Design and Implementation of the CCC Parallel Programming Language

Nai-Wei Lin
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, Taiwan 621, R.O.C.
naiwei@cs.ccu.edu.tw

Abstract- *CCC is a high-level parallel programming language that aims to provide a coherent integration of various parallel programming paradigms. CCC supports both data and task parallelism. In CCC, data parallelism is specified in SIMD model, while task parallelism is specified in MIMD model. Task parallelism in CCC supports both message-passing communication abstraction and shared-variables synchronization abstraction. The CCC parallel programming system aims to provide a unified implementation on top of various parallel machines. CCC has been implemented on both SMPs and SMP clusters. The implementation is mainly based on a virtual shared memory machine interface that supports both shared memory and dynamic task creation. This paper describes the design and implementation details of CCC.*

Keywords: Parallel programming languages, data parallelism, task parallelism.

1. Introduction

The CCC parallel programming language aims to provide a coherent integration of various parallel programming paradigms. The features of the CCC parallel programming language are illustrated in Figure 1. CCC supports both *data* and *task parallelism*. A CCC program consists of a collection of coordinated concurrent data-parallel or task-parallel tasks. Data parallelism in CCC is specified in single-instruction-multiple-data (SIMD) model; In other words, data-parallel tasks are executed *synchronously* and perform the same operations on different data. Shared-memory abstraction is provided to support remote read, remote write, and reduction operations among data-parallel tasks.

In contrast, task parallelism in CCC is specified in multiple-instruction-multiple-data (MIMD) model; In other words, task-parallel tasks are executed *asynchronously* and usually perform different operations on different data. Both message-passing communication abstraction and shared-variables synchronization abstraction are provided to facilitate various interaction patterns among task-parallel tasks. These salient features make CCC a *simple* and *modular* language for studying various parallel programming paradigms.

The CCC parallel programming system aims to provide a unified implementation on top of various parallel machines. CCC has been implemented on both symmetric multiprocessors (SMPs) and SMP clusters. The structure of the CCC parallel programming system is illustrated in Figure 2. The CCC programming system can be divided into five layers.

The first layer is the machine layer. The rapid advance of very large silicon integration (VLSI) and communication technologies has recently made SMPs or multithreaded processors (MTPs) and SMP or MTP clusters easily available. The CCC programming system is designed to target on these machines. Since CCC programs may contain both task and data parallelism, and task parallelism is more general than data parallelism, our implementation is mainly based on a virtual shared memory machine interface that supports both shared

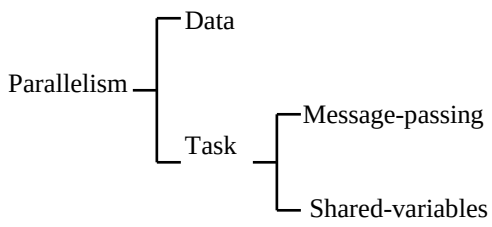

Figure 1. The features of the CCC parallel programming language.

CCC applications	
CCC compiler	
CCC runtime library	
Virtual shared memory machine interface	
Pthread	Millipede
SMP or MTP	SMP or MTP cluster

Figure 2. The structure of the CCC parallel programming system.

memory and dynamic task creation. For SMPs, these functionalities are supported directly by the underlying machines. To provide better portability, we will use the standard library Pthread that supports these functionalities [8]. For SMP clusters, the mechanisms for distributed shared memory and dynamic remote task creation are required. The millipede library is one library that supports such mechanisms on SMP clusters [9]. The second layer of the CCC programming system is the library that supports shared memory and dynamic task creation on top of the underlying machines.

The third layer of the CCC programming system provides a common interface on top of various libraries that supports shared memory and dynamic task creation. This interface makes the CCC compiler and runtime library highly retargetable. To retarget CCC compiler on a different library or machine, this layer is the only layer that needs to modify and the modification is very easy.

To simplify the code generator of the compiler, a collection of functions that supports the salient features of CCC on top of the virtual shared memory machine interface is composed into a runtime library. This runtime library serves as the fourth layer of the CCC programming system. The fifth layer is the CCC compiler and the sixth layer is the CCC applications. This structure makes the implementation of CCC *simple* and *retargetable*.

The remainder of this paper is organized as follows. Section 2 describes the design details of CCC. Section 3 describes the implementation details of CCC. Section 4 reviews related work. Finally, conclusions are given in Section 5.

