PC++: An Extension of C++ for Parallel Programming

Myungmun Bae
Department of Computer Science
Oregon State University
Corvallis, OR 97331
baem@mist.cs.orst.edu

A research paper submitted to
Oregon State University
in partial fulfillment of the requirements for the degree of
Master of Science

Major professor : Dr. Tim Budd
Minor Professor : Dr. Vikram Saletore
Other Committee Member : Dr. Curtis Cook

June 1, 1992

Thanks to

my wife, **Heeae Lim**,

and

little daughters, **Minju** and **Minhee**.

Contents

1 Introduction **1**

2 Related Work **5**

2.1 PRESTO 5
2.2 COOL 7
2.3 Concurrent C/C++ 9
2.4 μC++ 10
2.5 RTC++ 12
2.6 Dataparallel C 14

3 Execution Model **16**

3.1 Thread of Control 16
3.2 Relationship 16
3.2.1 Wait - between threads 17
3.2.2 Own - between thread and data object 17
3.3 Guard Expression 17
3.4 State Transition 18
3.5 Deadlock 20
3.5.1 Deadlock Detection 20
3.5.2 System Completion 21

4 Language Extensions **22**

4.1 Classification of Objects 22
4.2 Atomic Object 22
4.3 Dynamic Object 24
4.4 Using Future Types 26
4.5 Parallel Object 30

5 Implementation **33**

5.1 General Translation Technique 33
5.2 Task Management 35
5.2.1 Thread 35
5.2.2 Scheduler 37
5.2.3 Thread Overhead 39
5.3 Memory Management 39

5.4 Translation of Atomic Object Class 42
5.5 Translation of Dynamic Object Class 46
5.6 Translation of Future Type 51
5.7 Translation of Parallel Object Class 54
5.8 How to use PC++ 58

6 Conclusion **60**

A Grammar of PC++ **63**

B Example Programs **66**
B.1 Bounded Buffer Problem 66
B.2 Quick Sort 68
B.3 π Calculation 70
B.4 Matrix Multiplication 71

List of Figures

1 Wait-Relationship . . . 16
2 Own-Relationship . . . 16
3 State Transition . . . 19
4 Deadlock Situation . . . 20
5 Behavior of Future Variable . . . 29
6 Overview of Translator . . . 34
7 Timing - Thread Overhead . . . 40
8 Memory Management Scheme . . . 41
9 Timing - Memory Allocation/Deallocation . . . 43
10 Relationship in Parallel Object . . . 55
11 Timing - Matrix Multiplication . . . 60
12 Timing - Quick Sort . . . 61

1 Introduction

Understanding of techniques that can be used to develop software for various parallel machines has not fully matched the dramatic progress in hardware in recent years. Most parallel programming techniques have been closely tied to specific hardware. Different machines have different primitives and architectures. Programming and porting an application with the machine-dependent features across various machines is difficult and error-prone [21, 37]. This problem is especially amplified when the application is large.

Many ways to cope with the complexity of software development, and to resolve several difficulties, such as machine-independence and abstraction of parallelism, have been proposed and tried.

Ways to express parallelism

In the view of expression of parallelism, two difference approaches have been developed. The first technique, which is called parallelization technique, takes implicit parallelism. The programmer writes programs with implicit parallelism in a sequential language, most commonly FORTRAN. The compiler detects the parallelism and translates the program into a parallel form automatically or semi-automatically. This parallel form must be well-matched to the target machine. This technique is very well developed, but the automatic detection of concurrency is extremely hard, except in relatively regular cases [36, 44].

On the contrary, the other technique emphasizes explicit parallelism with high level parallel constructs or languages. This technique may allow the programmer to find better parallelism, which may be difficult to discover in the parallelization of sequential programs [6]. Sometimes, a good parallel program is quite different from a sequential one, and cannot be derived by source transformations on the sequential source code.

Parallel programming models

Several new programming paradigms for parallel computations, including process-oriented, functional, logic, and object-oriented programming methods, have been proposed [2]. Process-oriented programming models are based on communicating sequential processes [31]. Each process is executed sequentially, and the interprocess communication topology is static. Thus, it is difficult to specify a parallel program using large number of processes [2].

Functional programming relies on value-transferring functions to carry out computations. Because arguments to a function can be evaluated in parallel, functional programming provides ample opportunity for exploiting parallelism. But functions have no state, and therefore are hard to encapsulate history-sensitive behavior. Moreover, blindly evaluating all functions in parallel extends the total elapsed execution time, if the functions do relatively little work. Therefore, the compiler must decide which functions can be parallelized to get maximum speedup, but current compilers can not yet take care of it [6].

The object-oriented[1] programming technique has become exceedingly popular in the past few years [8]. In using the object-oriented programming technique, a program is represented in terms of autonomous objects. The interaction between objects is described by message passing. The object-oriented programming technique can promote productivity and reliability, by means of the abstraction mechanism of encapsulation and inheritance [8]. Programs can be built from pre-existing building blocks as much as possible [8, 39]. Due to the advantage of the object-oriented technique, many object-oriented parallel programming languages have been devised and developed. Examples include POOL [3], Actor [1, 2], and ABCL/1 [34, 48].

[1] It is often more precisely classified as *object-based*, *class-based*, and *object-oriented* [47]. A language will be called *object-based* if it provides linguistic support for *objects* which have a set of operations and states, and interact each other by message passing. A *class-based* language also supports *classes* in addition to objects. Finally, the *object-oriented* language is class-based and additionally requires *class inheritance* which is a mechanism for composing the interface of one or more inherited classes with the interface of the inheriting class.

Types of parallel computation

Parallel computation can be categorized into two patterns [20, 21]. First, *divide and conquer computation* or *tree computation* involves the concurrent activity which breaks the problem into several simpler independent subproblems, executes them in parallel, and waits for the results of the subcomputations.

Second, *cooperative problem solving*, or *crowd computation* involves a set of cooperating processes. Each process is an independent computational entity which communicates and synchronizes with each other processes. Data parallelism can be classified as a crowd computation, in that simultaneous operations are performed across large set of data, but it has different view in that the data parallelism comes from large set of data, rather than from multiple threads of control or processes.

My approach

As an object-oriented language, C++ provides data abstraction mechanisms, but C++ does not provide parallel programming facitilies in the language constructs. The primary goal of PC++, which is an extension of C++ and was implemented on Sequent Symmetry,[2] is the linguistic support to express explicit parallelism in more highly-abstracted manner, without knowing the underlying architecture and low-level primitives.

In general, languages with high abstraction concepts has the possibility to check the correct use of the primitives. The compiler can find syntax errors and a certain number of semantic errors. It can reduce the time and effort required to develop and maintain the program.

The abstraction mechanism of parallelism in PC++ is class-based, since the expression of parallelism is described and encapsulated by the class definition. PC++ classifies the data object classes into the atomic, dynamic, and parallel object classes. Among the object classes, the dynamic and parallel

[2] Symmetry is a trademark of the Sequent Computer Corp.

object classes invoke automatic parallel execution, when a member function of the object is called. In addition to the parallel execution, the parallel object also supports virtual topology and way to express data parallelism.

In summary, PC++ supports tree computation, crowd computation, and more general concurrent computation, such as reader-writer style computation with specific condition like guard command [17].

The remainder of this paper is organized as follows: Section 2 shows similar prior work. Section 3 describes how the parallel execution is modeled. The syntactic and semantic view of the each class-based abstraction is explained in Section 4. Section 5 shows the underlining runtime environment and its roles, and how the translator works by showing the examples. Finally Section 6 discusses the performance of two examples, and concludes this work.

2 Related Work

Several researchers have tried to abstract concurrency and synchronization in C and C++. Most related work that will be explained is involved with object-oriented approaches to specify concurrent activity in C++. But these do not attempt to add data parallelism. Dataparallel C is an imperative approach that extends C to provide data parallelism. The most following example programs are cited from the related papers.

2.1 PRESTO

PRESTO [7, 19] is a programming system with a set of pre-defined object classes. The programmer uses the *Thread* class to specify parallel execution. PRESTO also provides several classes; *Spinlock*, *AtomicInt*, *Monitor*, and *Condition*. Every synchronization and concurrency operation must be specified explicitly, since PRESTO is not a new language, but is instead a run-time library.

The following example, matrix multiplication, illustrates the use of the PRESTO features.

```
class Vector {
        int     num, *array;
    public:
        Vector(int n);                  // constructor
        int     operator[](int col);
        void    innerProduct(Vector &v, int *res);
};
class Matrix {
        Vector  *mat;
        int     size;
    public:
        Matrix(int n);
        Matrix  *multiply(Matrix &m);
```

```
        void    transpose();
        Vector& operator[](int row);
};
Matrix *Matrix::multiply(Matrix &m)
{
    int     i, j, k;
    int     n_thread = size * size; // # of threads
    Matrix *mtmp = new Matrix(size);
    Thread **waiter = (Thread **) new int[n_thread];
    m.transpose();              // transpose m
    for(k=0, i=0; i<size; i++)
        for(j=0; j<size; j++) {
            Thread *t = new Thread("mult");
            t->willjoin();
            t->start( this,                 // object
                      mat[i].innerProduct, // method
                      m[j],                 // parameter
                      &mtmp[i][j]);         // result
            waiter[k++] = t;
        }
    // wait all threads
    for(i=0; i<n_thread; i++) waiter[i]->join();
    m.transpose();              // set to original
    return mtmp;
}
#define N 100
main()
{
        Matrix M1(N), M2(N), *M3;
        // input M1, M2
        M3 = M1.multiply(M2);
        // output M3
}
```

The function main() inputs two matrices and computes a third matrix which is the product of the two input matrices. A thread, which is the basic

unit of execution, consists conceptually of a program counter and a stack, just like a process. As we see in this example, `start()` is an operation defined for parallel execution of a function with parameters.

In matrix multiplication, every element of the resultant matrix can be computed in parallel. Thus threads are created and started in the inner loop. In addition, we need some mechanism to control threads, since all threads must be finished before the resultant matrix can be returned. PRESTO provides `willjoin()` and `join()` to handle this situation. The function `willjoin()` must be specified before the thread is started, and then the function `join()` will wait for the completion of the thread. The thread with `willjoin()` will not be destroyed until the function `join()` is called, even if the thread is completed. In contrast, the thread without `willjoin()` will be destroyed automatically when it is completed.

The function `start()` is essentially untyped, since it can be used to start a thread within any object's member function having any number of any type of parameters. In other words, overloaded functions to be started by the function `start()` will not work, since the compiler cannot discern the types of the arguments to the function at compile time.

2.2 COOL

COOL [13] introduces concurrency and synchronization in C++. The parallelism is encapsulated as part of the implementation of a class, transparent to users. Every possible asynchronous invocation on an object is specified by the *parallel* function. The synchronization can be expressed in two ways; by specifying *mutex* function and *future types*.

```
class array {
        int count, *aptr;
    public:
        array(int n, int *p) { count=n; aptr=p; }
        parallel int$ sort();
};
```

```
parallel int$ array::sort()
{
        if(count < MinSize) {
                // serial algorithm
        } else {
            int     half = count/2;
            array *left = new array(half, aptr);
            array *right = new array(count-half,aptr+half);
            int$ done = left->sort();    // in parallel
            parallel~ right->sort().
            waitset(done);               // wait
            // merge two lists
        }
}
```

The above example, a merge-sort based on the divided-and-conquer algorithm, shows use of the parallel function and the future variable. The list to be sorted is split into two parts, each of which can be sorted independently and concurrently, and then the sorted halves are merged.

The variable `aptr` in the class `array` points to the list of integers to be sorted, and the variable `count` holds the number of the integers. If `count` is fewer than `MinSize`, then some serial algorithm is used. Otherwise `sort()` creates two `array` objects and initializes them. The left list is sorted in parallel by invoking `sort()` which creates a new thread, and is synchronized via a future variable `done`, since `sort()` is defined as a parallel future function. The future type is defined by appending `$` to the type `int`. The right half is sorted in sequential, because the attribute `parallel~` enforces the sequential execution without creating a new thread. Therefore the right half is sorted on the current thread, and the left half is sorted on a new thread. Consequently sorting both halves is in parallel. The function `waitset()` is applied to a future variable to wait for the termination of the related future call.

