Laboratory for Concurrent Computing Systems
Technical Report 31-034

Version 1.0 October 1991

Exploiting Low Level Parallelism in the Dynamic Control of Manipulators

S. Zeng
Swinburne University of Technology
Australia

Professor G. K. Egan
Swinburne University of Technology
Australia

Abstract:

The manipulator dynamic control involves computing the desired forces or motor torques to allow a manipulator to follow the required trajectory. These calculations are normally based on the manipulator dynamic equations and the feedback information about the actual motion of manipulator. To achieve convergence of the control algorithm may require sampling rates greater than 60 Hz, with the upper limit being determined by the mechanical resonance. Given this and the nonlinear characteristics of manipulator the computation load on the controller is substantial, and has in the past required an expensive mincomputer or even a super-minicomputer.

An alternative approach is to decompose the computation load of manipulator dynamic control into a number of tasks which will be performed by inexpensive multiprocessors concurrently.

In this paper, a number of parallel processing schemes are described for the manipulator control computation and implemented on multiprocessor systems. The results show that a moderate speedup have been achieved on a shared memory system and a good speedup on a dataflow system.

LABORATORY FOR CONCURRENT COMPUTING SYSTEMS
COMPUTER SYSTEMS ENGINEERING
School of Electrical Engineering
Swinburne University of Technology
John Street, Hawthorn 3122, Victoria, Australia.

EXPLOITING LOW LEVEL PARALLELISM IN THE DYNAMIC CONTROL OF MANIPULATORS

S.Zeng and G.K.Egan

Laboratory for Concurrent Computing Systems
Swinburne Institute of Technology, John Street, Hawthorn 3122, Australia

Abstract: The manipulator dynamic control involves computing the desired forces or motor torques to allow a manipulator to follow the required trajectory. These calculations are normally based on the manipulator dynamic equations and the feedback information about the actual motion of manipulator. To achieve convergence of the control algorithm may require sampling rates greater than 60 Hz, with the upper limit being determined by the mechanical resonance. Given this and the nonlinear characteristics of manipulator the computation load on the controller is substantial, and has in the past required an expensive minicomputer or even a super-minicomputer.

An alternative approach is to decompose the computation load of manipulator dynamic control into a number of tasks which will be performed by inexpensive multiprocessors concurrently.

In this paper, a number of parallel processing schemes are described for the manipulator control computation and implemented on multiprocessor systems. The results show that a moderate speedup have been achieved on a shared memory system and a good speedup on a dataflow system.

I. Introduction

There is an increasing need to develop manipulator dynamic control schemes[1] so that next generation manipulators, with redundant degrees of freedom, can move along a desired trajectory, possibly with unprogrammed or unknown loads, in the presence of obstacles including other manipulators. Additionally they must be operated at highest possible speeds to maximise utilisation of plant.

The above manipulator dynamic control requires real-time calculation of the desired forces or motor torques to allow manipulator to follow the required trajectory. These calculations are normally based on the manipulator dynamic equations and the feedback information about the actual motion of manipulator. To achieve convergence of the control algorithm may require sampling rates greater than 60 Hz, with the upper limit being determined by the mechanical resonance. Given this and the high degree of nonlinear characteristics of manipulator the computation load on the controller is substantial, and has in the past required an expensive minicomputer or even a super-minicomputer.

An alternative approach is to decompose the computation load of manipulator dynamic control into a number of tasks which will be performed by inexpensive multiprocessors concurrently [2][3].

Success of such an approach requires an optimal algorithm to assign tasks to respective processors and an efficient configuration of multiprocessor system to support concurrent computation.

This paper presents a number of implementations of the manipulator control computation on multiprocessor computer systems. Our research lies in exploring implicit parallel programming schemes for real-time control and as such is a departure from the more usual explicit solution techniques in the literature.

II. The Recursive Newton-Euler Equations

There are a number of ways to formulate the dynamic equation of manipulator motion, two main approaches are used by most of researchers to systematically derive the dynamic model of manipulator--the Lagrange-Euler (LE) formulation and the Newton- Euler (NE) formulation [4].

The LE formulation generates a set of closed form differential equations to describe motion. The computation method of the LE equations involves many matrix multiplication and is computationally inefficient.

The NE formulation yields a computationally efficient set of forward and backward recursive equations of motion. This algorithm is the fastest and the most efficient of existing algorithm for dynamic control computation [4].

