The Implementation of a Task Parallel Language on Symmetric Multiprocessors[*]

Ching-Chi Lin Juang-Weei Lin Nai-Wei Lin

Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, Taiwan, ROC

Abstract

The rapid advance of VLSI and communication technologies has recently made symmetric multiprocessors (SMPs) easily available. These SMPs usually consist of two to tens of commodity microprocessors interconnected to a shared memory. Due to cost effectiveness, clusters of SMPs have quickly become the trend for new generation of parallel computing systems. This paper will report experiences on implementing a task parallel language CCC on top of the Cilk runtime system on an SMP. Cilk is a multithreaded language for parallel programming. The philosophy behind Cilk development has been to make the Cilk language a true parallel extension of C, both semantically and with respect to performance. The language CCC is more structured than the language Cilk, while it has comparative performance with Cilk.

1. Introduction

Although the performance of a single processor has continuously and dramatically increased for some time, recent evidences show that the performance of a single processor is gradually approaching its physical limitation and this rapid progress will start to slow down in the near future [9]. This situation makes the research of parallel computing becoming more demanding than before. In particular, the requests for good parallel programming languages and programming environments are prevalent. The rapid advance of VLSI and communication technologies has recently made symmetric multiprocessors (SMPs) easily available [1, 8, 14, 15]. These SMPs usually consist of two to tens of commodity microprocessors interconnected to a shared memory. Due to cost effectiveness, the trend for parallel computing systems has moved from vector processors to distributed memory multiprocessors, then to clusters of workstations. Now, with the availability of SMPs, the trend has quickly moved to clusters of SMPs.

Two kinds of parallelism have been exploited in parallel applications: data parallelism and task parallelism. Data parallelism is achieved by simultaneously performing the same operations on multiple data. Many applications that involve large amount of matrix computation are successful examples of exploiting data parallelism. Several high-level parallel programming languages that support data parallelism have been developed [6, 7, 10]. Task parallelism is accomplished by simultaneously executing multiple threads of control. Many applications that contain a number of fairly independent subtasks are successful examples of exploiting task parallelism. Several high-level

[*] This work was supported in part by the National Science Council under grant NSC 88-2213-E-194-004

parallel programming languages that support task parallelism have been proposed [2, 3, 4, 5]. This paper will report experiences on implementing a task parallel language CCC on top of the Cilk runtime system on an SMP. Cilk is a multithreaded language for parallel programming [5]. The philosophy behind Cilk development has been to make the Cilk language a true parallel extension of C, both semantically and with respect to performance. The language CCC is more structured than the language Cilk, while it has comparative performance with Cilk.

2. The Task Parallel Programming Language CCC

The parallel programming language CCC is a simple extension of the programming language C. This section will describe the main features supported by CCC for exploiting task parallelism. We will first describe its concurrency mechanism, and then its communication mechanism.

2.1 Concurrency mechanism

A CCC program consists of a set of functions and tasks. Functions are original C functions. A task is a function that can be called to execute concurrently with its calling function. A task can be implemented as either a heavy-weight process or a light-weight process (thread) depending on the underlying architectures. The syntax for defining a task is as follows:

```
task function-name ( parameter-list )
{
        function-body
}
```

The reserved keyword **task** introduces a task definition. The parameter-list is the same as the parameter-list of the C functions except that each parameter must be qualified by one of the following three *parameter-passing direction* qualifiers **in**, **out**, and **inout**. These qualifiers allow the compiler to generate more efficient code for performing data transfer between tasks. Since parameters can be passed *bi-directionally*, tasks don't return values. The main function of a CCC program needs to be defined as a task. This task is the initial task of the program.

Tasks in CCC programs can be dynamically created via structured constructs called *parallel regions*. There are two forms of parallel regions. The first form of parallel region can be used to concurrently create a list of tasks. The syntax for this form of *enumerative* parallel region is as follows:

```
par {
    task-invocation-list
}
```

The reserved keyword **par** introduces an enumerative parallel region. The list of tasks in the task-invocation-list will be created concurrently. The original task will be blocked until all the tasks created in the parallel region terminate.