COOL also provides low level locking mechanisms; *Spinlock* and *Blocklock*. COOL is similar to PC++ in terms of parallel invocation and future type synchronization. But it does not have any constructs for data parallelism or crowd computation.

2.3 Concurrent C/C++

A Concurrent C [22, 24, 25] program may consist of a set of *processes* that execute in parallel. A process definition has two parts: a type (*specification*) and a body(*implementation*). Concurrent C processes interact with each other by means of *transactions* or *extended rendezvous*[3] which allows bidirectional information transfer at a rendezvous event. The specification and behavior are similar to those of ADA.[4]

The following simple program introduces the basic concepts of Concurrent C. The program reads data, processes the data, and then prints the results. A Concurrent C program can be constructed as follows: one process (*producer*) reads the data from the terminal, and sends them to another process (*consumer*). The *consumer* process converts all lower-case characters to upper-case, and prints the data on the standard output. This problem falls into the generic *producer-consumer* problem class.

```
#include <stdio.h>
process spec consumer()
{
        trans void put(int);
};
process spec producer(process consumer);
process body consumer()
{
        int c,a;
        for(;;) {
            accept put(a) { c = a; }
            if(c == EOF) break;
            putchar(islower(c) ? toupper(c) : c);
        }
```

[3] *Rendezvous* is the interaction between two independent tasks via synchronous message passing when one task has called an entry to the other. The interaction is performed first by synchronizing, then by transferring information, and finally by continuing their individual activities. [11, 22, 24, 42]

[4] Ada is a trademark of the US Department of Defense (Ada Joint Program Office).

```
}
process body producer(cons)
{
        int c;
        do {
            cons.put( c=getchar() );
        } while( c != EOF );
}
main()
{
        create producer(create consumer());
}
```

The example shows two process types: *producer* and *consumer*. The specification part of the *consumer* process has the transaction `put()`, which can be called by other processes to send a character to a process of type *consumer*. The body of *consumer* contains an `accept` statement which is how a process accepts a transaction call. Whenever a *consumer* process meets an `accept` statement during execution, the process waits until a related transaction is called by another processes. The *producer* process reads characters from the standard input and calls the transaction `put()` of the *consumer* to send a character. The function `main()` creates instances of *producer* and *consumer* processes. After the creation of a process, the body of the process starts to run automatically and independently with other processes.

Classes and processes in Concurrent C++ [23] are both abstraction facilities, and can be used to implement any abstract data types.

2.4 μC++

μC++ (*micro-C++*) [9] introduces concurrency into C++ by adding new types and statements. The inclusions correspond to the ideas of a coroutine, monitor, coroutine-monitor and task on shared-memory uniprocessor and multiprocessor computers. It provides static type checking as in C++, and

implicit mutual exclusion in the language construct. The way to achieve concurrency is similar to ADA and Concurrent C.

A task is an object with its own thread-of-control, whose public member routines provide *mutual exclusion.*

```
uTask T {
    protected:
        void    main();
    public:
        void    request_a();
        void    request_b(int);
};
void T::main()
{
        ...
        for(;;) {
            uAccept(request_a) {
                // service for request_a
            } uOr uAccept(request_b) {
                // service for request_b
            }
        }
        ...
        uDie;   // explicit terminate
}
main()
{
        T *tp = new T;  // create a task
        ...
        tp->request_a();
        ...
        tp->request_b(100);
        ...
        // wait for the termination
        delete tp;
}
```

The task type has one distinguished member, namely `main()`, in which the new thread starts execution. A user interacts with a task's `main()` member indirectly through its member routines; in this case, `request_a()` and `request_b()`. The acceptance of the user call to a *task* object is done by `uAccept()` statement.

When a task `tp` is created, a new thread is created and executes the member routine `main()` concurrently. The task `tp` terminates when its `main()` routine terminates, when its destructor is accepted, or when the statement `uDie` is executed.

The `uAccept()` statement has a member function as the argument. Therefore the member function to be used in the `uAccept()` cannot be overloaded.

2.5 RTC++

RTC++ [35] is an extension of C++ for programming real-time applications. RTC++ introduces an *active* object. An active object has by default a single thread of control. A user can specify multiple threads as *member threads* which are defined in an *activity* part of a class definition. The member threads are either *slave* or *master*. A slave thread is an execution unit related to a member function or a group of member functions. A master thread is intended to use a background thread within an active object. A guard expression may be defined in a member function definition to postpone a request until the condition becomes true.

```
active class Example {
    private:
        Region  *critical;
        int      count;
        int      background();
    public:
        int read(char *data, int size) when(count > size);
        int write(char *data, int size);
```

```
        int open(), close();
    activity:
        slave[5] read(char *, int);
        slave[5] write(char *, int);
        slave    open(), close();
        master   background() cycle(;;0t30m;);
};

int Example::read(char *data, int sz)
{
    ...     // [1] non-critical region
    region(critical) {
            // [2] critical region
    }
    ...     // [3] non-critical region
}

main()
{
        // task v is created with priority 4
        Example *v = new Example priority 4;
        ...
}
```

In this example, when an instance of class `Example` is created with priority 4, a new master thread for `background()` is created and executed in every 30 minutes. The class `Region`, that is used in the class `Example`, is a predefined active class to provide critical region.

The creation of threads is dynamically performed when the call to the active object is made. But its execution is bounded to the *activity* declaration. One thread only is responsible for the `open()` and `close()` requests. The member function `read()` can be preempted by up to 5 clients whose priorities are higher than the current execution of the `read()` function.

The function `read()` consists of the sequence of the non-critical region, critical region, and non-critical region. Suppose that one thread enters the

critical region and at that time a new `read()` request is coming where the sender's priority is higher than the previous sender's one. The new thread begins to execute the function `read()` with higher priority, and the former thread is suspended. But since the former thread is in the critical region, the higher priority's thread can not enter the critical region, and then is blocked. So the former thread executes again until exiting the critical region. After that, the higher priority's thread is resumed.

2.6 Dataparallel C

Dataparallel C [28] is a variant of the original data-parallel programming language C*[5] designed for Connection Machine.[6] The goal of Dataparallel C is different from the goals of the previously explained systems. Dataparallel C focused on data parallel programming language based on C. Dataparallel C has the *domain* construct to specify the data distribution over *virtual processors*. The *domain* construct is simliar to *struct* construct in C. The parallel code is surrounded by a *domain select* block. A *domain select* has the effect of *"waking up"* all virtual processors to perform the block in parallel.

Following example computes an approximation to π by calculating the area under the curve $4/(1+x^2)$ between 0 and 1 using numerical integration.

```
#define INTERVAL         10000
#define ID               (this - chunk)
domain span {
        char dummy;
} chunk[INTERVAL];
main()
{
        double pi;
        /* domain select */
        [domain span].{
```

[5] C* is a registered trademark of Thinking Machine Corporation.

[6] Connection Machine is a registered trademark of Thinking Machine Corporation.

```
            double width = 1.0 / INTERVAL;
            double x = (ID + 0.5)*width;
            double height = 4.0 / (1.0 + x*x);
                 /* reduction */
            pi = (+= (width * height));
        }
        printf("pi = %lf\n",pi);
}
```

To compute π, we divide the curve $4/(1+x^2)$ into small rectangles and sum the area of them. Each rectangles represents a virtual processor. The width of every rectangle is the same. The height of each rectangle is chosen so that the curve intersects the top of the rectangle at its midpoint. The operator `+=` is a *reduction* operator to accumulate the areas which are computed by multiplying `width` and `height`.

One of drawbacks of Dataparallel C is that it does not support dynamic creation of virtual processors. Its size and shape must be determined at compile time. The *domain* construct and *virtual processor* in Dataparallel C is similar to the *parallel* class in PC++, but PC++ supports dynamic creation of parallel objects.

3 Execution Model

In this section, I will explain the characteristic and behavior of the thread, which is an execution unit, the relationships between the data and the threads, or between threads, and finally the deadlock problem which may arise in this execution model.

3.1 Thread of Control

To invoke parallel execution, a new task must be created, resources allocated, and must be scheduled to run. Creating a new normal process requires a large overhead since it requires copying the whole address space.

A way to reduce the overhead is to perform explicit task management. When a new task is required, the task is created and put into the global task queue, then the task will be light-weight since it does not have to copy the whole address space but only needs allocate its own stack area, although the creation of a normal process needs the duplication of whole address spaces, including code, data and stack, and the maintenance of the internal system tables, such as process table. The light-weight task is called a *thread-of-control* in PC++. It is an unit of execution with its own stack, can be migrated over processors and preempted by other threads.

3.2 Relationship

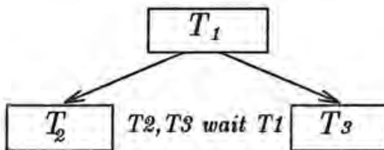

Figure 1: Wait-Relationship

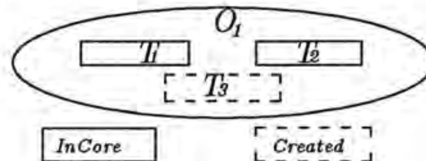

Figure 2: Own-Relationship

3.2.1 Wait - between threads

If a thread waits for the completion of another thread, the execution of the former thread must be blocked until the latter thread is done. The relationship can be expressed by a graph as Figure 1. In the graph, threads T_2 and T_3 can not be executed until the thread T_1 is finished.

3.2.2 Own - between thread and data object

Every thread runs on a data object which is used during the execution of the thread. If several threads update the state of a same data object at the same time, we can not predict the state of the data object after the threads are done. This results in a *side-effect.* To prevent this situation, PC++ normally allows only one thread on a data object. A thread, which is allowed only one on a data object, is called *impure thread.* But if there is no side-effect between execution of the threads on the same data object, we can specify the threads as *pure threads.*

In other words, we can say a data object *owns* a set of threads which are *pure* or *impure*, if the threads run on a data object as in Figure 2. In the graph, threads T_1 and T_2 are already started, and thread T_3 is created but not started. Threads T_1, T_2, and T_3 use the same data object, O_1. We say the data object, O_1, owns the threads T_1, T_2, and T_3.

This characteristic of thread allows multiple-readers and single-writer style programs like airline reservation systems. If the threads on a data object are *pure*, the threads can be executed at the same time. But only one impure thread on a data object can run at a time.

3.3 Guard Expression

A guarded command [17, 31] is used as a building block for alternative and repetitive constructs that allow nondeterministic program components.

The *alternative* construct is written by enclosing it by the special bracket pair **if** $\cdots$ **fi**. The general syntax is as follows:

$$\textbf{if } S_1 \; [] \; S_2 \; [] \cdots [] \; S_n \textbf{ fi}$$

S_1, S_2 and S_n are the guarded commands, each of which consists of a guard (*boolean expression*) and a set of statements. If none of the guards is true in the initial state, the program will abort; otherwise an arbitrary guarded list with a true guard will be selected for execution.

The *repetitive* construct is written down by enclosing a guarded command set by the special bracket pair **do** $\cdots$ **od**. The general syntax is as follows:

$$\textbf{do } S_1 \; [] \; S_2 \; [] \cdots [] \; S_n \textbf{ od}$$

It will only terminate in a state in which none of the guards is true. If one or more guards are true, a new selection for execution of a guarded list with a true guard will take place.

The guard expression in PC++ is a variant of the Dijkstra's guarded command. A guard expression may be defined in a member function definition in order to control concurrency. Although a member function with a guard expression is called, the execution is postponed until the expression becomes true. The characteristic is similar to the Dijsktra's *repetitive* construct since the guard expression will be evaluated over and over again until the expression becomes true.

3.4 State Transition

The lifetime of a thread can be divided into a set of states with a certain characteristic. When a thread is created with a function to be executed, argument list, and a guard expression which is a condition to start to execute, the thread is placed in the *created* state.

