Laboratory for Concurrent Computing Systems
Technical Report 31-029

Version 1.0 May 1991

Parallel Processor Implementations of the Recursive Newton-Euler Equations Used in the Dynamic Control of Robots

S. Zeng
Swinburne University of Technology
Australia

Prof G. K. Egan
Swinburne University of Technology
Australia

Abstract:

The dynamic control of robot involves the calculation of the desired force or motor torque required which drive all of the joints appropriately so that it is possible for the robot to follow the desired trajectory. These calculations are based on the robot's dynamic equations and on feedback information about the actual motion. Because of nonlinear characteristics of robots, however, the computing load is usually very heavy. For this reason, an expensive minicomputer or even a super minicomputer having high processing capabilities must be used if control computation for each sampling period are to be performed in less than 1/60 seconds, with the upper limit determined by the mechanical resonance. This technical report presents several software techniques and options to develop parallel processing schemes for the computation of the recursive Newton-Euler equations which is the fastest and the most efficient of existing methods describing robot dynamic control.

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

PARALLEL PROCESSOR IMPLEMENTATIONS OF THE RECURSIVE NEWTON-EULER EQUATIONS USED IN THE DYNAMIC CONTROL OF ROBOTS

S.S. Zeng and G.K. Egan

1 Introduction

The dynamic control of robot involves the calculation of the desired force or motor torque required which drive all of the joints appropriately so that it is possible for the robot follow the desired trajectory. These calculations are based on the robot's dynamic equations and on feedback information about the actual motion. Because of nonlinear characteristics of robot, however, the computing load is usually very heavy. For this reason, an expensive minicomputer or even a super minicomputer having high processing capabilities must be used if control computation for each sampling period are to be performed in less than 1/60 seconds, with the upper limit determined by the mechanical resonance. This technical report presents several software techniques and options to develop parallel processing schemes for the compution of the recursive Newton-Euler equations which is the fastest and the most efficient of existing methods describing robot dynamic control [1].

2 The Recursive Newton-Euler Equations

There are a number of ways to formulate the dynamic equation of motion, two main approaches are used by most of reseachers to systematically derive the dynamic model of the manipulator, one of them is the Newton-Euler formulation [2].

The Newton-Euler equations are based on Newton's second law

$$F_i = m_i a_i$$

where: F_i is the total force acting on link i;
m_i is the mass of link i;
a_i is the linear acceleration of the center of mass of link i.

and Euler's equation

$$N_i = I_i \dot{w}_i + w_i \times (I_i w_i)$$

where: N_i is the total moment acting on link i;
I_i is the inertia matrix of link i about its center of mass;
w_i is the angular velocity of link i;
$\dot{w}_i$ is the angular acceleration of link i.

S.S. Zeng is a graduate student in the Laboratory for Concurrent Computing Systems at the Swinburne Institute of Technology, John Street, Hawthorn 3122, Australia, Phone:+61 3 819 8516, E-mail: ssz@stan.xx.swin.oz.au.

G.K. Egan is Professor of Computer Systems Engineering and Director of the Laboratory for Concurrent Computing Systems at the Swinburne Institute of Technology, John Street, Hawthorn 3122, Australia, Phone:+61 3 819 8516, E-mail: gke@stan.xx.swin.oz.au.

Furthermore, recursive Newton-Euler equations are derived with nine equations for each link:

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

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

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

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

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

if link i is rotational

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

if link i is translational

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

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

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

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

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

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

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

Figure 1

Where: $A_i^0 \dot{w}_i$ is the angular acceleration of link i with respect to its own coordinate;

$A_i^0 \dot{v}_i$ is the linear acceleration of link i with respect to its own coordinate;

$A_i^0 \dot{\hat{v}}_i$ is the linear acceleration of center of mass of link i with respect to its own coordinate;

$A_i^0 F_i$ is the total external force exerted on link i with respect to its own coordinate;

${}^{0}_{i}A N_i$ is the total moment exerted on link i with respect to its own coordinate;

${}^{0}_{i}A f_i$ is the force exerted on link i by link i-1 with respect to its own coordinate;

${}^{0}_{i}A n_i$ is the moment exerted on link i by link i-1 with respect to its own coordinate;

t_i is the input torque for link i (if it is rotational) or the input moment for link i (if it is translational);

q_i, $\dot{q}_i$, $\ddot{q}_i$ is the position, velocity and acceleration of joint i, respectively;

b_i is the viscous damping coefficient for joint i;

m_i is the total mass of link i;

${}^{*}_{i}p$ is the origin of ith frame referred to (i-1)th frame;

${}^{0}_{i}A I_i {}^{i}_{0}A$ is the inertia matrix about the center of mass of link i with respect to its own coordinate;

${}^{0}_{i}A \hat{s}_i$ is the center of mass of link i referred to its own coordinate;

3 Language

In this study Pascal is augmented with the synchronization primitives from the parallel programming library of Encore Multimax to implement the robot's dynamic control equations. The primitives used were *fork, spininit, spinlock, spinunlock,* and the memory allocation directive *share* [3]. Some additional procedures were constructed using these and they are described below:

Fork:	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 a unique process ID. The process ID of the parent process is 0.
SemaphoreInit:	unlock semaphorelock, initialise Semaphore-Count.
Spinlock:	wait in a spin loop for the value of lock to become zero; When it does, it is automaically set non-zero.
Spinunlock:	unlock the lock by setting it to zero.
Fbarrier:	synchronize a fixed a number of threads of control. A thread of control requests to synchronize at a Fbarrier with the Fbarrier function. The fbarrier function is guaranteed not to return until a prespecified number of other threads have waited at the specified Fbarrier.
Fbarrier_init:	The fbarrier_init funtion initializes an existing Fbarrier.

There are two procedures constructed as Barriers -- InitialisationBarrier and ExitBarrier.

WaitInitBarrier:	is used to wait for that all the processes which we desire have been created before executing the parallel processing part.
WaitExitBarrier:	causes processes to wait and allows them proceed only after a predetermined number of processed arrive at the ExitBarrier. It is used to ensure that one stage of a calculation has been finished before the processed proceed to a next stage which requires the results of the

previous stage.

4 Decomposition of Robot Dynamic Control Computing Load into Tasks

The decomposing of the computing load is the first and important step in the application of parallel processing. That the work assigned to processors are unbalance will result in that some processors keep busy all the time, the others are in idle state. The parallelism is not maximum. On the other hand, if we split the computing load into too small grains, the data transfer activities between the processors (hence the operating system overhead) will increase. This effect will lead up to the performance degradation of parallel processing.