2. The Design of CCC

The language CCC is designed as a small extension of the language C. This small extension of C mainly provides abstractions for specifying the concurrency, communication, and synchronization of parallel tasks. The main aim of CCC design is to provide a coherent integration of various parallel programming paradigms to facilitate instruction and research of parallel computing. Hence, we strive to integrate both data and task parallelism, and both message-passing communication abstraction and shared-variables synchronization abstraction. Both data and task parallelism will be specified as *task-level* parallelism in terms of data-parallel and task-parallel tasks. *Channels* are used as the message-passing communication abstraction and *monitors* are used as the shared-variables synchronization abstraction. Both channels and monitors will be viewed as *objects*.

2.1. Data parallelism

We consider the data parallelism first. The concurrency abstraction for the data parallelism is specified by the definition of the domain construct and the invocation of the *data-parallel functions* defined in domain construct.

```
domain name [size] {
   data_declarations;
   data_parallel_functions;
}
```

Each data-parallel function definition here represents a collection of *size* parallel tasks. An invocation of a data-parallel function will concurrently create *size* tasks. The synchronization abstraction is implicitly specified by the synchronous semantics of the SIMD model. The communication abstraction is implicitly specified by a global name space that provides remote read, remote write, and reduction operations on the variables defined within the domain construct. The *data distribution* specifications can be given in the parameter specifications in the data-parallel function definitions.

This design has several merits. First, the domain construct provides a virtual task abstraction. Usually, the best performance occurs when the number of tasks is the same as the number of processors available. The mapping of virtual tasks to physical tasks is handled by the compiler. This allows the programmers to specify the parallelism degree of the applications independent of the number of processors on the underlying machines. Second, the SIMD model ensures a deterministic semantics for data parallelism. This significantly helps the understanding of the program behaviors. Third, the global name space provides a shared-memory abstraction. This greatly simplifies the non-local accesses of variables on different virtual tasks. These merits make CCC a simple and modular language for specifying data parallelism.

2.1. Task parallelism

We now consider the task parallelism. The concurrency abstraction for the task parallelism is specified via the definition of task-parallel functions and the *parallel section* constructs. A task-parallel function

```
task task_parallel_function();
```

is declared by putting the keyword task before its function definition. There are two forms of parallel sections. The par construct is used to concurrently invoke a group of task-parallel functions.

```
par {
   func_1;
   func_2;
     …
   func_n;
}
```

This example will execute the *n* task-parallel functions *func_1*, *func_2*, …, and *func_n* as *n* tasks

in parallel. Parallel sections are executed in fork-join form; that is, the parallel section exits only when all created tasks exit. The parfor construct is used to concurrently invoke multiple instances of a group of task-parallel functions.

```
parfor (init_expr; exit_expr; step_expr) {
    func_1;
    func_2;
     …
    func_n;
}
```

This example will execute multiple instances of the *n* task-parallel functions *func_1*, *func_2*, …, and *func_n* as tasks in parallel. The semantics for the control expressions *init_expr*; *exit_expr*; and *step_expr* in the parfor construct is the same as that in the for construct in C. The par and parfor constructs are more *structured* than the fork and join constructs in most libraries.

The communication abstraction for the task parallelism is specified via *channels* or *asynchronous message queues*. The messages in the channels can be accessed via the following two functions

```
msg = receive(channel);
send(channel, msg);
```

There are four types of channels: pipes, spliters, mergers, and multiplexers. Pipes are for one-to-one communication. They are the most basic channels. The other three types of channels can be implemented using pipes. Spliters are for one-to-many communication. These channels are useful for producer-consumer applications with single producer. Mergers are for many-to-one communication. These channels are useful for client-server applications with a single server. Multiplexers are many-to-many communication. These channels are useful for producer-consumer applications with multiple producers or client-server applications with multiple servers. Channels provide a simple abstraction for implementing communication in message-passing programming model.

The abundance of channel types gives several merits. First, communication structures among parallel tasks are more comprehensive. Second, the specification of communication structures is easier. Third, the implementation of communication structures is more efficient. Fourth, the static analysis of communication structures is more effective.

The synchronization abstraction for the task parallelism is specified via active *monitors*.

```
monitor name {
    data_declarations;
    condition_variable_declarations;
    function_definitions;
}
```

Monitors are *structured* and *efficient* constructs for implementing both the *mutual exclusion* and *condition synchronization* in shared-variable programming model. The functions defined in monitors are mutually exclusive by default. A read keyword can be put before the definition of a function if that function only reads the variables in the monitors. Multiple such functions can be invoked concurrently. Three functions wait(*cond*), signal(*cond*), and signalall(*cond*) are provided to manipulate *condition variables*. The signal functions have the semantics that they will continue to execute after signaling. Nested monitor calls are allowed in the language. A task releases monitor exclusion when it makes a nested monitor calls, and it needs to reacquire monitor exclusion when it returns from the call.