When the guard expression becomes true, the thread moves into the *ready* state. Every processor picks a thread from the *ready* state, and executes its function body. During the execution of function body, if it needs to wait

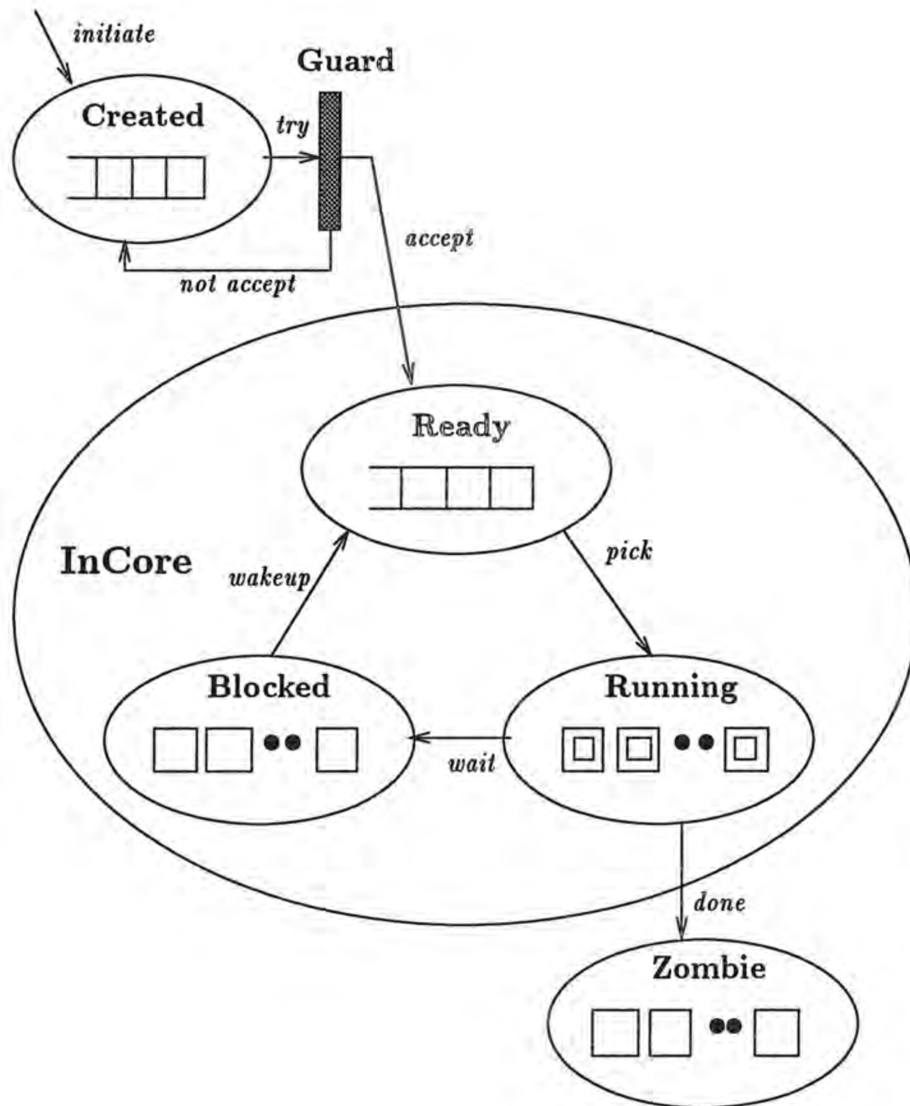

Figure 3: State Transition

for other threads, the thread is put on the *blocked* state and the process will be idle. The idle process picks a thread from the *ready* state and do it again.

If a thread is done and there is a blocked thread waiting for it, the blocked thread is changed to the *ready* state, and the completed thread moves into the *zombie* state. A thread in the *zombie* state will be destroyed automatically after cleaning up memories related to the thread. Figure 3 shows the overview of state transition and the lifetime of threads.

3.5 Deadlock

To explain deadlock, let T_n^w be a *impure* thread, and T_n^r be a *pure* thread. In Figure 4, T_1^w waits for the completion of T_2^w, T_2^w waits for the completion of T_3^r. But an object O_1 owns T_1^w, and T_3^r. So, T_1^w and T_3^r can not execute simultaneously since T_1^w is *impure* and T_3^r is *pure*. Thus T_3^r must wait for the completion of T_1^w. It implies circular wait condition which is one of deadlock condition [40].

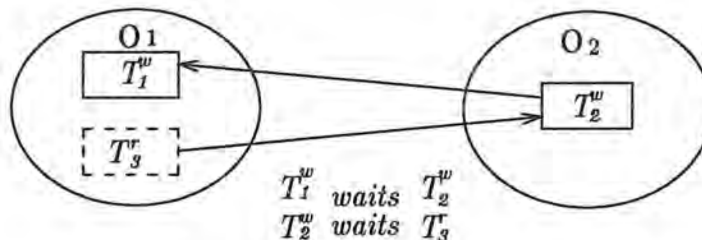

Figure 4: Deadlock Situation

3.5.1 Deadlock Detection

If there is no threads in *ready* state and *running* state, but some threads in *blocked* state, or some threads, in *created* state, which can not accepted any

way, then the system must be deadlocked.

Proof: Any threads in *ready* or *running* state can be executed in any order and at any time. But any threads in *blocked* state are waiting for the completion of some other threads. So if there are no threads in *ready* or *running*, no threads can be completed. Therefore any threads in *blocked* state can not be resumed.

3.5.2 System Completion

If there are no threads in any states, then all processing is done.

Proof: It is obvious. A program can be represented by a set of threads since each thread is an execution unit. In other words, a program can be executed only by means of threads, and a thread corresponds to a piece of the program. Therefore if a thread is done, a piece of the program is finished. Moreover if all threads needed to complete a program are done, the program is finished. It means that the program is done if there are no threads in the system.

4 Language Extensions

4.1 Classification of Objects

An object is a self-contained entity which has data and a set of associated operations, that manipulate the data, and that interact with the outside world exclusively through some forms of message passing [6, 5]. In addition to data and operations, an object possibly has a thread-of-control so that it can become an autonomous execution unit. A class is a template of objects, a kind of an abstract data type. Every object is an instance of some class.

In PC++, an object encapsulates data and operations, and a thread-of-control in case of dynamic object. An object without a thread-of-control is either a *passive* or *atomic* object. A *passive* object is just an ordinary object as in C++. The only difference between *passive* objects and *atomic* objects is that the *atomic* object guarantees mutual-exclusion between operations on an object.

An object with a thread-of-control is either a *dynamic* or *parallel* object. A *dynamic* object has one thread-of-control basically, but possibly can have multiple thread-of-control if those threads are specified as *pure*, *side-effect free* threads. The *dynamic* object can be migrated over processors and pre-empted. A *parallel* object is really a set of *dynamic* objects with a particular topology like *mesh* structure. An invocation of a parallel object results in the multiple invocations of dynamic objects.

4.2 Atomic Object

Only one thread can access the *atomic* object at any one time. This characteristic is similar to the idea of a *monitor* [30]. When a thread attempts to access the object that is already accessed by other threads, the thread is left spinning until the accessing thread is exited from the *atomic* object.

The mutual-exclusion mechanism can be implemented by busy waiting

on a shared memory machine. But the calls on lock() and unlock() are hidden from the language constructs in PC++, so the programmer does not have to specify lock() and unlock() explicitly to achieve mutual-exclusion on a object. The spinlock is less expensive to acquire and release the lock, and more efficient than relinquishing lock if the time of mutual-exclusion body, or critical section, is less than the time to relinquish and reacquire a processor.

Unlike the dynamic object, the atomic object does not support guard expressions since these may cause degradation of the system due to spinning. Moreover, an atomic object does not have a unique thread. The thread of its execution is the same as the caller's thread. Thus there is not any additional overhead required to access the atomic object in comparison to the dynamic object since it does not have to create a new thread.

The definition of an atomic object is as following.

```
atomic class X {
     ...
   public:
     [impure] int f();
     pure     int g();
};

int X::f() { ... }
int X::g() { ... }
```

`X::f()` is defined by *impure* member function, and `X::g()` is defined by *pure* function. Only one *impure* function can be executed on an atomic object, or several *pure* functions can be executed. But both of *impure* and *pure* functions can not be executed at a same time.

The qualifier *impure/pure* can be placed in front of the member function declaration. If there is no explicit *impure/pure* qualifier, the function is assumed as an *impure* function.

4.3 Dynamic Object

A dynamic object encapsulates the internal data, member functions as legal operations on the data, and even more the thread-of-control. It implies that creating a dynamic object causes creating a new thread. The threads can be multiple in case of *pure* functions.

A dynamic object supports two type message passing mechanisms; *synchronous* and *asynchronous*. An asynchronous message to a dynamic object results in asynchronous execution. The sender can do its own work without waiting the completion of the invocation. Unlike asynchronous invocation, a synchronous message causes the sender to wait for the completion of the invocation.

Furthermore, the dynamic object supports *future* type synchronization. In Multilisp [26], a future call results in asynchronous function call, and the calling process does not wait for the completion. The result of the future is transparent to the programmer. But the result of the future invocation in PC++ is bounded to the future variable. The future variable may be treated as a normal type variable. When the value of the variable is needed, the execution waits for the future invocation.

The future type invocation can encapsulate the details of synchronization specification since we do not need any explicit construct to wait the call. The waiting for the future invocation is automatically performed when it is needed.

A task may want to wait for any of messages from other tasks, rather than one specific message. If it is unknown in advance which message will be available first, such behavior is called *nondeterminism* [5]. To avoid this problem, most languages have introduced some expression as a controlling mechanism based on the *guarded command* [17]. Examples include the *select statement* in CSP [31], or the *guarded horn clause* in Concurrent Prolog. The dynamic object also supports guard expression using a *when* statement following by function declaration. A guard expression may consist of primitive data types such as integer, primitive operations such as addition, internal variables, and message variables as in the argument declaration. But a side-

effective guard expression such as updating the local states must be avoided since it causes nondeterministic result of the invocation because the number of evaluation of the expression can not be expected.

Like the atomic object, the *impure/pure* qualifier can be put in front of a function declaration. Its behavior is the same as multiple readers and single writer, as in case of the atomic object.

```
dynamic class X {
       ...
  public:
       // asynchronous invocation
       [impure] [async] void f(arg-decl) [when(expression)];
       // synchronous invocation
       [impure] [sync] int g(arg-decl) [when(expression)];
       // future invocation
       [impure] future int h(arg-decl) [when(expression)];
       // pure invocation
       pure [sync|async|future] int p(arg-decl)
                                        [when(expression)];
};

void X::f(arg-decl) { ... }
int  X::g(arg-decl) { ... }
int  X::h(arg-decl) { ... }
int  X::p(arg-decl) { ... }

main()
{
        X *x = new X;           // create
        future int fv;

        x->g();                 // sync call
        async x->g();           // async call
        fv = future x->g();     // future call
}
```

If there is no *impure/pure* qualifier in the member function declaration, the function is assumed to be an *impure* function. If the return type of the function is *void*, the function is assumed as *async* by default. If the return type of the function is not *void* and there is no *async* or *future* qualifiers, the function is assumed as a *sync* function by default.

Moreover, we can specify the type of call such as *sync*, *async* or *future* even if the type is different from the type of declaration. In the above example, `X::g()` is specified by *sync* type of the call, but we can change the default type call by any specific type of the call as shown above.

4.4 Using Future Types

The future type will receive a result of an asynchronous invocation. Its state is either *valid* or *invalid*. When an instance of a future type is created, the state of the variable is set to *valid* initially because there is no threads to be waited. If a function call is invoked with a future variable, the future variable is set to *invalid* until the invocation is done. A future type can be built from a primitive type such as integer or float, and a user defined type.

The legal operations on the future type is the same as the operations of its base type, and an additional operation which is *wait()* to suspend until the variable is determined. The operations on the future variables can be categorized into *strict* and *non-strict* operations.

The *non-strict* operations use the variables but do not need the value of the variables. Simple assignment and copy between future variables are the example of the *non-strict* operations. These operations do not wait for the completion of the call related to the variables. However, the other *strict* operations, like addition and multiplication, which need the actual value of the variable must block until the value is determined.

Many future variables can be possibly bound to an undetermined value which is the result of a future call. The compiler does not know how many future variables are related to a future call, but can know what are the future variables and when the *strict/non-strict* access is used. Therefore

the compiler can generate code ensuring that all accesses to future variables check the state of the variables with a state information which is a structure to maintain the lifetime of the future variables.

