ICOT Technical Report: TR-470

TR-470

Parallel Computation of Semigroups

by
E. Tick & N. Ichiyoshi

March, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Parallel Computation of Semigroups

E. Tick--University of Tokyo*

N. Ichiyoshi--ICOT†

Abstract

The Semigroup Problem is the calculation of the members of a semigroup given an initial set of generator members. The problem was solved in OR-parallel Prolog by Disz et al. and in FGHC by Ichiyoshi, yet these solutions have poor performance for reasons detailed in this paper. Optimizations of the programs are discussed and multiprocessor performance measurements are given. These optimizations: granularity collecting (in Prolog) and removal of synchronization points (in FGHC), are in general useful techniques for speeding up any parallel logic program.

1 Introduction

Several programming techniques have been proposed for parallel logic programming languages, but few have been analyzed on real multiprocessors. When such empirical tests are conducted, it is often the case that the multiplicative constant in the complexity order is significant in distinguishing *good* programs from bad programs. It is also the case that inefficiencies in a program give the deception of lots of easily exploitable parallelism. This paper analyzes two important programming techniques that can be put to use in practice. We present performance measurements from parallel logic programming languages (OR-Parallel Prolog and FGHC) running on shared memory multiprocessors (Sequent Symmetry) to justify our claims. The

*Research Center for Advanced Science and Technology, 4-6-1 Komaba, Minato-ku, Tokyo 153

†Mita Kokusai Bldg 21F, 1-4-28 Mita, Minato-ku, Tokyo 108

techniques discussed here have been used in programming several problems; however, here we use a single unifying example of the Semigroup Problem[2], a problem with a single solution.

The two techniques stressed here are granularity collection in OR-parallel systems and removal of synchronization points in AND-parallel systems. Although parallel Prolog is meant to hide parallel execution from the user, granularity collection helps exploit parallelism in programs where parallelism is hard to get at. Granularity collection is the rearrangement of code to allow the efficient placement of a branch-point, e.g., via a parallel `member/2` call. Without this rearrangement, the overheads of spawning OR processes from multiple branch points can outweight the benefit. Removal of synchronization points is the recoding of a problem to allow streams to flow freely without causing suspensions. This increases the throughput of the processes comprising the program, thus increasing the speed and speedup.

In addition to these techniques, the Semigroup Problem also raises the issue of appropriate data structures. We introduce a binary tree filter in FGHC that supports parallel accesses. This data structure is analogous in form and performance to the 2–3 tree used in the Prolog algorithms presented.

As with any engineering discipline, the techniques and performance measurements are subject to countless caveats. Firstly, the measurements presented here were taken on a Sequent Symmetry shared memory multiprocessor [5] running up to eight PEs (out of 12). Secondly, the OR-parallel system, Aurora [3], and the committed-choice system, KL1PS [4], are both in the continual process of refinement.

2 Semigroup in Prolog

The Semigroup Problem [2] is the calculation of the members of a semigroup from an initial set of generators. The specific problem solved in [2] uses four generators, each a 25-tuple of elements from the 5-element Bradt semigroup B_2. They produce a semigroup of 71 elements. A larger problem is discussed here, using four 40-tuple generators, producing a semigroup of 313 elements. Disz et al. [2] cite a speedup of 6.2 on eight PEs (on Encore Multimax) for their program. It may hardly seem like an improvement to report here a speedup of 3.6 for a new version of the same problem, on even bigger data! However, let us start at the beginning...

The program in [2] does more computation than is necessary to compute the semigroup: essentially each new semigroup member is multiplied by *all* other members from both sides. Ichiyoshi discovered that each new semigroup member need only be multiplied by the initial generators (the *kernel*) from *one* side. Thus the number of semigroup multiplications decreased from $N(N-1)$ to gN, where g is the number of generators and N is the size of the generated semigroup. For the example measured here, $N = 313$, so that the numbers of multiplications in the original and improved algorithms differ by a factor of 78. Experimentally, on SICStus Prolog V0.5 running on a SONY NEWS workstation, the original code ran in 1727.1 sec. compared to the new algorithm which ran in 24.3 sec. a factor of 71. Concentrating on the latter algorithm, cleaning up the code a bit resulted in 20.7 sec. At this point, adding a hash key to improve the performance of the 2–3 tree lookup reduced the time to 18.1 sec. These changes in sequential execution efficiency affect how efficiently parallelism can be exploited.

