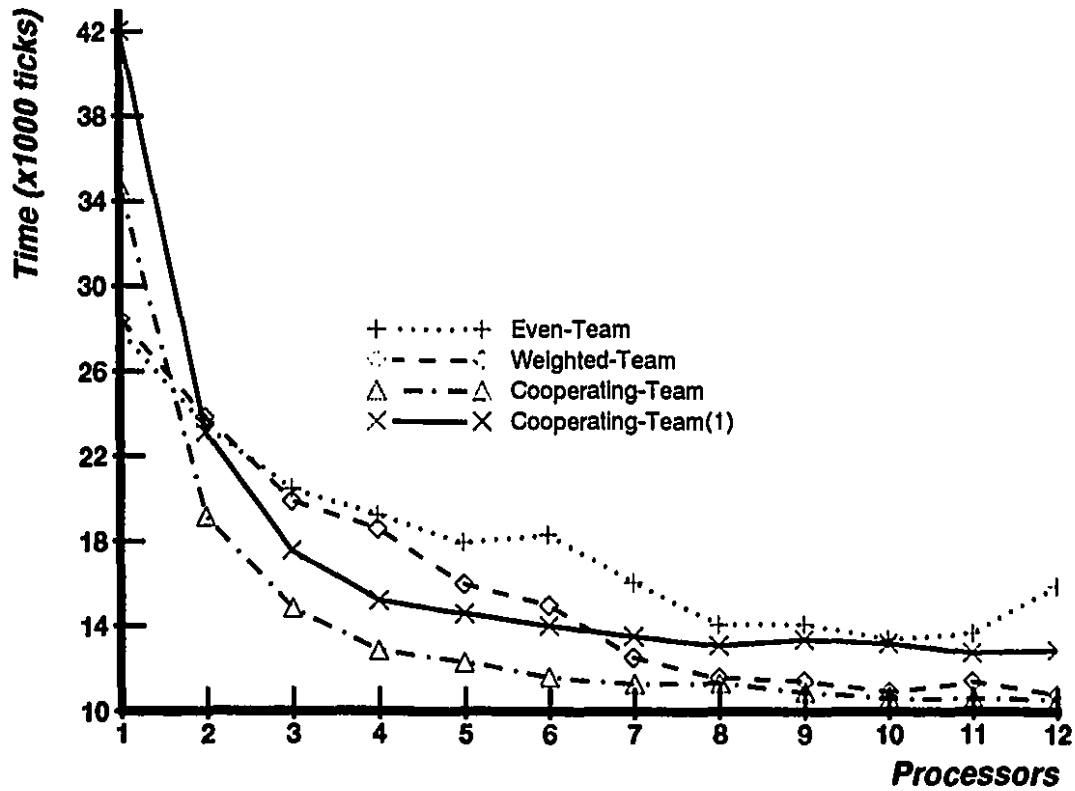

Figure 4.4: Quicksort: 4096 elements (10 cases)

In order to get more precise test results for these mechanisms, in some cases we repeated the experiment more than 10 times. Figure 4.5 presents the average results which are obtained by testing 100 random lists, where each list has 4096 elements. In this figure, the curve Cooperating-Team(1), the bold line, represents the test when we forced the weights to be equal for the Cooperating-Team mechanism. Comparing this Figure 4.4 to Figure 4.5, we can see the similar shape of these performance curves. This demonstrates the stability of these mechanisms.

The results of these experiments show that the Cooperating-Team has its best relative performance when the number of processors is around four. It also shows that the Cooperating-Team mechanism does have the ability to balance the work load among processors so that the total execution time can be reduced. Figure 4.6 shows the relative performance on four processors derived from Figure 4.5.

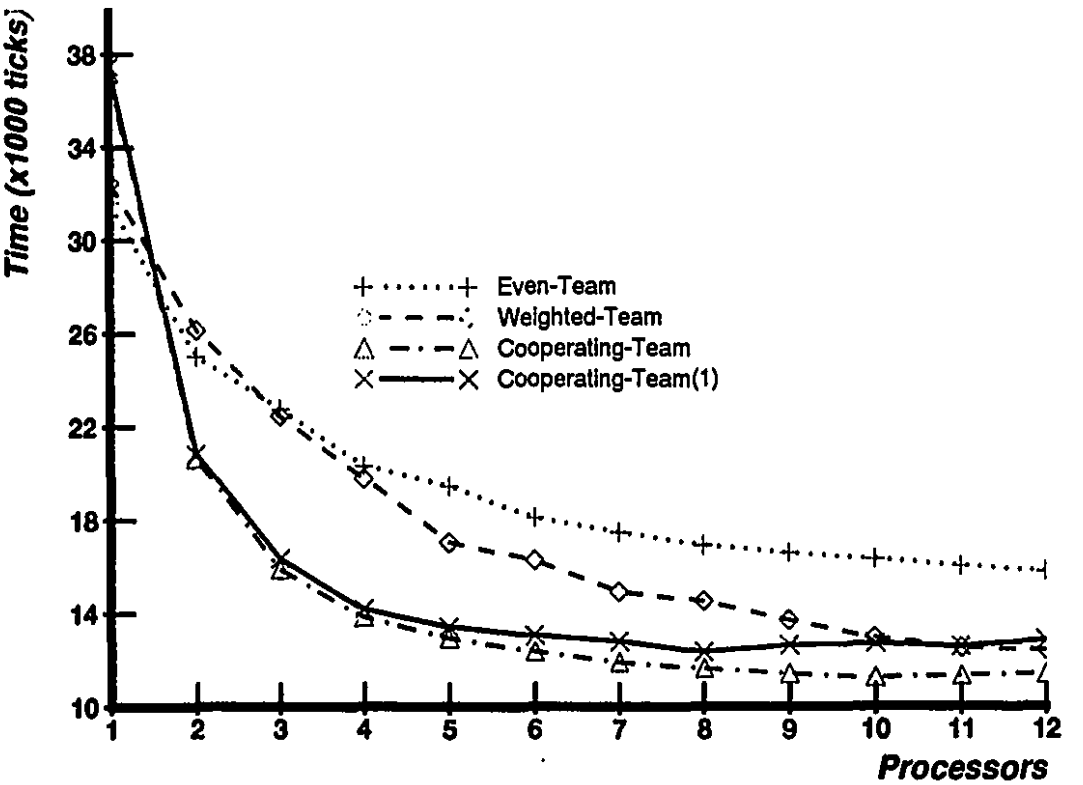

Figure 4.5: Quicksort: 4096 elements (100 cases)

	Time (x10^3ticks)	Speedup (compared to Even-Team)
Even-Team	21	1.00
Weighted-Team	19	1.11
Cooperating-Team	14	1.50
Cooperating-Team(1)	16	1.31

Figure 4.6: Relative performance on four processors

4.4 Example 2: Binary Tree Evaluation

The test results presented in this section are achieved by evaluating random binary expression trees, as shown in Figure 4.7. The leaf nodes contain random numbers and the internal nodes contain the operation codes and space to store intermediate results. In addition each node contains the size of its subtree.

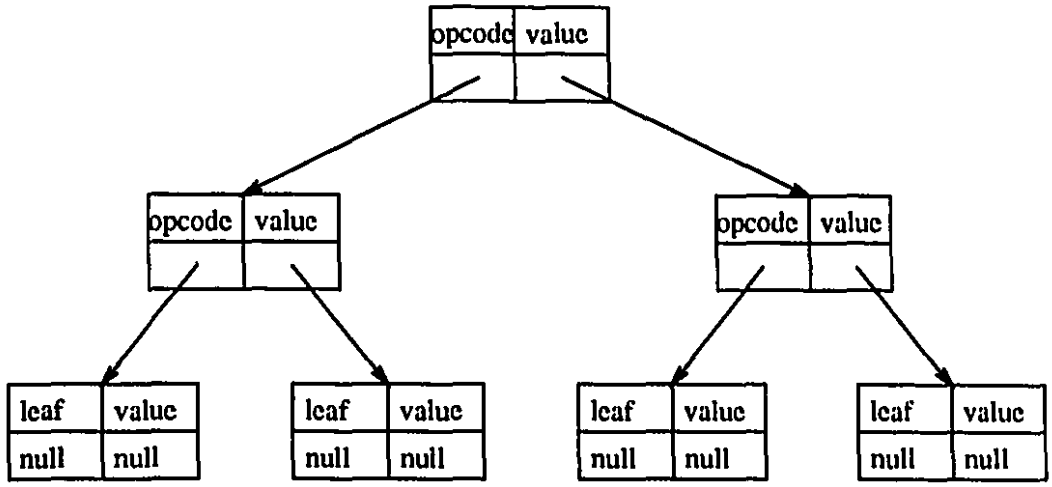

Figure 4.7: A sample binary expression tree

The program recursively evaluates through this tree and returns the value calculated to the root. The parallel pseudo-code is as follows,

```
eval(root)
{
  if (root != leaf) {
    eval(root->leftchild)@root->leftchild.size
     // eval(root->rightchild)@root->rightchild.size;
    ...
     /* according to the root->opcode, perform the operation
        on root->left->value and root->right->value
        and store in root->value */
  }
  return;
}
```

Figure 4.8 shows the result on a randomly generated tree with 20000 nodes. In this experiment, the weight chosen for the Weighted-Team and Cooperating-Team mechanisms is according to the number of nodes in the sub-tree being evaluated. Clearly, the Cooperating-Team and Weighted-Team take the advantage of this weight

information. In every case for two or more processors, the Weighted-Team mechanism outperforms the Even-Team mechanism. By examing the results for the one processor case, we can observe the overhead for the Cooperating-Team mechanism. Although this overhead is large, the Cooperating-Team outperforms both the Even-Team and the Weighted-Team once it gets more than one processor and it shows very strong rebalancing potential on the four processor case. For this case, the Cooperating-Team shows a 1.93 speedup over the Even-Team and a 1.50 speedup over the Weighted-Team.

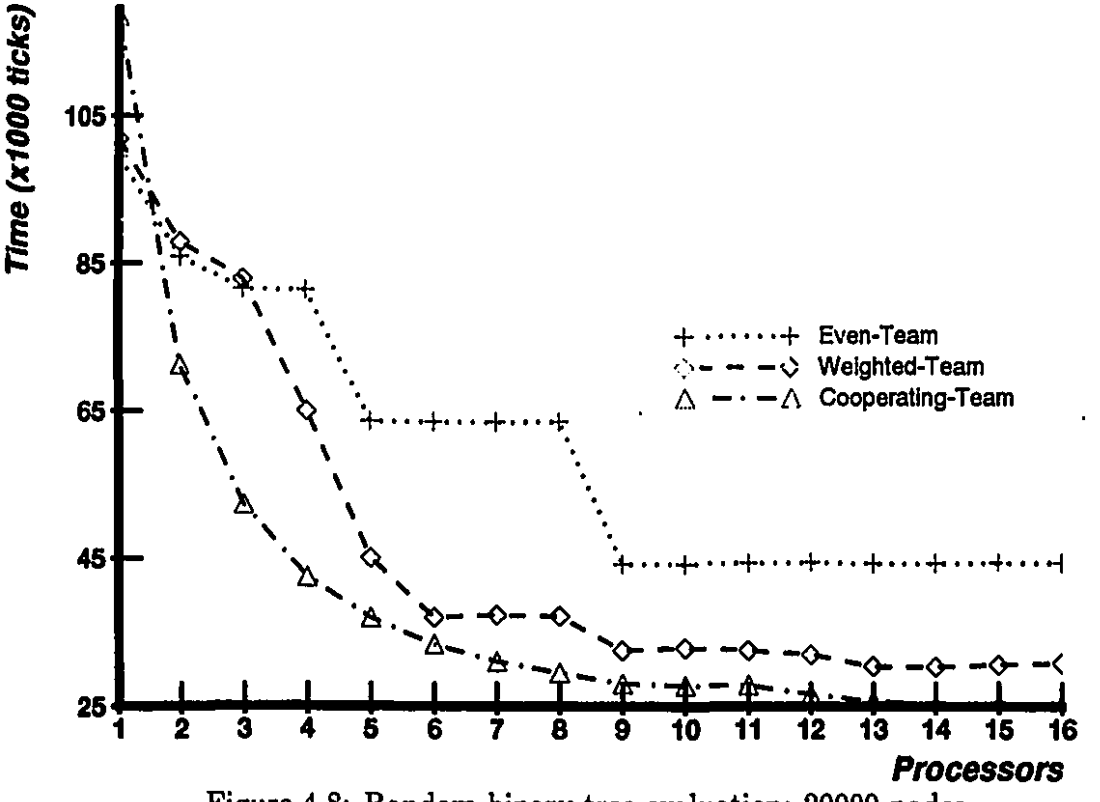

Figure 4.8: Random binary tree evaluation: 20000 nodes

The tree the program evaluates in the previous experiment is a random tree and therefore not balanced. What will the performance be if it is a balanced tree? In the other words, how do our mechanisms face a balanced work load? To answer this question, we tested our mechanism on a balanced binary tree which also has 20000 nodes. The performance results for this balanced tree test are presented in Figure 4.9. For this experiment, every mechanism gets the precise information for team dividing.

Since it is a balanced tree, the Weighted-Team mechanism loses its advantage. The performance of the Even-Team and the Weighted-Team mechanism are almost the same. We can hardly distinguish them in the Figure 4.9. The Cooperating-Team also lost its advantage of rebalancing ability. The performance shape of Cooperating-Team is very similar to the other two mechanisms' performance shapes and the small gap between them represents the overhead of the Cooperating-Team mechanism.

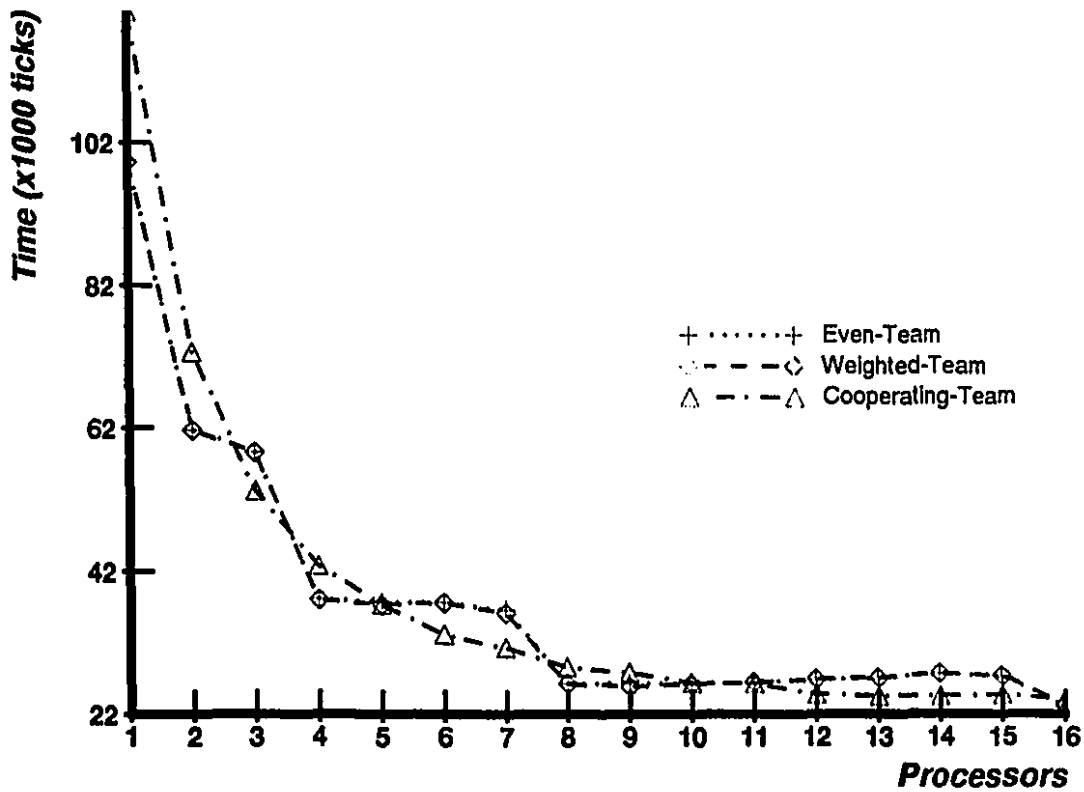

Figure 4.9: Balanced binary tree evaluation: 20000 nodes

Although in the previous experiments the tree sizes are fairly large, the computation load of each task is rather small. In the next experiment, we put a delay loop at each evaluation step so that we can examine the performance for a heavier computation load. The result of this test is given in Figure 4.10. For this experiment, we can see the shape of the figure is still as we would expect. However, some advantages of the Cooperating-Team mechanism becomes more obvious. In the previous binary experiments, we gave the times only for 1, 2, 4, 8 and 16 processors. In this experiment, we have data points for all processor numbers from 1 processor to 28

processors. As we can see, the stippled line, which represents the Cooperating-Team performance, declines smoothly as the number of processors is increased. However the Even-Team has a step-like curve which changes radically when we double the number of processors. Thus in many cases, the Even-Team can not get any performance improvement from adding more processors. This is because the Even-Team mechanism always divides the team into equal sub-teams. But if the size of a team is not an even number, one sub-team will be smaller and since there is no rebalancing ability, the speed is limited by the speed of the speed of the smaller team. Thus because of the unbalanced work load, the Even-Team can not rebalance the distribution even though it does get more free processors on hand. However, because of the rebalancing ability, the Cooperating-Team mechanism always gets performance improvement when it gets more processors. The Weighted-Team, on the other hand, can get improved performance in most cases because of the accurate work load prediction. But it gets slightly worse performance once there are more than 14 processors take part in the computation. That is because that the memory contention becomes the major factor affecting the performance when the number of processors is increased to certain extent. By contrast, the Cooperating-Team has the rebalancing ability to distribute the work when memory contention causes unbalanced work load.

