Mapping Techniques for Parallel Evaluation of Chains of Recurrences

Eugene V. Zima *
Dept. of Computational Math. and Cybernetics
Moscow State University
Moscow,119899, Russia
zima@cs.msu.su

Karthi R. Vadivelu †, **Thomas L. Casavant**
Dept. of Elec. and Comp. Engg.
University of Iowa
Iowa City, IA 52242, USA
[vadivelu,tomc]@eng.uiowa.edu

Abstract

This paper examines the parallelization of a technique for speeding up the evaluation of potentially-complex real-valued functions at a large number of points. The technique being parallelized generates a Chain of Recurrences (CR) which is then used to compute the function incrementally (i.e., by using the results of one iteration in calculating the value of the function in the next iteration). This paper examines the possibilities for mapping the CR technique onto parallel machines. The factors influencing the choice of mapping alternatives include efficiency, speedup, and most interestingly, the potential for improved error distributions.

1 Introduction

A common component in the analysis and solution of many problems, is the iterative evaluation of a function $G(x)$ over a domain of points. More specifically, given a starting point x_0 and an increment h, the evaluation of the function $G(x_0+ih)$ for $i = 0,1,\ldots,n-1$ occurs frequently in applications such as plotting graphs of functions, simulations, and signal processing applications. Straightforward evaluation of complex functions may not be adequate due to the cost in terms of computation time. One way to speed up this process sequentially, is to compute the function incrementally, i.e., use the results of one iteration in calculating the value of the function in the next iteration. For example, to compute the values

$$f_0(i) = \frac{exp(0.01i^2 - 0.2i)}{2^{0.3i+1}}$$

for $i = 0, 1, \ldots, n$ we can construct the chain of recurrences:

$$f_0(i) = \begin{cases} \frac{1}{2}, & i = 0 \\ f_0(i-1) * f_1(i-1), & i > 0, \end{cases}$$

$$f_1(i) = \begin{cases} \frac{exp(-0.19)}{2^{0.3}}, & i = 0 \\ f_1(i-1) * exp(.02), & i > 0, \end{cases}$$

There will be only two multiplications performed at each step of the loop which computes $f_0(i)$ values:

```
f0:=1/2; f1:=exp(0.-19)/2^0.3; f2:=exp(0.02); write(f0);
for i :=1 to n-1 do f0:=f0*f1; f1:=f1*f2; write(f0) od;
```

This approach has been shown to provide substantial reductions in computation time in the sequential case [5].

This paper examines the problem of mapping of this recurrence technique onto parallel machines. The underlying approach is the same as in [4], where a chain of recurrences (system of recurrence relations) for a given function is constructed.

*Work reported herein was supported in part by the RFBR (Russia) under Grant 95-01-01138a and in part by Grant J12100 from ISF and Russian Government.

†Currently at: Intel Corp., FM5-64, 1900 Prairie City Road, Folsom, CA 95630

Then, instead of evaluating the original function directly, the evaluation of the function can be reduced to just k additions and/or multiplications , where k is the "length" of the chain of recurrences. This method requires two steps:

- Constructing the chains of recurrences.
- Evaluation of the function over the set of points using the recurrent relations.

The second step, that of computing the function using the relations is highly parallelizable. However, there exist several alternatives for mapping onto parallel machines. The factors influencing the choice of mapping alternatives include efficiency, speedup and most interestingly, different error distributions. The iterative solution using the recurrence technique involves a cumulative error effect. By parallelizing appropriately, not only is greater speedup obtained, but reduced error accumulation can also be gained.

The parallel evaluation of recurrences has been studied before, but in a more general context, and only using functional parallelism. Methods to determine the degree of parallelism present in a recurrence relation, and a method for exploiting it have been discussed in [10]. Other approaches to parallelizing arbitrary recurrences have been studied in [1], which are similar to the functional-parallel approach which we describe later. It is useful to note that, several methods exist [6, 9] to compute a degree k polynomial in $O(log_2 k)$ parallel steps. In the case where the polynomial is to be evaluated in a loop, using a chain of recurrences allows the evaluation to be performed in $O(1)$ parallel steps.

Our focus is primarily on a variety of mapping techniques – both functional and data parallel – which influence speedup, as well as other properties such as the accumulated computational error in function evaluation.