3. The Implementation of CCC

The CCC parallel programming system aims to provide a unified implementation on top of various parallel machines. CCC has been implemented on both SMPs and SMP clusters. The implementation consists of a virtual shared memory machine interface, a runtime library, and a compiler.

3.1. The virtual shared memory machine interface

The virtual shared memory machine interface provides a common interface that supports both shared memory and dynamic task creation on top of both SMPs and SMP clusters. We now describe the functions provided in the interface.

The virtual shared memory machine is initialized with the VSMM_init() function and is finalized with the VSMM_final() function. The VSMM_init() function also sets the number of processors in the machine. The VSMM_num_of_procs() function queries the number of processors in the machine. The VSMM_my_pid() function returns the identifier of the processor on which the current task is executing. The VSMM_min_load_proc() function returns the identifier of the processor that has the minimal number of tasks. This function provides useful information for load balancing. The VSMM_share() function allocates a section of storages from the shared memory.

The tasks in the virtual shared memory machine are created using the VSMM_create() function. Each created task is assigned to execute on one of the processors in the machine. The VSMM_my_tid() function returns the identifier of the current task. The VSMM_exit() function terminates the current task. The VSMM_join() function waits for the termination of a specific task.

The tasks in the virtual shared memory machine can be synchronized with four types of synchronization objects: mutex locks, read-write locks, condition variables, and barriers. The functions VSMM_lock_init(), VSMM_lock(),

VSMM_unlock(), and VSMM_lock_destroy() support the functionalities of mutex locks. The functions VSMM_rwlock_init(), VSMM_rdlock(), VSMM_wrlock(), VSMM_rwunlock(), and VSMM_rwlock_destroy() support the functionalities of read-write locks. The functions VSMM_cond_init(), VSMM_wait_lock(), VSMM_wait_rdlock(), VSMM_wait_wrlock(), VSMM_signal(), and VSMM_cond_destroy() support the functionalities of condition variables. The functions VSMM_barrier_init(), VSMM_barrier(), and VSMM_barrier_destroy() support the functionalities of barriers.

The implementations of the virtual shared memory machine interface using Pthread and Millipede are both very straightforward. These two libraries have very similar functions to the ones in the virtual shared memory machine interface. Hence, most of the implementations are just macro definitions.

The remarkable details in the implementation using Pthread are as follows. First, since the library is running on a single SMP, the function VSMM_num_of_procs() is defined as 1 and both functions VSMM_my_pid() and VSMM_min_load_proc() are defined as 0. Second, read-write locks and barriers are not implemented in Pthread. The set of functions for read-write locks and barriers are implemented using mutex locks and condition variables in Pthread.

The remarkable details in the implementation using Millipede are as follows. First, to implement the function VSMM_min_load_proc(), an array VSMM_num_of_tasks[] is declared as shared that maintains the number of active tasks on each SMP. This array is updated by functions VSMM_create() and VSMM_exit(), and is read by the function VSMM_min_load_proc(). Second, the function VSMM_create() is implemented using the remote thread creation function, and the function VSMM_share() is implemented using the shared memory allocation function and the shared memory distribution function in Millipede. Third, the join of tasks is not implemented in Millipede, it is implemented using barriers in Millipede. Fourth, read-write locks are not implemented in Millipede. The set of functions for read-write locks are implemented using mutex locks and condition variables in Millipede.

3.2. The runtime library

The CCC runtime library contains a collection of functions that implements the salient abstractions of CCC on top of the virtual shared memory machine interface.

We consider the data parallelism first. For the sake of efficiency, the SIMD model of the domain construct will be transformed and executed in the SPMD model. For each domain construct named dname, the compiler will declare a structure type

CCC_dmain_dname_type

for it. Except for the original data members in the domain construct, the compiler will add a mutex lock and a barrier into it to implement the SPMD model. The compiler will also define a function

CCC_domain_dname_init()

for it to initialize the mutex lock and barrier. The set of functions CCC_*reduce*_op() provide efficient implementation for reduce operations: addition, product, bit-and, bit-or, bit-xor, logical-and, logical-or, maximum, and minimum.

We now consider the task parallelism. For each task-parallel function named fname and with n parameters, the compiler will declare a structure type

CCC_fname_n_param_type

for it. To create a task for this task-parallel function, its arguments will be packed into a structure of this type. The compiler will also define two macro definitions

CCC_fname_n_pack_arg()
CCC_fname_n_unpack_arg()

to pack and unpack its arguments before and after entering the function.