The second form of parallel region can be used to concurrently create a list of tasks several times. The syntax for this form of *iterative* parallel region is as follows:

```
parfor (init-expr; end-expr; step-expr) {
        task-invocation-list
}
```

The reserved keyword **parfor** introduces an iterative parallel region. The list of tasks in the task-invocation-list will be created concurrently several times. The number of times these tasks will be created depends on the three expressions: init-expr, end-expr, and step-expr. These expression have the same semantics as the expressions in **for** constructs in C. The original task will be blocked until all the tasks created in the parallel region terminate. Two **parfor** constructs can be nested, and a **parfor** construct can be within a **par** construct. Note that only task invocations and **parfor** constructs are allowed in **par** or **parfor** constructs. Parallel regions are not as efficient and flexible as lower level fork/join constructs, while they support more structured program development.

2.2 Communication mechanism

Tasks in CCC programs can communicate via shared variables or message passing. We describe the communication mechanism via shared variables first. Global variables in a CCC program can be accessed by all the tasks in the program. Mutually exclusive accesses of global variables can be supported by appropriate uses of *mutual-exclusion locks* or *mutexes*. The syntax for declaring mutexes is as follows:

mutex *mutex-name-list*;

The reserved keyword **mutex** declares a list of mutexes. Each mutex supports two operations to implement a mutual exclusion. The operation **lock**(*m*) requests to lock a mutex *m*. The operation **unlock**(*m*) unlocks a mutex *m*. A typical mutual exclusion is implemented as follows:

lock(*mutex-name*);
    mutually exclusive accesses of shared variables
unlock(*mutex-name*);

Mutual-exclusion locks are relatively low-level communication constructs. We consider support more structured communication constructs such as monitors in the future.

We next describe the communication mechanism via message passing. Four kinds of asynchronous message passing channels are supported in CCC. The first kind of channels, called pipes, is used for one-to-one directional message passing. Pipes are the most primitive channels. All other three kinds of channels can be implemented using pipes. The syntax for declaring pipes is as follows:

pipe *message-type* *pipe-name-list*;

The reserved keyword **pipe** declares a list of pipes with message type *message-type*.

The second kind of channels, called *mergers*, is used for many-to-one directional message passing. The receiver of a merger can receive a message sent by any of the senders. Mergers can be used, for example, in programs using client-server paradigm. The syntax for declaring mergers is as follows:

merger *message-type* *merger-name-list*;

The reserved keyword **merger** declares a list of mergers with message type *message-type*.

The third kind of channels, called *dividers*, is used for one-to-many directional message passing. The sender can send a message to any one of the receivers. Dividers can be used, for example, in programs using producer-consumer paradigm. The syntax for declaring dividers is as follows:

divider *message-type* *divider-name-list*;

The reserved keyword **divider** declares a list of dividers with message type *message-type*.

The fourth kind of channels, called *pools*, is used for many-to-many bi-directional message passing. Each of the participants of a pool can send a message to and receive a message from the pool. Pools are the most general channels. The other three kinds of channels are special cases of pools. The syntax for declaring pools is as follows:

pool *message-type* *pool-name-list*;

The reserved keyword **pool** declares a list of pools with message type *message-type*.

Two operations are provided for manipulating channels. The operation **send**(*c*, *m*) sends a message *m* to a channel *c*. The operation **receive**(*c*, *m*) receives a message *m* from a channel *c*.

2.3 An example

The following is a simple CCC program using producer-consumer paradigm.

```
task producer(in divider int channel)
{
  int  i;

  for (i = 0; i < 10; i++)
      send(channel, i);
  send(channel, -1);
}
```

```
task consumer(in divider int channel)
{
  int  i;

  receive(channel, i);
      while (i != -1) {
          receive(channel, i);
      }
  }

  task main( )
  {
      divider int channel;
      int  i;

      par {
        producer(channel);
        parfor (i = 0; i < 3; i++) {
              consumer(channel);
        }
      }
  }
```

3. The Cilk Runtime System

Cilk is a multithreaded language for parallel programming [5]. The philosophy behind Cilk development has been to make the Cilk language a true parallel extension of C, both semantically and with respect to performance. The concurrency mechanism used in Cilk is unstructured **spawn** and **sync** constructs. They are very similar to fork and join constructs in Unix systems. The communication mechanism used in Cilk is also via unstructured mutual-exclusion locks. These mechanisms make Cilk such a low-level parallel language as C a low-level sequential language.