We have presented the memory accessing information in Figure 4.1 in the previous section. To reduce the effect of memory access contention, in previous test experiments, we allocated the data on all processors of the system, even though the experimental program may not use all of these processors. In next two binary tree evaluation experiments, we restrict the data allocation to the processors which the tests will use. For example, when testing the program on 2 processors, half the tree nodes will be allocated on processor 1 and half on processor 2. In Figure 4.11, a very obvious phenomenon that can be noticed is that the Weighted-Team and the Even-Team mechanism get worse performance on two processors than on one processor. This degradation in performance occurs because the 2-processor case has non-local memory accesses while the 1-processor case makes only local memory access. The other factor that causes this degradation is the processor resource waste. Because

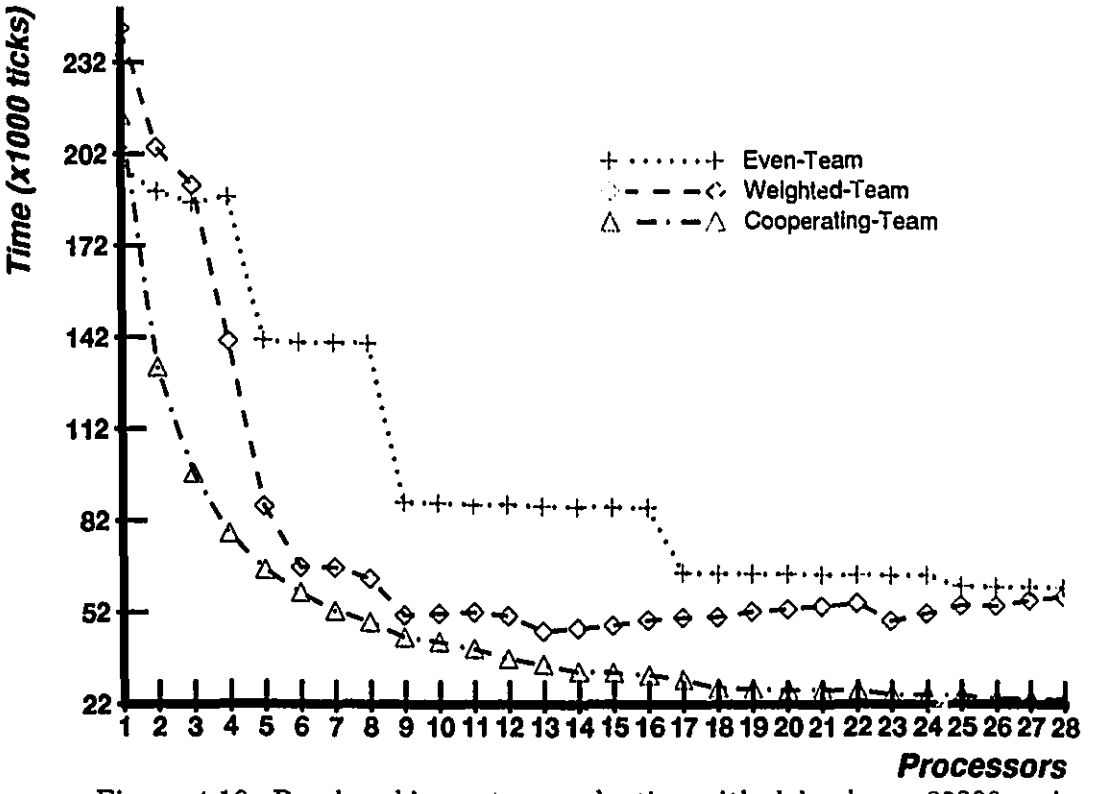

Figure 4.10: Random binary tree evaluation with delay loop: 20000 nodes

of the uneven work load distribution, the Even-Team and Weighted-Team mechanisms can waste processor resources on the 2-processor case. Thus, the parallelism on 2-processors is not enough to offset memory access contention between processors and the access penalty of the remote memory access. By contrast, because of the rebalancing ability of the Cooperating-Team mechanism, it still can get better performance on 2 processors compared to its performance on 1 processor. Comparing to Figure 4.8, we can find that in the one processor case, the performances in the Figure 4.8 are worse than that in the Figure 4.11. This is because in the former experiment, the data is scattered on all of the processors of the system so that there are more remote memory accesses. On the other hand, as we add more processors, the memory contention becomes the major factor affecting the performance. We can see once there are more than four processors, the performances in the Figure 4.8 become better than that in the Figure 4.11. A similar situation also happens for the balanced binary tree evaluation. We can conclude this by comparing Figure 4.9 and Figure

4.12.

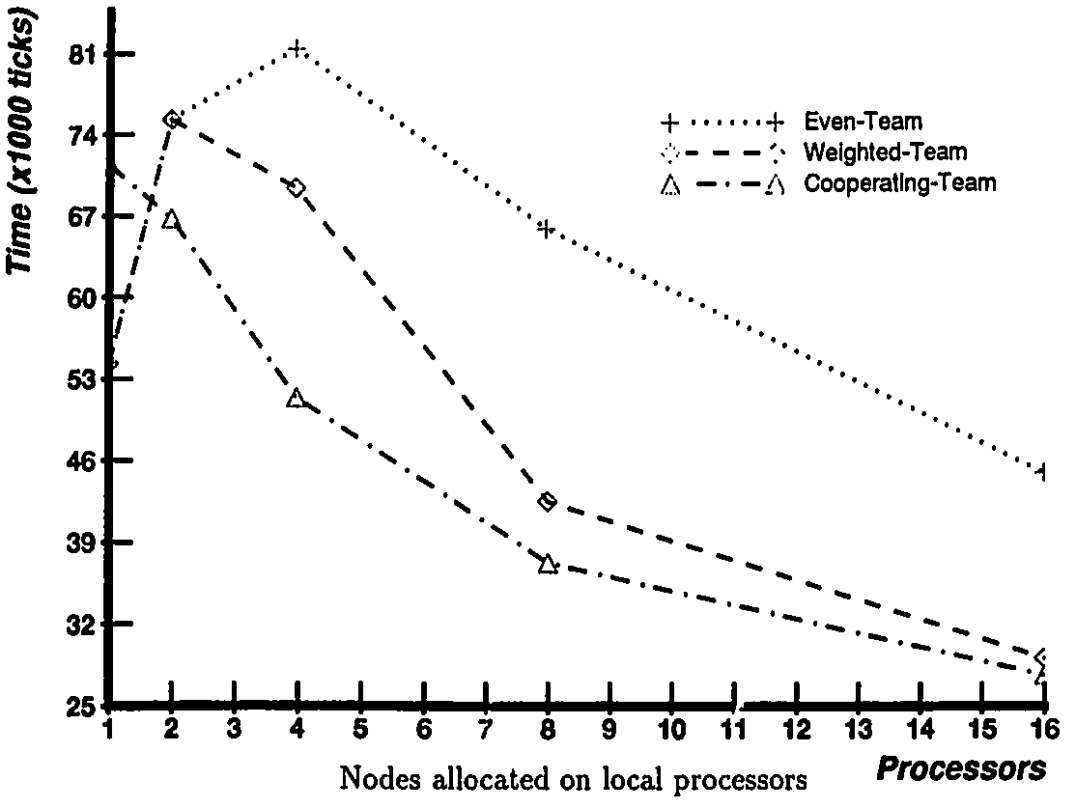

Figure 4.11: Random binary tree evaluation: 20000 nodes

Comparing Figure 4.11 and Figure 4.12, several observations can be made. The first one is for the 2-processor case. In both figures, the Even-Team mechanism and the Weighted-Team mechanism get worse performances on 2-processor than that on 1-processor. However in Figure 4.11, the performances difference is much more pronounced. This is because the experiment performed in Figure 4.12 was on balanced trees and therefore the workload is well predicted, the Even-Team and the Weighted-Team mechanisms are only affected by remote memory access and memory contention and not affected by wasted processor resources. Secondly, because of the balanced trees, we can find in Figure 4.12 that the performances of all the mechanisms are quite close, and the Cooperating-Team mechanism keeps almost the same shape in both figures.

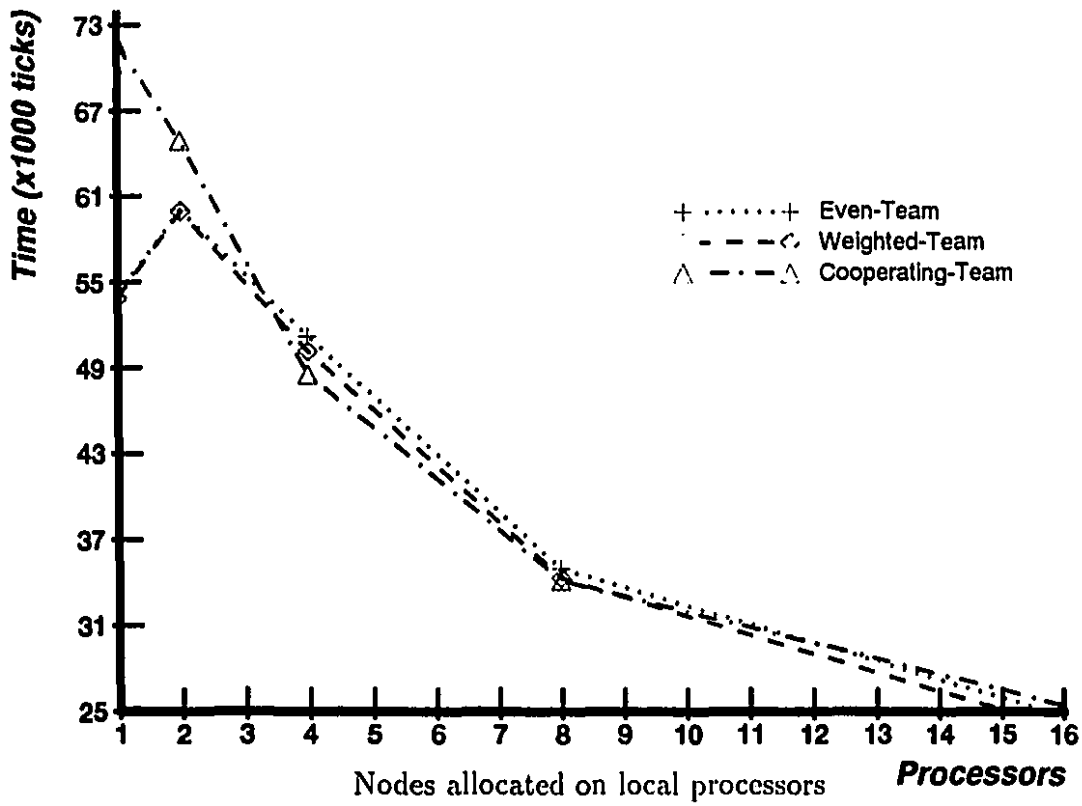

Figure 4.12: Balanced binary tree evaluation: 20000 nodes

To study this situation further, we performed several other experiments to compare performances under different test environments. In order to get the performance of dealing with small work load as well, we chose a small tree size, 1023 nodes in total, to do our experiments. First let's see the performances on a random tree. The performances shown in Figure 4.13 was measured without a delay loop, while Figure 4.14 was measured with a delay loop added in each task.

In Figure 4.13, we can see that because the tree is unbalanced, the Weighted-Team mechanism is better than the Even-Team mechanism. But because of increasing memory contention for cooperation and the small size of the tasks, the performance of the Cooperating-Team mechanism becomes worse than that of the Weighted-Team, if more than 6 processors take part in the execution. In Figure 4.14, we added a delay loop to each task. With this increased workload in each task, we see an

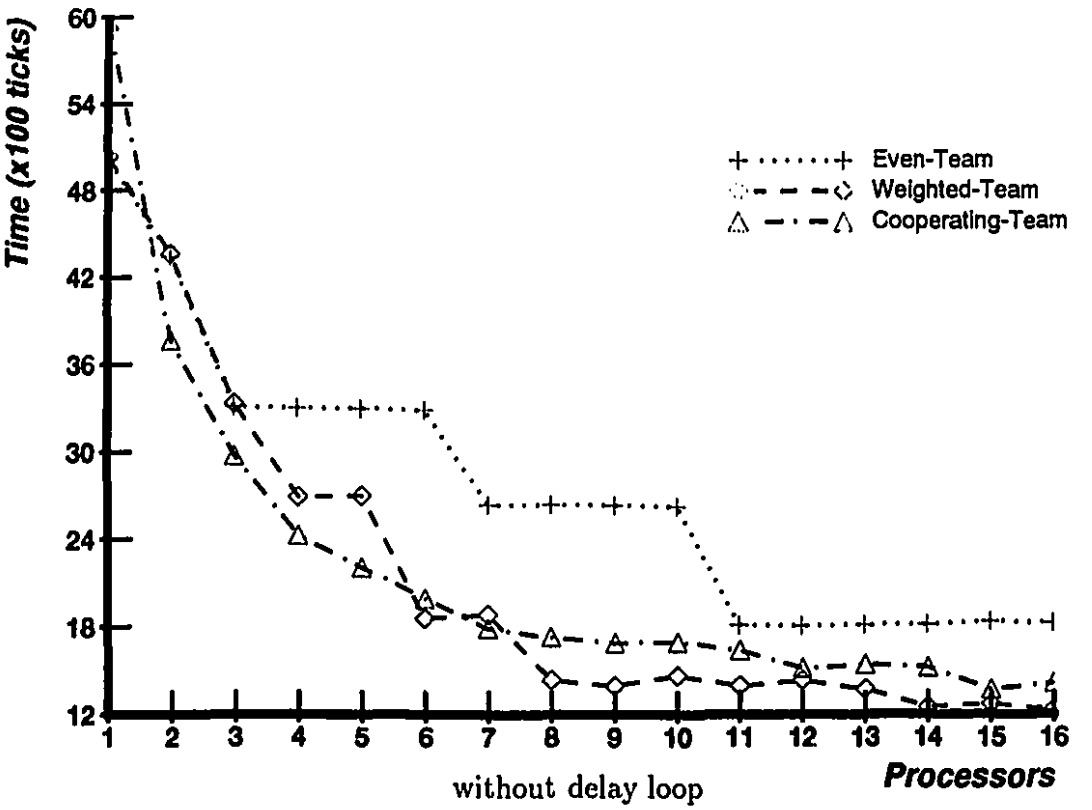

Figure 4.13: Random binary tree evaluation: 1023 nodes

even larger difference between the performances of the Even-Team and the Weighted-Team mechanisms. The Weighted-Team shows a larger advantage now because the overhead for synchronization and contention is relatively small when the work load is large. Also note that with larger work load, the Cooperating-Team provides the best performance. Figure 4.15 presents the relative speedup results for the experiment with delay loop.

Figure 4.16 presents the result of evaluation a balanced tree with a delay loop. As expected, the Weighted-Team loses its advantage. The Cooperating-Team still keeps the best performance, but its advantage is minor because the work load is balanced and the is not much need for Team-Cooperation.

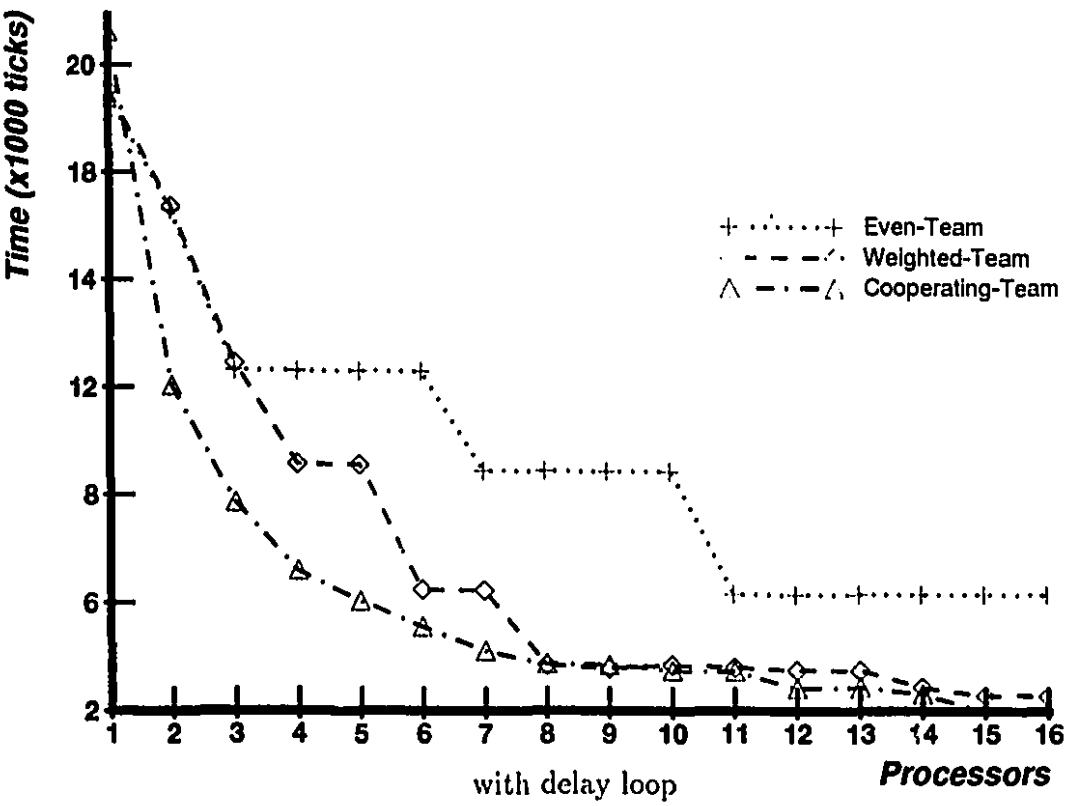

Figure 4.14: Random binary tree evaluation: 1023 nodes

	Time (x10^3ticks)	Speedup (compared to Even-Team)
Even-Team	12	1.00
Weighted-Team	5.9	2.03
Cooperating-Team	4.8	2.50