2 Chains of Recurrences

Given constants $\varphi_0, \ldots, \varphi_{k-1}$, a function f_k defined over $N \cup \{0\}$, and operators $\odot_1, \ldots, \odot_k$ equal to either $+$ or $*$, we define a *Chain of Recurrences (CR)* as the set of functions $f_0, f_1, \ldots, f_k$ connected in such a way, that for $0 \le j < k$

$$f_j(i) = \begin{cases} \varphi_j, & \text{if } i = 0, \\ f_j(i-1) \odot_{j+1} f_{j+1}(i-1), & \text{if } i > 0. \end{cases} \quad (1)$$

Further, to denote the CR above we will use the shorthand

$$f_0(i) = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \varphi_2, \ldots \odot_k, f_k\}(i).$$

The CR for $f_0(i)$ from section 1 can also be rewritten as

$$f_0(i) = \{\frac{1}{2}, *, \frac{exp(-0.19)}{2^{0.3}}, *, exp(0.02)\}(i).$$

Given a CR $\{\varphi_0, \odot_1, \varphi_1, \odot_2, \ldots, \varphi_{k-1}, \odot_k, f_k(i)\}$ we call it a

- *simple* CR, if $f_k(i)$ is a constant;
- *pure-sum* CR if $\odot_1 = \odot_2 = \ldots = \odot_k = +$; [1]

[1] it is useful to observe, that a simple pure-sum CR of the length k defines polynomial $f_0(i)$ of degree k, and constants $\varphi_0, \ldots, \varphi_k$ are nothing more than the table of finite differences of $f_0(i)$ taken at the point $i = 0$.

- *pure-product* CR if $\odot_1 = \odot_2 = \ldots = \odot_k = *$.

The integer k from our definition is called the *length* of the CR. It is easy to see that the number of arithmetic operations needed to compute the next value of a simple CR is equal to the length of the CR. Therefore, the length of a CR gives an indication of its evaluation cost.

For a given $G(x), x_0$ and h, it is possible to construct a CR Φ or an expression with CRs as operands such that $\Phi(i) = G(x_0 + i * h)$. A general algorithm to construct CRs for a given formula $G(x)$ was considered in [3, 8]. This algorithm can be applied to any function. Instead of finding recurrences for a particular class of function, it automatically generates a recurrence representation for a wide variety of common functions in order to obtain more efficient computational procedures for their evaluation. The algorithm is based on two main principles:

- replacing the trivial subexpression x by the simple recurrence $\{x_0, +, h\}$ on the parse tree of G;
- application of operations from expression $G(x)$ to recurrences already obtained during end-order traversal of the parse tree in order to construct CRs which embrace larger subexpressions of the given original expression

This algorithm uses CR-construction rules given in [8]. Most of these rules are very simple. For example, given pure-sum CRs $f(i) = i^2+1 = \{1, +, 1, +, 2\}$, $g(i) = 3*i+2 = \{2, +, 3\}$ and constant c, it is easy to get CRs for $f(i)+g(i)$ and $c*f(i)$: $f(i) + g(i) = \{3, +, 4, +, 2\}$, $c * f(i) = \{c, +, c, +, 2c\}$.

Since this technique is based on the use of previously computed values to compute the next value, any computational error in the previous step will be passed on to the new value in addition to any error in the current step. In general, this error is within reasonable limits, but for a large number of iterations, the error can become significant. This can be rectified by "refreshing" the recurrence relations, i.e., by reinitializing the value of components periodically. If the refreshing is done over the regular number of points, we find that this analogous to a well known sequential (and parallel) program transformation, called "strip-mining/striping" [7] or "loop unrolling".

Given $F(i)$ which has to be evaluated for $i = 1, \ldots, n$ (this corresponds to the initial "linear" problem), assume that $n = m \cdot q$. We can compute the required values using

- strip-mining:
  $F(j \cdot q + l), j = 0, \ldots, m-1; l = 1, \ldots, q;$ or
- striping:
  $F((l-1) \cdot m + j), j = 1, \ldots, m; l = 1, \ldots, q;$

It can be seen that with the original CR technique, the error is accumulated through $n = m \cdot q$ steps of computation, but after strip-mining/striping it is accumulated through no more than $m + q$ steps. Two features can be noted here:

- We need not reconstruct CRs to strip-mine/strip computations and
- This transformation allows us to exploit both parallelization and CR-based improvement of the code.