The NE equations are based on Newton's second law

$$\mathbf{F}_i = \mathbf{m}_i\mathbf{a}_i$$

and Euler's equation

$$\mathbf{N}_i = \mathbf{I}_i\dot{\mathbf{w}}_i + \mathbf{w}_i \times (\mathbf{I}_i\mathbf{w}_i)$$

Furthermore, recursive Newton-Euler equations are derived with nine equations for each link [1].

$$A_i^0 w_i = A_i^{i-1}(A_i^0 w_{i-1} + z_0\dot{q}_i) \qquad \text{if link i is rotational}$$

$$\text{or } A_i^{i-1}(A_i^0 w_{i-1}) \qquad \text{if link i is translational}$$

$$A_i^0 \dot{w}_i = A_i^{i-1}[A_i^0 \dot{w}_{i-1} + z_0\ddot{q}_i + (A_i^0 w_{i-1}) \times (z_0\dot{q}_i)] \qquad \text{if link i is rotational}$$

$$\text{or } A_i^{i-1}(A_i^0 \dot{w}_{i-1}) \qquad \text{if link i is translational}$$

$$A_i^0 \dot{v}_i = (A_i^0 \dot{w}_i) \times (A_i^0 p_i^*) + (A_i^0 w_i) \times [(A_i^0 w_i) \times (A_i^0 p_i^*)] + A_i^{i-1}(A_{i-1}^0 \dot{v}_{i-1}) \qquad \text{if link i is rotational}$$

$$\text{or } A_i^{i-1}(z_0\ddot{q}_i + A_{i-1}^0 \dot{v}_{i-1} + (A_i^0 \dot{w}_i) \times (A_i^0 p_i^*) + 2(A_i^0 w_i) \times (A_i^{i-1} z_0\dot{q}_i) + (A_i^0 w_i) \times [(A_i^0 w_i) \times (A_i^0 p_i^*)]$$

$$\text{if link i is translational}$$

$$A_i^0 \dot{\hat{v}}_i = (A_i^0 \dot{w}_i) \times (A_i^0 s_i) + (A_i^0 w_i) \times [(A_i^0 w_i) \times (A_i^0 s_i)] + A_i^0 \dot{v}_i$$

$$A_i^0 F_i = m_i A_i^0 \dot{\hat{v}}_i$$

$$A_i^0 N_i = (A_i^0 I_i A_0^i)(A_i^0 \dot{w}_i) + (A_i^0 w_i) \times [(A_i^0 I_i A_0^i)(A_i^0 w_i)]$$

$$A_i^0 f_i = A_i^{i+1}(A_{i+1}^0 f_{i+1}) + A_i^0 F_i$$

$$A_i^0 n_i = A_i^{i+1}(A_{i+1}^0 n_{i+1} + (A_{i+1}^0 p_i^*) \times (A_{i+1}^0 f_{i+1})] + (A_i^0 p_i^* + A_i^0 s_i) \times (A_i^0 F_i) + A_i^0 N_i$$

$$\tau_i = (A_i^0 n_i)'(A_i^{i-1} z_0) + b_i\dot{q}_i \qquad \text{if link i is rotational}$$

$$\text{or } (A_i^0 f_i)'(A_i^{i-1} z_0) + b_i\dot{q}_i \qquad \text{if link i is translational}$$

where: A_i^{i-1} and I_i are 3x3 matices, $\dot{q}_i$, $\ddot{q}_i$, τ_i, m_i and b_i are scalars, the rest are 3x1 vectors.

The manipulator configuration chosen is a popular ASEA IRb-6 robot arm which has five

In a task graph, there is a longest path called critical path whose path length is defined as follow:

$$t_{cr} = \max_{k} \sum_{i \in \phi_k} t_i$$

where ϕ_k represents the kth path from the entry node to the exit node.

The critical path length play a principal role since once the critical path length is determined, no matter what scheduling is used and how many processors will be employed, the execution time will be not shorter than this critical path length.

Using UNIX -gprof, we can estimate the execution time of each task, furthermore, the critical path length can be inferred:

the critical path: 0->2->12->22->32->42->44->45-> 47->37->39->29->19->9->10->52

the critical path length = 48.354 (sec) (40,000 iterations)

A task can not start until all of its predecessors are completed. Threshold variables are used to determine when a task is ready to be executed and that task is then scheduled. Two methods have been employed to schedule tasks. One is dynamic scheduling and the other is static scheduling.

__Dynamic scheduling__ For **dynamic scheduling**, all of the processors share a common task queue which is located in a shared memory and the tasks are scheduled at run-time. If we denote:

- T be a set of tasks, i.e. $T=(T_0, T_1, ..., T_i, ..., T_n, T_{n+1})$, where T_0 and T_{n+1} are dummy tasks, representing the enter node and the exit node in a task graph, respectively.
- P be a set of processors, i.e. $P=(P_1,..., Pj , ..., P_m)$;

then the dynamic scheduling can be described as the following steps:

1. Initially, T_0 is put into the queue, and a shared variable remaining_tasks is set to equal to NoOfTasks;
2. P_j checks the variable remaining_tasks: **IF** remaining_tasks=0, then go to step 4; **ELSE** go to step 3;
3. P_j reads a task T_i from the queue, **IF** T_i<>T_{n+1}, i.e.T_i is not the exit node, then P_j performs T_i. After completing T_i, P_j checks the thresholds of successors of T_i and put any task with threshold equal to its NoOfPred (No Of Predecessors) into the queue. go to step 2; **ELSE** P_j sets the variable remaining_tasks=0, and return to step 2;
4. stop.