One possible way to implement the future variable is by introducing *reference count* and maintaining two types of future objects: *global* and *local* future objects. The *global* future object is shared by all related *local* future variables. It has the actual area of the value of the future variable, and *reference count* which can tell how many future variables need the value. The *reference count* can prevent the garbage collection since the *global* object can be deleted automatically when the *count* reaches to zero. The *local* future object has a value if the variable is valid, and a pointer to the *global* object.

When a future call is invoked, a *global* future object is created, and the *reference count* is set to zero. If a local future variable is assigned to the call or to the other local future variable, the reference count in the global object is incremented, and the pointer in the local object is set to the global object. When a local future object is resolved, the reference count in the global object is decremented. Eventually the reference count reaches to zero, then the global object will be destroyed since no local variables are related to the global object. A new future is defined by putting *future* qualifier in front of a base type.

```
dynamic class X {
        ...
   public:
        future int f();  // future type invocation
};

main()
{
     X *x = new X;
     future int qx, qy, qz;
     int i, j;

     qx = x->f();    // (1)
     qy = qx;        // (2)
```

```
    qz = qy;          // (3)
    i = qy + 1;       // (4)
    j = qx;           // (5)
    qz = i;           // (6)

    qx = x->f();      // call again
    qx.wait();        // explicit wait
};
```

In the above program, `qx`, `qy`, and `qz` are the future variables, and `X::f()` is also defined as a future function. The Figure 5 shows the behavior of the variable. The *valid* variable is represented by the *black dot*, and *invalid* variable by the *white dot.*

At (1), `f()` is invoked as an asynchronous manner with future type with future type result, and `qx` is set to *invalid* since `qx` is waiting for the call `f()`. So `qx` will be *valid* if the call `f()` is done.

At (2), `qy` is assigned to `qx` as a *non-strict* operation, and then `qy` is also waiting for the call `f()`.

At (3), `qz` is also waiting for the call `f()` just like `qy`.

At (4), the statement `i = qy + 1` means `i` is assigned to the value of `qy` *plus* 1. At this point, `i` needs the value of `qy`, but `qy` will be determined after the call `f()`. Therefore the execution will suspend until the call `f()` is done since *plus* operation is a *strict* operation.

At (5), `j` is assigned to `qx`, but `j` is not a future type. In other words, this statement is that `j` is assigned to the value of `qx` as a *strict* operation.

At (6), `qz` does not need the value of the call `f()`, but only needs the value of normal variable `i`. Thus `qz` is now *valid* instead waiting the call `f()`.

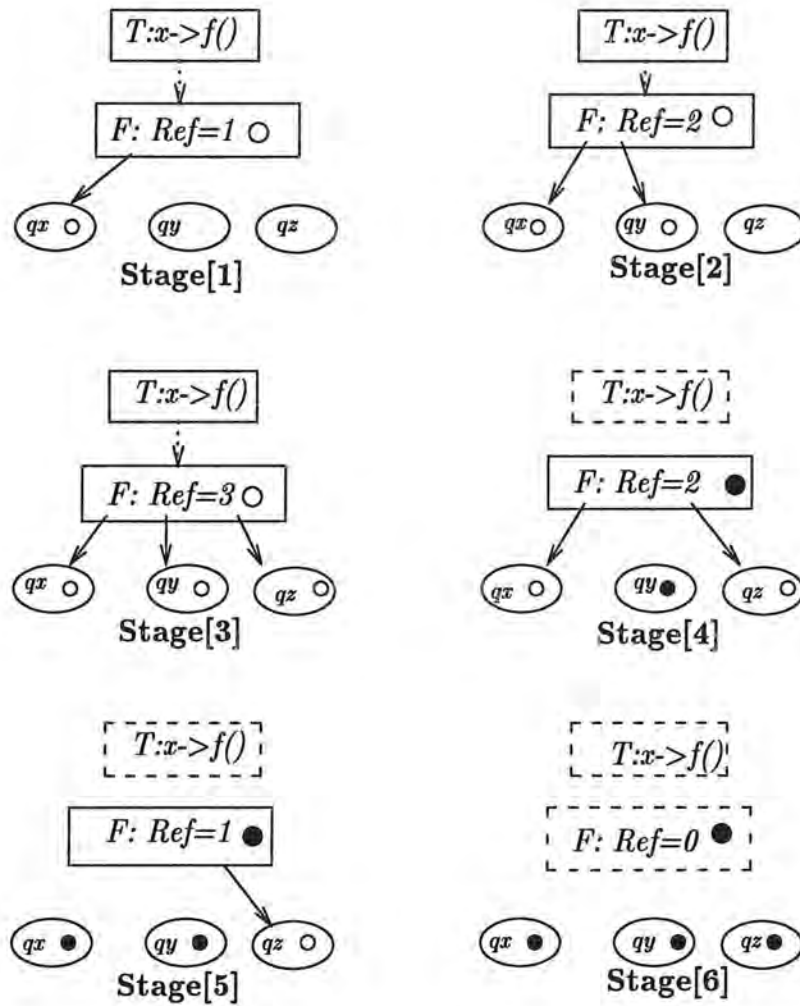

Figure 5: Behavior of Future Variable

4.5 Parallel Object

A parallel object is the construct used to achieve *data parallelism.* The data parallelism can be explained as simultaneous operations on large sets of data. It can also be implemented in terms of *control parallelism* involving multiple streams of control [29].

Data parallelism is well suited for numerical computations such as matrix computations. Moreover, the speedup of the data parallelism does not rely on the concurrency of the code but on the size of data to be applied. The feature makes more speedup than that in the control parallelism.

Research related to data parallelism is very rapidly growing. The data parallelism has historically close relation to the SIMD (Single Instruction Multiple Data) machines like Connection Machine. The languages for the data parallelism are also developed on MIMD (Multiple Instruction Multiple Data) machines as SPMD[7] (Single Program Multiple Data) style programming languages. The SPMD style programming is to write a single program which is applied to multiple data.

A parallel object in PC++ is expressed as a SPMD style program. A parallel object is a representative of the many worker objects, *computation agents.* Every computation agent has a local data and its own thread of control. The access of the local data from the other computation agents can only be done by message passing mechanism. No guard expression can be applied to the parallel object. The user view of message passing mechanism is only synchronous, but the implementation is by asynchronous invocation. The shape structure of the computation agents is assumed as a *mesh* in the first version of PC++. A message passed to a parallel object causes multiple invocations to the computation agents.

The general form to define parallel class is as following.

```
// 2-dimensional mesh
```

[7]SPMD programming is distinguished from SIMD programming, since a SIMD program is usually executed in a *lock-step* manner, but a SPMD program is not so rigidly executed as a SIMD program.

```
parallel class X on [][] {
    ...
  public:
    int f();
};
```

Creation of a parallel object is syntactically achieved by so-called *shape operator* in the *new* statement.

```
X  *x = new [5][5] X;
```

The [5][5] in the middle of *new statement* is called *shape operator*. The role of *shape operator* tells the actual size of the parallel object to the compiler. In this case, a variable **x** is now a parallel object with 5 by 5 grid structure.

The use of the parallel object is done by *select operator* and *reduction operator*. The general syntax is

```
result = '@' [reducer] [selector] <function-call>;
```

The reduction operator can be primitive binary operator such as $+$ and $*$. The reduction can be formalized as $result = v_1 \oplus v_2 \oplus \cdots \oplus v_n$, where the computation agents are numbered $1 \cdots n$, $\oplus$ is an associative binary operator. The select operator is defined as following.

```
selector :== domain_selector
selector :== selector domain_selector
domain_selector   :== '[' expression '..' expression ']'
domain_selector   :== '[' expression ']'
domain_selector   :== '[' '..' expression ']'
```

The example for the parallel object is described below.

```
main()
{
   X *jobs =  new [5][5] X;    // 5 by 5 mesh
   int v, w;
   @jobs->f();                    // invoke all agents
   v = @jobs->f();                // nondeterminism
   w = @+jobs->f();               // reduce all results
   @[0 .. 2][1 .. 2]jobs->f();    // selective invocation
}
```

The user view of the message passing to a parallel object is just a normal message passing. Thus the assignment to a invocation of the message passing expression is allowed as in the following;

```
X *x = new [N] X;
int v;
v = @x->f();                 // nondeterminism
```

But the result can not be expected, since it is not known which result of the multiple invocation will be assigned to the result **v**. This is called *nondeterminism*. The programmer should be aware of this nondeterminism. In many cases, the assignment to a scalar variable is useful by reducing the multiple data as the results of the invocations. It is called *reduction*.

PC++ also provides some basic useful operations related to the grid structure.

dimof(). This function tells the dimensionality of the parallel object

numof(dim). It tells the number of computation agents on the dimension `dim`

idxof(dim). It says the identity index of the dimension of the computation agent

An abbreviation **these** refers to the parallel object itself rather than referring to the computation agents themselves.

5 Implementation

PC++ has two components; runtime system and translator. The runtime system of PC++ handles basically dynamic task creation, task scheduling, and memory management. The task creation is light-weight since the cost of creation of new task is cheaper than the cost of the creation of new process.

The translator converts PC++ source into C++ code, and compiles and binds it with the runtime library routines. The important role of translator is the detection of extended classes (atomic, dynamic, and parallel classes) and qualifiers (pure, impure, sync, async, and future), and the generation of appropriate C++ code according to the characteristics of classes and qualifiers. The overview of PC++ translator is shown on Figure 6.

5.1 General Translation Technique

PC++ translator works in two phases. The first phase reads the PC++ source program and constructs an abstract syntax trees which is simplified as a S-expression.[8] Each link node, which is also a S-expression, represents a declaration, statement, or expression. A leaf node has the symbol ID and name.

The second phase performs pattern matching between the S-expressions and their related code generation routines, and produces associated C++ codes.

```
dynamic class X {
     future int f(int a, int b);
};
```

The abstract syntax tree of the above example is:

[8] *S-expression* is originally introduced in LISP. A S-expression is either a symbol, a constant, or a list ($S_1, S_2, \cdots, S_n$) of zero or more S-expressions [38]

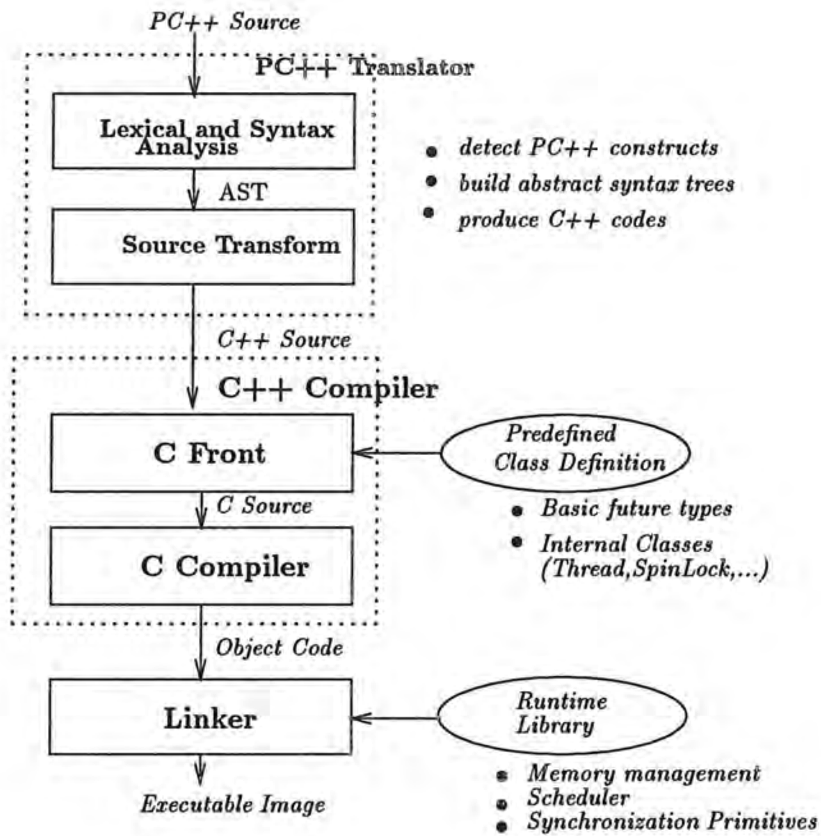

Figure 6: Overview of Translator

```
( CLASSkw ( DYNAMICkw "X" )
          ( FUNCTION "f"
                ( FUTUREkw INTkw )
                ( ( ( INTkw ) "a" )
                  ( ( INTkw ) "b" ) )
                ( /* null body */ ) ) )
```

The abstract syntax trees for the class and function declaration can be described in more general as follows:

```
( CLASSkw  ( <specifier> <name> )
           <member/data-declarations>
)

( FUNCTION <name>
        ( <return-type> )
        ( <argument-list> )
        ( <function-body> ) )
```

The other constructs can be represented as S-expressions like the class and function declaration.