3 Parallel Implementations

Function evaluation in loops using recurrence relations is inherently parallelizable. Consider a function $G(x)$ to be evaluated beginning at $x = x_0$, over a domain of n points with an increment h. There are two fundamentally different ways of parallelizing this problem: Functional Parallelism and Data Parallelism. The linear nature of recurrence relations gives rise to functional parallelism. If a function can be expressed as a CR of length k, it follows that the evaluation of the function will require k steps. Since each of these steps is independent (from the linear nature of the CR) the evaluation can be performed in parallel. Data parallelism arises from the fact that given p processors, the domain of n evaluation points can be divided into p sub-domains which can then be mapped to a parallel computer in two ways: *small increments* and *large increments*.

The following function will be considered to illustrate the three cases.
$f(x) = \frac{e^{(a_4x^4+a_3x^3+a_2x^2+a_1x+a_0)}}{2^{b_3x^3+b_2x^2+b_1x+b_0}}$, where $a_0 = 1$, $a_1 = \frac{503}{120}$, $a_2 = -\frac{1111}{480}$, $a_3 = \frac{193}{480}$, $a_4 = -\frac{41}{1920}$, $b_0 = 1$, $b_1 = \frac{205}{84}$, $b_2 = -\frac{293}{504}$, $b_3 = \frac{17}{504}$.

Currently, the CRs are constructed using a procedure implemented in Maple [2], which generates a symbolic representation of the CR. Using the symbolic representation, the chain of recurrences for each sub-domain can be computed by substituting the appropriate values of x_0 and h in the symbolic representation, thus eliminating the need to recompute the recurrences for each sub-domain. The symbolic representation of the CR for this function is:

$$\{f(x_0), *, \frac{e^{(a_4(hx_0^3+e_3(x_0+h))+a_3e_3+a_2e_1+a_1h)}}{2^{(b_3e_3+b_2e_1+b_1h)}}$$
$$, *, \frac{e^{(a_4(2he_3+e_2(x_0+2h))+a_3e_2+2a_2h^2)}}{2^{(b_3e_2+2b_2h^2)}}$$
$$, *, \frac{e^{(a_4(3he_2+6h^3(x_0+3h))+6a_3h^3)}}{2^{(6b_3h^3)}}, *, e^{(24a_4h^4)}\}$$

where, $e_1 = hx_0 + h(x_0 + h)$, $e_2 = 2he_1 + 2h^2(x_0 + 2h)$, $e_3 = hx_0^2 + e_1(x_0 + h)$

Functional Parallel Evaluation. The function $f(x)$ defined above has a CR of length 4 , and thus requires 4 steps to evaluate each iteration. Given 4 processors, each of these 4 operations may be executed in parallel. This requires synchronization at each iteration, which combined with the communication latencies may not yield an acceptable level of efficiency in the case of this function since the CR has a length of only 4. This method of parallelizing can be combined with the methods described below by having 4 processors per subsequence perform the 4 operations in parallel, in addition to each subsequence being performed in a data-parallel fashion.

Strip-mining. Parallelizing using the strip-mining implies that we calculate the initial recurrence relations for the p sub-domains first. The starting values for each of the recurrence relation for each processor would be $x_0 + h(j-1)\frac{n}{p}$, where j is the number of processor, and each processor would compute n/p values over the step h. For the above expression, calculating the CR produces a chain of length 4, with, $\odot_1 = \odot_2 = \odot_3 = \odot_4 = *$. Thus the computation of each successive value requires 4 multiplications, and uses the value obtained from the previous computation as the starting point.

Due to the fact that each processor computes a smaller sequence than in the sequential case, the peak accumulated error will be much less that in the sequential case.

Striping. Here again the domain of n values is divided into p sub-domains, but unlike the previous case, the sub-domains are interleaved. The starting points of the p sets of recurrence relations would then be, $x_0 + h(j-1)$, where j is the number of processor and each processor would compute n/p values over the step $H = hp$. The accumulated error in each subsequence is much smaller than the total accumulated error, had the entire sequence been calculated sequentially. In this case though, the shape of the error curve should be different from the previous case, since, due

to the interleaving, all the subsequences terminate in the same region of the overall sequence.

4 Experimental Results

In this section, empirical data is presented that illustrates and confirms the ideas and claims outlined in the previous sections.

