Lazy Remote Procedure Call in C: Implementation and Performance

Mario Latendresse
email: latendre@iro.umontreal.ca

Marc Feeley*
email: feeley@iro.umontreal.ca

January 6, 1997

Abstract

Lazy Remote Procedure Call (LRPC) is a high-level parallel construct, based on Lazy Task Creation (LTC), which was designed to address the problem of task creation overhead and load balancing in the context of a shared-memory parallel programming model. Previously, Feeley described how LRPC could be implemented as a source to source transformation for the C language. We have implemented this approach and have evaluated its performance on benchmark programs. This paper reports on our experience, pointing out the disadvantages of the approach in terms of restrictions on the C language, and the advantages in terms of its simplicity and performance.

1 Introduction

To express parallelism in C we use a fork-join high level construct that can be used on a function call. Syntactically this is an operator, noted "!!", as in:

function($\arg_1, \ldots, \arg_n$) !! { *declaration-list statement-list* }

The compound statement is the added element to the function call. The function's body and the compound statement are executed concurrently. The execution can proceed once both are terminated and the value of the expression, if there is one, is the function's value.

Among other things, this parallel construct allows to express parallelism in divide and conquer algorithms without substantial code modification.

Our approach in implementing such a construct is to perform a source to source transformation of the C program. Such a technique has the major advantages of allowing to use the manufacturer C compiler and causing no lost of performance in sequential part of the code. Indeed, most of the original code is kept unchanged and only where the parallel construct !! is used a transformation is performed. The transformed program is linked to a small runtime system that takes care of processor initialization and provides a simple task stealing mechanism.

A parallel call using !! is transformed to allow the function call to be stolen by another processor whereas the compound statement is locally executed immediately. Once the compound statement is terminated, if the function call is not stolen and is not evaluated by another processor, it is locally evaluated. If the function call is stolen and it's evaluation is not yet terminated, the local processor tries to steal another function call from another processor while waiting for the actual call to terminate. A queue of deferred function calls is maintained for every processor. Indeed the major transformation done at the source level is to allow to perform this task.

*DIRO, Université de Montréal

Program	Time in seconds T_{seq}	T_1	Speedup T_{seq}/T_p p=	2	3	4
sum	0.27	0.27		1.92	2.97	3.85
mm	2.08	2.13		1.82	2.60	3.46
scan	0.78	0.78		2.00	3.00	3.71
poly	1.92	1.94		1.97	2.90	3.69
fib	0.92	2.18		0.42	0.82	1.67

Table 1: Benchmark sequential and parallel-one-processor run time, and p=2,3,4 speedup

2 Performance

To test the performance of our approach we have used five benchmarks: a summation of 10^6 integers (sum), a multiplication of two 150x150 matrices of integers (mm), a parallel prefix sum applied on a vector of 10^6 integers(scan), a symbolic squaring of a 2000 terms polynomial(poly) and the computation of the 30th Fibonacci number by a recursive method (fib).

The first four benchmarks, namely sum, mm, scan and poly show good performance absolute speedup whereas the fifth, i.e. fib, shows weak absolute speedup. The reason for this weak performance of fib is quite simple: there are as many fork calls as there are additions of integers, in other words the granularity of the parallel calls are very fine; and since these additions are the only useful computation done, the fork calls overhead become substantial.

The experiment was done on a Silicon Graphics computer with four processors. The execution times are given in seconds. In these the time taken by the initialization phase of the programs has been eliminated as well as the time to create the initial heavy processes to handle parallelism under the operating system. The sequential times are the times taken by the original C programs. Note that speedups are given versus the sequential time, not the one processor time.