Task-parallel functions can be invoked in parallel sections. Parallel sections are executed in fork-join form. To appropriately execute the join operations, the task identifier for each created task will be maintained. The macro definition

CCC_parallel_section_prologue()

determines the number of function calls in the parallel section and declares an array to store the task identifiers. The macro definition

CCC_parallel_section_epilogue()

performs the appropriate join operations.

For each monitor construct named mname, the compiler will declare a structure type

CCC_monitor_mname_type

for it. If there is no read function in the monitor, a mutex lock will be added into the structure, and the body of each function will be enclosed by the lock and unlock operations of the mutex lock. If there are some read functions in the monitor, a read-write lock will be added into the structure, and the body of each function be enclosed by the lock and unlock operations of the read-write lock.

For each channel of basic type tname, the library contains an implementation of it using a monitor. More specifically, it contains a declaration of the structure type

CCC_channel_tname_type

and the definitions of the following two mutually exclusive functions

```
void CCC_send_tname
        (CCC_channel_tname_type, tname);
tname CCC_receive_tname
        (CCC_channel_tname_type);
```

For user-defined types, the compiler will generate the declarations of the corresponding structure type and the definitions the corresponding send and receive functions.

The code generator of the compiler can be significantly simplified given the macros and function templates defined in the runtime library. This simplification also greatly facilitates the maintenance and enhancement of the compiler in the future.

3.3. The compiler

The CCC compiler translates CCC programs into C programs that call functions provided by the virtual shared memory machine interface and the runtime library.

We consider the data parallelism first. The CCC compiler first needs to map the virtual tasks into the physical tasks based on the number of processors available. Usually, multiple virtual tasks will be mapped into one physical task. Hence, the mapping would be done by merging multiple data mapped to the same physical task and emulating the concurrent execution of multiple instances of a function.

The merging of the multiple data may add more dimensions to each variable. For example, if the domain construct is declared as a 2-dimensional array and the variable is also declared as a 2-dimensional array, then the merged variable would become a 4-dimensional array. The first two dimensions are used to identify the virtual tasks and the last two dimensions are the original variables. Hence, the access patterns for each variable would be changed accordingly. The CCC compiler will generate an access function for each variable.

The emulation of the concurrent execution of multiple instances of a function is much more difficult. Basically, each statement in the function body should be executed multiple times. However, to improve the efficiency on SMPs or SMP clusters, we would like to emulate the SIMD model using the SPMD model. That is, we will synchronize only when necessary.

The inference of the necessary synchronization points depends on the data dependence among the statements in the function body. The data dependence is inferred from the accesses of non-local variables defined in domain construct. These non-local variable accesses represent remote read, remote write, or reduce operations, and are the necessary synchronization points. The data dependence occurs between read-write, write-read, and write-write pairs. If multiple pairs overlap, then they can share their synchronizations to minimize the necessary synchronization points.

The statements among synchronization points could then be grouped into a loop to emulate the concurrent execution of multiple instances. Such a grouping could minimize the number of loops.

If a non-local variable access occurs in a conditional statement, like an if-statement or switch-statement, then this conditional statement needs to be partitioned into multiple conditional statements to make sure that every task will execute this non-local variable access. Similarly, if a non-local variable access occurs in a repeated statement, like a for-statement or while-statement, then each task needs to execute the same number of iterations to ensure that every task will execute this non-local variable access, although some of the tasks may do nothing else in the loop.

The compiler has performed several optimizations to improve the quality of the generated code. First, the merging of the multiple data may be omitted. Some of the variables defined in the domain construct maintain the same value for all virtual tasks. In such situations, one instance of these variables can be shared among the virtual tasks instead of duplicating multiple instances of these variables. Second, the looping of statements may be omitted. If the merging of a variable is omitted, then there is no need to use a loop to manipulate the instances of this variable. Third, the global checking for loop termination may be omitted. If the loop conditional expression ensures that every task executes the same number of iterations, then there is no need to perform the global checking for the loop termination.

We now consider the task parallelism. Since monitors and channels are higher-level shared memory synchronization and communication abstractions, the implementation of these abstractions in terms of lower-level abstractions provided by the virtual shared memory machine interface is quite straightforward. We only describe the static analysis of communication structures based on the channel declarations.

The static analysis of communication structures checks whether the channels are used as they are declared. This analysis is based on a *compact control flow graph* for each function. There are five kinds of nodes in the compact control flow graphs. A *call node* represents a call to a sequential function. A *spawn node* represents a call to a parallel function. A *compound node* represents a sequence of nodes. An *alternative node* represents a list of nodes that are mutually exclusive. A *repetition node* represents a node that may be executed multiple times. Based on this graph, we can analyze first-order channels in linear time.