Semigroup has no non-determinate OR-parallelism, as in an all-solutions search problem. Here, OR parallelism is exploited when finding the cross-product of two sets of tuples. In Overbeek's program, one set contains the new additions to the semigroup, and the other set contains the current semigroup. In the new algorithm, the second set contains only the generator tuples (in the specific case, four). Given this reduction in available parallelism, the original program did not optimally group the parallelism so that it could be efficiently exploited.

2.1 Original Code

The original code[2] is given in Figure 1. The state of computation is `state(Sos,Sub,Hbg)`, where `Sos` is a list of candidate tuples (ready to be included in the semigroup, but as yet they have not been used to generate further candidates), `Sub` is the 2–3 tree containing the current (partial) semigroup tuples (including the candidates), and `Hbg`, the list of current semigroup tuples (not yet including the candidates). If the candidate list is empty, then the current list of semigroup tuples is the complete semigroup. Otherwise, the state must be reduced into a new state by calculating all tuples generated by the cross-product of the candidates and the kernel.

The 2–3 tree is not strictly necessary —the list `Hbg` could be used exclusively. However, the 2–3 tree decreases the lookup time when checking for duplicate tuples. In fact, we developed a

```
:- parallel member/2, paired/4.

go(Hbg) :-
    kernel(Sos),
    extend_tree(Sos,nil,Sub),
    gen_all(state(Sos,Sub,[]),state(_,_,Hbg),Sos).

gen_all(state([],Sub,Hbg),state([],Sub,Hbg),_).
gen_all(S, F, Kernel) :-
    S = state([_|_],_,_),
    gen_one(S, S1, Kernel),
    gen_all(S1, F, Kernel).

gen_one(state([H|T], Sub, Hbg), state(Sos1, Sub1, [H|Hbg]), Kernel) :-
    findall(Tuple, newtup(H, [H|Hbg], Sub, Tuple), L),
    proc_new(L, Sub, Sub1, T, Sos1).

newtup(E,L,Sub,New) :-
    member(E2,L),
    paired(E,E2,New,Sub).

paired(E1,E2,New,Sub) :-
    mult(E1,E2,New),
    \+ acc23(New,Sub).
paired(E1,E2,New,Sub) :-
    mult(E2,E1,New),
    \+ acc23(New,Sub).

proc_new([],Sub,Sub,Sos,Sos).
proc_new([H|T],Sub,Sub1,Sos,Sos1) :-
    proc_new(T,Sub,Sub2,Sos,Sos2),
    (add23(Sub2,H,Sub1) ->
        Sos1 = [H|Sos2]
;
        (Sub1 = Sub2, Sos1 = Sos2)).
```

Figure 1: Original Prolog Program for Semigroup

"hash 2-3 tree" to speed things up even more, as is described in a later section.

The cross-product is calculated with a `findall/3`. Nondeterminate `member/2` and `paired/4` (declared `parallel`) exploit the independent OR-parallel computation of the cross product of `H` with `[H|Hbg]`. `acc23(X,T)` checks for tuple `X` in 2-3 tree `T`, and fails if the tuple exists. `add23(T0,X,T1)` inserts tuple `X` into tree `T0`, giving a new tree `T1` or fails if the tuple exists. Refer to Bratko [1] for this code. The check with `acc23/2` filters away products that already reside in the partially constructed semigroup. However, even if the product is not in the tree at this stage, it does not allow us to insert it. It may be the case that two or more identical products are computed concurrently. Thus we still need a filtering phase. `proc_new/5` filters all the new products for duplicates with `add23/3`. The previous filtering with `acc23/2` is not strictly necessary, given the filter here; however, `add23/3` is significantly more expensive. All unique tuples are added to the tree at this stage.