Error Characteristics

A number of different types of functions were evaluated using both the small and large increment methods. The error characteristics have been compared and plotted. The functions considered can be divided into two primary categories: those yielding CRs in which all the operators are additions (*pure-sum CRs*) and those yielding CRs in which all the operators are multiplications (*pure-product CRs*). The error characteristics of both types are studied in the following subsections. The results were measured on a SPARCstation SLC, implemented in C. The CRs for the functions were generated using a Maple-based implementation. The error characteristics have been compared based on the following criterion:

- Type of function
- Mapping method used

Pure-product CR Example

The example function is the same as the function of section 3. Figure 1 shows percentage error characteristics versus iteration number in the evaluation of the example function using three different techniques:

- 1. Direct evaluation,
- 2. Small increment method, and
- 3. Large increment method.

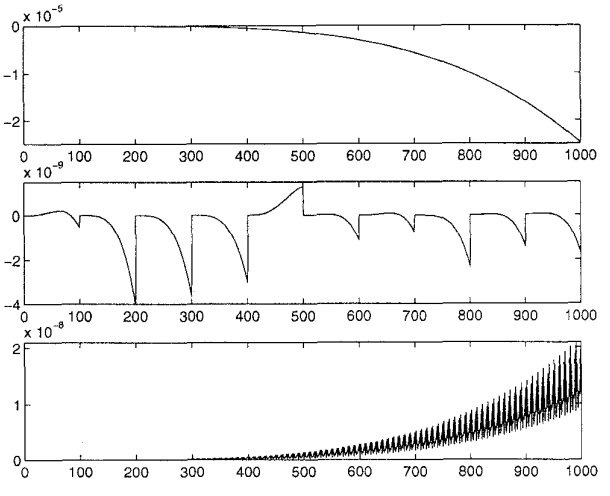

Figure 1: Percent Error with (a) Sequential CR Evaluation (b) Small Increments (c) Large Increments

The domain of x was $[0, 10]$, and was partitioned into 1000 points with $h = 0.01$. As predicted, the errors show a dramatic decrease when evaluated using either the small or large increment techniques. Over the range of points evaluated, Figure 1 shows the error improvement to be from three up to four orders of magnitude over the sequential CR solution.

Figures 5 and 6 give a perspective on the improved error performance by first using one of the increment methods versus the sequential CR evaluation, and then between the two parallel increment methods being described in this paper.

Figure 5 compares the small increment parallel evaluation method to the sequential CR evaluation method, and it can be seen that the difference is dramatic. Figure 6 shows the difference in the error characteristics between the two forms of parallel evaluation based on increments. As shown, the small increment method has a significant advantage over the large increment one for this function. This behavior is discussed shortly.

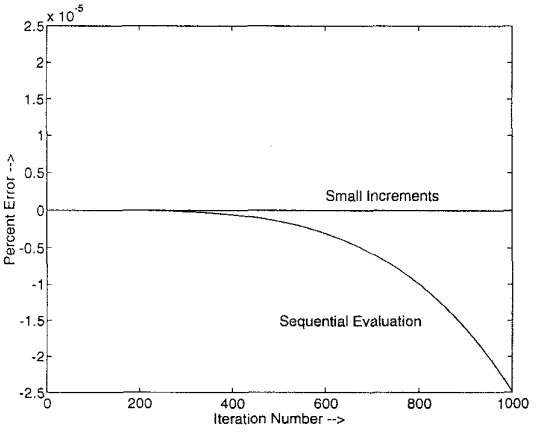

Figure 2: Comparison of percent errors with (a) Sequential Evaluation(b) Small Increments [Pure Product CR]

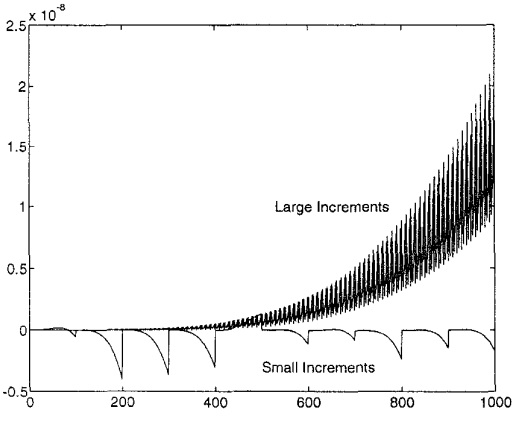

Figure 3: Comparison of percent errors with (a) Small Increments (b) Large Increments [Pure Product CR]

Pure-sum CR Example

The function used to illustrate pure sum CRs is the simple polynomial shown below:

$$f(x) = x^5 - 18x^4 - 11x^3 - 19x^2 - 3x + 2$$

this function yields a pure sum CR of length 5. To evaluate this function, the domain of x was $[0, 10]$, and was partitioned into 1000 points with $h = 0.01$.