In second phase, the translator generates C++ code. The examples for the code generation will be explained in Section 5.4 through Section 5.7.

5.2 Task Management

5.2.1 Thread

A task is represented by a *thread* object which contains information about the state and execution context. The information includes the thread ID, a pointer to the thread's stack, the stack size, a flag which tells *pure/impure* and *static/non-static*, a pointer to the *guard* expression function, a pointer

to the *execution* function, runtime context, and etc. The thread object is created by the translator when a parallel call is made.

The definition of the internal class Thread is as follows:

```
typedef void (*PFany)(...);
typedef int  (*PFIany)(...);

void waitChildren();        // wait all child threads

class Thread {
        ThreadState     state;          // BLOCKED,READY,...
        int             pure_flag;      // PURE, IMPURE
        int             static_flag;    // STATIC, NON-STATIC
        PFany           core_f;         // Executing Function
        PFIany          when_f;         // Guard Function
        int             *stack;         // Stack pointer
        DynamicObject   *domain;        // Bounded data object
        FutureObject    *waiting;       // Waiting Future
        Context         context;        // save registers
        int             *arguments;     // argument list
   public:
        // Constructor
        Thread(int ispure=IMPURE, int isstatic=NONSTATIC,
                int  stacksize=0);
        // set arguments and functions
        void    setargs(DynamicObject *dom, PFany cf,
                        PFIany wf, void *res, ...);
        int     insert(FutureObject *waited);  // onto Created
        int     guard_test();           // check the guard
        void    run();                  // start the invocation
        void    wait();                 // wait this thread
        void    wakeup();               // set to ready
        void    resume();               // set to running
        int     save_context();         // for context switch
        void    rest_context();
};
```

The function `waitChildren()` is used to wait until all child computations have terminated. The operations of the thread object are explained as follows:

Thread() is a constructor for a new task.

setargs() places related information onto the task, the information includes a *function* to be executed, *parameters*, a *guard expression function.* and a *bounded dynamic object* if the thread runs on the dynamic object.

insert() puts the task into *created* queue. After it is inserted, the task can be picked and executed by the scheduler.

guard_test() is used by the scheduler. Before the execution, the scheduler tests the *guard.* If it is true, the task can be run; Otherwise it is remained in the *created* state.

run() is used by the scheduler, the scheduler picks the task from the created queue, and invoke it by `run()`

wait(), wakeup(), resume() control the state of the thread. The state of the thread is changed from the *running* state to the *blocked* state by `wait()`, from the *blocked* state to the *ready* state by `wakeup()`, and from the *ready* state to the *running* state by `resume()`.

save_context(), rest_context() are used to switch running context which is the current content of the registers

5.2.2 Scheduler

The number of schedulers is the same as the number of physical processors in first version of PC++. The scheduler handles managing the task (*thread*), and checking the system status which is either *processing done* or *deadlock.*

The activation of the scheduler is shared by the activation of the task. When a processor is idle, the scheduler will be activated. The scheduler picks a task, and its control is switched to the task. The *context switching*

mechanism is implemented by *coroutines* which are provided as `setjmp()` and `longjmp()` in UNIX.[9]

```
class Scheduler {
private:
        Context context;        // Context of the scheduler
        int     proc_id;        // processor id

        void    do_task(Thread *t);     // run the thread
        void    do_resume(Thread *);    // resume the thread

        int     save_context();
        void    rest_context();
public:
        void    run();          // initiate scheduler
        void    resume();       // resume the scheduler
friend  int check_system();
};
```

do_task() invokes the execution of the thread when the *guard* is true. The state of the thread is changed from the *created* state to the *running* state.

do_resume() resumes the thread from the *ready* state to the *running* state.

run() initiates the scheduler.

resume() resumes the scheduler when the processor is idle.

check_system() check the system whether the program is completed or the system is deadlocked.

---

[9]UNIX is a registered trademark of AT&T Bell Laboratories

5.2.3 Thread Overhead

In order to execute a task, a new created thread must be added to the shared the *created* queue which is protected by a spinlock. The scheduler dispatches a thread from the *created* queue, tests the guard and purity to ensure reader-writer style lock, and eventually activates the task.

The use of a shared queue by a spinlock will be a significant bottleneck [19]. Figure 7 shows the overhead to create, start and delete the threads which have not any execution body, namely *null* threads, on various number of processors. The curve shows degradation in performance according to increasing the number of processors. The overhead comes mainly from the thread start-up cost and context switch between threads, and the contention to the shared resources, such as task queues (*created* and *ready* queues). As the number of processors increases, the contention to the shared queues increases, and therefore the execution time also increases.

5.3 Memory Management

Every task (thread) can be migrated over any processors, and a data object can be accessed by any tasks. It requires the single or shared memory address space mechanism. Thus the stack and the data must be placed on the shared memory space.

The dynamic memory allocation in PC++ performs on the shared memory, but the static memory placement does not. To use static memory between processors, the programmer must specify the memory as **shared_t**.

```
shared_t int xsum;   // shared memory
int          ysum;   // non-shared memory
                     // bounded to processors
main()
{
   int *x = new int; // on shared memory
}
```

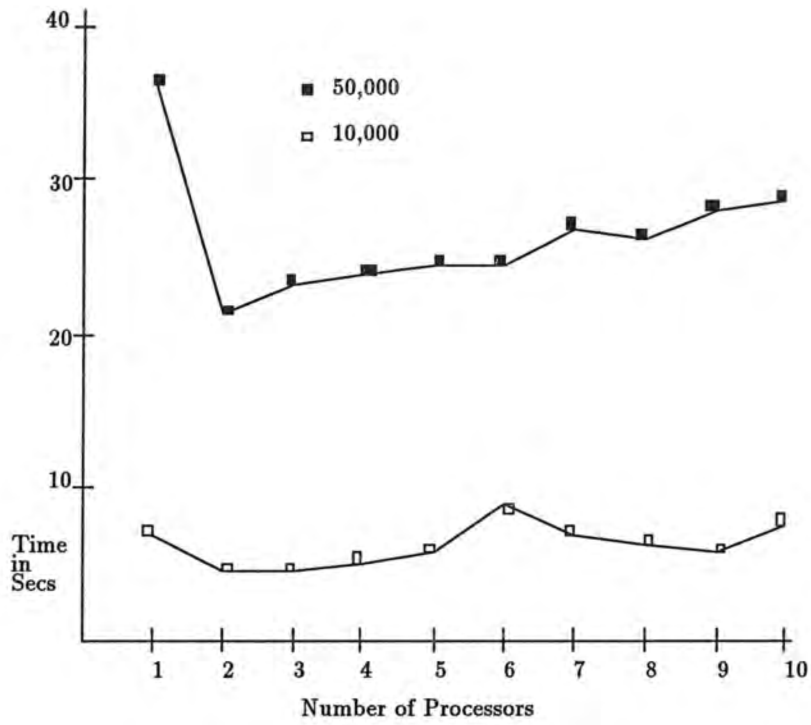

Figure 7: Timing - Thread Overhead

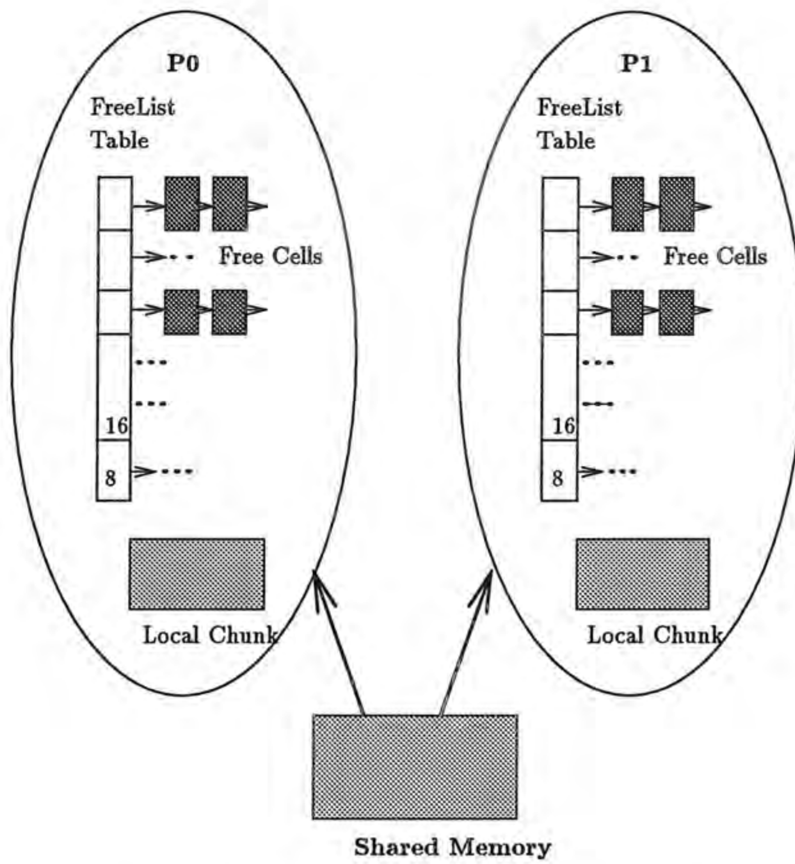

Figure 8: Memory Management Scheme

The memory spaces for `xsum` and `x` are located on the shared memory, but the space for `ysum` is located on the private memory of every processor.

Allocation on the shared memory space causes an additional overhead, because several processors can attempt to use the shared memory simultaneously, but only one allocation can be performed at a time. Thus PC++ provides more efficient memory allocation routine on shared memory space. The **new** and **delete** operator are redirected to the new efficient allocation routine.

The general scheme of the memory allocation is shown on the Figure 8. Every processor maintains *Free List Table* and *Local Memory Chunk*. The *Free List Table* is an array of the pointers to the list of free memory cells. The *Local Memory Chunk* is the unused memory but it is allocated from the shared memory. When a task requests an allocation of memory, the task lookup first the *Free List Table* on the processor whether the processor has available free memory. If it does, the system gives it to the task. But if it does not have available free memory, the system checks the *Local Memory Chunk* and partitions the chunk into the appropriate size. If the effort of trying to allocate memory on the local chunk fails, then the system requests a new memory chunk to the shared memory. When a task requests deallocation of a memory, the system simply appends the memory to *Free List Table* according to the size of the memory.

The performance of the memory management in Figure 9 is quite good comparing with the `shmalloc()` and `shfree()` in the Sequent Runtime Library. The benchmark test is performed under one million iterations for the allocation and deallocation of the memory randomly with the size 32 bytes and 64 bytes.

5.4 Translation of Atomic Object Class

The invocation of the member function on the atomic object does not require creation of a new thread. The thread of its execution is the same as the caller's thread.