Figure 4.15: Relative performance on six processors

4.5 Example 3: Quadrature

In the Quicksort and Binary tree evaluation examples, the computation load was not too heavy, but the problem size was quite large. In this section, we present a smaller application, the computation of an approximation for the integral of a function. The program attempts to produce an approximation that falls within an error tolerance

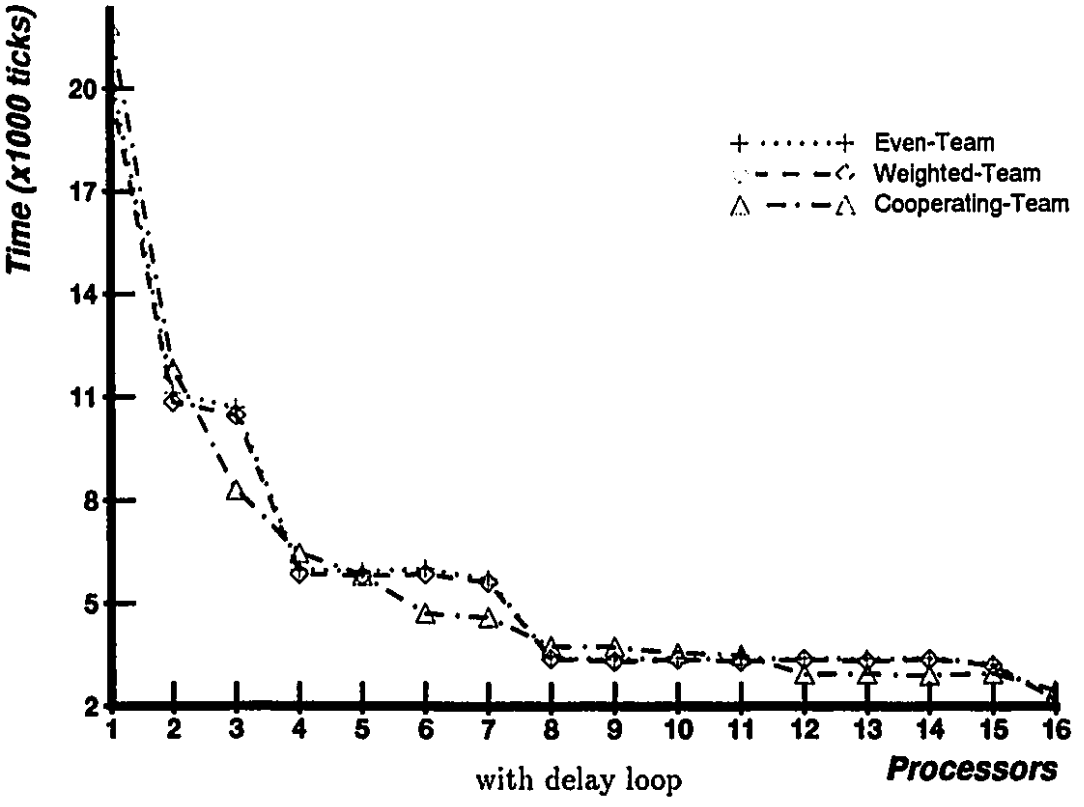

Figure 4.16: Balanced binary tree evaluation: 1023 nodes

specified by the user.

The technique, which we used to compute an approximation of an integral over an area, is called Simpson's rule [CC80] and amounts to computing the value of

$$\frac{f(x_l) + 4f(x_m) + f(x_r)}{6}$$

where x_l is the left boundary, x_r is the right boundary, and x_m is the midpoint. To get the approximation within the desired tolerance, we divide the interval at the midpoint, use Simposn's rule again to compute approximations for each half, add the two results together, and check the sum to see how close the cruder approximation is to the more refined computation. If the two values are "close enough", that is, the difference is less than the allowable difference specified by user, we take the sum, which is the more accurate value we hope, and call it our approximation. On the other hand, if the difference exceeds the specified tolerance, we recursively integrate each

half of the interval, add the two results, and called the sum the desired approximation.

The parallel program can be easily written as follows,

```
evaluate(integral, start, end)
float *integral, start, end;
{
  compute approximations;
  if ( not close enough ) {
    evaluate(&left_integral, start, (start+end)/2)
      // evaluate(&right_integral, (start+end)/2, end);
    *integral = left->integral + right->integral;
  } else {
    *integral = sum of approximations;
  }
}
```

The function that was integrated in our experiments follows,

$$f(x) = \sin(4 * \arctan(1) * x)^y / factor(y)$$

$$factor(y) = factor(\lfloor y/2 \rfloor) * (1 - (0.5/\lfloor y/2 \rfloor))$$

where $factor(0) = 1, y = 30$.

The performance curves are shown in Figure 4.17. Since in this example, the work load is almost balanced, we can see the Weighted-Team mechanism lost its advantage compared to the Even-Team mechanism, while the Cooperating-Team still keeps a little bit better performance than original one.

The other point we can get from this figure is that the performance of the three mechanisms is fairly close. One reason we have mentioned above is that the work load is balanced. The other reason is that the computation is a small part of each parallel task.

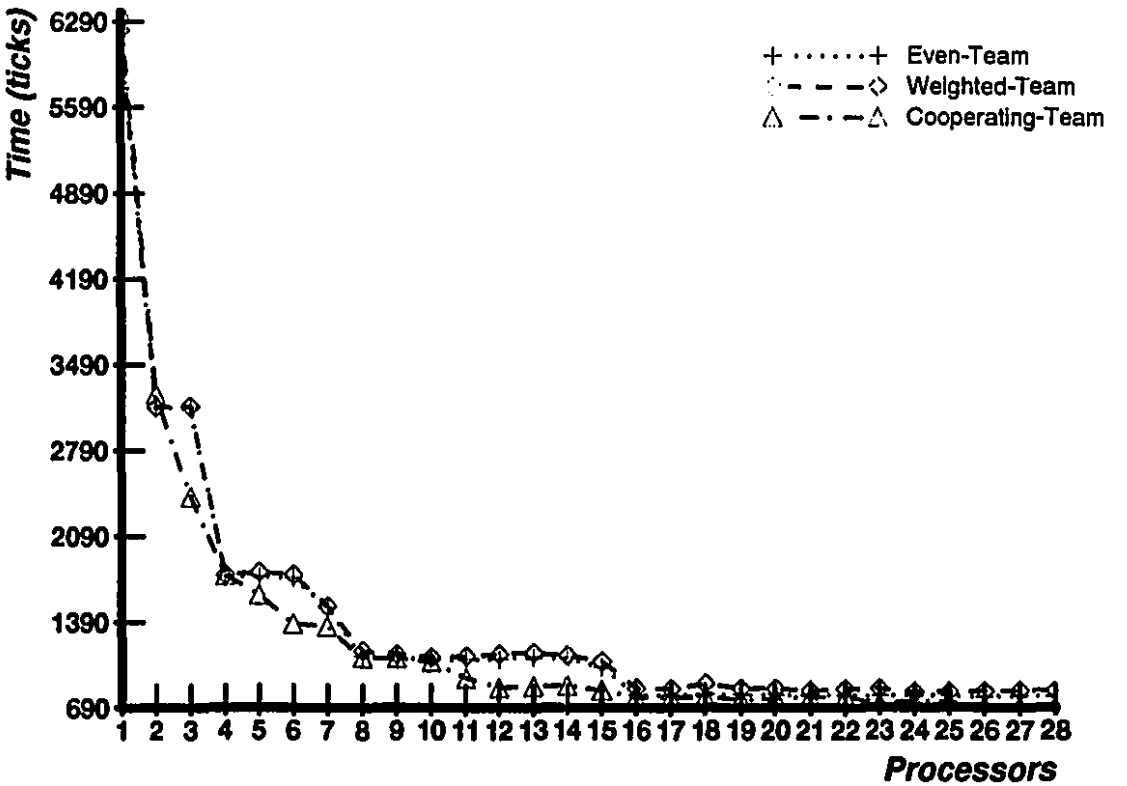

Figure 4.17: Quadrature

4.6 Example 4: Parallel Search

Search is one of the most important techniques in artificial intelligence area. Since it involves the creation and manipulation of trees, it can frequently take advantage of multiple processors.

For our test, we chose a 4 by 4 puzzle search application. By giving an initial pattern and a goal pattern, the program attempts to find the solution paths which can transform initial pattern into goal pattern. There are a number of ways to do a search. The method we used is the bound-first search method. The programs searches the nodes which can be generated within the cost bound. If within the current cost bound, no solution can be found, the cost bound is increased and the search is restarted. This process is iterated until a goal is found. This parallel search manipulates a search tree, where each node has at most three children. We now come to the issue of how exactly such a tree should be represented during execution of the

program. We have chosen an approach ("symmetric list") that generalizes easily to trees in which each node can have any number of children. In the data for each node we include a pointer to

```
its parent node
its first child
its sibling
```

Thus, we can represent the search tree shown in Figure 4.18 with the *binary search tree* shown in Figure 4.19.

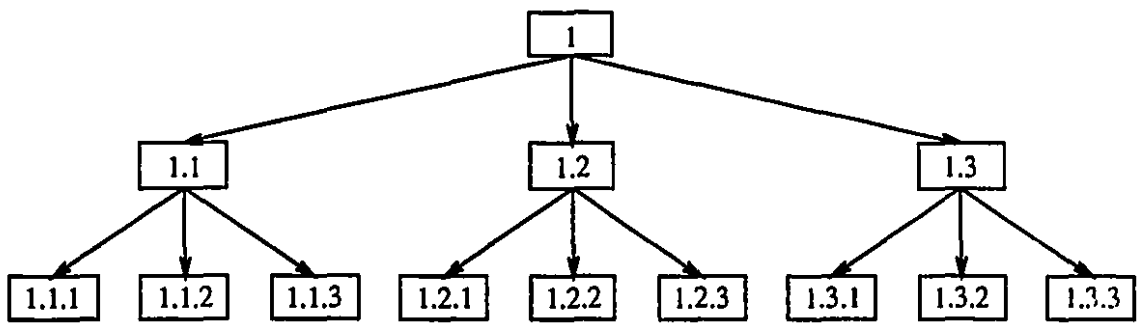

Figure 4.18: Search tree

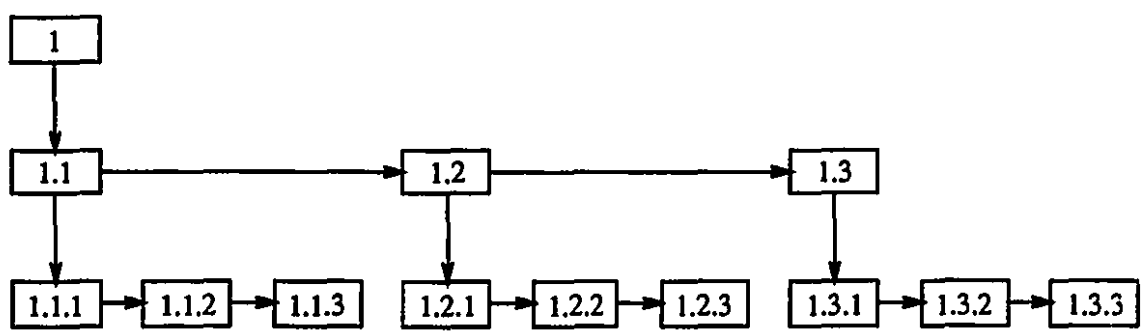

Figure 4.19: Binary search tree

Using this binary tree representation, the parallel search program can be described as follows,

```
search(root)
{
  if ( goal has been already found ) {
     return;
  } else {
     if (root->state == goal ) {
        output the solution path;
        return;
     } else {
        if ( satisfy the search bound ) {
           generate children nodes;
        }
     }
  }
  if ( there is child node to be searched ) {
     search(root->firstchild)//search(root->sibling);
  }
}
```

Figure 4.20 shows the test result by applying the Even-Team mechanism and the Cooperating-Team mechanism to this program. We can see a very impressive performance achieved by the Cooperating-Team mechanism. The Cooperating-Team mechanism gets 1.56 speedup over the Even-Team mechanism on 4 processors. On more than 4 processors, it keeps a 1.3 speedup.

We note very little speedup for cases when there are more than 5 processors. This is because that we are looking for only one of many possible solutions. Thus, by searching for solutions in parallel one generates extra tasks that have performed extra or speculative work over what would be required for a sequential search. As the number of processors is increased this amount of wasted work also increases. To

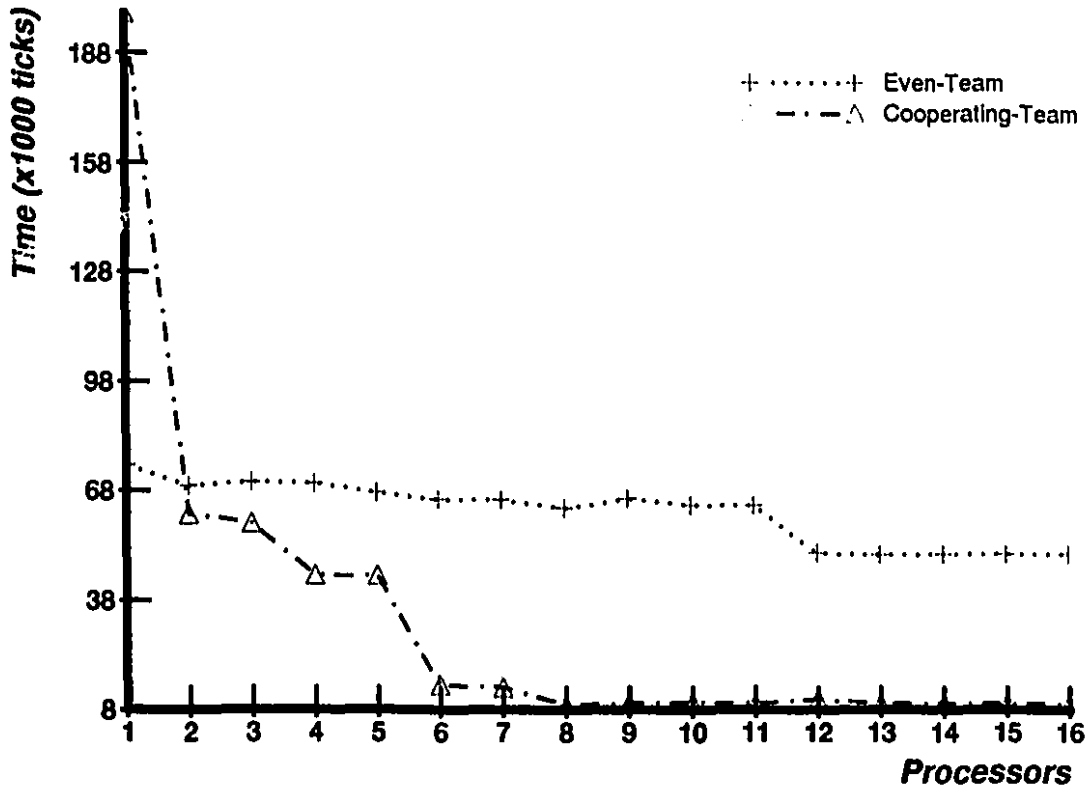

Figure 4.20: Searching: find one possible solution

compare the performance more precisely, we modified the program so that it searches for all possible solutions within the cost bound. This result is shown in Figure 4.21. For this case, both performances improve as the number of processors increases. In addition, the Cooperating-Team gets a 2.48 speedup over the Even-Team mechanism for the 16 processors case.

In our previous parallel search experiments, all the weights were chosen equally. However, in order to take more advantage of our Cooperating-Team mechanism, the user could adjust the weight which is associated with each task, so that the most hopeful search branch can be assigned more processors. Although there are many strategies for searching, our mechanism can balance the work load so that the performance may be improved further.

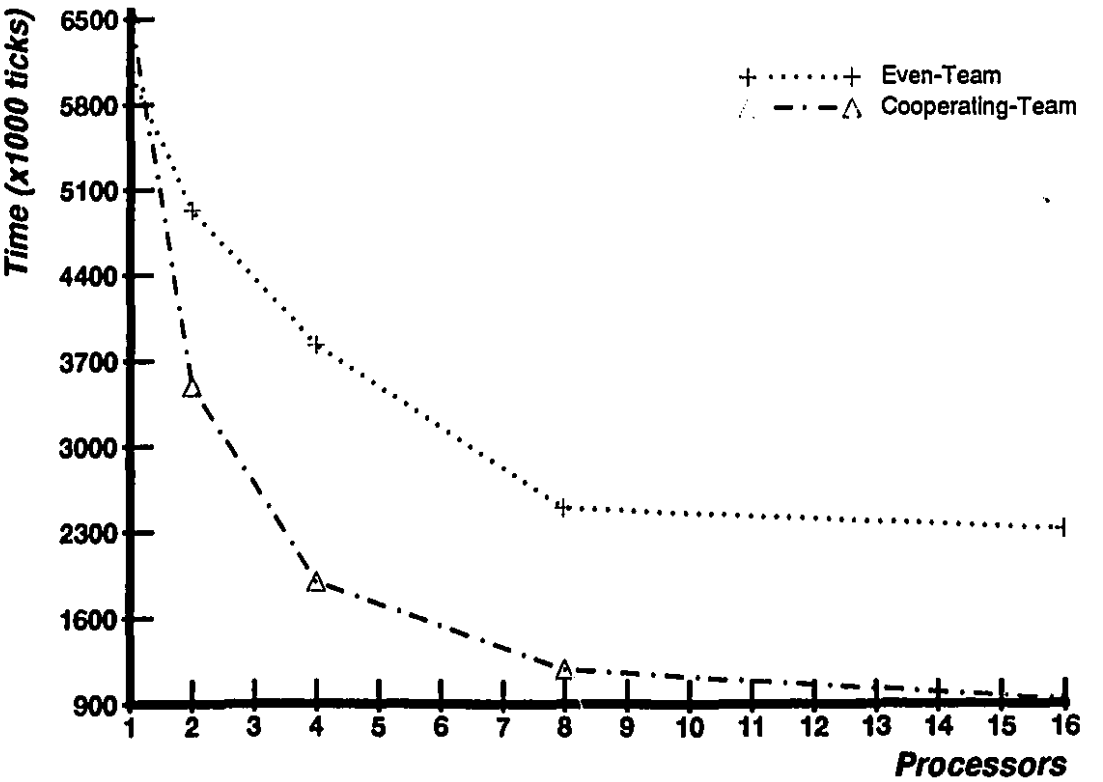

Figure 4.21: Searching: find all possible solutions

4.7 Summary

In the previous sections, we reported our experimental results. The first experiment we performed was parallel quicksort. Different cases have been tested, including the cases of different problem sizes, different item lists, and including the cases in which we artificially gave the wrong weight information to predict the work load. The results show the strong rebalancing ability of the Cooperating-Team mechanism.