As we know, the calculation for robot arm control is generally expressed by Newton-Euler equations which involve many vectors and matrices calculation. In the beginning, we employ the technique that Luh et al [1] used to break nine equations of robot dynamic control computation down into nineteen tasks for each joint. For our five joint robot, there are totally ninety-five tasks. If all of the tasks are assigned to one processor, we simply have a sequential processing. If we assign the tasks to more than one processors, the parallelism is achieved. The nineteen tasks for a link can be expressed as follow:

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

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

(2/i) $VEC1 = (A_{i-1}^0 w_{i-1}) \times (z_0 \dot{q}_i)$ if link i is rotational

or 0 if link i is translational

(3/i) $A_i^{0\cdot} w_i = A_i^{i-1}[A_{i-1}^0 \dot{w}_{i-1} + z_0 \ddot{q}_i + VEC1\ ]$ if link i is rotational

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

(4/i) $VEC2 = A_i^{i-1}(A_{i-1}^0 \dot{v}_{i-1})$ if link i is rotational

or $A_i^{i-1}(z_0 \ddot{q}_i + A_i^{0\cdot} v_{i-1})$ if link i is translational

(5/i) $VEC3 = (A_i^0 w_i) \times (A_i^0 p_i^*)$ if link i is rotational

or $2(A_i^{i-1} z0 \dot{q}_i) + (A_i^0 w_i) \times (A_i^0 p_i^*)$ if link i is translational

(6/i) $VEC4 = (A_i^{0\cdot} w_i) \times (A_i^0 p_i^*)$

(7/i) $A_i^{0\cdot} v_i = (A_i^0 w_i) \times VEC3+VEC2+VEC4$

(8/i) $VEC5 = (A_i^0 w_i) \times [(A_i^0 w_i) \times (A_i^0 \hat{s}_i)]$

(9/i) $VEC6 = (A_i^{0\cdot} w_i) \times (A_i^0 \hat{s}_i)$

(10/i) $A_i^0 F_i = m_i(VEC5+VEC6+A_i^0 \dot{v}_i)$

(11/i) $VEC7 = (A_i^0 I_i A_0^i)(A_i^0 \dot{w}_i)$

(12/i) $VEC8 = (A_i^0 w_i) \times [(A_i^0 I_i A_0^i)(A_i^0 w_i)]$

(13/i) $A_i^0 N_i = VEC7+VEC8$

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

(15/i) $VEC9 = (A_{i+1}^0 p_i^*) \times (A_{i+1}^0 f_{i+1})$

(16/i) $VEC10 = A_i^{i+1}(A_{i+1}^0 n_{i+1} + VEC9)$

(17/i) $VEC11 = (A_i^0 p_i^* + A_i^0 \hat{s}_i) \times (A_i^0 F_i)$

(18/i) $A_i^0 n_i = VEC9 + VEC10 + VEC11 + A_i^0 N_i$

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

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

Figure 2

subtask	predecessor	successor
(1/i)	(1/i-1)	(1/i+1),(2/i+1),(5/i),(8/i),(12/i)
(2/i)	(1/i-1)	(3/i)
(3/i)	(3/i-1),(2/i)	(3/i+1), (6/i), (9/i), (11/i)
(4/i)	(7/i-1)	(7/i)
(5/i)	(1/i)	(7/i)
(6/i)	(3/i)	(7/i)
(7/i)	(4/i), (5/i), (6/i)	(4/i+1), (10/i)
(8/i)	(1/i)	(10/i)
(9/i)	(3/i)	(10/i)
(10/i)	(7/i), (8/i), (9/i)	(14/i), (17/i)
(11/i)	(3/i)	(13/i)
(12/i)	(1/i)	(13/i)
(13/i)	(11/i), (12/i)	(18/i)
(14/i)	(10/i), (14/i+1)	(19/i), (14/i-1), (15/i-1)
(15/i)	(14/i+1)	(16/i)
(16/i)	(15/i), (18/i+1)	(18/i)

(17/i)	(10/i)	(18/i)
(18/i)	(16/i), (17/i), (13/i)	(16/i-1), (19/i)
(19/i)	(14/i), (18/i)	null

We find that the time which spends on the scheduling and spinlock is not short enough comparing with the execution time of each task , which means that overhead doesn't play an unimportant role.

In order to reduce the overhead, We redecompose the computing load into 52 tasks as follows:

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

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

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

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

(3/i) $A_i^0 \dot{v}_i = (A_i^0 \dot{w}_i) \times (A_i^0 p_i^*) + (A_i^0 w_i) \times [(A_i^0 w_i) \times (A_i^0 p_i^*)] + A_i^{i-1}(A_{i-1}^0 \dot{v}_{i-1})$

if link i is rotational

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

$\times [(A_i^0 w_i) \times (A_i^0 p_i^*)]$ if link i is translational

(4/i) $A_i^0 F_i = m_i[(A_i^0 \dot{w}_i) \times (A_i^0 \hat{s}_i + (A_i^0 w_i \times [(A_i^0 w_i) \times (A_i^0 \hat{s}_i)] + A_i^0 \dot{v}_i]$

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

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

(7/i) $VEC10 = A_i^{i+1}[A_{i+1}^0 n_{i+1} + (A_{i+1}^0 p_i^*) \times (A_{i+1}^0 f_{i+1})]$

(8/i) $VEC11 = (A_i^0 p_i^* + A_i^0 \hat{s}_i) \times (A_i^0 F_i)$

(9/i) $A_i^0 n_i = VEC10 + VEC11 + A_i^0 N_i$

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

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

Figure 3

subtask	predecessors	successors

(1/i)	(1/i-1)	(1/i+1), (2/i+1), (3/i), (5/i)
(2/i)	(1/i-1), (2/i+1)	(2/i+1), (3/i), (5/i)
(3/i)	(3/i-1), (1/i), (2/i)	(4/i), (3/i+1)
(4/i)	(3/i)	(6/i), (8/i)
(5/i)	(1/i), (2/i)	(9/i)
(6/i)	(4/i), (6/i+1)	(6/i-1), (7/i-1)
(7/i)	(6/i+1), (9/i+1)	(9/i-1)
(8/i)	(4/i)	(9/i)
(9/i)	(5/i), (7/i), (8/i)	(10/i), (7/i-1)
(10/i)	(9/i)	null

5 Scheduling

Encore Multimax is a shared-memory multiprocessors system with four processors. In program, using the syncronization primitives in the parallel programming library, we can command the operating system to create a new process (child process). The child process is a exact copy of the calling process except for it has unique ID. When we decompose the computing load into many tasks, every task may have predecessors and successors. A task can not start until all of its predecessors are completed. There are two ways to schedule a task queue. One is dynamic scheduling. Another one is static scheduling.

