CPar: A Parallel Language for Divide and Conquer Parallelism

Eric Methot, Marc Feeley, Bernard Gendron
Centre de recherche sur les transports
Dpt. d'informatique et de recherche oprationnelle
Université de Montréal
Montréal, QC, Canada

Abstract *Although recent advances in computer languages have made parallel programming an easier task, their use has been limited to coarse grained parallelism largely due to the overhead incurred by exposing parallelism. CPar is a language tailored to take advantage of a divide and conquer style of parallelism and provides an effective way to implement it on a shared memory multiprocessors. The objectives of CPar are twofold: limit processor communication and reduce the overhead of exposing parallelism. To achieve these goals, CPar implements "Lazy Remote Procedure Calls" and "Task Stealing" in an innovative way. Our tests have shown that even for fine grained applications, CPar can attain a total overhead of less than 45% on a 64 processor Sun Enterprise 10000 and 5% on a four processor Intel machine when compared to equivalent sequential programs. The present paper discusses the implementation of the CPar systems and also compares it to the Cilk system.*

*Keywords:*parallel, language, cpar

1 Introduction

Many languages designed to express control parallelism do a fine job at facilitating the task of programming certain styles of parallel algorithms. In particular, extensions to C such as Cilk [4] are well suited to express parallel algorithms in a divide an conquer fashion. Cilk programs have relatively low overheads and provide automatic load balancing.

In this paper we present CPar, also an extension to C that exploits this type of parallelism. Using an innovative implementation of "Lazy Remote Procedure Calls" and "Task Stealing" CPar achieves low total overhead. Although lacking the synchronization constructs and debugging tools found in Cilk, CPar does offer a substantial improvement in performance at a fine granularity.

We start by outlining the run-time and programming models of CPar programs and go on to describe two important aspects of the CPar system: "Lazy Remote Procedure Calls" and "Task Stealing". The following section discusses code transformations. Through experiments we study the run-time behavior of CPar programs. We measure the overhead of exposing parallelism as well as communication overhead. We also compare our system to Cilk. Our primary concern is speed. Our tests show that from a raw performance perspective, CPar's total overhead is up to two orders of magnitude smaller than Cilk's.

2 The Run-Time Model

A description of the run-time behavior of CPar programs helps put into perspective the different techniques used to obtain a low run-time overhead. At start-up, a fixed number of worker OS threads are launched and each of them implements the following strategy: when a thread has no work to do or is waiting for another task to end, it finds an unexecuted task and starts executing it. All threads start with-

out any work to do except the main thread which executes the main function. During the execution of a task, a worker might encounter a "fork" construct. At this point, a task is created and made available to the other workers thus enabling the parallel execution of the program.

3 The Programming Model

CPar is an extension to C and requires only a single keyword to express parallelism. The keyword *par* extends the syntax of a function call in the following manner:

function(arg1,...,argN) par {...};

The *par* keyword is a fork construct for which the semantics are to execute the function call concurrently with the instructions in the compound statement. The result of this parallel call is the result of the function call itself. Task synchronization is implicit and takes place at the end of the compound statement. Although slightly less expressive than the explicit synchronization mechanism found in the Cilk system it does require considerably less effort to manage because it does not employ locks on the critical path.

4 Lazy Remote Proc. Calls

As the run-time model suggests, a new thread is not created every time a fork construct is encountered in the execution flow. Instead, we use a data structure that we make available to all worker threads to enable parallelism. A lazy remote procedure calls (LRPC) as described by Feeley [3] is simply a way to tell the system that the creation of a new task which performs a function call has been requested without actually creating an independent thread of execution for it. In CPar, the creation of a LRPC is done in three steps: the allocation, initialization and the publication of a data structure called a task descriptor.

The task descriptor must contain sufficient information to reconstruct the original function call. The publication of this descriptor allows other workers to see it and execute it on behalf of its creator. Descriptors are allocated on the run-time stack and their publication is done by pushing a pointer to their descriptor onto a distributed deque of LRPC.