The experiment of evaluating binary trees focuses on studying the factors which affect parallel performance. These factors include memory locality, memory contention, task size, processor resources and so on. The Cooperating-Team mechanism achieves the best performance in most situations.

A fairly small experiment we performed was to compute an approximation for the integral of a function. The last experiment is the parallel search. Both of them show

some advantages of the Cooperating-Team mechanism.

From these expermental results on the BBN Butterfly GP-1000, we can summarize the factors that affect the self-scheduling mechanism. They are,

Parallelism: This factor is the base factor of a parallel program. Without parallelism, the program cannot take any advantage of parallel processor's capabilities, and all of the three mechanisms achieve poor performance when there is not enough available parallelism.

Memory locality: In many parallel architectures, memory access time is a heavy overhead compared to computation. This is true for a machine like the BBN Butterfly GP-1000, it's better to access local memory rather than non-local memory. The locality effect can be noticed by comparing the performances between the tests which allocate data on only the processors it uses and the tests which allocate data on all of the processors of the system. This factor affects the performance of every scheduling mechanism.

Memory contention: When a computation performs many non-local memory access we see a performance degradation due to switch contention and memory contention. In some cases we see that memory locality and memory contention interact. That is, sometimes we can reduce the effect of memory contention by spreading out the data on more processors. In addition, memory contention may also cause some unexpected changes in load balancing. Our experimental results indicate that in these cases the Cooperating-Team mechanism is able to rebalance the workload and therefore avoids some of the performance degradation due to memory contention.

Granularity control: Due to the overhead of the mechanism itself, a program needs to control the granularity of parallelism to get better performance. Comparing the performances of the test cases of different problem sizes, we can observe that this is significant. Due to the increased overhead of the Cooperating-Team mechanism, larger scale tasks are more suitable for the Cooperating-Team mechanism.

Parallelism control: The importance of parallelism control has been shown by the superior performance of the Cooperating-Team mechanism. The Cooperating-Team mechanism controls parallelism and takes more advantage of the parallel processor. Each of the three mechanisms has adopted a parallelism control scheme. The Even-Team mechanism and the Weighted-Team mechanism control the parallelism by executing the parallel tasks sequentially, when there is only one processor in a team. However, in the same situation, the Cooperating-Team mechanism would put one task on its pool for waiting, executing the other task first. If within this execution period, some processor becomes free, it steals this task, otherwise, the current team take it back to execute. Thus, the Cooperating-Team mechanism gives more opportunity to make use of the free processors.

Other overhead: This overhead may be due to the system itself, such as process creation overhead, memory allocation overhead, etc. This overhead can be noticed by comparing the performances which were achieved on one processor with those on more than one processor. The results showed that the Cooperating-Team mechanism has more overhead than the others.

The experimental results which were presented in previous sections, demonstrate that our Cooperating-Team self-scheduling mechanism has a better performance in general, especially in adjusting to unbalanced work load. It gives users a general balanced execution environment to develop applications based upon the divide-and-conquer concept.

Chapter 5

Implementation

In this chapter, we present an overview of the SELSYN implementation. This chapter is organized as follows, we first give a structural overview of the compiler. Following that, two parts of our compiler, the parser and the code generator are presented. The code generation part also includes a description of the implementation details of the run-time environment.

5.1 Overview of the Compiler

The implementation of SELSYN-C has two integrated parts: a source-to-source compiler, and a run-time environment. The source-to-source compiler functions as a front-end in the system. The back-end is the high-level compiler of the target machine. Linked with the run-time environment implemented on the target machine, the output object code can run directly on that machine. Figure 5.1 illustrates the structural overview of the implementation.

The source-to-source compiler performs two functions. The first function is parsing the SELSYN-C source code and constructing the internal representation. The other function is high-level source code generation for the target machine. Theses two functions are independent of each other. The interface between them is the internal

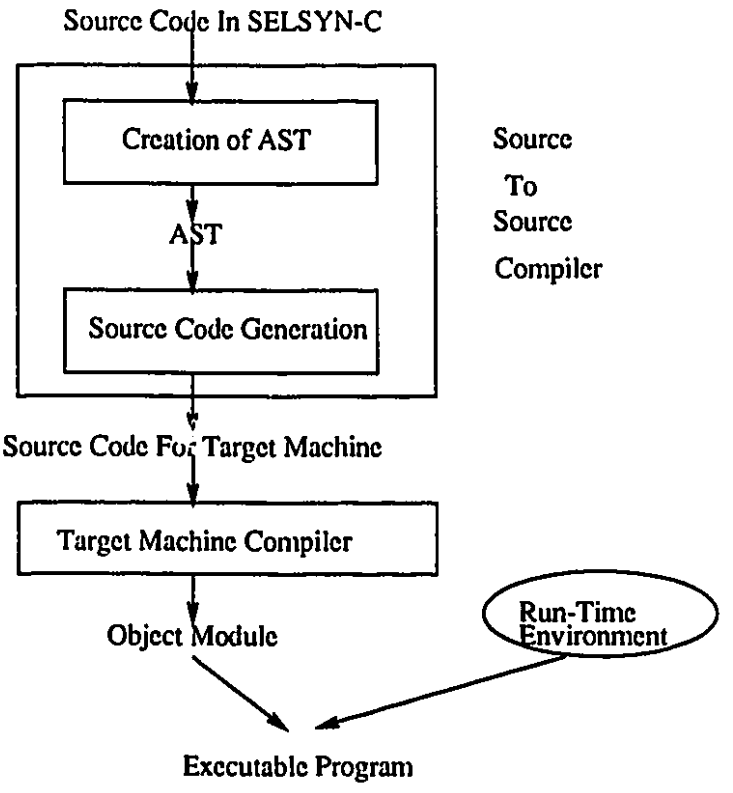

Figure 5.1: Structure overview

representation of the source program. We now introduce the construction of the internal representation.

5.2 Constructing the Internal Representation

We chose the GNC C compiler, GNU CC[1], as our work base. There are two reasons for this choice. One reason is that GUN CC is free to modify. The other reason is that the GNU CC adopts the AST (Abstract Syntax Tree) as an intermediate representation, which can retain the structure of the program, as its intermediate representation for carrying out high-level compiler optimizations.

[1]Copyright (C) 1988 Free Software Foundation, Inc.

We have used a modified version of the parser from GNU CC as the front-end of our compiler. Rather than only using AST to retain the structure of the program only up to statement level as original GNU CC does, we need to keep the structure of the entire program, by creating a complete AST for the program. This work has been done by Sridharan [Sri91]. Based on this complete AST, many transformation can be carried out on it, such as code transformation[2]. In addition to keep the support for original C language, we make use of this complete AST to contain the information which is supported by our language extensions.

The first modification we performed was to support the shared storage declaration. A new key word, `SHARED` is adopted in our language extension. In order to support it, the reserved keywords hash table has been modified. The GNU C compiler, like many other compilers, uses a hash table to handle keywords. However, the order of keywords in its hash table has been chosen for perfect hashing. This is achieved by using a program called "gperf", which is a separate part of the GNU C compiler. According to our language extensions, we modified the data file which contains all of the keywords of C language, then generated a new hash table and searching function.

The source-to-source compiler not only needs to parse the new keyword, but also it needs to retain this new information of the shared storage. Thus we had to modify the internal representation. Rather than creating a symbol table, the GNU C compiler represents the variables and their types as "Tree Nodes" in the AST[3]. The modification to support the shared storage type has been performed on these tree node definitions. An attribution bit has been added in the Declaration Node definition structure to indicate the shared storage attribution of the variable.

In addition to the `SHARED` declaration, several new node types have been added in order to represent the new operators. The parallel function call is represented by a new `PARALLEL_EXPR` node, which contains the weight information of the parallel function call.

The target dependent part of the source-to-source compiler is the high level code

[2]For a discussion of the AST transformation, the reader may refer to [Sri91].
[3]For the details of the GNU C compiler description, the reader may refer to [Sri91].

generation function. According to the specifics of different target machines, different high level codes need to be generated. We describe this implementation along with the description of the run-time environment implementation in the next few sections.

5.3 Generating Code

As we have introduced, the output of our source-to-source compiler is high level source code which is specific to the target machine. Now we present how to generate such output source code. The two main features of our language extension is the shared data storage and the parallel function call. In the following sections, we outline how these features have been implemented.

5.3.1 Implementing Shared Data Storage

The architecture of our target machine, BBN Butterfly GP-1000, has been illustrated in Chapter 2 in Figure 2.2. The GP-1000 operating system Mach 1000, is an extension of the Carnegie Mellon University (CMU) Mach operating system, which itself is an extension of the Berkeley 4.3 BSD Unix operating system[4]. Mach 1000 takes the basic premise of CMU Mach and extends it to include Butterfly-specific concepts such as clusters and virtual processor management library subroutines. To facilitate parallel programming, Mach 1000 provides an application library with routines for processor and memory management and allocation. This application library, known as the Uniform System (US) library, provides subroutines that can be called from C language programs [Moy88].

The Uniform System uses the Mach 1000 virtual memory system to implement globally shared memory. The globally shared memory and processor private memory can be viewed as shown in Figure 2.3 in chapter 2. In order to support our shared storage, we need to allocate the shared variables in the globally shared memory part.

[4]The difference between Mach 1000 and Berkeley 4.3 BSD can be found in [Moy88].

As discussed in chapter 4, memory contention affects performance. We allocate one heap in the shared memory of each individual processor so that the memory contention is reduced. Based on the same idea, each processor has a task stack and parameter heap allocated in the shared memory space so that the Cooperating-Team model could be carried out. Recall the memory allocation figure, Figure 2.4, presented in Chapter 2. We can draw a similar figure, Figure 5.2, illustrating the memory status after we allocate such heaps and stacks for processor m. The pointers to manipulate these shared spaces are kept in the processor-private memory on each processor. The shared variables are allocated easily in these heaps by the manipulating these pointers.

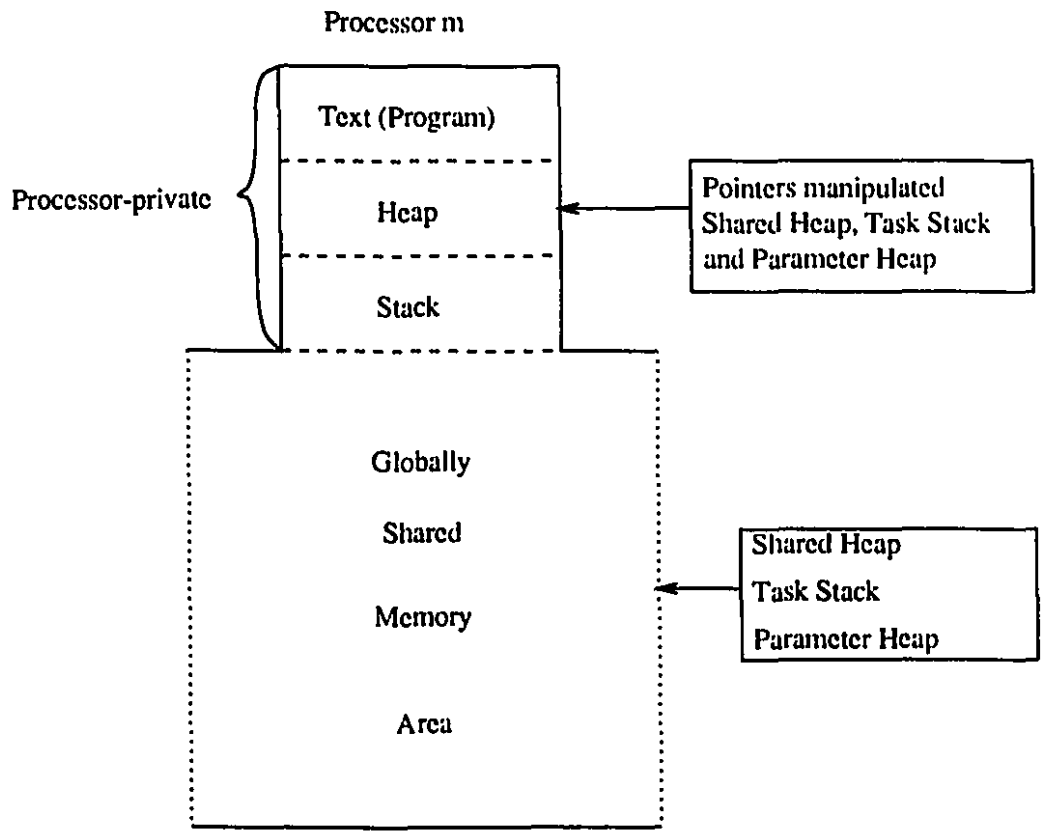

Figure 5.2: Shared storage

Let us consider the sum example again given in Figure 5.3. In this example, there

```
#define MAX 1000
shared int a[MAX];
main()
{ int  final_sum;
  ...
  sum(a, 0, MAX, &final_sum);
  ...
}

/* sum all entries a[l .. r], put in result */
sum(a,l,r,result)
int a[],l,r,*result;

{ shared int sum_left, sum_right;

  /* if only one entry left, then that is the sum */
  if (l == r)
    *result = a[l];
  else
    { /* sum left half and right half in parallel,
                        result is sum of left and right */
      int midpoint;
      midpoint = (l + r) / 2;
      sum(a,l,midpoint, &sum_left) // sum(a,midpoint+1,r,&sum_right);
      *result = sum_left + sum_right;
    }
}
```

Figure 5.3: An example SELSYN-C program

is a parallel function call,

```
sum(a,l,midpoint,&sum_left) // sum(a,midpoint+1,r,&sum_right).
```

In order to dispatch the function onto different processors, the function arguments have to be made available to the other processors via the shared memory. We have allocated a parameters heap for this purpose. For each function a parallel version is created which contains one parameter. This parameter is the address of a shared memory space which contains the arguments needed for the function. The example function example `sum()` would have a parallel version as,

```
int PP_sum (PARA)
int *PARA;
          /* Modified function name, so that it can
             accept shared variables passing. */
  {int *fpoint = my_local_heap;
          /* Obtain parameters via shared space */
   int  * a  = *((int  * *) PARA)++;
   int l  = *((int *) PARA)++;
   int r  = *((int *) PARA)++;
   int  * result  = *((int  * *) PARA)++;
```

Within the function sum(), there are two shared variables, sum_left and sum_right. Since the shared variables have to be allocated in our pre-allocated heap, the code to declare these two shared variables is,

```
int *sum_left ; int *sum_right ;
sum_left = ((int * ) my_local_heap)++;
sum_right = ((int * ) my_local_heap)++;
        /* Allocate the memory space for shared variables */
```

Note the type shared int is translated into type int *. This is because that the space for shared variables are assigned from the shared variable heap. One more level of address reference is created. Some more examples are,

shared int * j is translated into int * * j,

and shared int * * j is translated into int * * * j.

5.3.2 Implementation of the Cooperating-Team Model

Before we introduce how to implement parallel function call feature, we first present the implementation of the processor management. Studying the experimental results presented in Chapter 4, we have seen that our dynamic team-dividing strategy combining with team-cooperation, called the Cooperating-Team mechanism, has strong advantages over the static team-dividing strategy. We now explore the implementation issues to determine whether the overhead of Cooperating-Team mechanism can be acceptably minimized. Currently, the Cooperating-Team mechanism has been implemented on BBN Butterfly GP-1000.

Task Description

The core data structure of the system is the concise task descriptor. All of the actions which need to be performed for team division and team cooperation are accomplished via the task descriptor. The concise task descriptor is designed to keep the information of a ready to run task. It is defined as follows,

```
/* task descriptor */
typedef struct taskInfo{
        int teamNo;             /*                          */
        int (*func)();          /* the runnable task        */
        int weight;             /* the asscciated weight    */
        int teamSize;           /* number of team members   */
        int startProc;          /* leading processor        */
        int endProc;            /* last processor in team   */
        int *PARA;              /* parameters chain         */
        int sibLeader;          /* leading processor of
                                        sibling team        */
        int stat;               /* WAIT, EXEC, DONE, EXIT   */
        int ack;                /* acknowledgement */
        struct taskInfo *next;
} task;
```

This descriptor is used to for both team-dividing and team-cooperation. The meaning of the field `sibLeader` is slightly different when used for team-dividing or for team-cooperation. In team-dividing, it's obvious that it indicates the leading processor of the sibling team. While in the team-cooperation, this field indicates which leading processor stole the current task.

We designed a stack which functions as both a task stack and a task pool. It acts as a task stack for team-dividing, however it acts as a task pool for task waiting and task stealing. We can summarize these two actions, when encountering a parallel function call,

Case1: When there is more than one member in a team, team-dividing will occur. Two concise task descriptors are pushed onto the stack. Thus, the stack functions as task stack for team-dividing periods.

Case2: When there is only one member in a team, the task stack will function as a task pool which can store waiting tasks. The top of the stack will operate like a pool of tasks. Each waiting task descriptor is put in the pool. Tasks are taken out in FIFO order.