The outline of dynamic scheduling is described below:

```
shared
            :
        queue: q_rec;
        Threshold_Count:array[sub_task] of semaphore; *
        TaskDesc:array[sub_task] of Task_Rec;
```

---

* Although Threshold_Counts are located in the shared memory, we used some software technique to protect only one processor can change the value of a Threshold_Count at each time. This is detailed in [17].

Procedure SchedSuccessors schedules the next task for processors. After completion of a

task, the threshold of successor tasks are increased by 1. Any successor task with threshold equal to its NoofPred is placed into the execution queue.

```
begin
   with TaskDesc[task] do
      for i:=1 to NoOfSucc do with S[i] do
       with TaskDesc[Task] do begin
    Threshold_Count[Task]:=succ(Threshold_Count[Task])
         if Threshold_Count[Task]:=NoOfPred then begin
            Threshold_Count[Task]:=0;        {reset threshold}
            with queue do begin
              spinlock(q_lock);
              tail:=(succ(tail) mod max_task);
              t[tail]:=Task;                        {enqueue}
              spinunlock(q_lock)
           end;{with queue}
         end;{if}
      end;{with TaskDesc[Task]}
end;{SchedSuccessors}
```

Procedure Dynamic_control computes nine equations of manipulator dynamic control for every link.

```
begin
  repeat
     with queue do begin
        spinlock(q_lock);
        if tail <>head then begin           {if queue is not empty,
          head:=(succ(head) mod max_task);        then dequeue}
        spinunlock(q_lock);
              case task of
 0: SchedSuccessors(task);
 1: begin
       {do task 1 - compute angular velocity for link 1}
      SchedSuccessors(task);
    end; {task 1}
                 :
 52: remaining_tasks:=0;
      end;{case task}
    end {if}
    else spinunlock(q_lock); end;{with}
  until remaining_task=0
end;{dynamic_control}
```

The advantage of dynamic scheduling is that it is simple to programme, and avoids manual work such as mapping tasks to specific processors. The drawback is that a processor spends quite a long time on scheduling and spinlock/spinunlock.

Static scheduling For static scheduling, tasks are allocated to specific processors. The order in which the tasks are executed on a given processor is predetermined. Tasks wait until the predecessors have finished before executing. In this case, the task itself monitors the threshold rather than being explicitly scheduled by a predecessor. In the program, we use DHLF/MISF (Dynamic High Level First / Most Immediate Successive First) [16] method to generate a task order for each processor and then directly map the tasks to specific processors. If a task and its predecessor are allocated to the same processor, then the No of predecessors of this task can be reduced by 1.

Static scheduling has been made much more concise in order to reduce overhead.

```
Procedure dynamic_control;
begin
        case process of
1: begin
     repeat until Threshold_Count[11]=1 ;   {check threshold}
      {do task 11- compute angular velocity for link 2}
      Threshold_Count[2]:=succ(Threshold_Count[22]);
      Threshold_Count[3]:=succ(Threshold_Count[16]);
       Threshold_Count[11]:=0;                `{reset threshold}
                                   :
     repeat until Threshold_Count[28]=1 ;   {check threshold}
      {do task 28}
     Threshold_Count[29]:= succ(Threshold_Count[29]);
      Threshold_Count[28]:=0;               {reset threshold}
   end;{process 1}
                                   :
 4:  begin
                                   :
     repeat until Threshold_Count[20]=1 ;  {check threshold}
     {do task 20 compute torque for link 2}
      Threshold_Count[20]:=0;              {reset threshold}
                                   :
    end;{process 4}
  end;{case}
   fbarrier(process_barrier);
end;{dynamic_control}
```

We can see that the static scheduling involves substantial manual work to map tasks to specific processors.

Table 1: Times for Dynamic and Static Implementations (40000 iterations)

	scheduling	processor number	execution time (sec)	speedup
sequential			142.3	
parallel	dynamic	1	186.9	
		2	130.7	1.09
		3	104.2	1.37
		4	87.2	1.63
		5	80.8	1.76
	static	1	142.3	1.00
		2	84.7	1.68
		3	66.3	2.15
		4	61.3	2.32
		5	60.0	2.37

The results in Table 1 are for the dynamic control routine. It can be see that for dynamic scheduling, using one processor, the execution time is worse than that of the sequential scheme, using more than two processors, only a slight speedup can be achieved. This is because that processors spending quite a long time on scheduling and spinlock and spinunlock. It also can see

that for the static scheduling, the moderate speedup has been achieved by using several processors.