4. Related Work

The programming language CCC is highly influenced by the data parallelism features provided by Dataparallel C [7] and DINO [10], the monitor construct provided by Concurrent Pascal [4], the

channels provided by Fortran M [5], and the coherent integration of various parallel programming paradigms provided by SR [2].

In Dataparallel C, the non-local access patterns use the instance name instead of the domain name. This makes it difficult to reuse the functions. In DINO, the variables in domain are declared using global view instead of local view. This makes it necessary to explicitly distinguish local and non-local accesses. Monitors in Concurrent Pascal are passive construct, while monitors in CCC are active objects. Fortran M only provides one-to-one and many-to-one channels. We discover that one-to-many and many-to-many channels are also very useful. SR only supports task parallelism. Although data parallelism can be represented by task parallelism, the domain construct provides a higher-level abstraction that facilitates readability and maintainability.

Efficiently implementing a parallel programming system that supports both data and task parallelism on both SMPs and SMP clusters is a very difficult job. Most related projects concentrate on implementing efficient thread libraries supports either data or task parallelism on either SMPs or SMP clusters.

The libraries ThreadMark[1] and OpenMP [11] efficiently support data parallelism on SMP clusters. However, using these libraries to support task parallelism is very difficult if it is not impossible. The libraries Pthread, [8] and Cilk [3] efficiently support task parallelism on a single SMP. However, the extensions of these libraries on SMP clusters are still unavailable. The libraries Mellipede [9] and Filaments [6] efficiently support task parallelism on SMP clusters. However, the interfaces they provide are very different.

5. Conclusions

CCC is a high-level parallel programming language that aims to provide a coherent integration of various parallel programming paradigms. CCC supports both data and task parallelism. In CCC, data parallelism is specified in SIMD model, while task parallelism is specified in MIMD model. Task parallelism in CCC supports both message-passing communication abstraction and shared-variables synchronization abstraction. Both data and task parallelism are specified as task-level parallelism. Channels are used as the message-passing communication abstraction and monitors are used as the shared-variables synchronization abstraction. Both channels and monitors are viewed as objects. These salient features make CCC a simple and modular language for studying various parallel programming paradigms.

The CCC parallel programming system aims to provide a unified implementation on top of various parallel machines. CCC has been implemented on both SMPs and SMP clusters. The implementation is mainly based on a virtual shared memory machine interface that supports both shared memory and dynamic task creation. To retarget CCC compiler on a different library or machine, the virtual shared memory machine interface is the only component that needs to be modified. This makes the implementation of CCC simple and retargetable.

6. Acknowledgements

This work was supported in part by the National Science Council of R.O.C. under grant numbers NSC-87-2213-E-194-006, NSC-88-2213-E-194-004, NSC-89-2213-E-194-004, and NSC-90-2213-E-194-055. The author would like to thank all his Master students for their efforts in the implementation of the CCC parallel programming system.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R.Rajamony, W. Yu, and W. Zxaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, 29(2):18-28, 1996.

[2] G. Andrews and Olsson, *The SR Programming Language: Concurrency in Practice*, Benjamin/Cummings, 1993.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," In *Proc. Symposium on Principles and Practice of Parallel programming*, pp. 207-216, ACM, 1995.

[4] P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, 2 (June), 199-206, 1975.

[5] I. Foster, and K. M. Chandy, "Fortran M: A Language for Modular Parallel Programming," *Journal of Parallel and Distributed Computing*, 1994.

[6] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews, "Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations," *First Symposium on Operating Systems Design and Implementation*, pp. 201-212, Monterey, CA, November 14-17, 1994

[7] P. Hatcher and M. Quinn, *Data-Parallel Programming on MIMD Computers*, The MIT Press, 1991.

[8] IEEE. IEEE P1003.1c/D10: Draft Standard for Information Technology – Portable Operating Systems Interface (POSIX), September 1994.

[9] A. Itzkovitz, A. Shuster, and L. Shalev, "Thread Migration and its Applications in Distributed Shared Memory Systems," *Journal of Systems and Software*, vol. 42, pp. 71-87, 1998.

[10] M. Rosing, R. B. Schnabel, and R. P. Weaver, "The DINO Parallel Programming Language," *Technical Report CU-CS-457-90*, Department of Computer Science, University of Colorado, Boulder, Colorado, April 90.

[11] The OpenMP Forum. *OpenMP C and C++ Application Program Interface*, Version 1.0, October 1998.