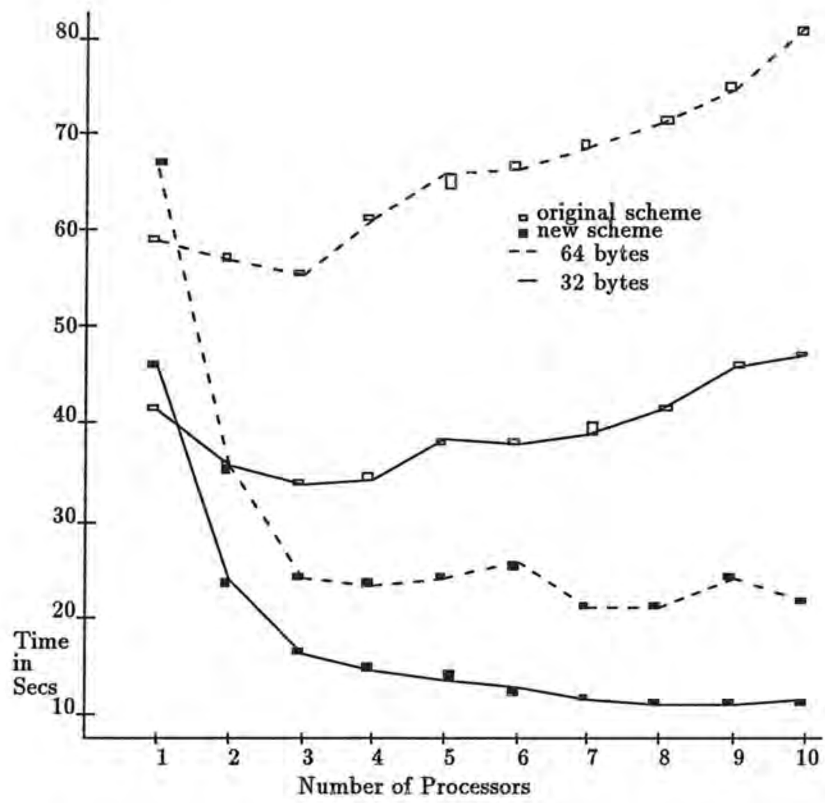

Figure 9: Timing - Memory Allocation/Deallocation

The operations on an atomic object are reader-writer style mutual exclusive. A *writer* task must have exclusive access. A *reader* task may share the atomic object with an unlimited number of other *reader* tasks. The reader-writer lock for the atomic object is implemented by busy waiting spinlock. If locking the atomic object is not successful because the other task already accesses the object, the current task is left spinning until the atomic object is released by the other task. The general reader-writer locking mechanism is explained in [16].

Every user atomic class object is a subclass object of the internal class *AtomicObject.*

```
class AtomicObject : public Object {
private:
        SpinLock        a_lock; // lock variable
        int             n_reader, n_writer;
public:
        AtomicObject();
        void    pure_lock();            // readers
        void    pure_unlock();
        void    impure_lock();          // writer
        void    impure_unlock();
};
```

The *pure* member functions of an atomic object act as *readers*, and are protected by `pure_lock()` and `pure_unlock()`. The *impure* member functions act as *writers*, and are enclosed by `impure_lock()` and `impure_unlock()`.

The qualifier *pure/impure* can be placed in front of the function declaration. If the qualifier is not described, the function is assumed as *impure* function. `<T1>` and `<T2>` can be any type. The following code shows a user program which uses an atomic object class.

```
atomic class X {
   public:
```

```
        pure      <T1>  f(args);     // reader lock
        [impure] <T2>  g(args);     // writer lock
};
<T1> X::f(args) { ... }
<T2> X::g(args) { ... }
main()
{
     X *x = new X;
     x->g();
}
```

The role of the translator for the atomic object is to produce C++ code surrounded by `pure_lock()/unlock()` or `impure_lock()/unlock()`. For the above example, the translator generates following code.

```
class X : public AtomicObject {
   public:
      <T1>  org_f(args);     // just rename
      inline <T1> f(args)
      {
           pure_lock();
           <T1> zztmp = org_f(args);
           pure_unlock();
           return zztmp;
      }

      <T2> org_g(args);
      inline <T2> g(args)
      {
          impure_lock();
          <T2> zztmp = org_g(args);
          impure_unlock();
          return zztmp;
      }
};
```

```
<T1> X::org_f(args) { ... }
<T2> X::org_g(args) { ... }

extern "C" pc_usr_main()      // main() ==> pc_usr_main()
{
    X *x = new X;
    x->g();
}
```

The `args` in the function declaration represents the argument declaration list, and `args` in the function call represents parameter list. The original function `f()` will be renamed by `org_f()`. The execution of the original function will be protected by `pure/impure lock()` and `unlock()` according to its characteristic.

5.5 Translation of Dynamic Object Class

Unlike the atomic object, the invocation of a member function on the dynamic object creates a new thread. If a call on the dynamic object is required, then a new thread is created and the dynamic object is bounded to the thread.

The purity, which is either *pure* or *impure*, is specified by the qualifier in front of the member function declaration just like the atomic object. The semantic of the reader-writer style lock on the dynamic object is the same as that on the atomic object. But the implementation of *pure/impure* lock is different from that for the atomic object. When a task tries to lock a dynamic object, and the object is already locked by other tasks, the task will be suspended and its control will be transferred voluntarily to the scheduler.

The user dynamic class will be a sub-class of *DynamicObject* by the translator.

```
class DynamicObject : public Object {
private:
```

```
        SpinLock        d_lock;
        short           n_reader;
        short           n_writer;
public:
        DynamicObject();
        int     check_purity(int ispure);
        void    undo_purity(int ispure);
};
```

The *pure/impure* locks are implemented by using `check_purity()` and `undo_purity()` in the class *DynamicObject*. The function `check_purity()` is considered as a reader lock if the argument is pure; otherwise, it is considered as a writer lock.

The following code is an example for the use of dynamic object. `<T>` can be any type. But in the first implementation, if the function is specified with the *future* qualifier, `<T>` is allowed only one of `int`, `short`, `long`, `float`, and `double`.

```
dynamic class X {
   public:
       pure async <T> f(args) [when(expression)];
       // For example,
       // pure int lookup(int key) when(key!=0);
};

<T> X::f(args) { ... }      // body of f()

main()
{
    X *x = new X;
    ....
    x->f();       // default call
    ...
    sync x->f(); // explicit sync call
    ...
```

```
}
```

The member function `f()` is by default considered as asynchronous invocation. But the invocation type can be changed by specifying the qualifier in front of the function call statement. For the above example, the following code is generated.

```
class X : public DynamicObject {
   public:
     <T>      org_f(args);      // just rename
     void     core_f(<T> *res, args);
     int      when_f(args);

     Thread *thread_f(void *, args)
     {
        Thread *zzt = new Thread(NON_STATIC,PURE);
        zzt->setargs(this,(PFany)X::core_f,
                      (PFIany)X::when_f, zzret, args);
        return zzt;
     }

     <T> async_f(args)    // async call
     {
         <T>  zzdummy;
         if(_P_this_dynamic == this)
               zzdummy = org_f(args);
         else  thread_f(&zzdummy, args)->insert();
         return zzdummy;
     }

     <T> sync_f(args)   // sync call
     {
         <T> zzret;
         if(_P_this_dynamic == this)
              zzret = org_f(args);
         else {
```

```
            Thread *zzt = thread_f(&zzret,args);
            zzt->wait(zzt->insert());
        }
        return zzret;
    }

    Future_<T> future_f(args)  // future call
    {
        Future_<T> zzlocal;
        if(_P_this_dynamic == this) {
            zzlocal = org_f(args);
        } else {
            FutureObject *zzremote;
            zzremote = newFuture(sizeof(<T>);
            Future_<T> zztmp(zzremote);
            Thread *zzt = thread_f(
                            zzremote->result_ptr(),args);
            zzt->insert(zzremote);
            zzlocal = zztmp;
        }
        return zzlocal;
    }

    // default f() == async_f()
    <T> f(args)           { return async_f(args); }
};

// rename f()
<T> X::org_f(args) { ... }

void X::core_f(<T> *x, args)        //non-inline
{   x ? x=f(args) : f(args); }

int  X::when_f(args)                //guard
{   return expression; }

extern "C" pc_usr_main()
```

```
{
    X *x = new X;
    ....
    x->f();
    ...
    x->sync_f();
}
```

To support various invocation types (sync, async, and future), the translator produces `async_f()`, `sync_f()`, and `future_f()` for a function `f()`.

The translator does not create a new thread if the invocation is made on the same dynamic object. To do this, the variable `_P_this_dynamic` keeps the current dynamic object on each processor. For example, consider a dynamic object class `X` has two member functions `f()` and `g()`, and the function `g()` to the same object is called inside `f()`.

```
dynamic class X {
    int f();
    int g();
};
int X::f()
{
    ...
    this->g();    // (1) same object
    ...
}
```

At point (1), `_P_this_dynamic` is the same as `this`. Thus the invocation of `g()` does not create a new thread.

5.6 Translation of Future Type

A future object is implemented by two type objects; *global* and *local* future objects. The *global* object is shared by all related *local* future objects. It has the state, reference count which tells how many *local* future objects are involved in the *global* future object, and the owner thread which produces the result of the future object.

The global future object `FutureObject` is defined as follows:

```
// Internal Global Future Object
class FutureObject : public Object {
private:
    SpinLock            f_lock;
    FutureState         state;      // 'valid' or not
    short               refcount;   // # referenced
    Thread              *owner;     // executing thread
    int                 owner_id;
    // here is result area
public:
    FutureObject(size_t sz);
    ~FutureObject();
    void    lock(), unlock();
    void    wait();                 // wait a thread
    char    *result_ptr();          // data location
    int     add_ref();              // increment refcount
    int     sub_ref();              // decrement refcount
    void    set_owner(Thread *t);
    void    set_owner_id(int id);
    void    set_state(FutureState s);
    FutureState get_state();
    // wake up all waiting thread via future
    void    wakeup();
    friend FutureObject *newFuture(size_t sz);
};
```

The function add_ref() and sub_ref() are used to handle *reference count*. If a new local object is created and accesses the global object, add_ref() is called to increment *reference count*. If a local object is resolved or does not need the global object, the *reference count* will be decremented via sub_ref(). Eventually, if the *reference count* becomes zero, the global object will be destroyed automatically.

The local future object holds its state (*valid* or *invalid*), the local value, and the pointer to the global future object. The behaviors of the local object are maintaining the reference count of the global object, assigning value or other local object, and referencing the value.

The local object can be implemented by two object classes; Future_<T> and Future_generic. Every user local object Future_<T> is a subclass of the generic future class Future_generic.

```
// User Generic Local Future Object
class Future_generic {
private:
    FutureObject    *remote;     // internal object
    short       data_size;
    FutureState     state;       // local state
public:
    Future_generic(FutureObject *rem,int sz);
    Future_generic(int sz);
    Future_generic(Future_generic *x,int sz);
    void *data_area();           // local data location
    void assign(Future_generic *x); // other future var
    void assign(void *x);        // normal value
    void wait();
};
```

The following code may be written as a user program. A dynamic class X has f() as a future function.

```
dynamic class X {
```

```
    ...
  public:
    future <T> f();
};
main()
{
    X *x = new X;
    future <T> fi;
    <T>    v;
    fi = x->f();
    ...
    v = 100 + fi;
}
```

The example use the type <T> for future call. Thus the translator generates a future type; Future_<T>. The following code is produced for the above example.

```
class Future_<T> : public Future_generic {
private:
   <T>  val;         // local value
public:
    Future_<T>(FutureObject *rem)
         : Future_generic(rem, sizeof(<T>))    {}
    Future_<T>(<T> &x) : Future_generic(sizeof(<T>))
        { val = x; }
    Future_<T>(Future_<T> &x)
      : Future_generic(&x,sizeof(<T>))  {}

    Future_<T> & operator= (<T> &x)
    { Future_generic::assign((void *)&x);   return *this; }
    Future_<T> & operator= (Future_<T> &x)
    { Future_generic::assign( &x ); return *this; }

    <T>     value() { Future_generic::wait(); return val; }
    void    wait()  { Future_generic::wait(); }
```

```
    operator    <T>()   { return value(); }
};
class X : public DynamicObject {
     ...
   public:
     Future_<T> future_f();    // f()
     ...
};
extern "C" pc_usr_main()
{
   X *x = new X;
   Future_<T> fi;      // future variable
   <T>         v;
   fi = x->f();
   ...
   v = 100 + fi;
}
```

5.7 Translation of Parallel Object Class

The implementation of the parallel object can be done with two objects; *ManagerObject* and *WorkerObject* as shown on Figure 10.

The *ManagerObject* holds the information about the *topology* and *dimension* of the objects, and the other related informations as shown on the definition of the class *ManagerObject*. The *WorkerObject* holds the pointer to the *ManagerObject*, and its *identification indices* related to the dimension. The parallel object provides several built-in operations; `idxof()`, `numof()`, and `dimof()`. These built-in operations can be used in the user program.

The selective message passing mechanism to the parallel object is implemented by *ViewObject*. The *ViewObject* handles *subset* operations of the *WorkerObjects*, *invocation* of the *subset* of the *WorkerObject*, and *reduction* of the results.