IV. Implicit Parallel Implementations in Sisal

Sisal is a functional language which has been targeted at a wide variety of systems including current generation multiprocessors such as the Encore Multimax and research dataflow machines [6][7]. The multi-targeting feature is accomplished by compiling SISAL to an intermediate language IF1. The IF1 representation is then compiled to the appropriate target instruction set. The textual form of SISAL, in terms of control structures and array representations, provides a relatively easy transition for those familiar with imperative languages. The optimising SISAL compiler (OSC) from Colorado yields performance competitive with FORTRAN.

In Sisal program, there two forms of Loop expression, one is of the form:

```
for initial
    index:=lowest_index;
    variable:=initial_value;
while index<=highest_index repeat
    index:=old index + 1;
    variable:=function1(old index, old variable);
return array of variable
end For
```

Another form of Loop expression is:

```
for index in lowest_index, highest_index
   variable:=function2(index);
return array of variable
end For
```

The first one performs sequential loop in which one iteration depends on the results of previous iteration. The second form may be used when there are no data dependencies between iterations, this makes it is possible to execute several loops in parallel.

<u>Sisal on a Conventional Shared Memory Multi- processor</u> As we know, the dynamic control of manipulator involves calculating nine equations for each link. The data dependencies between two equations among nine equations and between two links are so strong that it is difficult to write the calculation program in second form of parallel Sisal loop. The manipulator program expressed in Sisal is presented below. It can be seen that there are no directives as to how the tasks should be scheduled.

```
type    vector = record[x,y,z:real];
type    matrix = record[n,o,p:vector];
                  :
function VAV(a,b:vector returns vector)        %vector + vector
   let
     c:=record vector[x:a.x+b.x;y:a.y+b.y;z:a.z+b.z]
   in c
   end let
end function
                  :
function dynamic_control(z0:vector;dq,ddq,m,bo:array[real];
         po,s:array[vector];Tm,Tm_1,Im:array[Matrix] returns
                                w,dw,v,dv,dcv,Fe,Ne,fc,nc,t)
let
 wj,dwj,dvj,dcvj,Fej,Nej:=
   for initial
       link:=0;
     wi:=record vector[x:0.0;y:0.0;z:0.0];
```

```
                    :
      while link<=4 repeat
         link :=old link+1;
         wi,dwi,dvi,dcvi,Fei,Nei:=
         if motion[link]=1 then                              %rotation
           let
             %compute angular velocity
            zdq:=SMV(dq[link],z0);
            value01:=VAV(old wi,zdq);
             w1:=MMV(Tm_1[link],value01);
                        :
             %compute external moment - Ne1
           in w1,dw1,dv1,dcv1,Fe1,Ne1
           end let
        else                                                 %translation
           let
             %compute angular velocity
             w1:=MMV(Tm_1[link],old wi);
                        :
             %compute external moment - Ne1
           in w1,dw1,dv1,dcv1,Fe1,Ne1
           end let
         end if
         returns array of wi
                        :
                array of Nei
      end for;
      w,dw,dv,dcv,Fe,Ne:=array_setl(wj,0),
                                     :
                        array_setl(Nej,0)
                    :
        in w,dw,dv,dcv,Fe,Ne,fc,nc,t
        end let
      end let
    end function
```

where: SMV, VAV and MMV are procedures that perform the multiplication of a scalar and a vector, the addition of two vectors and the multiplication of a matrix and a vector, respectively.

Table 2: Times for Sisal on a Conventional Multiprocessor (40000 iterations)

processor number	execution time (sec)		
	whole program	outside dynamic control routine	dynamic control routine (infer)
1	291.8	175.3	116.5
2	308.5	189.3	119.2
3	312.5	190.9	121.6
4	316.1	196.4	119.7

Currently, Sisal compiler OSC (Optimising Sisal Compiler) can't exploit the parallelism in **for initial···while** loop (only can exploit the parallelism in **for ··· in** loop) for shared-memory machines. Procedural level concurrency of the type found in the manipulator control program is not currently exploited as can be seen from the results in Table 2(It is hard to work out the execution time for the dynamic control routine alone since Sisal optimise eliminates the routines as ' dead

code ' if their results are not used). It can be observed however that the run times for Sisal on a conventional multiprocessor compare favourably with the Pascal implementations. The execution time slightly increasing with additional processors is caused by the operating system overhead.
Sisal on a Dataflow Multiprocessor The dataflow model has been introduced to exploit the maximum parallelism inherent in algorithms since the early 1970s. Dataflow microprocessors are commercially available and other microprocessors such as the Inmos transputers may be used as dataflow processors[14]. Unlike the conventional control-flow model the course of a computation is determined **solely** by the availability of data, therefore, the dataflow model can avoid most of problem existing in the conventional multiprocessor system, such as memory conflict, side effects, etc.