To manipulate this stack, we used four pointers which point to task stack bottom, task stack top, task pool bottom and task pool top respectively. We demonstrate the details of this manipulation in Appendix A.

Each processor is assigned one such task stack. The stack is used once a processor becomes the leading processor of a team and encounters a parallel function call. All of the processor assignments are accomplished via this two-function stack.

Task Issuing and Hoping

As we described, the execution always starts on the leading processor. The leading processor is the only processor that can issue parallel tasks. The algorithm for leading processor to issue tasks as follows (assume it encounters a parallel function call, `func_A(a) // func_B(b)`),

```
if ( only one processor on current team) {
       put func_A(a) on task pool waiting;
       execute func_B(b);
       if (func_A(a) still waiting on pool) {
                take func_A(a) back from pool;
                execute func_A(a);
       } else {
                POOL_WAITING ;
       }
} else {
       TASKISSUE ;
       TEAM_WAITING ;
}
```

Figure 5.4: Task issuing algorithm for leading processor

The step TASKISSUE calculates the ratio between the two weights associated with the two tasks (functions) and according to this ratio the current team is divided into two subteams, which execute func_A(a) and func_B(b).

The leading processor of the current team will become the leading processor of one subteam which executes func_B(b). One child processor of current team will be signaled to become the leading processor to execute func_A(a). We say that this child processor has been hoping to get a task to execute. The task hoping algorithm of the child-processor is presented in Figure 5.5.

```
child_processor()
{
  Initialization;
  while ( Not Terminated )
    {
      wait to get a task to execute
      switch (TASK_CODE)
        {
          case WAIT:
                Become leading processor
                      and execute the task;
                TEAM_WAITING; /* join its sibling team */
                break;
          case EXIT:
                Terminate the processor;
                break;
        }
    }
}
```

Figure 5.5: Task hoping algorithm for child processor

Note that both the leading processor and child processor have the TEAM_WAITING step. The TEAM_WAITING and POOL_WAITING accomplish the synchronization between teams. Team-Cooperation may happen in these steps. In the next subsection we present the algorithms for these two functions.

Synchronization and Cooperation

Under two cases, synchronization is needed. One case is at the team-dividing point. When both subteams finish their tasks, they need to merge together into one team. The two subteams wait for each other to finish their tasks. The other synchronization happens when a task has been stolen by the other team. The processor which lost the task should wait for the task to be finished. During the waiting period of the synchronization, Team-Cooperation may happen.

The team waiting scheme and the pool waiting scheme are presented in Figure 5.6 and Figure 5.7.

```
TEAM_WAITING()
{
        While ( sibling team hasn't finished task ) {
                if ( task waiting on sibling team's task pool ) {
                        steal one to execute.
                }
        }
}
```

Figure 5.6: Team-waiting algorithm

```
POOL_WAITING()
{
        While ( Stolen task hasn't finished ) {
                if ( task waiting on the processor that stole the task ) {
                        steal a sub-task back to execute
                }
        }
}
```

Figure 5.7: Pool-waiting algorithm

The idea behind these two algorithms is to use the waiting period to check whether the idle processor may offer help to a busy processor.

As we said previously, the goal of the Team-Cooperation is to rebalance the work load between the processors. To achieve this purpose, the remaining runnable tasks

are stored in the pool waiting for an idle processor to steal them. However, a team may have several sibling teams which can steal from its pool. According to the maximum number of sibling teams, we can decide the number of waiting tasks that may be allowed in the individual leading processor's pool. Since the processor team is organized during run-time, this number varies at run-time for each individual leading processor depending on how many times the team it leads has been divided. By considering this factor, the task issue algorithm for leading processors can be slightly modified as shown in Figure 5.8.

```
if ( only one processor in current team) {
       if ( task-pool not full ) {
            put func_A(a) on task pool waiting;
            execute func_B(b);
            if (func_A(a) still waiting on pool) {
                take func_A(a) back from pool;
                execute func_A(a);
            } else {
                POOL_WAITING ;
            }
       } else {
            execute func_B(b);
            execute func_A(a);
       }
} else {
       increase task-pool size;
       TASKISSUE ;
       TEAM_WAITING ;
}
```

Figure 5.8: Revised task issuing algorithm

5.3.3 Link To the Run-Time Environment

Each parallel function call is implemented by generating code for the case when there is more than one processor in the current team (team division) and the case when there is only one processor in the current current team (adding a task descriptor

to the leading processor's pool). The generated code makes use of calls to the run-time system to handle synchronization. Note that procedures calls, `POOL_CHECK()`, `POOL_WAITING()`, `factor()`, `issueTask()`, `UsLock()`, and `UsUnlock()`, refer to calls to the run-time environment. The implementation of those primates for the BBN Butterfly GP-1000 is given in Appendix C.

```
/**** Here goes the transformation for parallel function call ****/
/* sum(a,l,midpoint, &sum_left) // sum(a,midpoint+1,r,&sum_right) */
if ( my_index == my_endpoint ) {
        /* Only one member in team. */
        int *arglist; int *para, *para1;
        para1 = arglist = arg_list_ptrs[my_index];
        *((int  * *) arglist)++ = a ;
        *((int *) arglist)++ = l ;
        *((int *) arglist)++ = midpoint ;
        *((int * *) arglist)++ = (& *sum_left) ;
        para = arglist ;
        /* Setup for sum(a,l,midpoint, &sum_left) */
        *((int  * *) arglist)++ = a ;
        *((int *) arglist)++ = (midpoint + 1) ;
        *((int *) arglist)++ = r ;
        *((int  * *) arglist)++ = (& *sum_right) ;
        arg_list_ptrs[my_index] = arglist;
        /* Setup for sum(a,midpoint+1,r,&sum_right) */
        if ( my_task < my_task_top ) {
                UsLock(taskStack_lock[my_index], 0);
                if (((poolStack_top[my_index] - poolStack_base[my_index])
                    < my_pending_no)
                  || (poolStack_base[my_index]
                    == teamStack_top[my_index])) {
                   /* Enough Task Pool space */
                   *my_current_pending++;
                   my_task->func = PP_sum ;
                   my_task->stat = WAIT;
                   my_task->startProc = my_index;
                   my_task->PARA = para1;
                   my_task->next = NULL;
                   /* Create the waiting task description */
                   if (poolStack_base[my_index] == teamStack_top[my_index]){
                        poolStack_base[my_index] = poolStack_top[my_index]
                        = my_task++;
                    }else { poolStack_top[my_index] = my_task++;
                    }
                    UsUnlock(taskStack_lock[my_index]);
                    /* Setup one task waiting on the pool */
```

```
                    PP_sum(para);
                    /* Execute the other task on current processor */
                    if ( POOL_CHECK() == 1 ) { /* Check task pool */
                         PP_sum(para1); arg_list_ptrs[my_index] = para1;
                         /* Take the task back to current processor */
                    } else {
                        POOL_WAITING();
                        /* Wait for joining.
                           Team-Cooperation may take place */
                    }
                  } else {
                    /* Task-Pool is full. Execute both tasks
                       on current processor */
                    UsUnlock(taskStack_lock[my_index]);
                    PP_sum(para);
                    arg_list_ptrs[my_index] = para;
                    PP_sum(para1);
                    arg_list_ptrs[my_index] = para1;
                    }
            } else { fprintf(&_iob[2], "stack overflow \n");
                     exit(3);
            }
    } else { /* More than one member in a team */
            taskPtr taskPointer, taskPointerT;
            int *arglist; int *temp;
            taskPointerT = taskPointer =  my_task++;
            taskPointer->func = PP_sum ;
            taskPointer->weight = 1 ;
            arglist = taskPointer->PARA = arg_list_ptrs[my_index];
            taskPointer->stat = WAIT;
            taskPointer->next = my_task;
            temp = arglist;
            *((int  * *) arglist)++ = a ;
            *((int *) arglist)++ = 1 ;
            *((int *) arglist)++ = midpoint ;
            *((int  * *) arglist)++ = (& *sum_left) ;
            arg_list_ptrs[my_index] = arglist;
            /* Setup for sum(a,1,midpoint, &sum_left) */
            taskPointer =  my_task++;
            taskPointer->func = PP_sum ;
            taskPointer->weight = 1 ;
            arglist = taskPointer->PARA = arg_list_ptrs[my_index];
            taskPointer->stat = WAIT;
            taskPointer->next = NULL;
            *((int  * *) arglist)++ = a ;
            *((int *) arglist)++ = (midpoint + 1) ;
            *((int *) arglist)++ = r ;
```

```
        *((int  * *) arglist)++ = (& *sum_right) ;
        arg_list_ptrs[my_index] = arglist;
        /* Setup for sum(a,midpoint+1,r,&sum_right) */
        poolStack_base[my_index] = teamStack_top[my_index]
         = poolStack_top[my_index] = taskPointer;
        /* Create task description */
        factor(taskPointerT); /* Setup for team-dividing */
        my_pending_no++;
        issueTask(taskPointerT);
        /* Issue tasks and join. Team-Cooperation may take place */
        my_pending_no--;
}
```

We give some further explanation on this generated code here. The manipulation of `arglist` is to prepare the function's parameters for transferring to another processor. All of the parameters that need to be transferred are put in the pre-allocated shared space. For the case of more than one processor in a team, after two task descriptors have been prepared, the team-dividing setup function `factor()` and team-dividing function `issueTask()` are invoked. In the case of one processor team, if the task pool has enough space, one task descriptor will be put on it for waiting, the other task is carried out on the local processor. The function `POOL_CHECK()` is called to check whether the waiting task has been stolen by other processors since the local processor finished its own task on hand. `POOL_WAITING()` is called to synchronize the processors if task stealing has happened. For the situation, where there is one processor in a team and not enough pending space on task pool, the two tasks are carried out sequentially on the local processor.

5.3.4 Initialization

As we illustrated in the structural overview, the object module has to be linked with the run-time environment. It includes the link with the initialization part, which is the main entry to invoke the application. Our mechanism is based on Cooperating-Team model, and in order to carry out this mechanism, in the initial phase of an execution, the Cooperation-Team system has to get control of all of the available processors. The first step of this initialization phase is achieved via Uniform System's

routine `InitializeUs()`. This routine creates and starts a Uniform System process on every available processor in the cluster, sets up the memory that is globally shared among all Uniform System processes in the cluster, and initializes the Uniform System storage allocator[5]. After this first step, the processors are under the control of the Cooperating-Team system. The next step in initialization is to prepare the memory space for the mechanism usage. This includes the task stacks and the parameter stacks and shared memory storage. After this initialization phase, all of the available processors to the execution have been constructed as one team. Every processor has the same copy of the execution code. The leading processor starts the execution, and the rest of the processors start to wait for the tasks that may assigned to them and hope to become a leading processor. The algorithms for both leading processor and these waiting processors have been presented in the previous sections.

5.4 Summary

Our implementation is currently on the BBN Butterfly GP-1000. The implementation includes

- A language parser for the SELSYN-C.
- Code generation for the BBN Butterfly C.
- Support for shared storage.
- A run-time library, which provides the following functions,

  Main: a main entry to initialize the system and invoke the application.

  child(): performs the task hoping for waiting processor.

  factor(): setup for team-dividing.

  issueTask(): performs team-dividing and team-joining.

[5]The detailed description of the Uniform System can be found in [Sen88].

TEAM_WAITING(): performs synchronization between teams.

POOL_CHECK(): performs the task-taken back from local task waiting pool.

POOL_WAITING(): performs the synchronization between teams which have team-cooperation.

The language parser is machine independent. While the rest of the components are machine dependent parts of the implementation. Since the language parser contains all the information of program structure, attaching different machine dependent parts to it can generate object codes for different target machines. Current the object code is generated for BBN Butterfly GP-1000.

Chapter 6

Related Work

Many researchers are rising to the challenge of exploiting parallel processing. Efforts on developing parallel programming languages include inventing new languages and adding new features to various existing sequential languages. Further work has been done for developing various scheduling strategies.

In this chapter, we outline other related work in the areas of both parallel programming languages and scheduling mechanisms. We compare these to our SELSYN-C language and Cooperating-Team mechanism which has been presented in this thesis.

6.1 Parallel Programming Languages

The development of parallel programming languages is an attractive field of research. Some researchers have worked on developing entirely new languages for parallel programs, such as Strand [FTS9], PCN [CT90], SISAL [MSA+85] and BLAZE [MvR87]. Although these are interesting and often elegant languages, they do not satisfy our goal of being familiar to programmers.

There are also many activities which involve developing extensions to existing languages. In this area, one of the major thrusts has been the development of extensions

to the language FORTRAN. This is due to the fact that traditionally, many parallel applications have come from the scientific community and have therefore focused on techniques for dealing with arrays. In addition, many programs and parallel approaches have been developed in FORTRAN. There are several proposed extensions to this language, which include EPEX/Fortran [DGNPS8], Cedar FORTRAN [GPHL88] and Force [Jor87, JBAJ89]. Some of them were invented for earlier developed vector processors and they contain specific constructs that allow the programmer to exploit vector machines. For instance, Cedar FORTRAN was developed for the Cedar Vector Processor. Force was developed for MIMD multiprocessors with a shared memory programming model. This language was implemented as an extension supported by a preprocessor. Several other researchers have focused on developing extensions to language C. Efforts in this direction include Concurrent C [CGR89], parallel-C [KS85], PCP [Bro89], and Jade [LR91]. Concurrent C is targeted for the Concurrent Sequential Process model. Parallel-C is an attempt for the SIMD and MIMD modes of parallelism. It includes the definition of parallel variables, functions and expressions. PCP offers some constructs to specify the parallelism of a program. Similar to Force it was targeted to the shared memory programming model. Jade is a data-oriented language for parallelizing programs. It is targeted towards exploiting parallelism at a coarse-grain level. In the domain of symbolic languages like Lisp there has been effort in designing Mul-T [KHMS9] and Multilisp [Hal85]. By using a new construct, future, programmer can express the parallelism explicitly in the Lisp.

Rather than developing a new high-level language, we have selected the programming language C as the basis of our work because it is familiar to a wide range of programmers, and thus it provides a large user base for our new language. SELSYN-C is an extension set to C. It provides a very simple way to express parallelism explicitly in imperative language C. This extension includes the definition of shared variables and parallel function calls. Close to some related work, our programming model is targeted to MIMD multiprocessors with the shared memory model and the parallelism expression is at a coarse-grain level. Compared to the other development on the C extensions, our language has the advantage of being easy to use and understand. We have concentrated on minimizing the number of new keywords and operators.

Further, we have introduced application specific extensions that are useful for the compiler (weights). This information is used to organize a dynamic self-scheduling parallel program rather than a statically- scheduled parallel program. Our language's backbone, the dynamic scheduling mechanism is discussed in the next section.

6.2 Scheduling Mechanism

Backing up our language extensions, we developed a self-scheduling mechanism that can achieve high efficiency and good utilization of processors. One benefit of this mechanism is that it reduces synchronization overhead so that processor stall time can be decreased.

Several related methods to solve the synchronization overhead have been proposed. A synchronization method by fuzzy barriers was suggested by Gupta and Epstein [GE90]. The main idea of this method is to divide an instruction stream into two kinds of regions, non-barrier regions and barrier regions. Streams with no barrier regions require no barrier synchronizations. While for the steams with barrier region areas, synchronization should finish within a barrier region. The goal of this method is to reduce the processor stall time. The processor can wait for synchronization while it is executing the instructions within the barrier region. The tolerance of the mechanism to the variation in the rate at which each stream progresses is limited by the number of instructions in the barrier regions. Thus, the larger the barrier regions, the less likely it is that the processor will stall. Figure 6.1 (a) presents the traditional barrier synchronization. The synchronization must occur exactly at two barrier points of the two processes. While Figure 6.1 (b) illustrates fuzzy barriers. In this case, synchronization only needs to finished somewhere within the two barrier regions. Therefore, this method gives more tolerance time for synchronization. However one potential weak point is that the compiler may not be able to find a big enough synchronization region, so that it may not substantially improve the performance. The method is suitable for synchronization among small scale tasks, such as synchronization of loop iterations and may not function well for synchronization among tasks

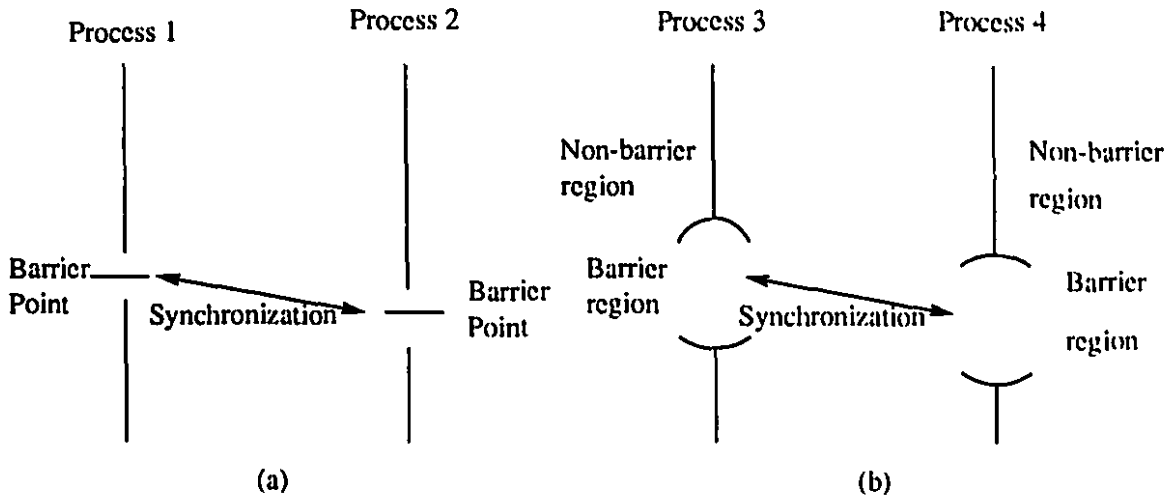

Figure 6.1: Principle of fuzzy barrier

such as synchronization between two procedures. Our Team-Cooperation idea is proposed for this purpose. Instead of finding the small pieces of barrier region statically at compile-time, our Team-Cooperation scheme dynamically finds executable tasks to reduce the processor stall time at run-time.