2.2 Utilities

There are a few as yet undefined utilities needed for the Semigroup program: `member/2`, `kernel/1`, and `mult/3`. `member/2` is a nondeterminate member, which can be unrolled to increase the available parallelism at a single branch point within Aurora [2]. The measurements presented throughout this paper have `member` unrolled four times.

As for `mult/3`, Figure 2, there are basically two choices with which to implement the tuples: structures and lists. Within each, a hash key can be calculated from the the tuple elements, and placed as a first, special, element. This hash key element will automatically help the 2-3 tree code do fast lookups (note that original program in Disz [2] does not incorporate a hash key). The hash function we devised gives only five aliases for 313 tuples in the example analyzed. The better choice among structure and list is dependent on the Prolog system, but both are close. We show the list version here. Each list is of length 41, where the first element is the hash key and subsequent elements are the tuple arguments. The FGHC code for `mult/3` is similar.

```
kernel(K) :-
   K=[1833472791,1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,4,4,4,4,4,
                 5,5,5,5,5,3,3,3,3,3,5,5,5,5,5,4,4,4,4,4],
      -500019130,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,
                 1,2,3,4,5,1,2,3,4,5,1,3,2,4,5,1,2,3,4,5],
      1084198104,1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,5,5,5,5,5,
                 4,4,4,4,4,2,2,2,2,2,4,4,4,4,4,3,3,3,3,3],
      1154844798,1,2,3,5,4,1,2,3,5,4,1,2,3,5,4,1,2,3,5,4,
                 1,2,3,5,4,1,2,3,4,5,1,2,3,5,4,1,2,3,5,4]].

mult([_|X],[_|Y],[Key|Zs]) :-
    mult(X,Y,Zs,0,0,Key).

mult([], [], [], Key, I, A) :-  A is I+Key*3.
mult([X|Xs], [Y|Ys], [Z|Zs], Key, I, A) :-
    NewKey is I+Key*3,
    m(X, Y, Z),
    mult(Xs, Ys, Zs, NewKey, Z, A).

m(2,1,1):-!. m(3,1,1):-!. m(4,1,1):-!. m(5,1,1):-!. m(1,1,1):-!.
m(2,2,2):-!. m(3,2,1):-!. m(4,2,1):-!. m(5,2,5):-!. m(1,2,1):-!.
m(2,3,1):-!. m(3,3,3):-!. m(4,3,4):-!. m(5,3,1):-!. m(1,3,1):-!.
m(2,4,4):-!. m(3,4,1):-!. m(4,4,1):-!. m(5,4,3):-!. m(1,4,1):-!.
m(2,5,1):-!. m(3,5,5):-!. m(4,5,2):-!. m(5,5,1):-!. m(1,5,1):-!.
```

Figure 2: Kernel Tuples and Hash–Multiplication Procedure

2.3 Optimized Algorithm

Ichiyoshi's optimization is the fact that only the kernel generators need be multiplied, and only in one direction. This modification is shown in Figure 3, where `paired/4` has been fused with `newtup/4`. Now considering parallelism and speedup, we must work from this efficient algorithm. Unfortunately, in reducing the complexity of the algorithm, we removed exploitable parallelism from the program. The modified program *slows down* on multiple PEs on Symmetry! The major bottleneck is that the cross-product of new semigroup tuples and

```
gen_one(state([H|T], Sub, Hbg), state(Sos1, Sub1, [H|Hbg]), Kernel) :-
    findall(Tuple, newtup(H, Kernel, Sub, Tuple), L),
    proc_new(L, Sub, Sub1, T, Sos1).

newtup(E,L,Sub,New) :-
    member(E2,L),
    mult(E,E2,New).
    \+ acc23(New,Sub).
```