For dynamic scheduling, there is only one task queue, and it is scheduled at run-time, which means that when all of predecessors of a task have been completed, this task is said to be ready. and it is put into the queue. In the meantime, if a process has finished its job, it reads next ready task in the queue, and implements this task. Since the queue is in the shared memory so that it can be accessed by every process, it certainly needs a spinlock (q-lock) to protect that only one process can read from and write into the queue.

For static scheduling, each process has its own task queue. The predetermined order need to be made in which tasks are implemented on each process. At run-time, each process waits for data to be available for next task in its own queue. There are two ways to make queues for static scheduling. One way is preassigning tasks among the processes according the execution time of each task. At run-time, when a process finishs a task, it puts each ready successor of the task into corresponding process's queue. In another words, the queues are also arranged at the run-time and in the shared memory, the spinlocks (q-locks) need for the same sake. In version 1.0 ,2.0 and 3.0, the static scheduling is belong to this kind of way. Another way is predetermining a queue for each process, inputting the queues as data files. Each process checks the threshold of the task in its own queue, when all of predecessors of this task have finished, the process implements this task and starts to check the next task in the queue after finishing this task. The queues are in the local memory. The spinlocks are not necessary. In version 4.0, the static scheduling is following this way. Since it is hard to estimate the exact execution time of each task, the simulant way is using write statement to trace the list of a queue when we run dynamic scheduling program, and put it into the list of queue in static scheduling. It should approximate the real static scheduling.

5.1 Scheduling in Version 1.0

In this version, We tried to calculate concurrently Newton-Euler equations within one link. As we know from Figure 1, the equations above the dashed line can be decomposed into threeteen tasks and are calculated repeatedly from link one to link five. The equations under the dashed line can be decomposed into six tasks and are calculated repeatedly from link five to link one, and can't be calculated until the tasks above the dashed line have been completed. The task graph for one link can be expressed as figure 4.

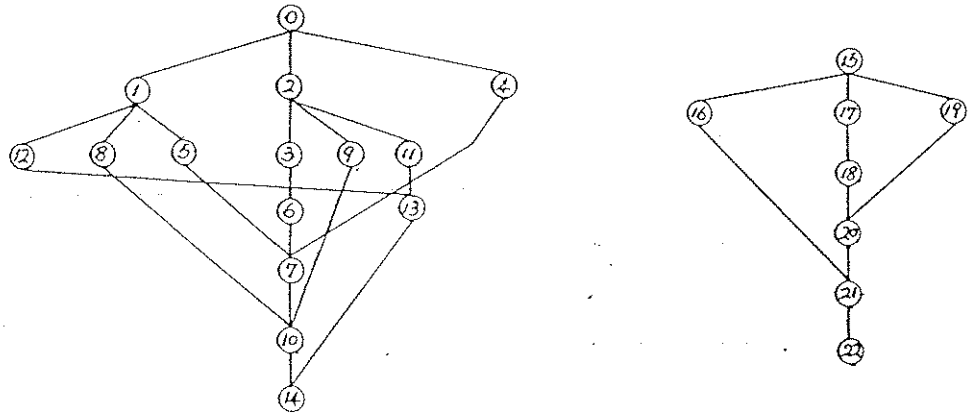

Figure 4

where task 0,14, 15 and 22 are dummy tasks.

The scheduling consists of several procedures. They are procedure Sched, Next, Threshold and SchedSuccessors. In the program, a queue can be expressed in a following form:

```
q_rec = record
            q_lock: lock;
            entries, head,tail : integer;
            t  : array [sub_task] of sub_task;
            sl :array [sub_task] of sub_link;
        end;
```

Each task has a task semaphore which has the following form:

```
SemaphoreRec = record
            SemaphoreLock: lock;
            SemaphoreCount : semaphor
        end;
```

Each task has a task descendant which can be described by

```
Task_Rec = record
            NoOfPred: sub_pred;
            NoOfSucc: sub_succ;
            S: array [sub_succ] of record
                  Task: sub_task;
                  Proc: sub_proc;
                  Link: sub_link
               end;
        end;
```

and these variables should be in the shared_memory so that every process can address them.

```
shared
            :
        queue: array [ sub_proc] of q_rec;
        task_sem: array [sub_task] of SemaphoreRec;
        TaskDesc: array [sub_task] of Task_Rec;
        task_lock: lock;
        remaining_tasks: integer;
            :
```

Procedure Sched puts the ready tasks into the queue of each process. For dynamic scheduling, all of processes share a queue.

The outline of this procedure could be:

```
Begin
   # if DYNAMIC
       with queue[1] do begin
   # else DYNAMIC
       with q[proc] do begin
   # end if  DYNAMIC
          spinlock(q_lock);
          entries:=succ(entries);
          tail:=(succ(tail) and max_task);
          t[tail]:=task;
          sl[tail]:=link;
          spinunlock(q_lock)
       end;
End;
```

Procedure Next selects a task from the queue on principle of FIFO and is going to execute it.

The outline of this procedure could be:

```
Begin
    # if DYNAMIC
       with queue[1] do begin
    # else DYNAMIC
       with q[proc]  do begin
    # end if DYNAMIC
          spinlock(q_lock);
          if entries<=0 then
          task:=max_task;
          else
             begin
                entries:=pred(entries);
                head:=(succ(head) mod max_task);
                task:=t[head];
                link:=sl[head];
             end;
          spinunlock(q_lock);
       end;
End;
```

Procedure Threshold makes semaphoreCount of each task whose predeccessors have been finished decrease by 1, if the semaphoreCount of a task is equal to zero,then starts to put it into the queue and resets SemaphoreCount.