Our work on scheduling is mainly influenced by the work of WorkCrew [VRS8] and PCP [Bro89]. The WorkCrew approach [VRS8] introduces a mechanism that handles task division properly. Under this model, a centralized scheduler controls the processor acquisition and task dispatching. Whenever a processor meets a task that can be divided into two subtasks, it issues a processor request to the central scheduler and continues on executing one of the subtasks on its own. The WorkCrew approach won't actually dispatch the other subtask unless a processor becomes idle. This means that the processor, which issued the processor request, will also execute the other subtask on its own if no other processor has taken this queued subtask. The WorkCrew mechanism is based on a fork mechanism and therefore it faces the cost of processor acquisition.

Eugene Brooks introduced a split-join model rather than a fork-join model [Bro89]. In the split-join model the job starts out with all of the processors it will ever have,

and this team of processors is disassociated into independent subteams as nested concurrency is encountered. The weak point of this system is its inability to balance the work load between independent processors. In addition, their approach emphasizes loop iterations, while our Processor-Team approach is targeted towards coarse-grain procedure level parallelism and divide-and-conquer parallelism.

Our mechanism has the advantage of both these two mechanisms. We reduce the cost by adopting the split-join model (which we call team-division), while at the same time we introduced team-cooperation to rebalance the work distribution. Rather than having a centralized task-pool, we create individual controller for each team so that the cost is reduced further.

Other related research on task scheduling has been introduced for Strand [FT89] and for Mul-T [KHM89]. Mul-T uses lazy task creation to control the parallelism and to try to balance the work distribution. Both approaches adopted central schedulers for which high expense can be expected. Our mechanism uses some similar ideas, but is implemented in the domain of the popular language C.

6.3 Compiler Generated Self-Scheduling Programs

Self-scheduling programs can be manually generated. For instance, in the WorkCrew model [VR88], it is the programmer's responsibility to write necessary procedure calls to ensure correct synchronization and value retrieval. However, it is clearly more preferable if the programmer can invoke a compiler to automatically generate a self-scheduling version of a source program. This idea has been accepted by many researchers and some of them developed language preprocessors to support this idea. For example, such a preprocessor has been adopted in PCP [Bro89] and Force [Jor87, JBAJ89]. Foster designed an source-to-source transformation which transforms a high-level concurrent language into imperative language C and Fortran [FT89, FO90]. Our source-to-source transformation is from the parallel C extension, SELSYN-C, to BBN Butterfly C, which is specific to the target machine. The extensions become part

of the extended C grammar. Our source-to-source transformation emphasizes more on the internal representation for the parallelism so that different transformations can be performed. For instance, variables storage type conversion, function type and name transformation and so on. In addition, the internal representation can be utilized for generating code for different machine parallel programming tools.

6.4 Summary

In this chapter, we presented and compared the main related developing trends for both parallel programming languages and scheduling mechanisms. On the portability issue, we also compared the preprocessor approach and source-to-source transformation approach for language implementation. From our discussion, we can see that the techniques for parallel programming languages and self-scheduling programs have been widely studied and developed, and it is still one of the most challenging areas in providing software environments for parallel programming. SELSYN-C and Cooperating-Team mechanism is our endeavor to meet this challenge.

Chapter 7

Conclusions and Further Work

This thesis has reported on the SELSYN-C compiler and the associated Cooperating-Team scheduling mechanism. The SELSYN-C compiler has been implemented and is currently producing parallel code for the BBN Butterfly GP-1000 parallel processor.

As discussed in this thesis, the design our the SELSYN-C language was driven by the need for an "easy-to-use" language that was accessible to a wide variety of programmers. Our SELSYN-C language consists of simple extensions to C that free the programmer from dealing with architectural details like processor management. Rather, the programmer can concentrate on the application and leave the details of scheduling and resource management to be handled completely by the the SELSYN-C compiler and associated run-time library. SELSYN-C supports two major extensions to C: (1) the distinction between processor-private and globally-shared data, and (2) the introduction of weighted parallel function calls. It provides users with a simple way to declare parallelism explicitly. New keywords, constructs and idioms are minimized for this purpose. By using these simple but useful extensions, the programmer can express shared data, explore parallelism at the coarse-grain level, and take advantage of the parallel architectures.

In order to provide an effective and efficient scheduling strategy, we have incorporated the notion of weights into the SELSYN-C language, and we have developed

a new Cooperating-Team scheduling mechanism that can make use of these weights. In addition, our approach provides a mechanism for balancing the workload in the cases where the weights are not known or not completely accurate. The scheduling mechanism was implemented based on a Processor-Team model, which can reduce the overhead of processor acquisition. In addition, a dynamic team-dividing strategy was chosen as an effective method to organize the processor-teams. Combining this strategy with the team-cooperation scheme, the Cooperating-Team mechanism was established to achieve the goal of both high efficiency and high utilization.

Finally, we have demonstrated the effectiveness of our approach on several divide-and-conquer applications. Throughout our experimental results, our Cooperating-Team mechanism demonstrates its strong ability of rebalancing the work load between the processor-teams. It gives users a general balanced execution environment to develop applications based upon the divide-and-conquer concept.

Our development of SELSYN-C concentrated on the goal of simplicity, efficiency and good processor utilization. As we discussed in chapter 6, our language has the advantage of being easy to use. However, several useful features could be added to improve the SELSYN-C language. One such extension would be to allow multiple parallel function calls to enhance the exploitation of the parallelism. In addition, various high-level loop constructs could make SELSYN-C more general in its expression of parallelism.

By analyzing the experimental results, we can notice that one of most significant factors affecting performance was the locality of data. Because of the data locality on the system, memory access delays and contention can significantly affect performance. Further improvement would be achieved by studying and adopting different data mapping strategies that could be supported at the language level.

Combining the compiler techniques to automatically exploit parallelism is another possible improvement or extension to our system. Hendren introduced an analysis method to automatically figure out tasks that can be executed in parallel [Hen90]. While the approach of Jade is compiler analyzing aids by the notions which can be

utilized by the programmer to express the data dependency [LR91]. Both approaches can be adopted using our SELSYN-C to explore parallelism at a more fine-grain level.

In conclusion, we have reported our endeavor on developing an "easy-to-use" language and an efficient scheduling mechanism. In addition to the advantages we have already demonstrated, the further success of these approaches will depend on further performance improvement which would make it possible for the programmer to express parallelism at all levels of granularity rather than only at a coarse-grain level.

Appendix A

A Case Study of the SELSYN Mechanism

In this appendix, we illustrate different cases that our system may face. We study in detail how team-dividing and team-cooperation work together on these cases.

The following figures in this appendix illustrate team-dividing, task stack information, and task pool information. The team-cooperation is illustrated by showing a sequence of "snoopshots" of the task stack and task pool. First of all, we describe some notations that are used in our figures.

- **T_T** is the task stack top pointer.
- **P_B** is the pool bottom pointer.
- **P_T** is the pool top pointer.
- **D** represents that the task has be done.
- **E** represents that the task is being exccuted.
- **W** represents that the task is waiting.
- **Busy, Idle** represent the status of the team.

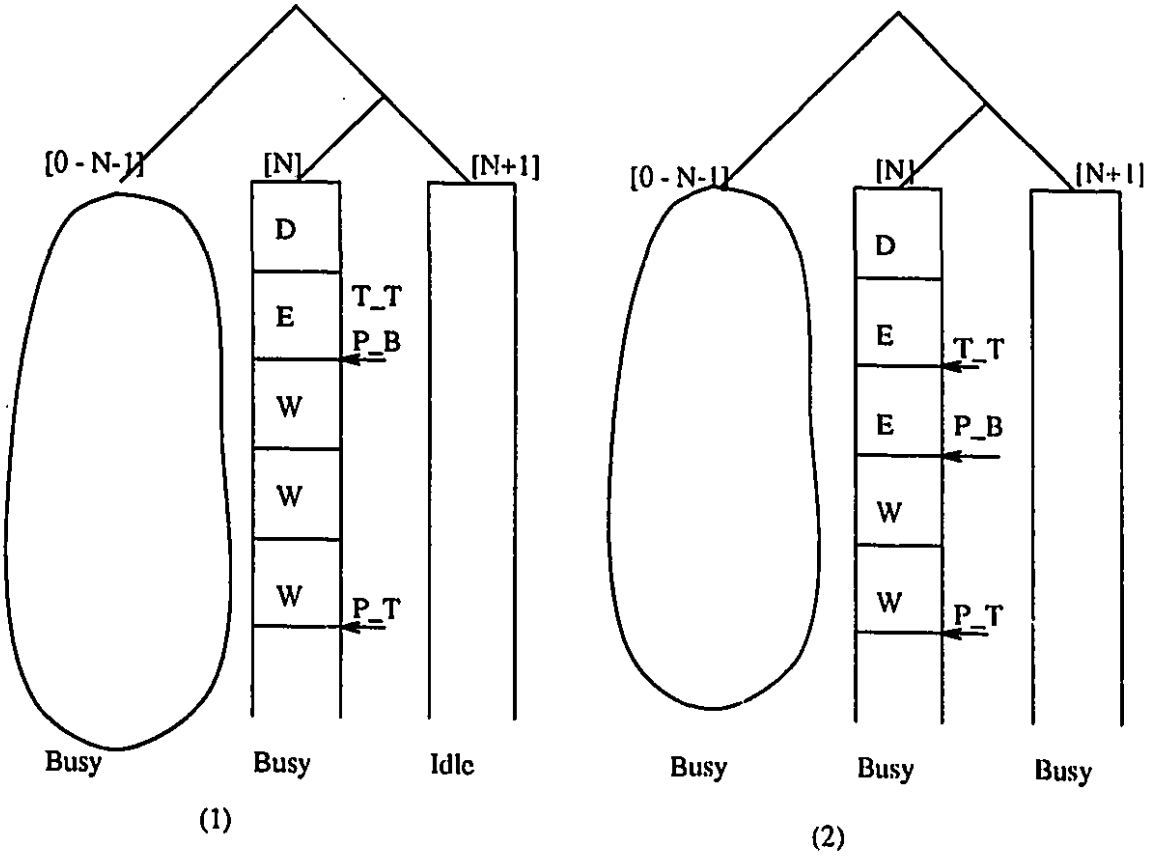

Figure A.1: Case example 1

The first case is presented in Figure A.1. In this figure, the processor team has been divided into team [0 - N-1], team [N], and team [N+1]. The task stack and task pool are shown for team [N] and team [N+1]. The detailed situation for team [0 - N-1] is not shown in this case. The figure is divided into two parts,

Part (1): This shows the situation when team [N] is busy and it has three other tasks, which are represented by **W** in the figure, on the waiting pool. While its sibling team [N+1], led by processor N+1, is idle waiting for the team-join with team [N] into team [N - N+1].This is indicated by **D** at the bottom of processor N's task stack. Note that processor N is the leading processor of team [N - N+1]. That's why it can issue a parallel task and keep two concise task descriptions on its task stack.

Part (2): Under our mechanism, the team-cooperation will happen between these two teams. Processor N+1, the leading processor of team [N+1], checks the task pool of processor N, which is the leading processor of team [N], and steals a waiting task to execute on team [N+1]. The task status at the bottom of the task pool of team [N] is changed to **E**. The status of team [N+1] is changed from idle to busy.

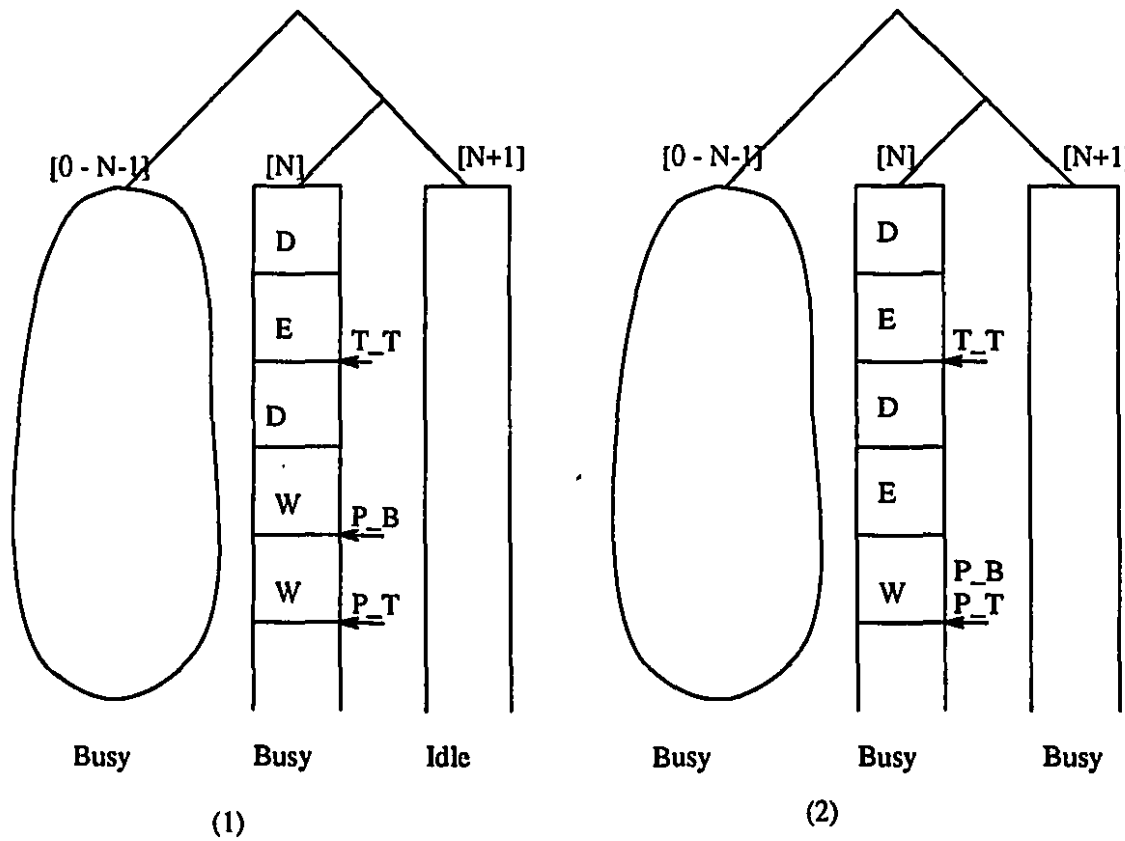

Figure A.2: Case example 2

Case 2, which is presented in Figure A.2, could happen following case 1. That is,

Part (1): After team [N+1] finishes the task which was stolen from team [N], team [N] is still busy with its own tasks and there are still two tasks left on its leading processor's task pool. This situation means that the team-join for team [N] and team [N+1] can not yet take place.

Part (2): Similar to the case 1, processor N+1 will steal a task from processor N. This should help to balance the load between the two teams.

Note that in both case 1 and case 2, task-cooperation always steals the waiting task at the bottom of the waiting pool. That's because in a recursive decomposition, the earlier a task is generated, the heavier work load it may represent.

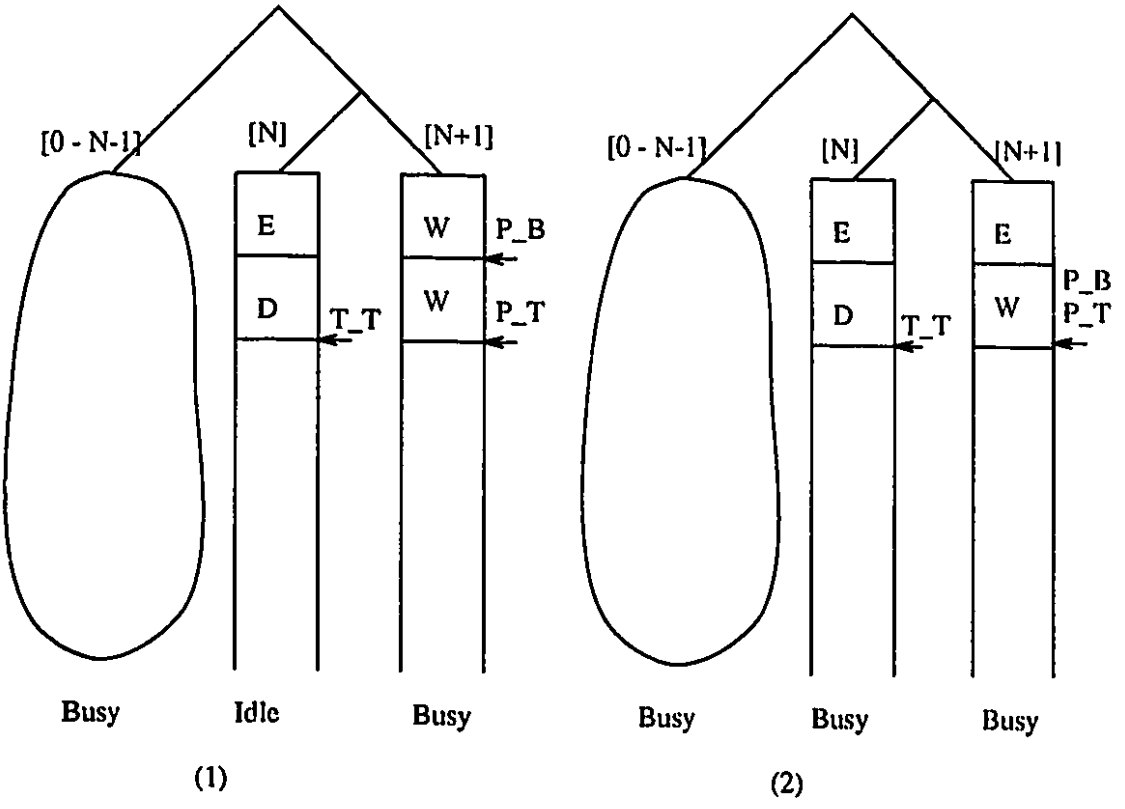

Figure A.3: Case example 3

Case 3, Figure A.3, is a similar situation to case 1, but the statuses of team [N] and team [N+1] are reversed, that is, team [N] is idle while team [N+1] is busy. In this figure,

Part (1): This time, team [N] finished its tasks first and is waiting for its sibling team, team [N+1], to join into team [N - N+1].