When the underlying architecture use registers to pass arguments to a function, LRPC arguments will need to be copied when the call is made. The is a portable technique but it requires one or two extra memory references per function argument. When arguments to a function are passed using the run-time stack we can use the context of the function call as our task descriptor. With this approach, we only need to add an extra space for the result in case the task is stolen. Another field is also needed for the address of a proxy function. We note that using this optimization may not always be beneficial.

5 Task Stealing

Task stealing is the technique used by CPar to provide automatic load balancing. Whenever a worker thread is idle, its starts looking for work in the deques of other worker threads. A worker also looks for work while waiting at a synchronization point. Task stealing requires a distributed task deque and accompanying access control protocol to guarantee mutually exclusive access to individual task descriptors. Because task stealing is expensive it is wise to limit its frequency at which it occurs. The access control protocol is relatively straightforward. We will refer the reader to Feeley [2] for the details and proof of correctness.

The frequency at which stealing occurs can be reduced by transferring large chunks of work at each task steal. For this reason, CPar workers steal tasks from the bottom of the task deque. When using a divide and conquer approach, older tasks will typically contain more work than newer ones [2, 5]. One interesting aspect of task stealing is that it requires little extra code in the heavily executed areas of the parallel program. The process of stealing takes

place in the same function that handles worker synchronization. Since synchronization only occurs when a worker is actually waiting for another to complete a stolen task, task stealing should not account for much of the total overhead.

6 Code Generation

The CPar compiler targets the C dialect understood by the GNU C Compiler. To allow the GNU C compiler to do a good job optimizing the source code, it is important to minimize the extent of code transformations. Thus, CPar only transforms the source code locally where parallel constructs are found. We note that leaf calls carry no overhead as their execution flow never reaches the parallel code segments.

Efficient access to the local deque of task descriptors is important for overall performance. We use a data structure aligned on a 4K byte boundary that contains the head pointer and the entries of the deque in an array. The deque tail pointer is kept in a machine register. With a simple mask operation we can recover a pointer to the 4K byte area and thus to out data structure. On the SPARC processor where there are more registers available, a second machine register is dedicated for this purpose.

For the Intel processor, a total of 11 extra instructions including only 4 memory references are necessary to create, initialize and publish a task descriptor. This is the key to the performance obtained by CPar and discussed further in our experimental results. Similar, although less impressive results are found for the SPARC architecture even though we cannot take advantage of the run-time stack as we do on Intel processors. Consequently, the number of instructions is greater and the number of memory references depends on the number of arguments to the parallel function call.

7 Experimental Results

In this section we explore the run-time behavior of CPar programs. We are interested in measuring the communication overhead between processors and comparing the performance of CPar programs to their Cilk counterparts. Tests where conducted on two platforms: An Intel computer with four Pentium processors running at 150MHz and a Sun Enterprise 10K with 64 processors running at 400MHz each. Our tests used up to 58 processors and all measurements where done using the average times of 10 executions.

7.1 Communication Overhead

From a parallel processing standpoint, a low communication frequency is usually preferable. As such we have put much effort into limiting this overhead. In CPar, communication occurs when a worker attempts to steal work from another and also when they synchronize. CPar incorporates features that limit both types of communications. One of these is to steal work from the bottom of the deque and the other is to check for task completion before calling the synchronization procedure.We measured the normalized communication overhead as follow: $B_c = \frac{n \times T_n - T_1}{T_s}$ where $n = 58$. Table 1 shows the communication overhead obtained on a set of test programs. The overhead is typically under 5%. Considering the high degree of parallelism found in these programs, we find the level of communication to be relatively low.

7.2 A Comparison to Cilk

To verify the competitiveness of our implementation we proceeded to compare CPar to version 5.2 of the Cilk system. The results of this benchmark can be found in Table 1.