The outline of this procedure could be:

```
Begin
    With TaskDesc[task], task_sem[task] do
       begin
          spinlock(SemaphoreLock);
          SemaphoreCount:=pred(SemaphoreCount);
             If SemaphoreCount:=o then begin
                Sched(queue[proc], task, link);
                 SemaphoreCount:=NoOfPred;
             end;
          spinunlock(SemaphoreLock);
       end;
```

```
End;
```

Procedure SchedSuccessors schedules the next task for a process when it finishs its job. The outline of the procedure could be:

```
Begin
    with TaskDesc[task] do begin
        for i:=1 to NoOfSucc do with S[i] do
        Threshold(task,proc,link);
     end;
End;
```

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

The outline of the procedure could be:

```
Begin
    repeat
    next (queue[proc],task,link);
    if task<>max_task then
            :
        case  task of
    0:    begin
            SchedSuccessors(task);
        end;
    1:    begin
            {task 2}
            SchedSuccessors(task);
        end;
            :
            :
    14:   begin
            if 13 tasks above of each link have finished then
            Sched(queue[proc],15,5);
            else go back to execute task 0 to 13
        end;
    15:   begin
            SchedSuccessors(task);
        end;
            :
            :
    22:   begin
            SchedSuccessors(task);
         end;
      end;{case}
        spinlock(task_lock);
        remaining_tasks:=pred(remain_task);
        spinunlock(task_lock);
 until remaining_tasks<=0
End;
```

5.2 Scheduling in Version 2.0

In version 1.0, we can see that there are not many tasks which can be processed concurrently. In order to overcome this drawback, We tried to calculate the N-E equations throught all of links. The N-E equations for five links can be decomposed 95 tasks. Each task can be executed whenever its predecessors are completed. The task graph for 95 tasks is shown as Figure 5.

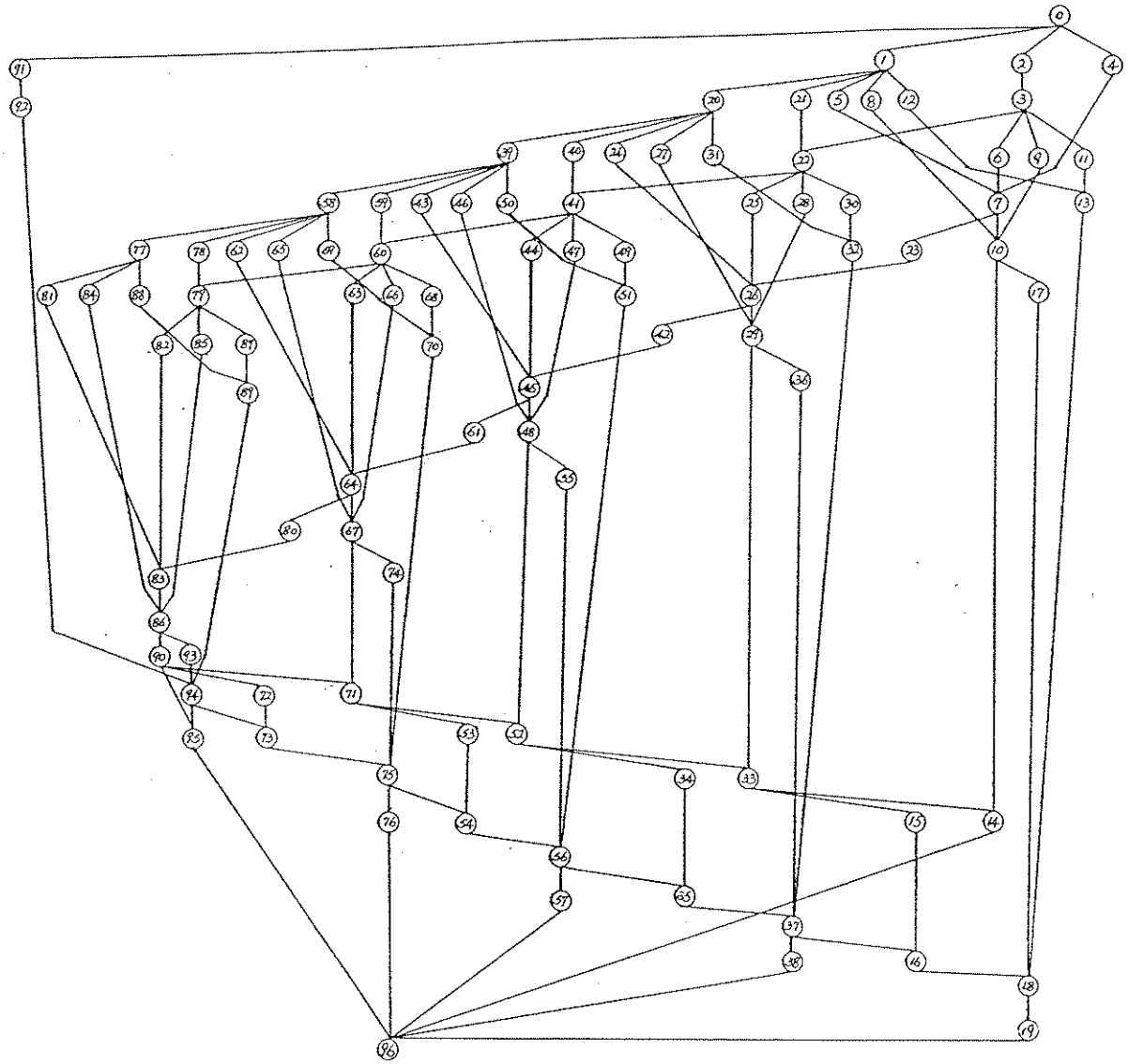

Figure 5

Where task 0 and 96 are dummy tasks. They are enter node and exit node respectively.

Since each link has the same calculation equations, we set a variable subtask, for link i, task No. is from [1+(i-1)*20] to [19+(i-1)*20]. It can be translated to subtask No: subtask No = [task No. mod 20]. In the program, we only need to list 19 calculation subtasks, and task 0 is a dummy task for enter node, task 100 is a dummy task for exit node. As a result of doing above, we can advoid to occupy too many memory and make the program concise. However, when we run the program, we find that the scheduling takes longer time than that we expect so that we have to reduce calculation in the scheduling as possible as we can. The update program directly uses varible task to seach the respective task.

The scheduling of this version is similar to that of version 1.0. The outline of procedure Dynamic_control is written as:

```
Begin
    repeat
        Next(queue[proc], task);
        if task<>max_task then
                :
            case task of
```

```
        0:  begin
              SchedSuccessors(task);
            end;
        1:  begin
              {task 1};
              SchedSuccessors(task);
            end;
              :
              :
       96:  begin
              remaining_tasks:=0;
            end;
      end;{case}
    until remaining_tasks<=0
End;
```

Since task 96 is exit node, we can remove the task_lock and directly set remaining_tasks=0 in this task.