Figure 3: Reduced Order Semigroup Algorithm (Prolog)

```
go(Hbg) :-
    kernel(Sos),
    extend_tree(Sos,nil,Sub),
    loop(Sos, Sub, Sos, Hbg, Sos).

loop([], _, Hbg, Hbg, _) :-!.
loop(Sos, Sub, Hbg, F, Kernel) :-
    findall(Tuple, newtup(Sos, Kernel, Sub, Tuple), L),
    filter(L, Sub, NewSub, [], NewSos, Hbg, NewHbg),
    loop(NewSos, NewSub, NewHbg, F, Kernel).

newtup(L,K,Sub,New) :-
    member(E1,L),      % new candidates
    member(E2,K),      % kernel
    mult(E2,E1,New),
    \+ acc23(Sub,New).

filter([], Sub, Sub, Sos, Sos, Hbg, Hbg).
filter([H|T], Sub, SubF, Sos, SosF, Hbg, HbgF) :-
    (add23(Sub,H,Sub1) ->
        Sos1=[H|Sos],
        Hbg1=[H|Hbg]
    ;   Sub1=Sub,
        Sos1=Sos,
        Hbg1=Hbg),
    filter(T, Sub1, SubF, Sos1, SosF, Hbg1, HbgF).
```

Figure 4: Big Granule OR-Parallel Semigroup Program

the kernel is proceeding by multiplying one new tuple at a time against the kernel. Previously this same code attained speedups, on even smaller data because **member** selected from the entire semigroup, creating an efficient branch-point. To solve this problem, we must *collect the granularity* of the computation into a focal point where a branch-point (with a high branching factor) can be built.

2.4 Granularity Collecting

In the case of the Semigroup program, we wish to collect the candidate list into a single granule that another **member** can select from. The reworked code (Figure 4) alleviates much of the previous bottleneck by doing the entire cross product within **newtup/4** with the use of two **members**. We have also done a bit of procedure fusing, renaming, and removal of superfluous structures. This code achieves a speedup of 4.1 on eight Symmetry PEs on the 40-element tuple problem. Speedup is still constrained by the small kernel size of only four elements. Recall that Semigroup is *not* an all-solutions search program where each OR-parallel branch is usually a

search subtree. Instead, the `member` branch-points create only OR-parallel stubs for computing products.

In general we conclude that as one streamlines the sequential structure of an algorithm, the corresponding parallelism can become harder to exploit. This may require restructuring of the program to collect the granularity in a more centralized spot. Thus in "automatically" parallelized languages such as Prolog, the parallelization is not as automatic as one might hope.

3 Semigroup in FGHC

The FGHC version of the Semigroup Problem published in [6] also has its problems. For this program running on Sequent Symmetry on the KL1PS system [4], speedups of (1.7,3.0,4.5) on (2,4,8) PEs were measured. Here there are two problems: the program gets poor speedup and the program is slow, 6.3–5.1 times slower (for one and eight PEs respectively) than the optimized Prolog program previously discussed. Yet both programs do the same amount of multiplications, i.e., each new semigroup tuple is multiplied only by the kernel tuples.

3.1 Original Code

The basic idea of the original program [6], Figure 5, is to create a pipeline of filters, one corresponding to each new tuple in the semigroup. Initially this pipeline has four tuples corresponding to the generators. Each tuple begins as a `g/5` process — during this phase, it multiplies itself by tuples it receives as messages. This message stream holds the new semigroup tuples, delimited by two bookends: `begin` and `end`. When the new group ends, the `g/5` process has done its job and changes into an `f/3` filter process. Now it remains forever, simply checking oncoming product tuples for a match (in which the oncoming tuple is discarded).