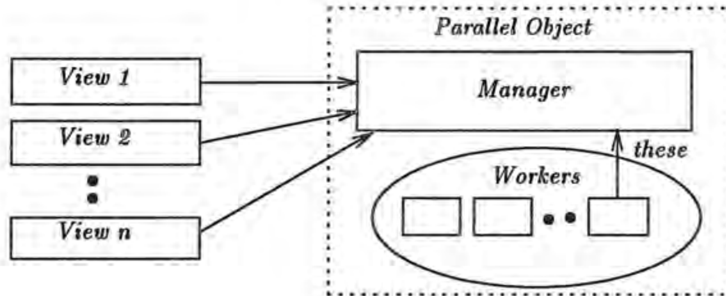

Figure 10: Relationship in Parallel Object

A *ManagerObject* is a representative of the set of *WorkerObjects*.

```
class ManagerObject : public Object {
private:
        WorkerObject     *workers;        // Workers
        int              w_size;          // sizeof(worker)
        int              n_dim;
        int              n_num[MAX_DIM];
public:
        ManagerObject(WorkerObject *objs, int wsiz, ...);
        int      dimof();
        int      numof(int dim);
};
```

A *WorkerObject* is also a *DynamicObject*.

```
class WorkerObject : public DynamicObject {
private:
        int              ids[MAX_DIM];    // indices
public:
        ManagerObject    *manager;
        int              numof(int dim);
```

```
        int             dimof();
        int             idxof(int dim);
};
```

Whenever a call to the parallel object is made, a *ViewObject* is created and used to specify the selection of the *WorkerObjects*.

```
class ViewObject : public Object {
private:
        int             lower[MAX_DIM];
        int             upper[MAX_DIM];
        int             n_workers;
public:
        ManagerObject   *manager;
        int     parCall(PFany func, void *results,
                        int r_size, ...);
        int     int_reduce(int red_op, int res[]);
        float   float_reduce(int red_op, float res[]);
        friend ViewObject *newViewObject(ManagerObject *, ...);
};
```

The following user program is an example which uses a parallel object.

```
parallel class X on [][] {
  public:
    int  lv;
    int f(args);
};

int X::f(args) { ... }

#define N 10
main()
{
   X *x = new [N][N] X;
```

```
    int v;
    @x->f(args);                              // (1)
    v = @x->f(args);                          // (2)
    v = @+x->f(args);                         // (3)
    v = @+[1 .. 3][0 .. 2]x->f(args);         // (4)
    v = @[2][3]x->lv;                         // (5)
}
```

The variable x is a parallel object with N by N grid structure. The object x is a set of *WorkerObjects* and has a *ManagerObject*. Whenever an invocation to the parallel object, a *ViewObject* is used.

```
class X : public WorkerObject {
   public:
      int   lv;
      int   f(args);
      void  core_f(int *, args);
      int   parallel_f(int red_op, ViewObject *zzview)
      {
         int *zzres = new int[zzview->num_worker()];
         zzview->parCall((PFany)X::core_f,
                          zzres,sizeof(int),args);
         zzfinal = zzview->int_reduce(red_op,zzres);
         delete zzres;
         return zzfinal;
      }
      X *get_peer(int d1, ...);
};
// not changed
int X::f(args) { ... }
void X::core_f(int *zzret, args)
{
    zzret ? *zzret = f(args) : f(args);
}
#define N 10
extern "C" pc_usr_main()
```

```
{
   X *x = new X [ N * N ];
   new ManagerObject(x, sizeof(X), N, N);
   int  v;
   x->parallel_f(NONE_REDUCE,
         newViewObject(x->manager), args);          // (1)
   v = x->parallel_f(NONE_REDUCE,
         newViewObject(x->manager), args);          // (2)
   v = x->parallel_f(PLUS_REDUCE,
         newViewObject(x->manager), args);          // (3)
   v = x->parallel_f(PLUS_REDUCE,
         newViewObject(x->manager,1,3,0,2), args);  // (4)
   v = (x->get_peer(2,3))->lv;                      // (5)
}
```

The `parallel_f()` invokes multiple threads of the *WorkerObjects*, waits for the threads, and finally reduces the results. The function `get_peer()` returns a pointer to the *WorkerObject* related to the arguments which are the indices.

5.8 How to use PC++

PC++ is the superset of C++. The syntax of PC++ is described in the appendix A.

To compile a PC++ program, type as following

```
% pc++ command-line
```

For example, we have a source program `matrix.C` and want to produce the runnable object file `matrix`.

```
% pc++ -o matrix matrix.C
```

Then the translator of PC++ produces file `matrix..C` and compile it by C++ compiler to the file `matrix..o`. Finally we can get the object file `matrix`.

The runtime parameters such as the number of processors and stack size of the each thread can be specified by `setenv` command before the execution. For example, if we want to run `matrix` on 4 processors with 4096 byte stack of each thread.

```
% setenv PROCS     4
% setenv STACKSIZE 4096
% matrix
```

If the environment `PROCS` and `STACKSIZE` are not specified, PC++ assumes them as 1 processor and stack size 2048 bytes respectively.

6 Conclusion

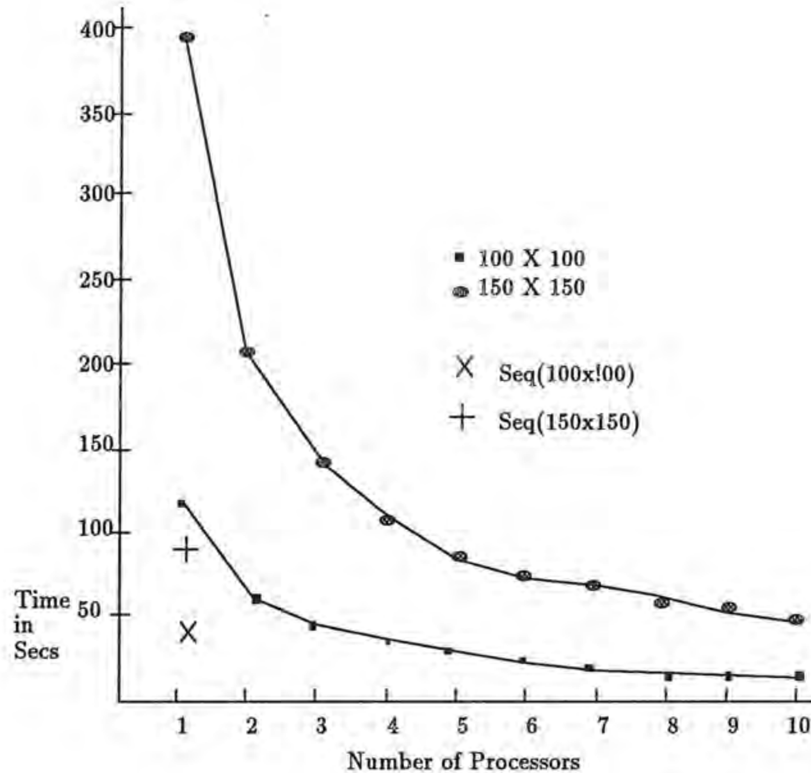

Figure 11: Timing - Matrix Multiplication

Figure 11 shows the benchmark test to compute integer matrix multiplication with various processors on Sequent Symmetry. Figure 12 is the curve to sort integer lists with random distribution by means of quick sort algorithm [32]. The curves are affected by the overhead of creation/deletion of threads, busy waiting spinlock to check purity, and memory allocation/deallocation. The slope is smoothly flattened when many processors are used because every processor attempts to get a new task from the shared queue. But the initial slope of the curve is fairly good. The improvement can be achieved in several ways:

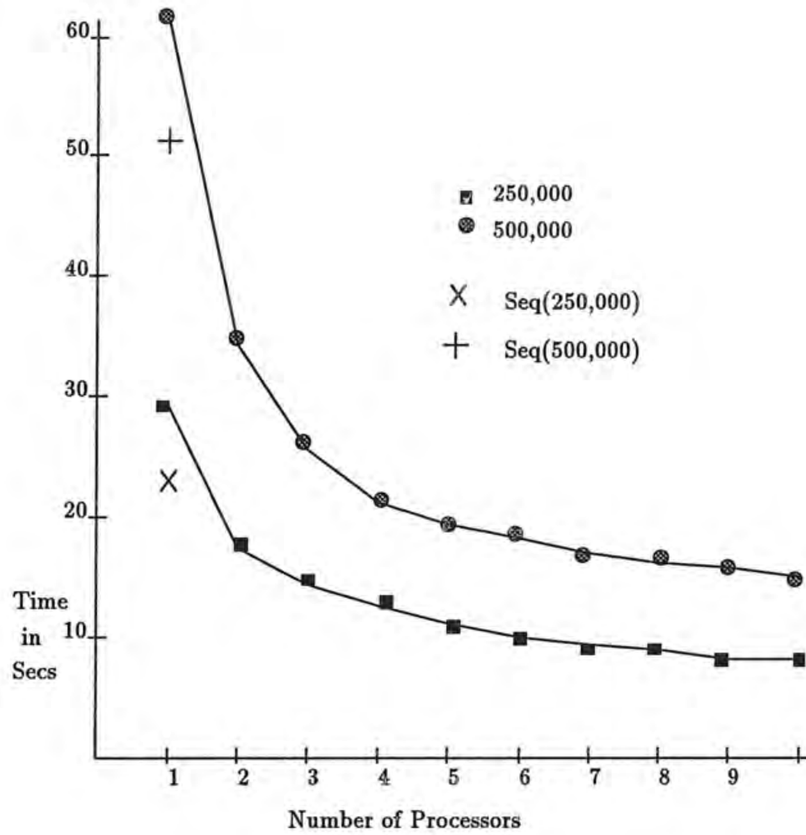

Figure 12: Timing - Quick Sort

- **Separate task queue.** The shared task queue will be a significant bottleneck [19]. The separate task queue on each processor will help to reduce overhead.

- **Queue-based locking.** The first implementation uses spinlock for critical section. But it can be re-implemented by queue-based spinlock which is introduced in [19].

Extending C++ to support various concurrent and parallel computation based on both process-orientation and data-orientation has demonstrated that it is feasible to implement.

The data parallelism is emphasized more today because of its semantic simplicity, its ability to easily express large amounts of parallelism, and its scalability. PC++ can satisfy the expressiveness of the data parallelism as well. Moreover, PC++ provides high-level class-based abstraction for various parallel programming, as C++ provides object-oriented environment.

But this work does not mention the inheritance. The inheritance will be a one of remaining future works. The extension to the distributed memory machines will be researched furthermore, since the current trends of massively parallel computing is on the line of distributed memory machines, and the maximal speedup and scalability can be achieved more effectively on the distributed memory machines.