5.3 Scheduling in Version 3.0

In version 3.0, the computing load is decomposed into 52 tasks. The task graph for 52 tasks can be expressed by Figure 6.

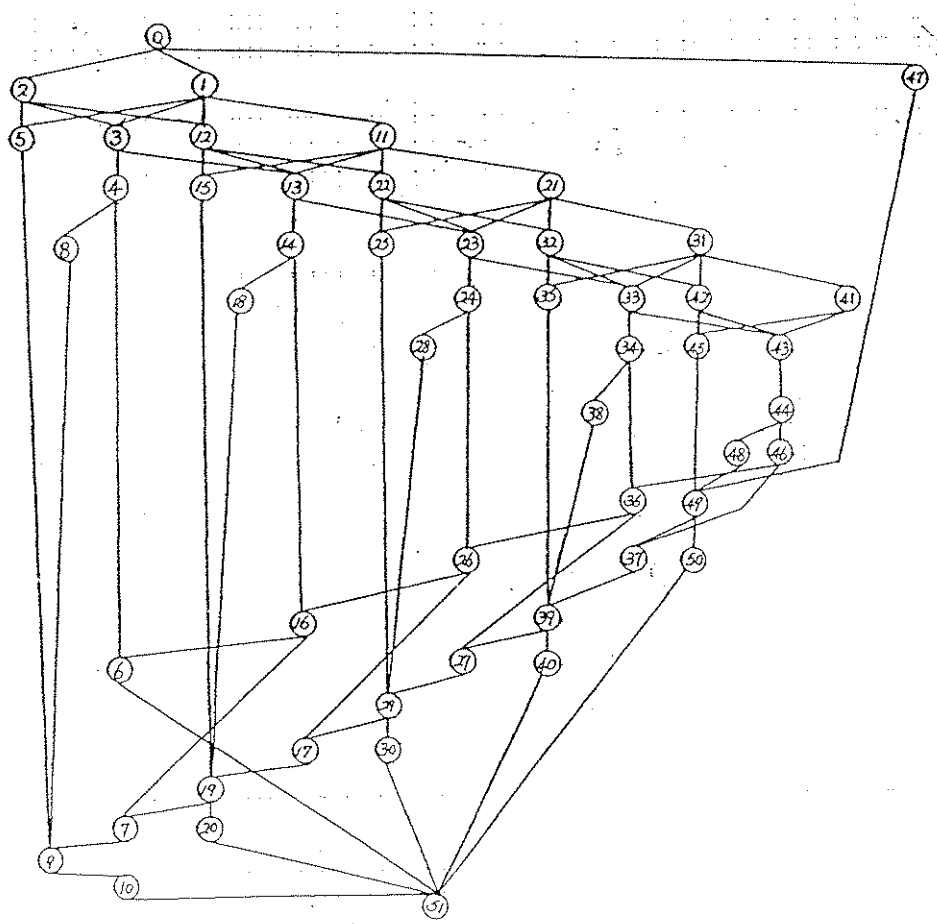

Figure 6

Where task 0 and 51 are dummy tasks.

The scheduling of this version is the same as that of version 2.0, and procedure Dynamic_control is similar to that of version 2.0 as well except there are only 52 tasks.

5.4 Scheduling in Version 4.0

As we said before, the q_lock has been removed in this version since the queue of each process is in a local memory. The scheduling has been made much conciser in order to cut down overhead. The queue is defined as:

```
q_rec = record
            NoOfSubtask: sub_task;
            point : integer;
            t: array [sub_task] of sub_task;
            sl: array [sub_task] of sub_link;
        end;
```

and the queue is located in the local memory:

```
var
      :
   queue: array [sub_proc] of q_rec;
      :
```

The scheduling only contains two procedures-- procedure Next and procedure SchedSuccessors.

Procedure Next checks a task in the queue which is input as data file ready or not, if it is ready, it is going to be executed. The procedure is given by

```
Begin
   with queue[proc] do begin
      task:=t[point];
      with task_sem[task] do
         if SemaphoreCount<>0 then
         task:=max_task;
      link:=sl[point];
   end;
End;
```

Procedure SchedSuccessors sets point to next task in the queue when a process finishs its job, resets SemaphoreCount for this job and makes the SemaphoreCount of all of this job's successors decrease by 1.

The procedure is described by:

```
Begin
   with queue[proc] do
      if point >=NoOfSubtask then point:=1
      else point:=succ(point);
   with TaskDesc[task] do begin
      with task_sem[task] do
         SemaphorCount:=NoOfPred;
      for i:=1 to NoOfSucc do with S[i] do
         with TaskDesc[task], task_sem[task] do begin
            spinlock(SemaphoreLock);
            SemaphoreCount:=pred(SemaphoreCount);
            spinunlock(SemaphoreLock);
         end;
   end;
```

```
    End;
```

The body of procedure Dynamic_control can be given by:

```
Begin
    repeat
        Next(queue[proc], task,link);
        if task<>max_task then
          case task of
        0:  begin
                SchedSuccessors(proc,task);
            end;
        1:  begin
                {task 1};
                SchedSuccessors(proc,task);
            end;
              :
              :
        51: begin
              remaining_tasks:=0;
               task_sem[51].SemphoreCount:=6;
               queue[proc].point:=1;
            end;
      end;{case}
     until remaining_tasks<=0
End;
```

5.5 Scheduling in version 5.0

The schedulings in above versions have a common disadvantage since they can not guarantee the tasks in critical path will be done first. The reason is that we can't estimate the exact execution time of each task, and we only use write statement to trace the action of each task when we run the dynamic scheduling program. There exists a problem: if two tasks are ready at same time while only one process is available, which task should be done first ? Certainly is the one which is in the critical path and has most immediate successors. However, the dynamic scheduling can not guarantee this. In this version, we generated the queue list of each process by using static CP/MISF (Critical Path Most Immediate Successors First) scheduling. The CP/MISF scheduling has a rule that can be stated as: if several tasks are available at the same instant, then the task in critical path should be done first, if they are all not in critical path, then the task having most immedialte successors should be done first. We make a rule: if a task and its parent are in the same process' queue list, it is not necessary to spend time on decreasing Threshold of the task and checking the Threshold,and put all of tasks in critical path into the same process' queue, and the tasks which have a same parent and same children can be combinated together.