The program complexity and poor speedup comes from the method used to collect products. Each tuple is represented by its value in addition to a difference list holding the products formed by multiplying it by the pipeline `g/5` processes. All the new products are strung together by `connect/2`. This serializes the computation because it does not allow the new products to be filtered as fast as they are created.

```
go(Out) :-
    kernel(Gens),
    gen_g(Gens, Gin, Fin, Gout, Fout),
    gen_gen(Gens, Gin, NGin),
    connect(Gout, Fin),
    ends(Fout, _, _, NGin, Gens, Out-[]).

g([gen(X,P,P0)|Gin1], Fin, Gout, Fout, E) :-
    mult(E, X, EX),
    P0 = [EX|P1],
    Gout = [gen(X,P,P1)|Gout1],
    g(Gin1, Fin, Gout1, Fout, E).
g([begin|Gin1], Fin, Gout, Fout, E) :-
    Gout = [begin|Gout1],
    g(Gin1, Fin, Gout1, Fout, E).
g([end|Gin1], Fin, Gout, Fout, E) :-
    Gout = [end|Gin1],
    f(Fin, Fout, E).

f([], Fout, E) :- Fout = [].
f([E|Fin1], Fout, E) :- f(Fin1,Fout, E).
f([X|Fin1], Fout, E) :- otherwise | Fout = [X|Fout1], f(Fin1, Fout1, E).

gen_g([X|Xs], G0,F0,G,F) :-
    g(G0,F0,G1,F1, X),
    gen_g(Xs, G1,F1,G,F).
gen_g([], G0,F0,G,F) :- G0=G, F0=F.

ends([begin,end|_], _, Gin, OGin, _, O1-O2) :- Gin=[], OGin=[], O1=O2.
ends([begin,X|Fout2], _, _, OGin, Gens, Out) :- X \= end |
    gen_gen(Gens,NGin0,NGin),
    ends([X|Fout2],NGin0,NGin,OGin, Gens, Out).
ends([end|Fout1], Gout, Gin, OGin, Gens, Out) :-
    connect(Gout, OGin),
    ends(Fout1, _, _, Gin, Gens, Out).
ends([X|Fout1], Gout, Gin, OGin, Gens, O1-O3) :- otherwise |
    O1 = [X|O2],
    g(Gout, Fout1, NewGout, NewFout, X),
    ends(NewFout, NewGout, Gin, OGin, Gens, O2-O3).

gen_gen(Gens, G0,G) :-
    G0 = [begin|G1],
    gen_gen1(Gens, G1,G).

gen_gen1([X|Xs], G0,G) :-
    G0 = [gen(X,P,P)|G1],
    gen_gen1(Xs, G1,G).
gen_gen1([], G0,G) :-
    G0 = [end|G].

connect([gen(_,P0,P)|G1], F) :- F=P0, connect(G1,P).
connect([begin|G1], F) :- F=[begin|F1], connect(G1,F1).
connect([end|G1], F) :- F=[end|G1].
```

Figure 5: Original FGHC Semigroup Program: Pipeline of Filters

```
go(Out) :-
    kernel(K),
    append([begin|K],[end|K],S),
    spawn(S,K,Out,[]).

spawn([begin,end|_],S0,T0,T1) :- T0=T1, S0=[].
spawn([begin,X|Xs],S0,T0,T1) :- X \= end |
    S0 = [begin|S1],
    spawn([X|Xs],S1,T0,T1).
spawn([end|Xs],S0,T0,T1) :-
    S0 = [end|S1],
    spawn(Xs,S1,T0,T1).
spawn([X|Xs0],S0,T0,T2) :- otherwise |
    kernel(K),
    T0 = [X|T1],
    S0 = [_,_,_,_|S1],
    g(K,X,S0),
    f(Xs0,Xs1,X),
    spawn(Xs1,S1,T1,T2).

g([],_,_).
g([K|Ks],E,[P|Ps]) :-
    g(Ks,E,Ps),
    mult(K,E,P).
```

Figure 6: High-Throughput AND-Parallel FGHC Semigroup Program

3.2 Removing Synchronization Points

The bottlenecks in the original program are corrected in the program shown in Figure 6. `gen_gen/3` is no longer needed – we pass the raw tuples directly down the stream. By issuing the new products directly into the tail of **`spawn/4`**'s input stream, we obviate the need for `connect/2`. Note also that the `g/3` and `f/3` processes are spawned together, avoiding unnecessary synchronization. The strange binding of `S0` is to skip four places in the candidate stream for the four kernel products. This removes any synchronization with the recursive call to `spawn`.