Dataflow programs are represented by a directed graph where the arcs denote paths over which data travels and the node the computational function (instruction operation). A node 'fires' as soon as all its operands arrive on all its input arcs. When the node fires, results are transmitted to successor nodes in data packets called token and these nodes will cause further firings. Potentially many nodes may be eligible to fire in parallel.

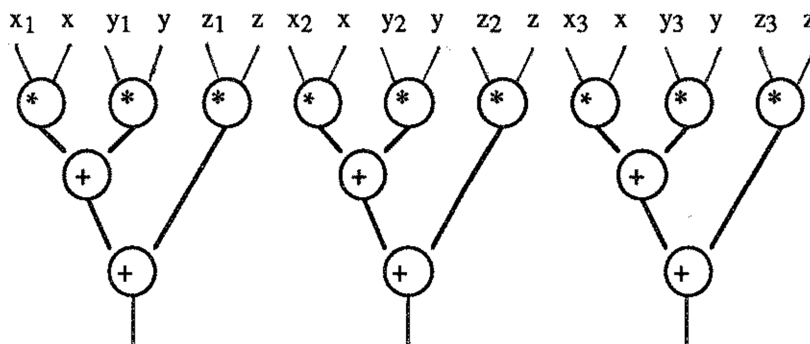


Figure 2 Dataflow Graph Execution Mechanism For

$$\text{matrix_A}\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix} * \text{vector_B}(x,y,z)$$

The hardware of the dataflow machine studied (CSIRAC II) consists of a homogeneous array of processing-elements or processors interconnected by a modulo 4 multi-stage interconnection network (MIN). The graph describing the computation to be performed is partitioned and the partitions distributed to the processing elements [8][9][10]. In CSIRAC II, a processing element consists of two main functional units (Figure 3). If the destination node is monadic, the token can be directly passed to the evaluation unit; If the destination node is diadic, then the matching unit retains the token and processes the next token; the retained token is retrieved when its partner arrives. In the evaluation unit the node function is evaluated and the results are dispatched through the communication network to their destinations[11].

The CSIRAC II development experiment environment provides a simulator for detailed language studies. In the simulator, the discrete time increment, nominally 10 nsec, is used to calculate all execute time. The transmission time of communication network is assumed as 100 nsec per 128 bit word and the relevant communication path is modelled as being busy for the entire period of token transmission. The default communication latency is 500 nsec although this may be varied. Pipelining within processing elements is not modelled directly but approximated by commencing the processing of tokens a time equal to half the transmission time after the token has been dispatched. The rate for reading tokens from and written tokens to queues is assumed as 50 nsec per word and the time for tokens to perform the matching operation is assumed as a very conservative 500 nsec plus another 500 nsec for each further search or token storage. The basic node evaluation time is 100 nsec with the more complex node evaluation time set to appropriately long time [18][19]. The execution time of a program running on the simulator is calculated based on above assumption. The simulator also generates a number of graphics for performance analysis[8].

degrees of freedom, so there are in total 45 equations.

III. Explicit Parallel Implementations on a Shared Memory Multiprocessor

In these implementations, the Pascal programs augmented explicitly with the synchronisation primitives from the parallel programming library of Encore Multimax, a shared- memory multiprocessor system with six processors, is used to implement the computation of manipulator dynamic control equations.

The primitives used were fork, *spinlock, spinunlock, fbarrier_init, fbarrier* and the memory allocation directive *share*[5], where fork is used to create a new process. The new process (the child) is an exact copy of the calling process (the parent) except for the child process has an unique process ID. In the program, one process runs on one physical processor. Spinlock and spinunlock are used where necessary to provide exclusive access to the data structures located in a shared memory. Fbarrier_init and fbarrier are used to synchronise the parallel processes on each iteration of the control loop.

Decomposition of the computing load into tasks The decomposing of the computing load is the first and important step in the application of parallel processing. If we split the computing load into coarse grains, only a few tasks can be performed concurrently. On the other hand, if the grains are too small, the data transfer activities between the processors (hence the operating system overhead) will increase. This effect will lead to the performance degradation of parallel processing.