Discussion of Error Characteristics

The errors in function evaluation using CRs could arise due to two sources: 1) Computation Error (Floating point error that occurs when we compute the initial values of the

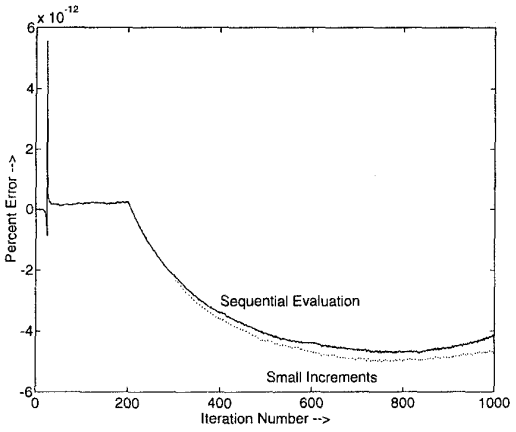

Figure 4: Comparison of percent errors with (a) Sequential Evaluation(b) Small Increments [Pure Sum CR]

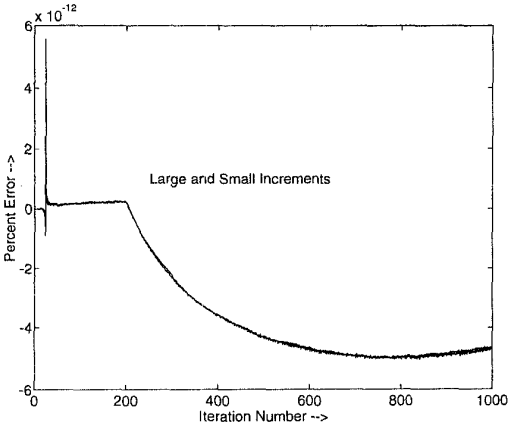

Figure 5: Comparison of percent errors with (a) Small Increments (b) Large Increments [Pure Sum CR]

constants), and 2) Representation Error (Error caused by loss of information due to the finite word size). Typically when we compare the function's values from evaluation using CRs with normal straightforward evaluation, the error of representation will affect both quantities. Thus it is the error of initialization that plays a crucial role in causing the value obtained using CRs to be different from the value obtained through straightforward evaluation. Since successive values are computed using past values of the components of the CR, the nature of the function also plays an important part in the progression of the error.

Let's consider a simple pure-sum CR
$\Phi(i) = \{\varphi_0, +, \varphi_1, +, \ldots, +, \varphi_k\}(i)$
and the approximate simple pure-sum CR
$\tilde{\Phi}(i) = \{\tilde{\varphi}_0, +, \tilde{\varphi}_1, +, \ldots, +, \tilde{\varphi}_k\}(i)$,
obtained after floating point initialization of $\Phi(i)$. Let σ be an upper bound for the relative error of initialization (we assume that $\sigma << 1$). The value of σ depends on the concrete floating point representation. It can be shown (see [11] for details) that $|\tilde{\Phi}(i) - \Phi(i)| \leq \sigma \cdot \Gamma(i) \cdot \max_j |\varphi_j|$,

where $\Gamma(i) = \underbrace{\{1, +, 1, +, \ldots, +, 1\}}_{k+1 \text{ times}}(i)$.
The function $\Gamma(i)$ is defined by the following formula

$$\Gamma(i) = 1 + i^{(1)} + i^{(2)}/2! + \ldots + i^{(k)}/k!,$$

where $i^{(j)} = i(i-1)\ldots(i-j+1)$ is j−th falling factorial. In the case of a function having a pure sum CR, the above equations show that the error $|\tilde{\Phi}(i) - \Phi(i)|$ does not depend upon the value of the function $\Phi(i)$. Thus this means that both the large increment and the small increment methods will have the same error characteristics. This is borne out by the empirical results plotted in figure 5.