The performance of the Cilk language is achieved mainly by using a provably good work-stealing scheduling algorithm in the Cilk runtime system. Each processor in the system maintains a separate ready queue. When a processor empties its ready queue, it can “steal” ready threads from other processors. To constraint thread migration overheads incurred only for stolen threads, the Cilk compiler generates two clones of each task. A fast clone is generated for regular threads and a slow clone for stolen threads. Since the Cilk language has been efficiently implemented on symmetric multiprocessors and is also a simple extension of C, we have chosen to implement CCC on top of the Cilk runtime system. This would make CCC a more structured parallel language, while with similarly comparative performance.

4. The CCC Compiler

This section describes the compiler that translates CCC programs into C programs on top of the Cilk runtime system. We will first describe how channels in CCC are implemented using mutual-exclusion locks. We then describe how tasks are created and scheduled.

4.1 Implementation of channels

Each channel is implemented as an unbounded buffer. The buffer is a doubly linked list. Each element of the list is a pointer of type void so that it can accommodate various message types. Each channel has an integer variable, size, recording the size of each element. The size information is used to dynamically allocate memory for each element. Each channel also has an integer variable, count, that maintains the number of elements in the list if it is positive or the number of receivers waiting for messages if it is negative. The count information is used to determine whether a receiver should block on an empty buffer.

Since each pipe or merger has only one receiver, the operation **receive** for them can be implemented as follows:

```
receive(m)
{
   lock(L1);
   count--;
   if (count < 0) {
      unlock(L1);
    /* wait for non-empty buffer */
      lock(L2);
      lock(L1);
   }
   move the message in the head of
   the queue to m ;
   unlock(L1);
}
```

Here lock L1 is used to ensure mutual exclusion and lock L2 is used to ensure the condition that the buffer is not empty. On the other hand, each divider or pool may have more than one receiver, so the operation **receive** for them has to be implemented as follows:

```
receive(m)
{
     lock(L3);
     lock(L1);
     count--;
     if (count < 0) {
          unlock(L1);
    /* wait for non-empty buffer */
          lock(L2);
```

```
            lock(L1);
        }
        move the message in the head of
        the queue to m ;
        unlock(L1);
        unlock(L3);
    }
```

Here lock L3 is used to ensure that there is at most one task blocking on lock L2.

Because each channel is an unbounded buffer, the operation **send** for all four kinds of channels can be implemented as follows:

```
    send(m)
    {
        lock(L1);
        count++;
        insert message m into the end
        of the queue;
        if (count <= 0) unlock(L2);
        /* signal non-empty buffer */
        unlock(L1);
    }
```

4.2 Implementation of tasks

Since this implementation is targeted on top of the Cilk runtime system and symmetric multiprocessors, tasks are implemented as threads. The compiler will generate two clones of each task. A fast clone is generated for regular threads and a slow clone for stolen threads. We will illustrate these two clones using the main task in the producer-consumer program introduced earlier as an example. The fast clone generated for the main task is as follows:

```
  struct main_frame {
    StackFrame  header;
    struct {int argc; char **argv; }
       scope0;
    struct {int i; divider channel; }
       scope1;
};

int main_fast(int argc, char **argv)
{
   struct main_frame *frame;
   /* allocate frame's memory */
   frame = alloc(sizeof(struct
            main_frame));
   /* set a pointer to slow clone */
   frame->header.sig = main_sig;
   frame->scope0.argc = argc;
   frame->scope0.argv = argv;
   {
     int   i;
     divider channel;
     { /* par parallel region */
       {   /* spawn producer */
         frame->header.entry = 1;
         /* call site */
         frame->scope1.channel =
            channel;/* variable */
         push(frame);
         producer(channel);
         if (pop( ) == FAILURE)
           return 0;
           /* parent stolen */
       } /* spawn producer */
       for (i = 0; i < 3; i++) {
       /* parfor parallel region */
         { /* spawn consumer */
           frame->header.entry = 2;
           /* call site */
           frame->scope1.i = i;
```

```
            /* variable */
            push(frame);
            consumer(channel);
            if (pop( ) == FAILURE)
               return 0;
               /* parent stolen */
           } /* spawn consumer */
        } /* parfor parallel region */
      } /* par parallel region */
    }
    free(frame, sizeof(*frame));
    /* free frame's memory */
    return 0;
  }
```