Part (2): Under this situation, leading processor N of team [N] takes a waiting task from the bottom of processor N+1's task pool to balance the progress of the two teams.

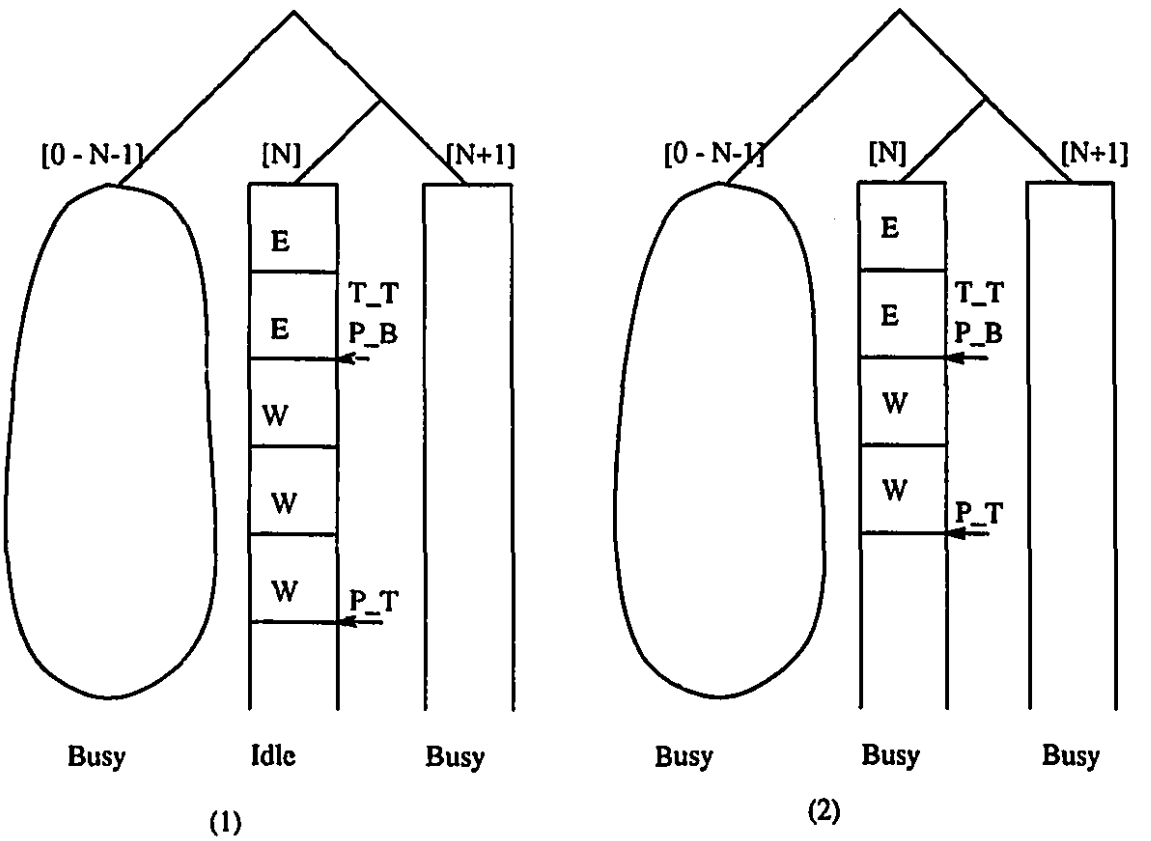

Figure A.4: Case example 4

Figure A.4 shows a situation when the processor N finishes its own task on hand. There are still tasks on its task pool waiting for execution and haven't been stolen by any processor, while the processor N+1 is busy with its own task. Under this situation, processor N takes one task back from the top of its own task pool. This action is indicated by the different positions of **P_T** in this figure.

The similar situation can happen as in Case 5, Figure A.5. Neither team [0 - M-1] nor team [M - N-1] is ready for a team-join. At this moment, there is no other team

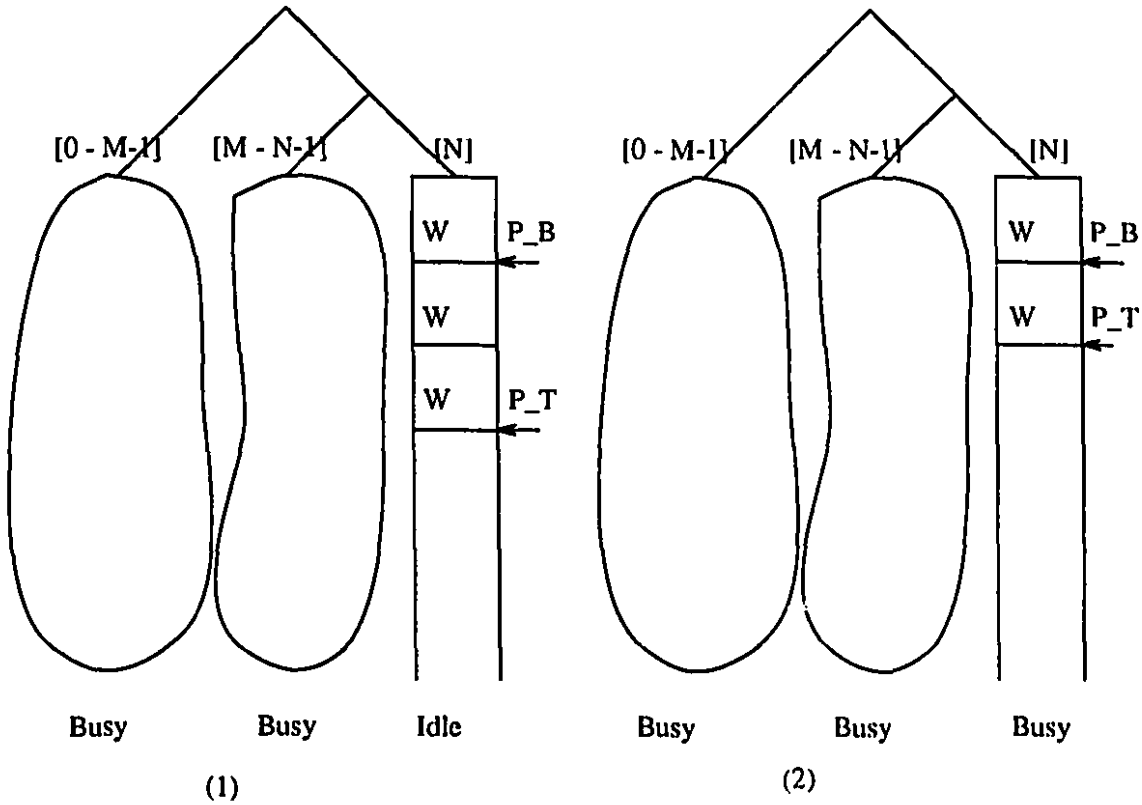

Figure A.5: Case example 5

to give a hand to team [N] to carry out the tasks which are waiting on the task pool of processor N. Team [N] takes one waiting task on its own.

Case 6, Figure A.6, shows the situation when team [N] has finished its tasks on hand, and all of the tasks which were put on the task pool when they were issued by processor N were stolen and have been executed by its sibling teams. Team [N] is waiting for its sibling team, team [N+1], to join into team [N - N+1]. Under these circumstances, team [N] gives a hand to its busy sibling team, team [N+1]. Processor N takes a task from the bottom of the processor N+1's task pool in order to balance the two teams' progress.

In section 3.4 of Chapter 3, we described the rules for task stealing. Because of these rules, a team may have opportunities of Team-Cooperation with several leading

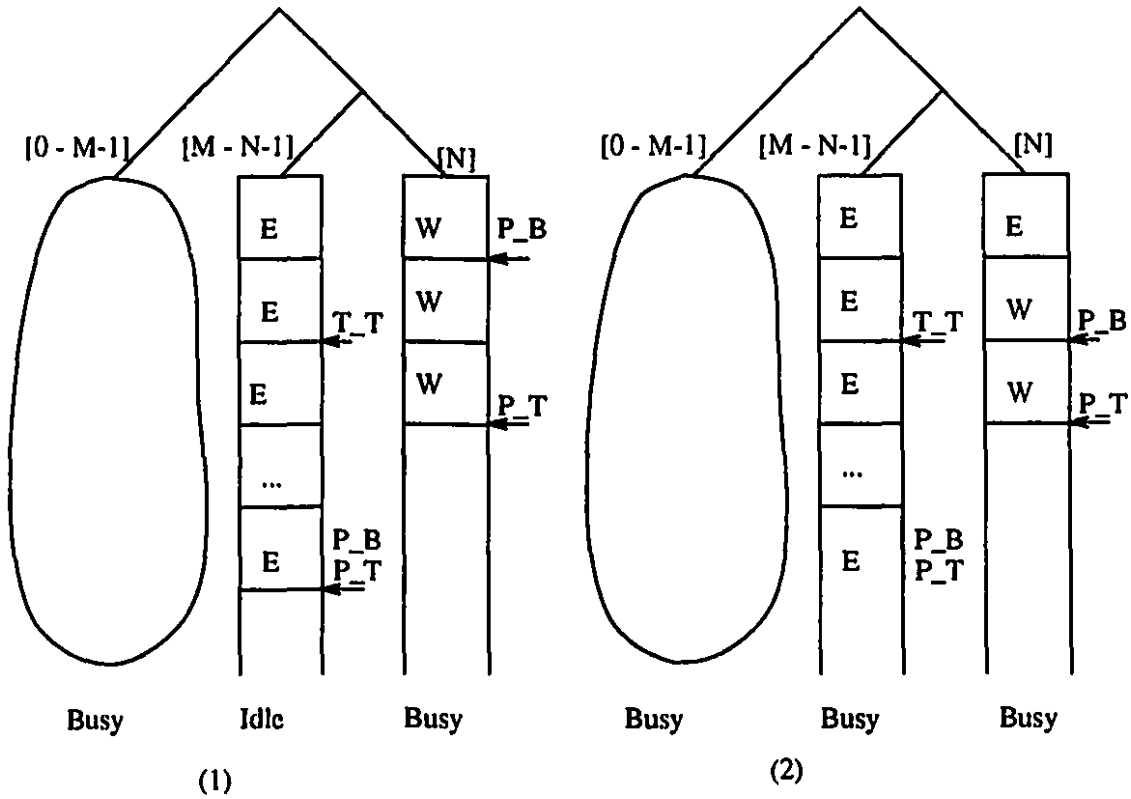

Figure A.6: Case example 6

processors depending on how many times its parent team has been divided. Here is a example to show a situation in which a team have Team-Cooperation with more than one leading processor.

The Case 7, shown in Figure A.7, may be changed into two situations because of the task stealing contention among the leading processors. In this case, the waiting task of team [0] can be stolen by processor 1, the leading processor of idle sibling team [1], or by processor 2 which is the leading processor of idle team [2 - 3]. During the period that they are waiting for a team-join, both of them are checking if there is any task waiting on the task pool of team [0]. Thus, either team [1] or team [2 - 3] may get the task from team [0]. Which team gets it varies upon factors at run-time. It can be affected by the physical location of the processors, bus or network switch [1]

[1]The link between processors on the BBN is called *Butterfly Switch*. Refer to Figure 2.2

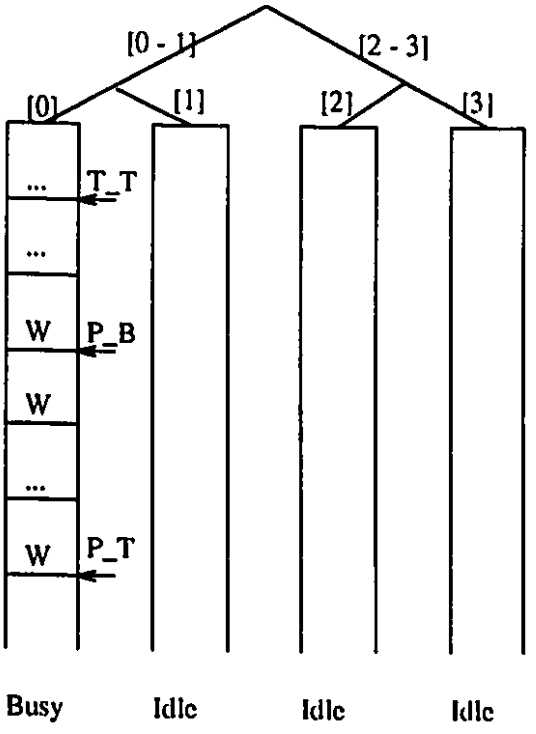

Figure A.7: Case example 7

contention and other factors.

A more general case of Case 7 is presented in Figure A.8. In this figure, part (1) shows a situation when team [0 - N-1] has waiting tasks. All of team [0 - N-1] members are busy. While team [0 - N-1]'s sibling team team [N - N+1], is waiting for team-join.

Although Team [N - N+1] maybe far away from team [0 - N-1], it will take a task from team [0 - N-1] to execute because it is the sibling team of team [0 - N-1], as shown as part (2) in Figure A.8.

Once team [N - N+1] gets its task from team [0 - N-1], it executes it independently from team [0 - N-1]. The team-dividing and team-cooperation may happen within the team, depending on the situation this team encounters. If team [N - N+1] meets a parallel function call, the team-dividing will take place. The team will be divided into two sub-teams, team [N] and team [N+1], since there are two processors in team

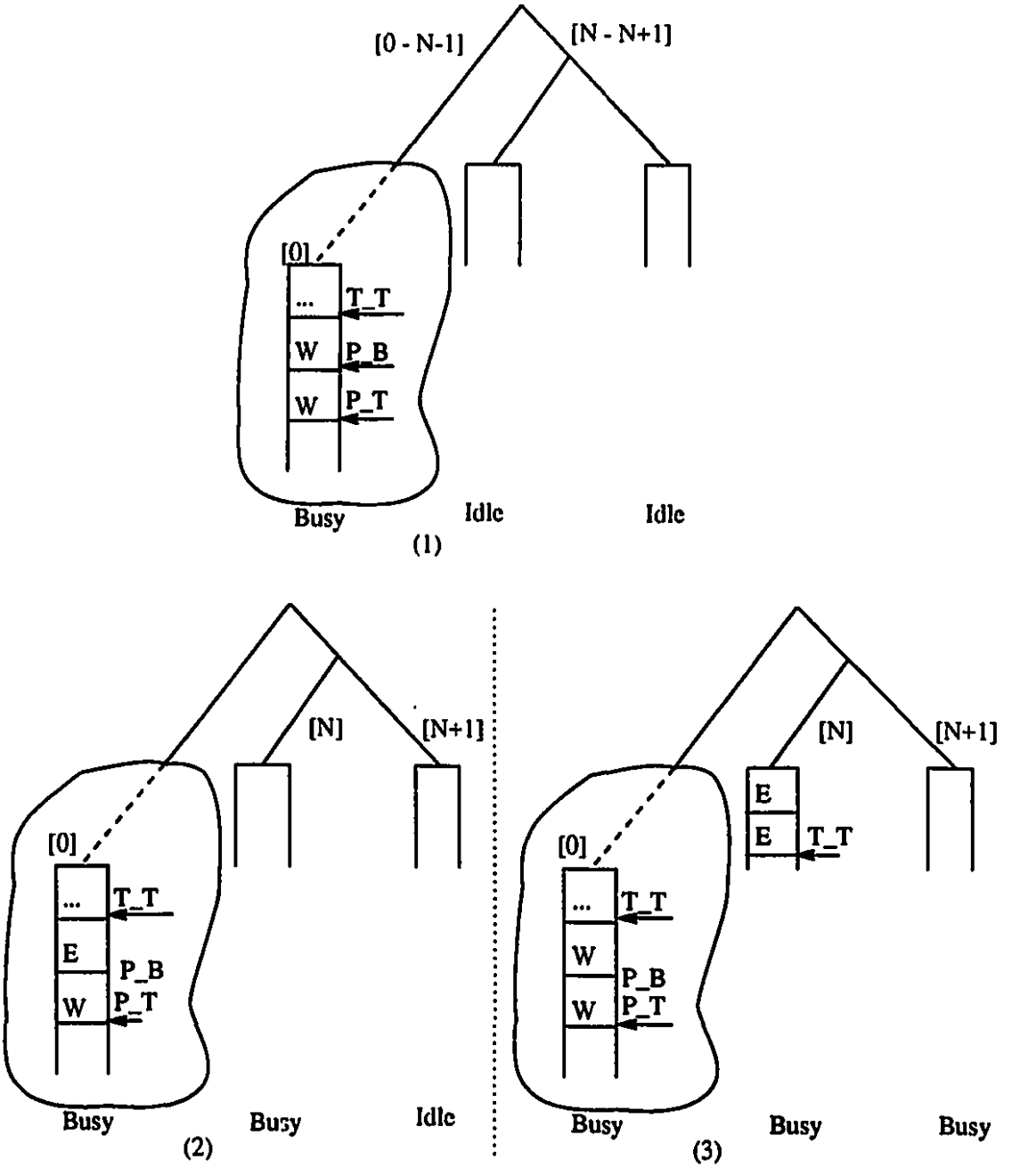

Figure A.8: Case example 8

[N - N+1]. This action is shown as part(3) in Figure A.8. Perhaps there will be team-cooperation happening between these two sibling subteams, team[N] and team [N+1], or between processor 0, which the leading processor of team [0], and processor N, which is the leading processor of team [N].