3.3 Single Bookend Termination

We can simplify the optimized program noticing that only one bookend is really necessary. Instead of checking for **`begin`** followed immediately by **`end`**, we use only **`end`** and a flag to indicate if two **`end`**s fall back-to-back. This modification (Figure 7) does not give us any speedup, but is prettier.

In any case, the program still suffers from the handicap of a long pipeline of filters. This

```
go(Out) :-
    kernel(K),
    append(K,[end|R],S),
    spawn(S,R,Out,[],_).

spawn([end|Xs],S0,T0,T1,sawelement) :-
    S0=[end|S1],
    spawn(Xs,S1,T0,T1,sawend).
spawn([end|Xs],S0,T0,T1,sawend) :-  T0=T1, S0=[].
otherwise.
spawn([X|Xs0],S0,T0,T2,_) :-
    kernel(K),
    T0 = [X|T1],
    S0 = [_,_,_,_|S1],
    g(K,X,S0),
    f(Xs0,Xs1,X),
    spawn(Xs1,S1,T1,T2,sawelement).
```

Figure 7: Single Bookend FGHC Semigroup Program

pipeline requires a linear number of checks, whereas the 2-3 tree in Prolog requires only a log number of checks. This makes the speedup of the pipelined filter better than that of the 2-3 tree, but the 2-3 tree is much faster. This observation is confirmed by the timings, where the Prolog program's advantage over the FGHC program on one PE is a factor of 6.3 in speed, but this decreases to only 5.1 on eight PEs. Ideally we would like a tree filter in the FGHC program also, as is described in the next section.

3.4 Binary Hash Tree Filter

The tree filter introduced below is an unbalanced binary tree containing the tuples. Unlike the 2-3 tree in Prolog [1], parallelism can be exploited in the unbalanced tree filter. A candidate tuple can enter the tree filter *before* the insertion of the previous tuple has been completed. Measurements presented show this simple data structure gives over a factor of four improvement over the previous pipeline filter. We first modify the definition of a tuple slightly. Here a tuple is `K-T` where `K` is the hash key and `T` is a list of elements.