We try to using direct mapping method to avoid using procedure and record data structure as well. The new version dynamic_control procedure can be described by:

```
shared      :
            Threshold_Lock: array [sub_task] of lock;
            Threshold_Count: array [sub_task] of semaphore;
             :

Procedure TaskThreshold(task: sub_task);
   Begin
        spinlock(Threshold_Lock[task]);
        Threshold_Count[task]:=pred(Threshold_Count[task]);
        spinunlock(Threshold_Lock[task]);
   End;
```

```
Procedure Dynamic_Control
  Begin
    repeat
      with queue[proc] do begin
        task:=t[point];
        if Threshold_Count[task]<>0 then
          task:=max_task;
      end;
      if task <> max_task then
        case task of
 0:   begin
       queue[proc].point:=succ(queue[proc].point);
       Threshold_Count[0]:=1;
       TaskThreshold(1); TaskThreshold(2); TaskThreshold(47); TaskThreshold(52);
      end;
 1:   begin
        {task 1};
        queue[proc].point:=succ(queue[proc].point);
       Threshold_Count[1]:=1;
       TaskThreshold(11); TaskThreshold(3); TaskThreshold(12);
      end;
                    :
                    :
 51:  begin
        remaining_tasks:=0;
        Threshold_Count[51]:=6;
        queue[proc].point:=1;
      end;
    end;{case}
    until remaining_tasks<=0
End;
```

The new task level for four processes can be presented by:

Level No.	Task No. (process 1)	Task No. (process 2)	Task No. (process 3)	Task No. (process 4)
0	0			
1	52	1	2	47
2	53	11	12	3
3	54	21	22	13
4	55	31	32	23
5	56	41	42	33
6	24	34	45	43
7	25	35		44
8	28	38	46	48
9	14		36	49
10	15	50	26	37
11	18	4	16	39
12		5	40	27
13		8		29
14		6	30	17
15			20	19
16				7
17				9
18				10
19				51

where, the task 3 is broken down to two tasks, task 3 and task 52, so do task 13 ,task 23,task 33 and task 43.

The task level for three processes is presented as:

Level No.	Task No. (process 1)	Task No. (process 2)	Task No. (process 3)
0	0		
1	1	2	47
2	11	12	3
3	21	22	13
4	31	32	23
5	41	42	33
6	34	45	43
7	24	35	44
8	14	46	48
9	38	36	49
10	25	26	37
11	28	16	39
12	15	50	27
13	18	40	29
14	4	30	17
15	5	20	19
16	8	6	7
17			9
18			10
19			51

We also directly use primitives fbarrier_init and fbarrier from parallel programming library in stead of using procedures InitialisationBarrier and ExitBarrier;

6 Parallel Processing of Dynamic Control Equations Calculation

The flow chart of the sequential program of closed loop control of robot is shown in figure 7.

The procedure Trajectory_Generation gives the desired position, orientation, velocity and acceleration of robot's hand, from which the desired position of each joint can be worked out by using inverse kinematics. The procedure Initialisation gets parameters of each link from data files and initialises q, q' q". The procedure Dynamic_control calculates the recursive Newton-Eular equations. The procedure Motor_Drive_Joint transmits the calculated motor torques to motors so that motors drive joints to follow the desired trajectory , and the procedure Sample samples the q and q'. The procedure Compute_Actual_Position & Orientation calculates the actual position and orientation of robot's hand by using kinematcs. The Compute_Error computes the errors between the desired position, orientation and the actual positin, orientation. If the errors are less than predetermined errors, then the robot has arrived the desired the position and orientation, else the procedure Compute_q" calculates q" of each joint and the program goes back the procedure Dynamic_Control, calculates the new motor torques.

The flow chart of the parallel program of closed loop control of robot is represented in figure 8.

The procedure Read_file reads the desired number of processes, the number of tasks, the predecessors and successors of each task and the queue list of each process (for static scheduling). The procedure Initial_Work unlocks the SemaphoreLock of each task and set SemaphoreCount of each task equal to number of predecessors. The procedure Fork_Process creates a number of

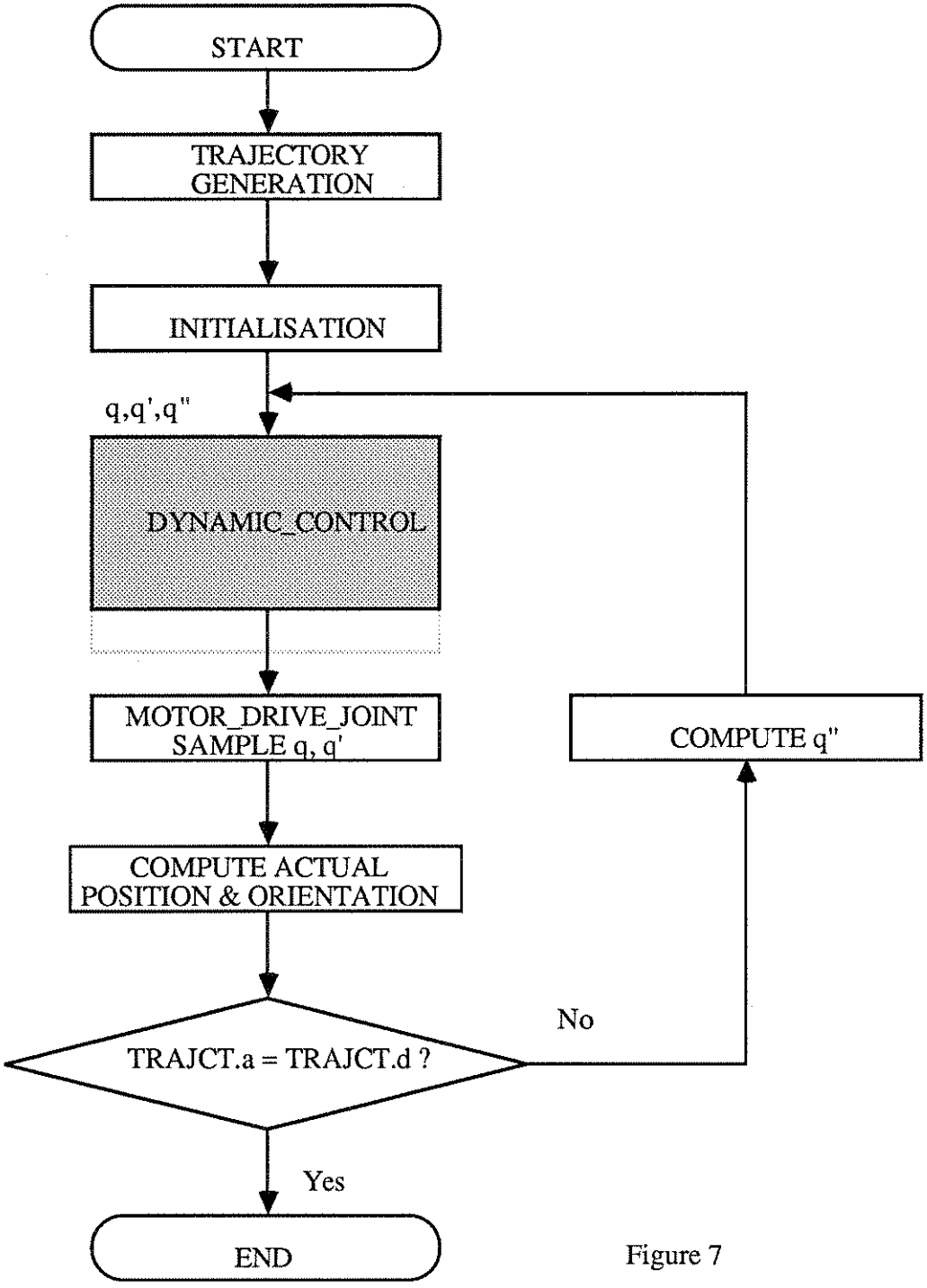
START
TRAJECTORY GENERATION
INITIALISATION
q,q',q"
DYNAMIC_CONTROL
MOTOR_DRIVE_JOINT SAMPLE q, q'
COMPUTE q"
COMPUTE ACTUAL POSITION & ORIENTATION
No
TRAJCT.a = TRAJCT.d ?
Yes
END