After a substantial number of experiments[15], we finally partitioned the recursive Newton-Euler equations into 51 tasks. A task graph was sketched to represent the ordering constraint arising from the data dependencies between the tasks and is shown in Figure 1, where task 0 and 52 are dummy tasks, representing enter node and exit node respectively.

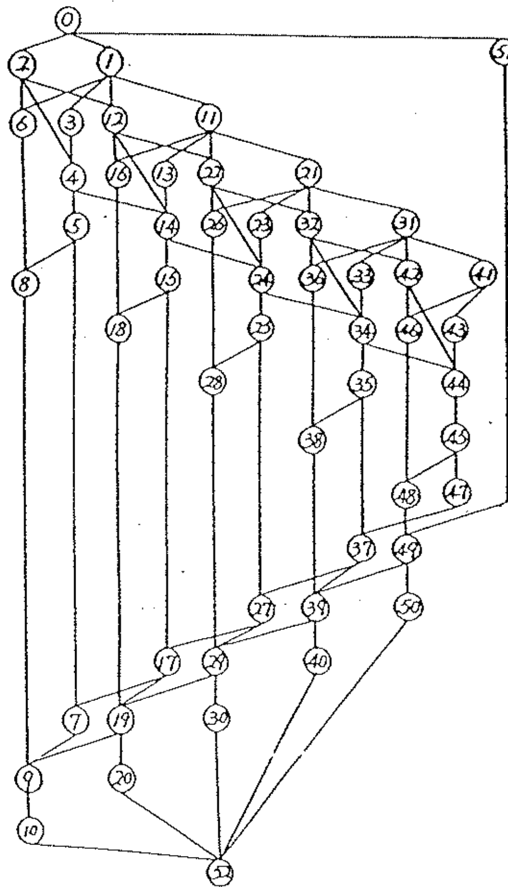


Figure 1: Newton-Euler Task Graph and Dependencies

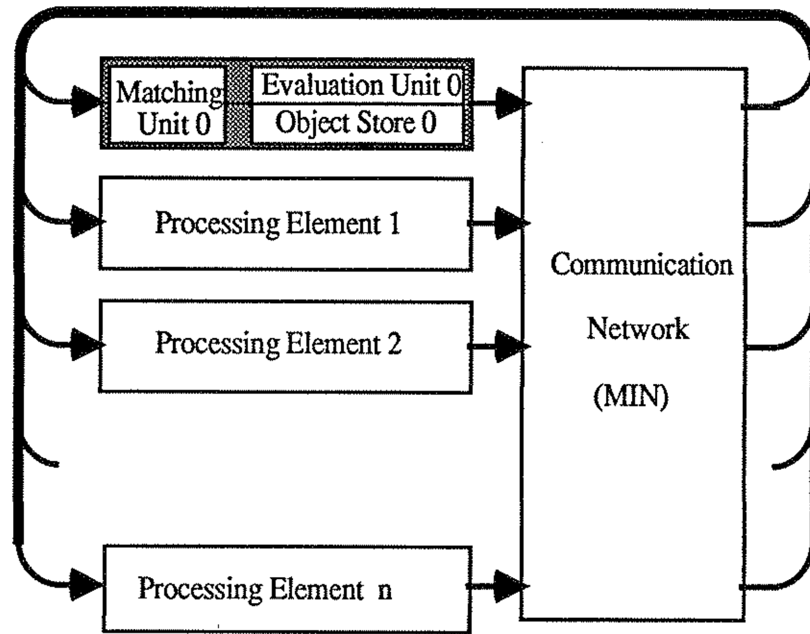
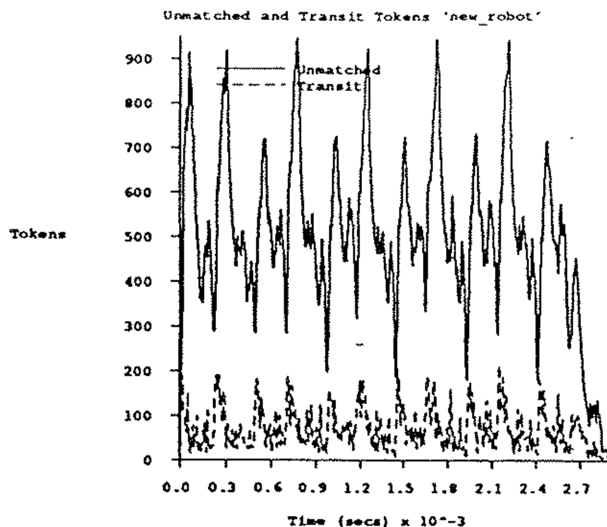