Consider simple pure-product CR
$\Phi(i) = \{\varphi_0, *, \varphi_1, *, \ldots, *, \varphi_k\}$
and the approximate simple pure-product CR
$\tilde{\Phi}(i) = \{\tilde{\varphi}_0, *, \tilde{\varphi}_1, *, \ldots, *, \tilde{\varphi}_k\}(i)$,
obtained after initialization of $\Phi(i)$. It is easy to show, that $|\tilde{\Phi} - \Phi| \leq \sigma\Gamma(i)\Phi(i)$. The last means, that error depends on current value of the function, multiplied by $\Gamma(i)$ and by initialization error σ. That's why in the case of the pure product CR, we find a difference in accumulated error behavior with respect to the different mapping schemes.

Parallel Performance Results

The timing results shown in Tables 1 and 2 are presented to show the potential for speeding up the parallel implementation of the CR method, and for further improving the performance of evaluating a function over a large domain of values. The data shown in Table 1 were collected on a 96-processor Intel Paragon at ETH-Zürich, and the function of Section 3 was evaluated over a domain, x, of $[0, 10)$, and was partitioned into $n+1$ points, with $h = 10/n$. The results in Table 2 were taken with the same conditions as for the Paragon, but on a 26-Processor Sequent Balance system at Kent State University. The following 6 versions were implemented.

- 1. **D.S**: direct sequential evaluation.
- 2. **C.S**: sequential evaluation using CRs.
- 3. **C.F**: functional Parallel method. This evaluation involved creating a large-grained pipeline among the k-stages of the CR and passing data between processors after each step of the CR evaluation.
- 4. **D.P**: direct evaluation of the function in parallel, by dividing the domain $[0, 10]$ into 10 sub-domains.
- 5. **C.DP**: data-parallel evaluation of the function using CRs, by dividing the domain $[0, 10]$ into 10 sub-domains.
- 6. **C.FD**: parallel evaluation of the function using CRs, by a hybrid of methods 3 and 5 above.

Cases 1 - 5 of the previous 6 cases were implemented on both systems. Case **C.FD** was only implemented on the Paragon. For all cases except cases **C.F** and **C.FD**, 10 processors were used. In case **C.F**, because the function being evaluated had a CR of the length 4, only four processors were used. In case **C.FD**, 40 processors were used with 10 being employed for the sub-domains, and 4 processors for each sub-domain. As can be seen from the Tables, the best choice of method is highly dependent on the number of available processors, and upon the number of points (i.e., the resolution over the interval) to be evaluated. First, examining the execution times for the Paragon shown in Table 1, it can be seen in cases **D.S** and **C.S**, that there is a linear increase in execution time as the problem instance grows. This is to be expected due to the linear time complexity of the evaluation of a function, either directly,

n	D.S	C.S	C.F	D.P	C.DP	C.FD
.1	0.0024	0.001	0.029	0.385	0.382	0.014
1	0.022	0.001	0.283	0.417	0.394	0.107
10	0.217	0.015	2.811	0.439	0.408	0.906
100	2.158	0.151	28.310	0.651	0.448	9.232
500	10.793	0.754	141.323	1.669	0.608	45.972
1000	21.448	1.527	282.223	3.145	0.863	90.439

Table 1: Execution times of example function on a Paragon(in sec.) Problem size in thousands.

n	D.S	C.S	C.F	D.P	C.DP
.01	0.0162	0.0012	0.2092	0.4312	0.4292
.1	0.1120	0.0141	0.2252	0.4399	0.4390
1	1.1135	0.0922	0.4380	0.5433	0.4421
10	11.129	0.8763	2.8404	1.5460	0.5222
100	111.26	8.7152	26.796	11.605	1.3100

Table 2: Execution times of example function on a Sequent Balance(in sec.) Problem size in thousands.