The structure main_frame declares the activation frame for task main. Since the Cilk runtime system does not implement the scheduling of threads on the same processor, the threads on the same processor will be executed sequentially. When a new task is created, the new created child task will be executed first, and the parent task is pushed on the stack. An idle processor will always steal tasks on the bottom of a frame stack. Hence, for a stolen task, if it wants to continue to execute on a different processor, we need to save all the variables and the call site information of the parent task on the frame. The slow clone generated for the main task is as follows:

```
   static void main_slow(struct
    main_frame *frame)
   {
    int argc;
    char **argv;
    switch(frame->header.entry)    {
    /* restore call site */
      case 1: goto _sync1;
      case 2: goto _sync2;
      case 3: goto _sync3;
    }
    argc = frame->scope0.argc;
    argv = frame->scope0.argv;
    {
      int  i;
      divider channel;
      { /* par parallel region */
        {  /* spawn producer */
          frame->header.entry = 1;
          /* call site */
          frame->scope1.channel =
            channel; /* variable */
          push(frame);
          producer(channel);
          if (pop( ) == FAILURE)
            return;
            /* parent stolen */
          if (0) {/* come here only for
                     stolen task */
_sync1: channel =
              frame->scope1.channel;
          }
        } /* spawn producer */
        for (i = 0; i < 3; i++) {
          /* parfor parallel region */
          {  /* spawn consumer */
            frame->header.entry = 2;
            /* call site */
            frame->scope1.i = i;
            /* variable */
            push(frame);
            consumer(channel);
            if (pop( ) == FAILURE)
               return;
               /* parent stolen */
            if (0) {/* come here only
                 for stolen task */
    _sync2:   channel =
              frame->scope1.channel;
              i = frame->scope1.i;
            }
          } /* spawn consumer */
        }/* parfor parallel region */
      }
      { /* barrier for par parallel
          region */
        frame->header.entry = 3;
        if (sync( )) {
          return;
    _sync3:
        }
      }
    }
    free(frame, sizeof(*frame));
    /* free frame's memory */
    return;
  }
```

When a stolen task starts to execute on a different processor, it will first dispatch to its call site and then execute from there.

5. Performance Evaluation

We have done a very preliminary performance evaluation of the CCC compiler. The evaluation is done on a dual Pentium II/350 PC running Red Hat Linux 2.2.9. The benchmark set contains five programs: Fast Fourier Transforms, Knapsack, LU Factorization, Computing π using the Monte-Carlo method, and Finite Differences. The result is shown in the following table:

programs	CCC	Cilk	ratio
FFT	11.40s	11.32s	1.01
Knapsack	16.06s	14.77s	1.09
LU Factorization	54.12s	52.83s	1.02
Pi	2.76s	2.58s	1.07
Finite Differences	45.30	32.00s	1.41

Table 1. Execution time for benchmark programs

The first three programs in the benchmark are more natural to communicate via shared variables, while the last two programs are more natural to communicate via message passing. Except for the last program, their performance results are quite close. The bad performance for the last program is due to the lack of thread scheduling on the same processor in the Cilk runtime system. We believe that the performance result should also be quite close if thread scheduling on the same processor is supported.

6. Conclusions

This paper has reported experiences on implementing a task parallel language CCC on top of the Cilk runtime system on an SMP. Cilk is a multithreaded language for parallel programming. The philosophy behind Cilk development has been to make the Cilk language a true parallel extension of C, both semantically and with respect to performance. The language CCC is more structured than the language Cilk, while it has comparative performance with Cilk.

References

[1] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, M. Snir. SP2 System Architecture. *IBM Systems Journal*, 34(2):152-184, 1995.

[2] N. Carriero and D. Gelernter. Linda in Context. *Communications of ACM*, 32(4):444-458, 1989.

[3] K. M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object Oriented Programming Notation. In *Research Directions in Object Oriented Programming*, The MIT Press, pp. 281-313, 1993.

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

[5] M. Frigo C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212-223, June 1998.

[6] D. Gannon et al. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proc. Supercomputing'93*, 1993.

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

[8] F. M. Hayes. Design of the AlphaServer Multiprocessor Server Systems. *Digital Technical Journal*, 6(3):8-19, Summer 1994.

[9] W. E. Johnston. The Case for Using Commercial Symmetric Multiprocessors as Supercomputers. Technical Report, Information and Computing Science Division, Lawrence Berkeley Laboratory, 1997.

[10] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*, The MIT Press, 1994.

[11] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. White Paper, SGI, 1998.

[12] Sun Microsystems Incoporation. Sun and High Performance Computing. 1997.