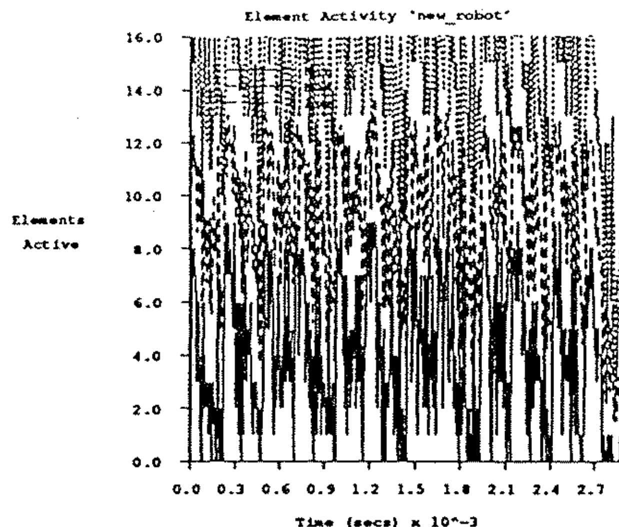
Figure 3: CSIRAC II Organisation

For CSIRAC II, Sisal programs are compiled by the front end of OSC into IF1(an intermediate form). The IF1 is then translated into i2, an intermediate target language which directly represents a data flow graph [9][13].

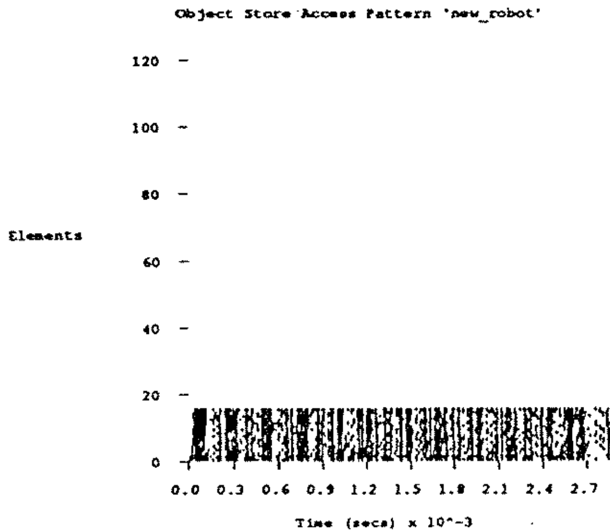
The set of graphs in Figure 4 shows the machine activity during the simulation. Where graph 1 shows the number of unmatched tokens and transited tokens; Graph 2 shows the maximum, minimum and average number of active processing elements during each time step; Graph 3 shows the activity of processing elements accessing the object store; Graph 4 shows the work-load distribution, a dark spot indicates that processing element is active [8].



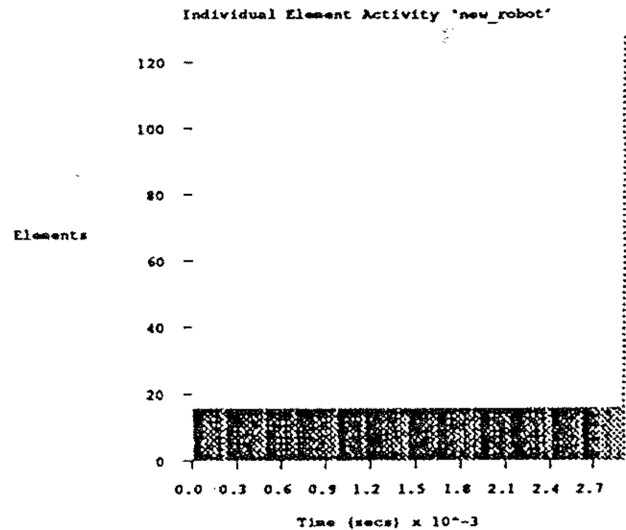
Graph 1



Graph 2



Graph 3



Graph 4

Figure 4: Machine Activity During the Simulation

The following Table 3 indicates the execution time of whole simulation program for ten iterations by using different number of processing element:

Table 3: Runtime for SISAL on dataflow (10 iterations)

processing element number	execution time (ms)	speedup
1	27.506	
2	14.472	1.91
8	4.386	6.27
16	2.923	9.41
32	2.329	11.81
64	2.149	12.80
128	2.085	13.19

V. Conclusion

We have investigated several computation models for the dynamic control of a manipulator. As we know, program parallelism involves the partition of a computing load. In theory, if we wish to speedup our program, we should cut down the critical path length, to do so, we should produce a fine grained program to exploit low level parallelism in tasks, further reducing critical path length. However, if the grains are too small, there will be substantial overhead associated with coordinate processes. Therefore, the actual critical path length will increase, and the potential gain from parallelism will be overwhelmed by this lengthening of the critical path. This can be seen from our shared- memory schemes, where, each processors can directly access a data structure located in shared memory which also can be accessed by all other processors. In order to avoid this contest, some method is required for ensuring mutual exclusion The method we use here is spinlock. However, the spinlock can degrade the performance of parallel processing since it can slow down processors doing useful work. This is why we only exploit medium granularity for

our shared-memory schemes. Although part of the software scheduling can be replaced by hardware, a certain effects could be received, only a limited speed-up can be achieved since a limited number of processors could be employed.