or using CRs. Considering case **C.F**, the functional-parallel case, note that this case performs worse than for either of the sequential cases. While this may seem very discouraging at first sight, it is actually quite to be expected. The grain size of the computation per processor, compared to the communication per computation step, is very small. Therefore, much more time is being spent in communication than is being gained through the parallelism, which is only 4 processors. In case **D.P**, the data-parallel implementation of the direct method, a significant (roughly 7 times) improvement is shown over the direct sequential method (**D.S**), but notice that for all problem sizes examined, that the execution time exceeds that of the sequential CR method (**C.S**). This is potentially disturbing, because it might be hoped that the benefit from parallelism might dwarf the expected improvement from a sequential algorithm enhancement alone. However, it should be noticed that the example function is *very* complex, and that direct evaluation, even at a parallelism of width 10, might not be expected to do better than the reduction gained through the 4 multiply-add operations as required by the CR technique. In examining case **C.DP** then, the data-parallel increment-based CR method, we begin to see the more subtle trade-offs involved in using CRs on parallel machines. Note that for smaller values of n, the performance of case **C.DP**, is about the same as for case **D.P**, but slightly worse than for case **C.F**. However, note that as n increases past 1000, the disadvantage compared to case **C.F** disappears, and the growth in execution time remains very low as compared to either case **C.F** or **D.P**. The reason here again is related to grain size of the computation compared with communication. Obviously, the data-parallel CR technique is able to be mapped on to very fine-grained machines, but for a system such as the Intel Paragon, the grain size required due to the large ratio of communication to computation speed of the machine, is quite large. Thus, for smaller values of n, the technique is not preferred over techniques **C.F** or **D.P**, but for large values, the amount of computation as compared to communication for case **C.DP** allows it to accommodate larger and larger problems, while only suffering a very slow rate of growth in execution time. Finally, in case **C.FD**, the hybrid of Data and Functional Parallel solutions, it can be seen that for smaller values of n, the technique compares favorably with all of the parallel cases in an absolute sense, but when considering that 40 processors are being used, the efficiency is not as attractive. In fact, as n continues to grow, the technique begins to suffer badly, as was that case with the purely functional parallel case (**C.F**). However, such a method might be considered useful for cases of small to moderate values of n, in which there are an ample supply of processors available.

In considering the execution timings for the Sequent as shown in Table 2, many of the observations are similar to those for the Paragon, but with a few differences. Case **D.P**, the parallel direct method, shows similar behavior to the Paragon, with somewhat worse performance for smaller values of n, but better values and slower growth than the **D.S** case. Similarly, for case **C.DP**, note the relatively slow rate of growth of execution times compared to both cases **C.F** and **D.P**, especially as n becomes larger [2].

Summary

Evaluation of functions using CRs can be more efficient than direct evaluation of functions. The error associated with the CR method can be significantly reduced when the function is evaluated in parallel using CRs. The two data parallel mapping techniques discussed each result in dramatically reduced errors although having different error distributions. This is an excellent example of a situation where parallel evaluation techniques result in significant improvements in solution quality, in addition to reduced execution times.

References

[1] Huang A.J. and Z. George Mou. Parallel partition expansion for the solution of arbitrary recurrences. In *Proc. ICPP '92*. CRC Press, 1992.

[2] Char B.W., Geddes K.O, et al. *Maple-V Language Reference Manual*. Springer Verlag, 1991.

[3] Zima E.V. Automatic construction of systems of recurrence relations. *Journal of Computational Mathematics and Mathematical Physics*, 24(6):193–197, 1984.

[4] Zima E.V. Recurrent relations technique to vectorize function evaluation in loops. In *Proc. PARCELLA '94*, pages 161 – 168, Potsdam,Germany, 1994. Akademie Verlag.

[5] Bachmann O., Wang P.S., Zima E.V. Chains of Recurrences – a method to expedite the evaluation of closed-form functions. Proc. ISSAC'94, Oxford, UK, July 1994, ACM Press, pp. 242-249.

[6] Feilmeier M. *Systems for Parallel Processing (Russian translation)*. Mir, Moscow, 1985.

[7] Wolfe M. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, 1996.

[8] Zima E.V. Simplification and Optimization Transformations of Chains of Recurrences. in the Proceedings of ISSAC'95, Montreal, Canada, ACM Press, 42–50. .

[9] Hockney R. and Jesshope C. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger, Philadelphia, 1988.

[10] Karp R.M., Miller R.E, and Winograd S. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563 – 590, 1967.

[11] Casavant T., Vadivelu K., Zima E. *Mapping Techniques for Parallel Evaluation of Chains of Recurrences*, Tech. report, ECE Dept, Univ. of Iowa.

[2] Observe, that in both cases **D.P** and **C.DP** include initialization overheads of about 0.4 seconds