Figure 7

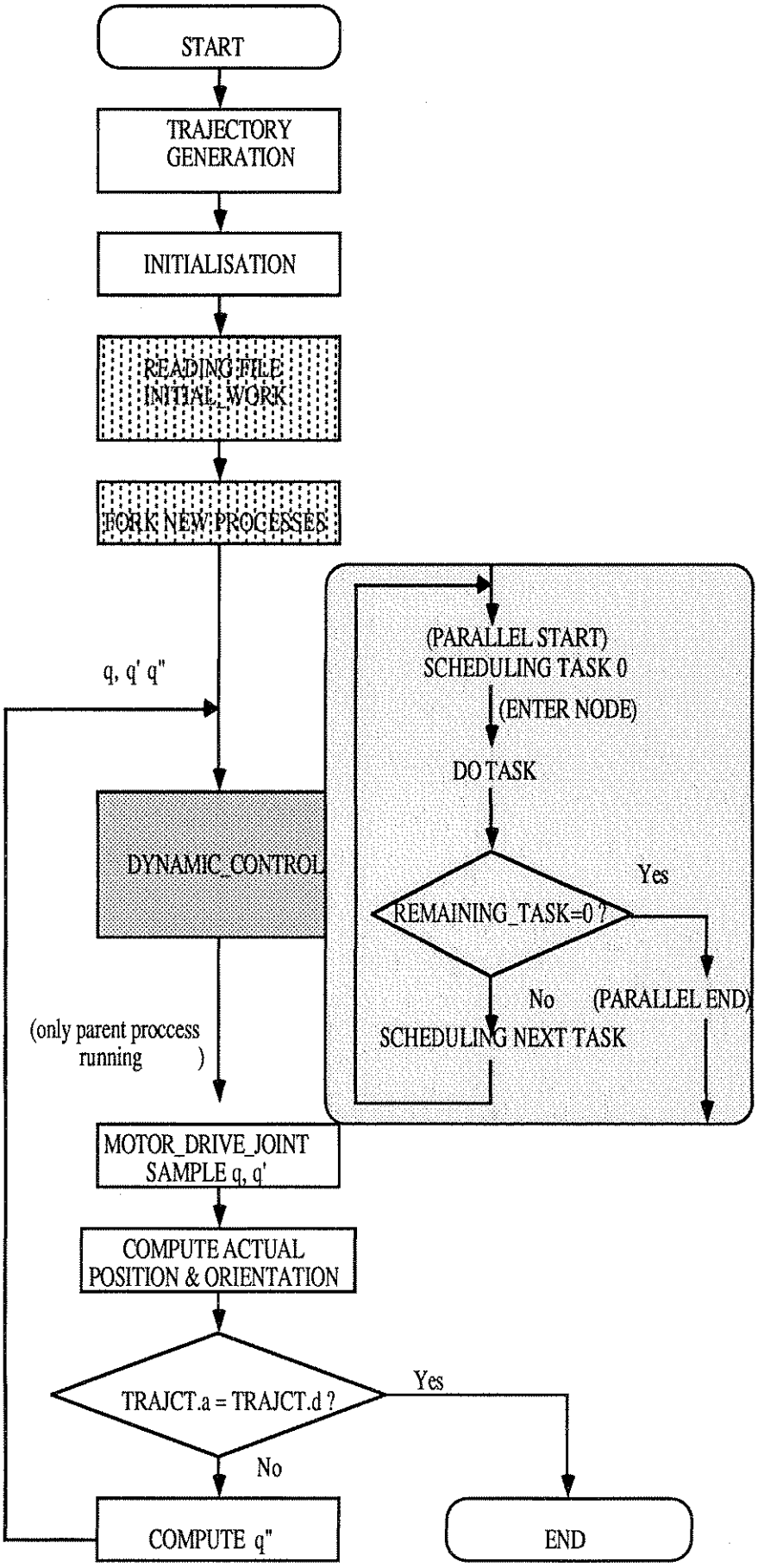
START
TRAJECTORY GENERATION
INITIALISATION
READING FILE INITIAL_WORK
FORK NEW PROCESSES
q, q' q"
DYNAMIC_CONTROL
(only parent proccess running)
MOTOR_DRIVE_JOINT SAMPLE q, q'
COMPUTE ACTUAL POSITION & ORIENTATION
TRAJCT.a = TRAJCT.d ?
Yes
No
COMPUTE q"
END
(PARALLEL START)
SCHEDULING TASK 0
(ENTER NODE)
DO TASK
REMAINING_TASK=0 ?
Yes
No
(PARALLEL END)
SCHEDULING NEXT TASK

Figure 8

processes which we desire. The outline of this procedure could be:

```
Begin
    proc:=1;
    child:=0;
    while (child=0) and (proc<=NoOfProces) do begin
        child:=fork;
        if child=0 then proc:=succ(proc)
    end;
   WaitInitialisationBarrier;
End;
```

The parent process's ID is 0, and the rest processes' ID is 1, 2, ... NoOfProces - 1 respectively.