From the above cases, we can see how the team-cooperation works for the different situations. Each case shows that team-cooperation can rebalance the work load among the teams.

Appendix B

SELSYN-C Code For Binary Tree Evaluation

```
#include <stdio>
   /* Tree node definition */
typedef struct Node {
        int op_code;
        int size;
        double pin;
        struct Node *left;
        struct Node *right;
} node;
typedef node * nodePtr;
#define OPCODE 4
#define FALSE 0
#define TRUE 1
nodePtr treeGen();
int usproc;
int totalproc;
/* Generate a balanced random binary tree */
nodePtr treeGen(size)
int size;
{
nodePtr pointer;
int leftSize;
int flag = FALSE;
long int big = 0x7FFFFFFF;
long rand();
        if ( usproc == totalproc )
```

```
            usproc = 0;
        pointer = (nodePtr) shared_malloc(usproc, sizeof(node));
        usproc++;
           /* Allocate tree nodes on all of the processors */
        if ( pointer == NULL ) {
                perror(" malloc");
                exit(3);
        }
        pointer->size = size;
        if ( size == 1 ) {
                pointer->op_code = -1;
                pointer->pin = (double) rand() / big;
                return(pointer);
        }
        pointer->op_code = (int) (OPCODE * (double)((double)rand() / big));
           /* Generate random node value and operation code */
        size--;
        if ( size > 2 ) {
                leftSize = size / 2;
                if ( leftSize % 2 == 0 ) leftSize++;
        } else {
                leftSize = 1;
        }
        pointer->left = treeGen(leftSize);
        pointer->right = treeGen(size - leftSize);
        return(pointer);
}
/* Recursively evaluate a binary tree */
func(root)
nodePtr root;
{
        if ( root->size > 2 ) {
                int l_weight, r_weight;
                l_weight = root->left->size;
                r_weight = root->right->size;
                func(root->left)@root->left->size
                  // func(root->right)@root->right->size;
                     /* Evaluate left and right sub-tree in parallel */
        switch ( root->op_code )
        {
                case 0:
                        root->pin = root->left->pin + root->right->pin;
                        break;
                case 1:
                        root->pin = root->left->pin - root->right->pin;
                        break;
                case 2:
```

```
                        root->pin = root->left->pin * root->right->pin;
                        break;
                case 3:
                        root->pin = root->right->pin - root->left->pin;
                        break;
        }
         /* Perform operation on return values */
        Free(root->left);
        Free(root->right);
        }
        return;
}
void
/*********/
main(treeSize)
/*********/
int treeSize;
{
int flag;
nodePtr root;
float val;
int *PARA, *PARA0;
int i;
        if ( (treeSize % 2) == 0 ) {
                fprintf(stderr, " Tree size must be odd\n");
                exit(2);
        }
        srand(1);
        usproc = 0;
        totalproc = TotalProcsAvailable ();
           /* Get total number of processors */
        root = treeGen(treeSize);
           /* Generate a binary tree */
        func(root);
           /* Evaluate a binary tree */
        fprintf(stderr, " Evaluation result is %f \n", root->pin);
        Free(root);
        return;
}
```

Appendix C

Source Code of The Run-Time Library

```
#include <us.h>        /* Uniform System */
#include <stdio.h>     /* Standard I/O */
extern current_proc;
#include "include/util.h"  /* Utility macros */
#include "include/task.h" /* Task descriptor */
/* Synchronize with sibling team */
TEAM_WAITING(teamPointer)
taskPtr teamPointer;
{
 while ( teamPointer->stat != DONE ) {
   /* Wait sibling team finish its task */
   taskPtr assistant;
   int *fpoint = my_local_heap;
   assistant = poolStack_base[teamPointer->startProc];
      /* Get the handle of sibling team's task pool */
   if (((assistant != teamStack_top[teamPointer->startProc])
       || (assistant == poolStack_top[teamPointer->startProc]))
       && (assistant->stat == WAIT)){ /* extra */
          /* If there is any waiting task on pool */
       UsLock(taskStack_lock[teamPointer->startProc], 10);
       assistant = poolStack_base[teamPointer->startProc];
       if ((assistant != teamStack_top[teamPointer->startProc])
           || (assistant == poolStack_top[teamPointer->startProc])){
           if ( assistant->stat == WAIT ) {
                /* Steal a task */
             assistant->stat = EXEC;
```

```
                    poolStack_base[teamPointer->startProc]
                     = teamStack_top[teamPointer->startProc];
                } else {
                    poolStack_base[teamPointer->startProc]++;
                }
                *current_pending[teamPointer->startProc]--;
                UsUnlock(taskStack_lock[teamPointer->startProc]);
                   /* Adjust sibling team's pool manipulate pointers */
                (*assistant->func)(assistant->PARA);
                   /* Execute the stolen task */
                UsLock(taskStack_lock[assistant->startProc], 0);
                assistant->stat = DONE;
                UsUnlock(taskStack_lock[assistant->startProc]);
                   /* set state to DONE */
             } else {
                UsUnlock(taskStack_lock[teamPointer->startProc]);
             }
          } else {
             UsUnlock(taskStack_lock[teamPointer->startProc]);
          }
      } /* extra query  to reduce the lock contention */
my_local_heap = fpoint;
 }
return;
}
/* Check if task has been stolen */
POOL_CHECK()
{
 UsLock(taskStack_lock[my_index], 10);
 if (poolStack_top[my_index]->stat == WAIT ) {
   /* Task is still waiting on pool */
   poolStack_top[my_index]->stat =  DONE;
   my_task--;
   if (poolStack_top[my_index] <= poolStack_base[my_index]) {
     poolStack_base[my_index] = teamStack_top[my_index];
   }
   poolStack_top[my_index]--;
   *my_current_pending--;
   UsUnlock(taskStack_lock[my_index]);
   /* Release task from task pool */
   return (1);
 } else {
   UsUnlock(taskStack_lock[my_index]);
   /* Task has been stolen */
   return (0);
 }
}
```

```
/* Synchronize with team which stole the task */
POOL_WAITING()
{
 while (poolStack_top[my_index]->stat != DONE ) {
   /* Stolen task hasn't be finished */
   taskPtr assistant;
   int *fpoint = my_local_heap;
   assistant = poolStack_base[poolStack_top[my_index]->sibLeader];
   /* Get the handle of the task pool of which team stole the task */
   if ((assistant != teamStack_top[poolStack_top[my_index]->sibLeader]
      || assistant == poolStack_top[poolStack_top[my_index]->sibLeader] )
      && assistant->stat == WAIT ) { /* extra */
         /* If there is any task waiting on that team */
      UsLock(taskStack_lock[poolStack_top[my_index]->sibLeader],10);
      assistant = poolStack_base[poolStack_top[my_index]->sibLeader];
      if ((assistant != teamStack_top[poolStack_top[my_index]->sibLeader]
         || assistant == poolStack_top[poolStack_top[my_index]->sibLeader] )
         && assistant->stat == WAIT ) {
            /* Steal back a task */
         assistant->stat = EXEC;
         assistant->sibLeader = my_index;
         if ( poolStack_base[poolStack_top[my_index]->sibLeader]
            == poolStack_top[poolStack_top[my_index]->sibLeader]) {
            poolStack_base[poolStack_top[my_index]->sibLeader]
            = teamStack_top[poolStack_top[my_index]->sibLeader];
         } else {
            poolStack_base[poolStack_top[my_index]->sibLeader]++;
         }
            *current_pending[poolStack_top[my_index]->sibLeader]--;
            UsUnlock(taskStack_lock[poolStack_top[my_index]->sibLeader]);
               /* Adjust the manipulate pointers */
            (*assistant->func)(assistant->PARA);
               /* Execute the stolen task */
            UsLock(taskStack_lock[assistant->startProc],0);
            assistant->stat = DONE;
            UsUnlock(taskStack_lock[assistant->startProc]);
               /* Set the task status to be finished */
      } else {
         UsUnlock(taskStack_lock[poolStack_top[my_index]->sibLeader]);
      }
   } /* extra query to reduce the lock contention */
   my_local_heap = fpoint;
 }
 UsLock(taskStack_lock[my_index], 0);
 arg_list_ptrs[my_index] = poolStack_top[my_index]->PARA;
 poolStack_top[my_index]--;
 my_task--;
```

```
 if (poolStack_top[my_index] < poolStack_base[my_index]) {
     poolStack_base[my_index] = teamStack_top[my_index];
 }
 UsUnlock(taskStack_lock[my_index]);
    /* Release task description, adjust manipulate pointers */
 return;
}
/*
Assign the processors to sub-teams
*/
void factor(taskPointer)
taskPtr taskPointer;
{
 int i, j;
 int sum = 0;
 int p, p_endpoint;
 int interval;
 int l_team, r_team;
 int input_point;
 taskPtr Pointer;
 input_point = my_endpoint;
 Pointer = taskPointer;
 taskPointer->weight = abs(taskPointer->weight);
 sum = taskPointer->weight;
 while ( taskPointer->next != NULL ) {
   taskPointer = taskPointer->next;
   taskPointer->weight = abs(taskPointer->weight);
   sum += taskPointer->weight;
 }
    /* Accumulate weights */
 p_endpoint = my_endpoint;
 interval = (my_endpoint - my_index + 1);
 if (Pointer->weight == Pointer->next->weight || interval<= 2) {
    l_team = interval >> 1;
 } else {
    l_team = (Pointer->weight * interval) / sum;
    if (l_team == 0 && Pointer->weight != 0) {
        l_team++;
    }
    if ((my_endpoint - l_team) < my_index  && Pointer->next->weight != 0) {
        l_team--;
    }
 }
 my_endpoint -= l_team;
 Pointer->teamSize = l_team;
 Pointer->startProc = my_endpoint + 1;
 Pointer->endProc = p_endpoint;
```

```
 Pointer->sibLeader = my_index;
 Pointer->next->sibLeader = Pointer->startProc;
 Pointer = Pointer->next;
 Pointer->startProc = my_index;
 Pointer->endProc = my_endpoint;
 Pointer->teamSize = interval - 1_team;
 my_endpoint = input_point;
    /* Assign sub-teams to two tasks */
 return;
}
/*
Issue sub-tasks
*/
issueTask(taskPointer)
taskPtr taskPointer;
{
 short *p_pending;
 int *p_arglist;
 int input_endpoint = my_endpoint;
 int *fpoint = my_local_heap;
 if (taskPointer->teamSize != 0) {
    if (taskPointer->startProc != my_index) {
       my_endpoint -= taskPointer->teamSize;
       START(taskPointer->startProc,taskPointer);  /* let processor p start */
       taskPointer->next->stat = EXEC;
       (*taskPointer->next->func)(taskPointer->next->PARA);
          /* Execute one task on local team */
       UsLock(taskStack_lock[my_index],0);
       taskPointer->next->stat = DONE;
       UsUnlock(taskStack_lock[my_index]);
       TEAM_WAITING(taskPointer);
          /* Synchronize with sibling team */
       taskPointer->ack = 20;
       while (taskPointer->next->ack != 20 ) {};
             /* release pending lock */
    } else {
       my_endpoint = input_endpoint;
       taskPointer->next->stat = DONE;
       taskPointer->stat = EXEC;
       (*taskPointer->func)(taskPointer->PARA);
       taskPointer->stat = DONE;
    }
 }else {
    taskPointer->stat = DONE;
    taskPointer->next->stat = EXEC;
    (*taskPointer->next->func)(taskPointer->next->PARA);
    taskPointer->next->stat = DONE;
```

```
 }
 UsLock(taskStack_lock[my_index], 0);
 arg_list_ptrs[my_index] = taskPointer->PARA;
 taskPointer->stat = -1;
 taskPointer->next->stat = -1;
 taskPointer->ack = 0;
 taskPointer->next->ack = 0;
 (taskPtr) my_task--;
 (taskPtr) my_task--;
 poolStack_base[my_index] = poolStack_top[my_index]
  = teamStack_top[my_index] = my_task;
 UsUnlock(taskStack_lock[my_index]);
 my_local_heap = fpoint;
    /* Release task descriptor from task stack,
         adjust manipulate pointers */
 return;
}
/*
   Task hoping for child processors
*/
void
child()
{ taskPtr pending;
  int my_exit, procedure_num;
  long oldval;
  my_exit = 0;
  Atomic_add(start_counter,-1); /* signal this processor started */
  while (my_exit == 0) { /* while this processor not killed */
    int *fpoint = my_local_heap;
    WAIT_NONNULL(*my_task_pointer); /* wait for work to do */
    pending = *my_task_pointer;
    UsLock(my_busy_lock,0);
    *my_task_pointer = NULL;
    UsUnlock(my_busy_lock);
    procedure_num = pending->stat;
    my_endpoint = pending->endProc;
    switch (procedure_num)
    {
      case WAIT:
           oldval = Atomic_add_long((long)&pending->stat, -7);
           my_pending_no++;
           (* pending->func)(pending->PARA);
              /* Execute assigned task */
           oldval = Atomic_add_long((long)&pending->stat, 1);
           TEAM_WAITING(pending->next);
              /* Synchronize with sibling team */
           my_pending_no--;
```

```
                pending->next->ack = 20;
                break;
            case EXIT:
                   /* Entire program is finished */
                atomadd32(&pending->ack, -1);
                my_exit = 1;
                break;
        }
        my_local_heap = fpoint;
    }
}
```

Bibliography

[BAD87] E.D. Brooks III, T.S. Axelrod, and G.A. Darmohray. The cerberus multiprocessor simulator. In *Proc. Third SIAM Conference on Parallel Processing*, pages 1–4, December 1987.

[BBN89] BBN Advanced Computers Inc. *Private Communication.* BBN Hotline, 1989.

[Bro89] Eugene D. Brooks III. PCP: A parallel extension of C that is 99% fat free. In Tom MacDonald, editor, *Workshop on Scientific And Numerical Programming in C, ACM Supercomputing 89*, Novermber 1989.

[CC80] S. D. Conte and de Boor Carl. *Elementary Numerical Analysis.* McGraw-Hill Book Company, 1980.

[CGR89] R. F. Cmelik, N. H. Gahani, and W. D. Roome. Experience with multiple processor versions of concurrent C. *IEEE Transaction on Software Engineering*, Vol. 15:335–344, March 1989.

[CT90] K. M. Chandy and S. Taylor. Composing parallel programs. In *Beauty is our Business.* New York: Springer-Verlag, 1990.

[DGNP88] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, 1988.

[Ell86] John. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* ACM Distinguished Dissertations. The MIT Press, Cambridge, Massachusetts, 1986.

[FO90] I. Foster and R. Overbeek. Experiences with bilingual parallel programming. In *Proc. of 5th Distributed Memory Comput. Conf.*, 1990.

[FT89] I. Foster and S. Taylor. Strand: New concepts in parallel programming. In Englewood Cliffs, editor, *Strand: New concepts in Parallel Programming.* NJ: Prentice-Hall, 1989.

[GE90] Rajiv Gupta and Michael Epstein. High speed synchronization of processors using fuzzy barriers. *International Journal of Parallel Programming*, Vol. 19(No. 1):295–307, 1990.

[GPHL88] Mark D. Guzzi, David A. Padua, Jay P. Hoeflinger, and Duncan H. Lawrie. Cedar FORTRAN and other vector and parallel FORTRAN dialects. Technical report, University of Illinois–Center for Supercomputing Research and Development, January 1988. CSRD Rpt. No. 731, Submitted to 1988 International Conference on Parallel Processing, St. Charles, IL.

[Hal85] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[HB84] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw Hill Book Company, New York, 1984.

[Hen90] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, January 1990.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms.* Adam Hilger Ltd., Bristol, 1988.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[JBAJ89] Harry F. Jordan, Muhammad S. Benten, Gita Alaghband, and Ruediger Jakdo. The force: A highly portable parallel programming language. In *Proceeding of the 1989 International Conference on Parallel Processing*, volume II, pages 112–117, 1989.

[Jor87] Harry F. Jordan. The force. In Leah H. Jamieson, Dennis Gannon, and Robert J. Douglass, editors, *The Characteristics of Parallel Algorithms.* The MIT Press, 1987.

[Kat85] M. G. H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI.* MIT Press, Cambridge, 1985.

[KHM89] D. A. Kranz, R. H. Halstead Jr., and E. Mohr. Mul-T: A high-performance parallel lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Languages Design and Implementation*, pages 81–90, 1989.

[Kog81] P. M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill Book Company, New York, 1981.

[KS85] James T. Kuehn and Howard Jay Siegel. Extensions to the C programming language for SIMD/MIMD parallelism. *IEEE*, pages 232–235, 1985.

[LR91] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *Proceeding of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, April 1991.

[Moy88] B. Moy88. *Differences between Mach 100 and Berkeley 4.3 BSD.* BBN Advance Computers Inc., version 1.0 edition, October 1988.

[MSA+85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language reference manual version 1.2. Technical report, Lawrence Livermore National Laboratory, 1985. No. M-146, Rev. 1.

[MvR87] P. Mehrotra and J. van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, Vol. 5:339–361, 1987.

[Sab88] G. Sabot. *The Paralation Model—Architecture-Independent Parallel Programming.* The MIT Press Series in Artificial Intelligence, 1988.

[Sen88] M. Sen. *Programming in C with the Uniform System.* BBN Advance Computers Inc., revision 1.0 edition, October 1988.

[Sri91] Bhama Sridharan. Creation and transformations of the abstract syntax tree. Technical Report ACAPS Note 27, McGill University, 1991.

[Sto87] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley Publishing Company, 1987.

[Tha87] S. S. Thakkar, editor. *Selected Reprints on Dataflow and Reduction Architectures.* IEEE Computer Society Press, Washington, D.C., 1987.

[VR88] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, Vol. 17(No. 4):347–366, 1988.