A Grammar of PC++

The grammar of PC++ is an extension of C++. The following PC++ grammar will be added to the C++ grammar which is given in [18].

```
unary_expression:
        ...
        | domain_spec postfix_expression
        | run_qualifier postfix_expression
        ;

allocation_expression:
        ...
        | "new" parallel_declarator
        ;

primary_expression:
        ...
        | "these"
        ;

domain_spec:
        "@"
        | "@" binary_operator
        | domain_spec "[" ".." expression "]"
        | domain_spec "[" expression "]"
        | domain_spec "[" expression ".." expression "]"
        ;

parallel_declarator:
        "[" expression "]" class_name
        | "[" expression "]" parallel_declarator
        ;


run_qualifier:
```

```
        "sync"
        | "async"
        | "future"
        ;

type_specifier:
        ...
        | "sync"
        | "async"
        | "future"
        | "impure"
        | "pure"
        ;

class_specifier:
        ...
        | class_head shape_operator "{" member_list "}"
        | class_head shape_operator "{" "}"
        ;

class_head:
        ...
        | pc_class_key class_name
        | pc_class_key identifier
        ;

pc_class_key:
        "atomic" class_key
        | "dynamic" class_key
        | "parallel" class_key
        ;

shape_operator:
        "on" "[" "]"
        | shape_operator "[" "]"
        ;
```

```
member_declaration:
        ...
        | decl_specifiers member_declarator_list
             guard_expression ";"
        ;

guard_expression:
        "when" "(" conditional_expression ")"
        ;
```

B Example Programs

B.1 Bounded Buffer Problem

```
#include <stream.h>
#include <stdlib.h>

#define N  10

dynamic class Buffer {
private:
    int buf[N+1];
    int ip, op, n;
public:
    Buffer()    { ip=op=n=0; }
    void    put(int c) when(n<N);
    int     get() when(n>0);
};

void Buffer::put(int c)
{
    buf[ip] = c;
    ip = (ip+1) % N;
    n++;
}


int Buffer::get()
{
    int c = buf[op];
    op = (op+1) % N;
    n--;
    return c;
}
```

```
// Producer
dynamic class Producer {
    Buffer *buf;
    int id;
public:
    Producer(Buffer *b, int p)  { buf=b; id = p; }
    void run(int n);
};

void Producer::run(int n)
{
    for(int i=0; i<n; i++) {
        id++;
        buf->put(id);
    }
}

dynamic class Consumer {
    Buffer *buf;
    int id;
public:
    Consumer(Buffer *b) { buf=b; }
    void run(int n);
};

void Consumer::run(int n)
{
    int c;
    for(int i=0; i<n; i++) {
        c = buf->get();
        cout << "Consume " << c << "\n";
    }
}

main(int argc, char *argv[])
{
    cout << "Usage:" << argv[0] << " #data\n";
```

```
    int n = atoi(argv[1]);
    Buffer *buf = new Buffer;
    Producer *prod = new Producer(buf,0);
    Consumer *cons = new Consumer(buf);
    Consumer *cons2 = new Consumer(buf);

    cons->run(n/2);
    cons2->run(n-n/2);
    prod->run(n);

}
```

B.2 Quick Sort

```
#include <stream.h>
#include <stdlib.h>

dynamic class Array {
   private:
    int from, to, *aptr;
    int insertion_sort();

   public:
    Array(int f, int t, int *ap)
         { from=f; to=t; aptr=ap; }
    future int quick_sort();
};

int Array::insertion_sort()
{
    int i, j, v;
    for(i=from+1; i<to+1; i++) {
        v = aptr[j=i];
        while(j>from && aptr[j-1]>v ) {
```

```
            aptr[j] = aptr[j-1];
            j--;
        }
        aptr[j] = v;
    }
    return to - from + 1;
}

#define MIN_SIZE 50
int Array::quick_sort()
{
    if(to - from < MIN_SIZE) {
        return insertion_sort();
    }

    // partition
    int i = from - 1;
    int j = to;
    int v = aptr[j];
    int t;
    do {
        while( aptr[++i] < v );
        while( aptr[--j] > v );
        t = aptr[i];
        aptr[i] = aptr[j];
        aptr[j] = t;
    } while( j > i );
    aptr[j] = aptr[i];
    aptr[i] = aptr[to];
    aptr[to] = t;

    Array *left = new Array(from, i-1, aptr);
    Array *right = new Array(i+1, to, aptr);
    future int lval = left->quick_sort();
    future int rval = right->quick_sort();

    return (int)lval + (int)rval;
```

```
}

main(int argc, char *argv[])
{
    int     n;
    int     *xlist;
    n = atoi(argv[1]);
    if(n <= 0) {
        cout << form("Usage: %s #elmts\n", argv[0]);
        return 1;
    }

    // make sample data
    xlist = new int[n];
    for(int i=0; i<n; i++) {
        xlist[i] = rand();
    }

    Array *list = new Array(0,n-1,xlist);
    future int count = list->quick_sort();

    // wait here by two ways
    // one is "waitChildren()"
    // two is by strict access
    cout << "Count is " << (int)count << "\n";

}
```

B.3 π Calculation

```
// PI - calculation
#include <stream.h>
#include <stdlib.h>

parallel class PiArea on [] {
```

```
public:
    double area(double width);
};

double PiArea::area(double width)
{
    int idx = idxof(0);
    double x;
    x = (idx + 0.5) * width;
    return 4.0 / (1.0 + x * x) * width;
}


main(int argc, char *argv[])
{
    int n = atoi(argv[1]);
    if(n <= 0) {
        cout << "Usage: %" << argv[0] << " #size \n";
        return 1;
    }
    PiArea *curve = new [n] PiArea;
    double pi_val = @+curve->area(1.0/(double)n);

    cout << form("PI value is %15.10lf\n", pi_val);

}
```

B.4 Matrix Multiplication

```
#include <stream.h>
const int N = 50;

parallel class Matrix on [][] {
public:
    double  v;
```

```
    void    multiply(Matrix *a, Matrix *b);
};

void Matrix::multiply(Matrix *a, Matrix *b)
{
    int i = idxof(0);
    int j = idxof(1);
    v = 0;
    for(int k=0; k<N; k++) {
        v +=  @[i][k]a->v * @[k][j]b->v;
    }
}

main(int argc, char *argv[])
{
    Matrix *a = new [N][N] Matrix;
    Matrix *b = new [N][N] Matrix;
    Matrix *c = new [N][N] Matrix;
    for(int i=0; i<N; i++) {
       for(int j=0; j<N; j++) {
            @[i][j]a->v = 1;
            @[i][j]b->v = 1;
        }
    }
    @c->multiply(a,b);  // in parallel
    cout << "Matrix Multiplication is done\n";
}
```

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.

[2] Gul Agha and Carl Hewitt. *Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming.* In *Research Direction in Object-Oriented Programming,* The MIT Press, Cambridge, MA, 1987.

[3] Pierre America. *POOL-T: A Parallel Object-Oriented Language.* In *Object-Oriented Concurrent Programming,* The MIT Press, Cambridge, MA, 1987.

[4] Gregory R. Andrews, Fred B. Schneider. *Concepts and Notations for Concurrent Programming.* ACM Computing Surveys, Vol. 15, No. 1, March 1983.

[5] Gregory R. Andrews. *Paradigms for Process Interaction in Distributed Programs.* ACM Computing Surveys, Vol. 23. No. 1, March 1991.

[6] Henri E. Bal, Jennifer G. Steiner, Andrew S. Tanenbaum. *Programming Languages for Distributed Computing Systems.* ACM Computing Surveys, Vol. 21, No. 3, September 1989.

[7] Brian N. Bershad, Edward D. Lazowska and Henry M. Levy. *PRESTO: A System for Object-oriented Parallel Programming.* Software-Practice and Experience, Vol. 18, No. 8, August 1988.

[8] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1991.

[9] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher and B. M. Younger. *μC++: Concurrency in the Object-oriented Language C++.* Software-Practice and Experience, Vol. 22(2), February 1992.

[10] David Callahan and Burton Smith. *A Future-based Parallel Language for a General-purpose Highly-parallel Computer.* In *Language and Compilers for Parallel Computing,* David Gelernter, Alexandru Nicolau and David Padua (Eds.), The MIT Press, 1990.

[11] William E. Carlson. *Ada: A Promising Beginning.* In *The ADA Programming Language: A Tutorial*, Sabina H. Sib and Robert E. Fritz (Eds.), Computer Society Press, 1984.

[12] Nicholas Carriero and David Gerlenter. *How to Write Parallel Programs: A Guide to the Perplexed.* ACM Computing Surveys, Vol. 21, No. 3, September 1989.

[13] Hohit Chandra, Anoop Gupta, and John L. Hennessy. *COOL: a Language for Parallel Programming.* Technical Report CSL-TR-89-396, Stanford University, 1989.

[14] Siddhartha Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors.* Ph.D. dissertation, CMU-CS-91-189, Carnegie Mellon University, October 1991.

[15] Roger S. Chin and Samuel T. Chanson. *Distributed Object-Based Programming Systems.* ACM Computing Surveys, vol. 23, No. 1, March 1991.

[16] P. J. Courtois, F. Heymans, and D. L. Parnas. *Concurrent Control with "Readers" and "Writers".* CACM, Vol. 14, No. 10, October 1971.

[17] E. W. Dijkstra. *Guarded commands, nondeterminacy, and formal derivation of programs.* CACM Vol. 18, No. 8, August 1975.

[18] Margaret A. Ellis, and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1991.

[19] John E. Faust and Henry M. Levy. *The Performance of an Object-Oriented Threads Package.* ECOOP/OOPSLA '90 Proceedings, October 1990.

[20] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors. Vol I, General Techniques and Regular Problems.* Prentice-Hall, Englewood Cliffs, N.J., 1988.

[21] Eran Gabber. *VMMP: A Practical Tool for the Development of Portable and Efficient Programs for Multiprocessors.* IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 3, July 1990.

[22] N. H. Gehani, W. D. Roome. *Concurrent C.* Software-Practice and Experience. Vol. 16(9), September 1986.

[23] N. H. Gehani and W. D. Roome. *Concurrent C++: Concurrent Programming with Class(es).* Software-Practice and Experience, Vol. 18, No. 12, December 1988.

[24] N. H. Gehani. *Message Passing in Concurrent C: Synchronous versus Asynchronous.* Software-Practice and Experience, Vol. 20, No. 16, June 1990.

[25] N. H. Genhani, W. D. Roome. *Implementing Concurrent C.* Software-Practice and Experience. Vol. 22, No. 3, March 1992.

[26] Robert H. Halstead, Jr. *Multilisp: A Language for Concurrent Symbolic Computation.* ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4, October 1985.

[27] P. Brinch Hansen. *Distributed Processes: A Concurrent Programming Concept.* CACM Vol. 21, No. 11, November 1978.

[28] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* The MIT Press, 1991.

[29] W. Daniel Hillis and Guy L. Steele, Jr. *Data Parallel Algorithms.* CACM, Vol. 29, No. 12, December 1986.

[30] C. A. R. Hoarse. *Monitors: An Operating System Structuring Concept.* CACM, Vol. 17, No. 10, October 1974.

[31] C. A. R. Hoarse. *Communicating Sequential Processes.* CACM, Vol. 21, No. 8, August 1978.

[32] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[33] Koru Hosokawa, Hiroaki Nakamura and Tsutomu Kamimura. *Concurrent Programming in COB.* In Lecture Notes in Computer Science, No. 491, *Concurrency; Theory, Language and Architecture*, 1989.

[34] Yuuji Ichisugi, Akinori Yonezawa. *Exception Handling and Real Time Features in an Object-Oriented Concurrent Language.* In Lecture Notes in Computer Science, No. 491, *Concurrency; Theory, Language and Architecture*, 1989.

[35] Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer. *Object-Oriented Real-Time Language Design: Constructs for Timing Constraints.* ECOOP/OOPSLA '90 Proceedings, 1990.

[36] Harry F. Jordan, Muhammad S. Benten, Gita Alaghband, and Ruediger Jakob. *The Force: A Highly Portable Parallel Programming Language.* 1989 International Conference on Parallel Processing, 1989.

[37] L. V. Kale. *The Chare Kernel Parallel Programming Language and System.* 1990 International Conference on Parallel Processing, 1990.

[38] Samuel N. Kamin. *Programming Languages. An Interpreter-Based Approach.* Addison-Wesley, 1990.

[39] Michael F. Kilian. *Object-Oriented Programming for Massively Parallel Machines.* International Conference on Parallel Processing, 1991.

[40] J. Peterson and A. Silberschartz. *Operating System Concepts.* Addison-Wesley, MA, 1983.

[41] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, Inc., 1987.

[42] Stephen A. Schuman. *Tutorial on ADA Tasking, Vol. I, Interprocess Communication.* In *The ADA Programming Language: A Tutorial.* Sabina H. Saib and Robert E. Fritz (Eds.), Computer Society Press, 1984.

[43] Robert Sedgewick. *Implementing Quicksort Programs.* CACM, Vol. 21, No. 10, October 1978.

[44] D. B. Skillicorn. *Practical Concurrent Programming for Parallel Machines.* The Computer Journal, Vol. 34, No. 4, 1991.

[45] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1987.

[46] Mark T. Vandevoorde and Eric S. Robets. *WorkCrews: An Abstraction for Controlling Parallelism.* Internation Journal of Parallel Programming, Vol. 17, No. 4, 1988.

[47] Peter Wegner. *The Object-Oriented Classification Paradigms.* In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (Eds.), The MIT Press, Cambridge, MA, 1987.

[48] A. Yonezawa, E. Shibayama, T. Takada and Y. Honda. *Modeling and Programming in an object-oriented concurrent language ABCL/1.* In *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro (Eds.), The MIT Press, Cambrdige, MA, 1987.