Comparing with the sequential program , we can see that there are extra work-- create the number of processes, initialise queues and scheduling, etc in the parallel program.

7 The Results

Since the dynamic control part only occupies about 50 procent in the whole program, another 50 procent part is still kept in sequential form, and our focus is on parallel processing the dynamic control part, what we do is letting dynamic control program run large loop number so that the execution time spending on the part of program that is outside dynamic control part can be neglected.

The graph and table below record the results of sequential and each parallel version.

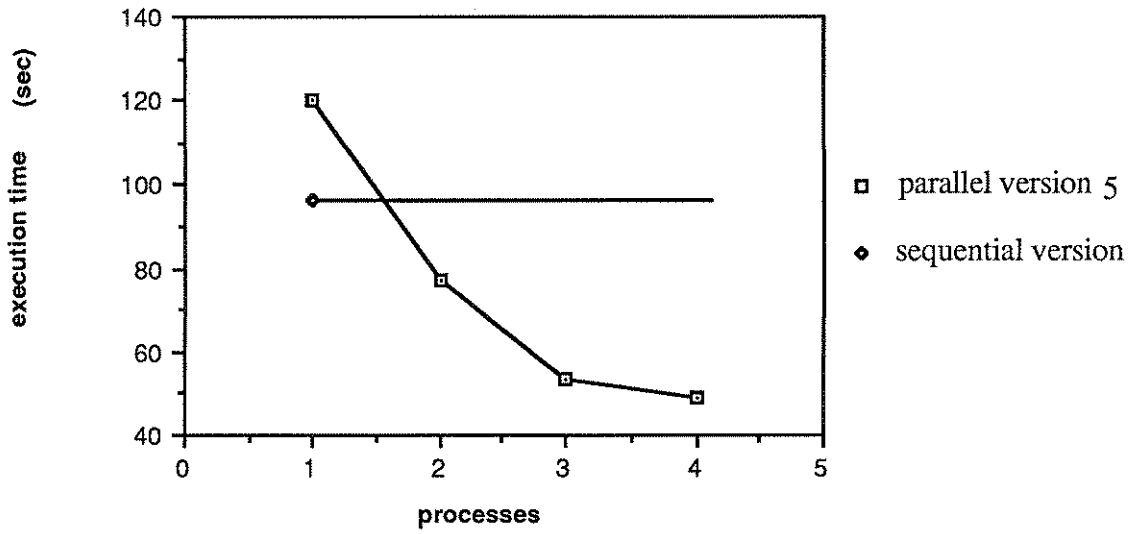

Figure 9

Loop number = 21200

	scheduling type	process number	execution time (second)
sequential version		1	97.7 + 0.4
parallel version 1	dynamic	1	298.1 + 0.8
		2	254.8 + 0.7
		3	287.8 + 0.7
		4	368.2 + 0.7
	static	1	297.1 + 0.7
		2	255.8 + 0.7
		3	268.5 + 0.9
		4	313.3 + 1.3
parallel version 2	dynamic	1	274.5 + 0.6
		2	177.9 + 0.5
		3	161.6 + 0.6
		4	180.1 + 0.7
	static	1	252.6 + 1.7
		2	180.0 + 0.7
		3	156.5 + 0.7
		4	155.9 + 0.5
parallel version 3	dynamic	1	192.8 +0.4
		2	130.4 + 0.4
		3	115.4 + 0.4
		4	122.7 + 0.3
	static	1	178.4 + 0.6
		2	124.4 +0.6
		3	106.4 +0.4
		4	112.3 +0.5
parallel version 4	dynamic	1	183.3 + 0.5
		2	125.9 + 0.5
		3	113.7 + 0.4
		4	118.0 + 0.3
	static	1	161.7 + 0.3
		2	99.6 + 0.3
		3	82.7 + 0.3
		4	80.1 + 0.3
parallel version 5	static	1	123.7 + 0.2
		2	83.9 + 0.2
		3	55.7 + 0.2
		4	50.4 + 0.2

8 Conclusions and Some Proposals

As we see from the results, the execution time of parallel program for robot dynamic control running on one process is much slower than that of the sequential program, and we only get moderate speedup when we run parallel program on three or four processes. There are several factors that cause the performance degradation of parallel processing . One is that the spinlock is too expensive.

In shared_memory multiprocessors system, 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. Some approaches have been represented in a few of papers, one of them is queueing[5]. Unfortunately, queueing has a larger overhead than spinlock when the number of processors is less than 7. For our computing load, the critical path has determined the number of processors is not more than six, e.i., even we use more than six processors, we could not achieve the shorter execution time than the critical path length. This is why we didn't choose queueing to guarantee mutual exclusion.

Some proposals can be considered. They are using data flow model and message passing model. Further discussions will be presented in another technical report.

Achnowledgements

The authors wish to thank the members of the Laboratory for Concurrent Computing Systems at Swinbourne Institute of Technology, especially S.K.Tang and P.S.Chang, for their contribution to the work presented in this technical report.

References

[1] H. Kasahara and S. Narita, "Parallel processing of robot_arm control computation on a multimicroprocessor system", IEEE J. Robotics and Automation, vol RA-1(2), June, 1985.

[2] E. E. Binder and J. H. Herzog, "Distributed computer architecture and fast parallel algorithms in real-time robot control", IEEE Trans. Syst., Man.,Cybern. vol. smc-16 no.6, pp. 543-549, July / August,1986.

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

[4] J. Y. S. Luh and C. S. Lin, "Scheduling of parallel computer for a computer-control mechanical manipulator", IEEE Trans, Syst.,Man, Cybern. vol. 12. pp. 214-234, 1982.

[5] T. E. Anderson, "The performance of spin lock alternatives for shared_memory multiprocesses", IEEE Trans., Parallel and Distributed Systems, vol. 1, pp. 6-16, January, 1990.