The unbalanced binary hash tree code is shown in Figure 8. A node, containing a bucket `B` of tuples and its hash key `X`, is comprised of one of four types of processes: leaf (no children), right-child only, left-child only, and with both right and left children. A node process responds to four types of messages: `[]` – kill yourself, `m(Y,C,S)` check/insert tuple `C` with hash key `Y`,

```
% leaf node (node with no child nodes)
t([],_,_).
t([m(Y,C,S) |T],X,B) :- X =:= Y | insertBucket(B,B,C,B1,C,S), t(T,X,B1).
t([m(Y,C,S) |T],X,B) :-   X < Y | gen(C,S), tr(T,X,B,R), t(R,Y,[C]).
t([m(Y,C,S) |T],X,B) :-   Y < X | gen(C,S), t(L,Y,[C]), tl(T,X,B,L).
t([out(S0,S)|T],X,B) :- send(B,S0,S), t(T,X,B).
t([end(S0,S)|T],X,B) :- S0=[end|S], t(T,X,B).

% node with right child only
tr([],_,_,R) :-  R = [].
tr([m(Y,C,S) |T],X,B,R) :- X =:= Y | insertBucket(B,B,C,B1,C,S), tr(T,X,B1,R).
tr([m(Y,C,S) |T],X,B,R) :-   X < Y | tr(T,X,B,R1), R = [m(Y,C,S)|R1].
tr([m(Y,C,S) |T],X,B,R) :-   Y < X | gen(C,S), t(L,Y,[C]), tlr(T,X,B,L,R).
tr([out(S0,S)|T],X,B,R) :- send(B,S0,S1),  tr(T,X,B,R1), R = [out(S1,S)|R1].
tr([end(S0,S)|T],X,B,R) :- S0=[end|S], tr(T,X,B,R).

% node with left child only
tl([],_,_,L) :-  L = [].
tl([m(Y,C,S) |T],X,B,L) :- X =:= Y | insertBucket(B,B,C,B1,C,S), tl(T,X,B1,L).
tl([m(Y,C,S) |T],X,B,L) :-   X < Y | gen(C,S), tlr(T,X,B,L,R), t(R,Y,[C]).
tl([m(Y,C,S) |T],X,B,L) :-   Y < X | L=[m(Y,C,S)|L1], tl(T,X,B,L1).
tl([out(S0,S)|T],X,B,L) :- L=[out(S0,S1)|L1], tl(T,X,B,L1), send(B,S1,S).
tl([end(S0,S)|T],X,B,L) :- S0=[end|S], tl(T,X,B,L).

% node with both left and right children
tlr([],_,_,L,R) :-  L = [], R = [].
tlr([m(Y,C,S) |T],X,B,L,R) :- X =:= Y | insertBucket(B,B,C,B1,C,S),tlr(T,X,B1,L,R).
tlr([m(Y,C,S) |T],X,B,L,R) :-   X < Y | tlr(T,X,B,L,R1), R = [m(Y,C,S)|R1].
tlr([m(Y,C,S) |T],X,B,L,R) :-   Y < X | L = [m(Y,C,S)|L1], tlr(T,X,B,L1,R).
tlr([out(S0,S)|T],X,B,L,R) :- L=[out(S0,S1)|L1], tlr(T,X,B,L1,R1),
                              send(B,S1,S2), R=[out(S2,S)|R1].
tlr([end(S0,S)|T],X,B,L,R) :- S0=[end|S], tlr(T,X,B,L,R).

insertBucket([],B,C,B1,E,S) :- B1=[C|B], gen(E,S).      % new element
insertBucket([C|_],B,C,B1,_,S0-S1) :- B1=B, S0=S1.      % already in bucket
insertBucket([_|Cs],B,C,B1,E,S) :- otherwise | insertBucket(Cs,B,C,B1,E,S).

send([],S0,S1) :- S0=S1.
send([T|Ts],S0,S2) :- S0=[T|S1], send(Ts,S1,S2).
```

Figure 8: Unbalanced Binary Hash Tree Filter

```
go(Out) :-
    kernel(K),
    append(K,[end|R],T),
    t(T1,0,[]),
    spawn(T,T1,sawelement,R,Out).

spawn([end|T],T1,sawelement,R0,Out) :-
    T1=[end(R0,R1)|Ts],
    spawn(T,Ts,sawend,R1,Out).
spawn([end|_],T1,sawend,_,Out) :- T1=[out(Out,[])].
spawn([X-Y|T],T1,_,R0,Out) :-
    T1=[m(X,Y,R0-R1)|Ts],
    spawn(T,Ts,sawelement,R1,Out).

gen(X,S0-S1) :- kernel(K), g(K,X,S0,S1).

g([],_,S0,S1) :- S0=S1.
g([K|Ks],E,S,T) :-
    S=[P|Ps],
    g(Ks,E,Ps,T),
    mult(K,E,P).

mult(_-X,Y,Out) :- Out=Key-R, mult(X,Y,R,0,Key,0).
```

Figure 9: Binary Tree Filter FGHC Semigroup Program

`out(S0,S)`--output your value, `end(S0,S)`--pass bookend through.

To check/insert the new tuple, the hash key is used to route the message to the appropriate node in the tree. Note that general unification of the entire tuple is avoided until the final bucket check in `insertBucket`. If that tuple already exists, the stream `S` is shorted. If that tuple is new, it is inserted into the tree, and `gen(C,S)` is used to generate the cross-product with the kernel. These product tuples are sent down the `S` stream.