The dataflow computation model is of a "natural" parallelism, and can exploit very fine grained (instruction) levels of concurrency, which can make the critical path length of a program minimum. Sisal automatically partitions the program into very fine-grained tasks (instruction level) and their data dependencies. The dataflow machine uses a dynamic scheduling policy very similar to the one used in the Pascal based scheme. The simulation results demonstrate the potential and advantage of dataflow machine to resolve a complicated and high performance real time control problem such as manipulator dynamic control.

References

[1] J. Y. S. Luh, M. W. Walker, and R. P. C. Paul, "On-line computational scheme for mechanical manipulators", ASME Journal on Dynamic Systems, Measurement, Control, Vol.102, pp. 69-76, June, 1980.

[2] J. Y. S. Luh and C. S. Lin, "Scheduling of parallel computer for a computer-controlled mechanical manipulator", IEEE Transactions on Systems, Man and Cybernetics, Vol.12, pp. 214-234, 1982.

[3] H.Kasahara and S.Narita, "Parallel processing of robot-arm control computation on a multi-microprocessor system", IEEE Journal on Robotics and Automation, Vol. RA-1, No.2, pp., 104-113, June, 1985.

[4] E. E. Binder and J. H. Herzog, "Distributed computer architecture and fast parallel algorithms in real-time robot control", IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-16, No.6, pp. 543-549, July / August, 1986.

[5] UMAX 4.3 Programmer's Reference Manual 1.

[6] Sisal Language Reference Manual Version 1.2, March 1,1985.

[7] J. T. Feo and D. C. Cann, "A Report on the Sisal Language Project", Journal of Parallel and Distributed Computing, Vol. 10, pp 349-366,1990.

[8] M. W. Rawling, "Implementation and Analysis of a Hybrid Dataflow System" M.Eng Thesis, Royal Melbourne Institute of Technology, Australia.

[9] G. K. Egan, N. J. Webb and A. P. W. Bohm, "Some Features of the CSIRAC II Dataflow Machine Architecture", in Advanced Topics in Data-Flow Computing, Prentice-Hall 1991, pp143-173.

[10] D.A. Abramson , G.K.Egan "The RMIT data flow computer: A hybrid architecture", Computer Journal , Vol. 33, No. 3, 1990.

[11] D.A. Abramson and G. K. Egan, "Design of a High Performance Data Flow Multiprocessor", in Advanced Topics in Data-Flow Computing, Prentice-Hall, 1991, pp121-141.

[12] M. L. Welcome, et al., "IF2: An Applicative Language Intermediate Form with Explict Memory Mangement, Lawrence Livermore National Laboratory Manual M-195". Lawrence Livermore National Laboratory, Livermore, CA, November, 1986.

[13] G. K. Egan, M. Rawling and N. J. Webb "i2: An Intermediate Language for RMIT Data Flow Machine", Technical Report 31-004, Laboratory for Concurrent Computing Systems Systems, School of Electrical Engineering , Swinburne Institute of Technology.

[14] A. Katbab, "A Multiprocessor Architecture for Robot-Arm Control", Microprocessing and Microprogramming 24 673-680,1988.

[15] S. Zeng and G. K. Egan, " Parallel Processor Implementations of the Recursive Newton-Euler Equations Used in the Dynamic Control of Robots", Technical Report 31-028, Laboratory for Concurrent Computing Systems Systems, School of Electrical Engineering , Swinburne Institute of Technology.

[16] C. L. Chen, C.S.G.Lee and E.S.H.Hou, "Efficient scheduling algorithm for robot inverse dynamics computation on a multiprocessor system", IEEE Trans. syst. man., cybern., vol.18, No.5, September/October, 1988.

[17] S. Zeng, "The Control of High -Performance Manipulators using Multiprocessor Computer Systerm", M.Eng Thesis, Laboratory for Concurrent Computing Systems Systems, School of Electrical Engineering , Swinburne Institute of Technology, to appear, 1992.

[18] C. Baharis, "Tomographic Reconstruction on the CSIRAC II Dataflow Computer", Master Thesis, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, April, 1991.

[19] G. K. Egan, "The CSIRAC II Dataflow Computer Simulation Suite", Technical Report 31-010R, Laboratory for concurrent computer system, School of Electrical Engineering , Swinburne Institute of Technology, May, 1990.