Curiously, some of our parallel programs are faster than their sequential equivalents. This unusual circumstance is probably caused by cache effects and/or better instructions ordering. Thus, they are not representative of a normal overhead measurement. The Cilk equivalent program on the other hand all have an important total overhead when the granularity of the task is fine.

CPar outperforms Cilk in speed for these

Prog.	T_s	T_1	T_{58}	$\frac{T_1}{T_s}$	B_c
fib	257.7	345.2	6.1	34.0%	3.4%
sum	116.5	165.8	2.46	42.3%	-19.8%
queens	221.6	265.1	4.74	19.6%	4.4%
knap	556.8	558.0	10.0	0.0%	4.0%
scan	111.3	125.2	2.5	12.5%	2.3%
mmul	795.6	787.8	13.5	-1.0%	-0.5%
poly	94.6	107.2	2.06	13.3%	13.0%

Table 1: Overheads on the Sun E-10K.

programs. We note however that as the granularity of the tasks grows, the differences in performance between CPar and Cilk tends to diminish.

8 Conclusion

We have introduced the CPar language designed for parallel programming in a divide and conquer style. Although the techniques used by CPar are not entirely new, their implementation as a source to source transformation as well as their integration and evaluation in a high quality C compiler are. We have seen that few instructions are necessary at each parallel call site for implementing parallelism and automatic load balancing which results in low total overhead. Our preliminary tests show that CPar has good potential and compares favorably to the Cilk system from a performance perspective when executing fine grained programs.

Simple and elegant, CPar offers the programmer the ability to express parallel algorithms without worrying about the run-time overhead associated by exposing parallelism. Yet CPar is still in an early stage and should include in the near future some interesting features found in the Cilk system such as synchronization constructs and debugging tools.

CPar Prog.	T_s	T_1	T_4	$\frac{T_1}{T_s}$	$\frac{T_4}{T_s}$
fib(34)	1.88	1.88	0.47	0.0%	4.0
sum(4e6)	1.48	1.16	0.30	-21%	4.9
queens(13)	2.76	2.84	0.71	2.9%	3.9
knap(34)	1.63	1.66	0.42	1.8%	3.9
scan(4e6)	2.70	2.82	0.76	4.4%	3.6
mmul(384)	1.56	1.60	0.41	2.6%	3.8
poly(8e3)	2.89	2.78	0.70	-3.8%	4.1
Cilk Prog.	T_s	T_1	T_4	$\frac{T_1}{T_s}$	$\frac{T_4}{T_s}$
fib(34)	1.88	8.27	2.12	331%	0.88
sum(4e6)	1.40	4.24	1.11	203%	1.26
queens(13)	2.76	5.94	1.52	115%	1.82
knap(34)	1.63	2.61	0.703	60%	2.33
scan(4e6)	2.70	9.17	2.42	240%	1.12
mmul(384)	1.56	2.09	0.56	34%	2.78
poly(8e3)	2.89	2.90	0.772	0.1%	3.74

Table 2: CPar vs Cilk on Intel machine.

References

[1] Gendron B. and Chabini I. Parallel Performance Measures Revisited. In *High Performance Computing Symposium'95*, pages 381–392, Montreal, Canada, jul 1995.

[2] Marc Feeley. An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors. Technical Report IRO-869, Dept. d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1993.

[3] Marc Feeley. Lazy Remote Procedure Calls and its Implementation in a Parallel Variant of C. In Queinnec C. Ito T., Halstead R., editor, *Parallel Symbolic Languages and Systems 95*, Sprigner-Verlag Lecture Notes in Computer Science 1068, pp. 3–21, 1995.

[4] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of the 1998 ACM SIGPLAN Symposium on Programming Language Design and Implementation*, jun 1998. MIT.

[5] Francis L'Ecuyer. Conception et réalisation d'une variante parallèle de C basée sur la création paresseuse de tâches. Master's thesis, Dept. d'Informatique et de Recherche Opérationnelle, Université de Montréal, dec 1997.