Given the tree code, the new semigroup program (Figure 9) is even simpler than the previous pipeline program. Again we use a single bookend termination method. In this case, `spawn` no longer spawns anything, but simply checks for termination. Spawning generators and filters is replaced by sending the tuples as `m/3` messages into the tree. The crux is the stream argument of the check/insert message, `R0-R1` in the last clause of `spawn`. Similar to the original program, we issue the new products, via `R0-R1`, into the tail of `spawn`'s own input stream.

program	Fig.	1 PE	2 PE	4 PE	8 PE
Prolog					
original	1	**out of heap space**			
+ Ichiyoshi's opt.	3	**slowdown**			
+ granule collect.	4	38.8	23.4	14.8	10.7
		1.00	1.70	2.62	3.63
FGHC					
original	5	244.2	148.3	81.7	54.2
		1.00	1.65	2.99	4.51
+ hash keys	2	171.2	92.7	59.0	35.6
		1.00	1.85	2.90	6.25
+ synch. removal	7	163.6	87.1	45.9	27.4
		1.00	1.88	3.56	5.97
+ binary tree	9	37.5	19.1	10.1	5.8
		1.00	1.96	3.71	6.47

Table 1: Semigroup Execution on Symmetry: Seconds/Speedup

4 Discussion

Parallel solutions to the Semigroup Problem posed by Disz et al. [2] are discussed in this paper. It is shown how the original solutions given by [2, 6] are inefficient in algorithm and in parallelism. Improvements to the programs are given here, as a sequence of stepwise refinements, with measurements to indicate the utility of the optimizations. In these changes, we believe the clarity of the programs actually increased.

A summary of the Symmetry timings is given in Table 1. All programs calculate the 313 member semigroup given by the generators in Figure 2. All the programs use lists to represent tuples. For Prolog, we have given only the final program (Figure 4) timings because the original program cannot run (for the large problem size) on Aurora to memory limitations and the intermediate program (without granularity collection) has slowdown. For FGHC however, the optimizations afford a continuous range of performance improvement. Hash keys give a

speedup of 50%. Removal of synchronization points gives a speedup of 30% on eight PEs. Most significantly, the tree filter gives a speedup of 4.7.

The programming techniques of granularity collection and synchronization point removal are general tools that prove useful in an imperfect world where linear speedups cannot be produced automatically by smart compilers and schedulers. Refer to Tick[7] for further examples of parallel logic programming techniques and their performance analyses.

A general conclusion of this paper is that parallel logic programming has not yet achieved the goal of declarativity wherein parallelism can automatically be uncovered from an efficient sequential algorithm (in a language such as Prolog). In addition, although dependent stream-AND parallelism is very powerful because small-grain parallelism is exploited, care must be taken to avoid synchronizations and bottlenecks. We urge those who are designing parallel architectures to take an honest look at the benchmarks they are using or the programming paradigms they are professing. We have seen many cases of published analyses of parallel programs with grossly inefficient algorithms, inappropriate data structures, scheduler-dependent termination, and other problems.

References

[1] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley Ltd., Wokingham, England, 1986.

[2] T. Disz, E. Lusk, and R. Overbeek. Experiments with OR-Parallel Logic Programs. In *Fourth International Conference on Logic Programming*, pages 576–600. University of Melbourne, MIT Press, May 1987.

[3] E. Lusk et. al. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[4] M. Sato and et al. KL1 Execution Model for PIM Cluster with Shared Memory. In *Fourth International Conference on Logic Programming*, pages 338–355. University of Melbourne, MIT Press, May 1987.

[5] Sequent Computer Systems, Inc. *Sequent Guide to Parallel Programming*, 1987.

[6] S. Takagi. A Collection of KL1 Programs-Part I. Technical Memo TM-311, ICOT, May 1987.

[7] E. Tick. Exeriences in Programming Parallel Logic. Technical report, University of Tokyo, 1989.