Robotics: Science and Systems 2007
Atlanta, GA, USA, June 27-30, 2007

Automatic Scheduling for Parallel Forward Dynamics Computation of Open Kinematic Chains

Katsu Yamane
Department of Mechano-Informatics
University of Tokyo
Email: yamane@ynl.t.u-tokyo.ac.jp

Yoshihiko Nakamura
Department of Mechano-Informatics
University of Tokyo
Email: nakamura@ynl.t.u-tokyo.ac.jp

***Abstract*— Recent progress in the algorithm as well as the processor power have made the dynamics simulation of complex kinematic chains more realistic in various fields such as human motion simulation and molecular dynamics. The computation can be further accelerated by employing parallel processing on multiple processors. In fact, parallel processing environment is becoming more affordable thanks to recent release of multiple-core processors. Although several parallel algorithms for the forward dynamics computation have been proposed in literature, there still remains the problem of automatic scheduling, or load distribution, for handling arbitrary kinematic chains on a given parallel processing environment. In this paper, we propose a method for finding the schedule that minimizes the computation time. We test the method using three human character models with different complexities and show that parallel processing on two processors reduces the computation time by 35–36%.**

I. Introduction

Dynamics simulation of kinematic chains is an essential tool in robotics to test a mechanism or controller before actually conducting hardware experiments. In graphics, such technique can be applied to synthesizing realistic motions of virtual characters and objects. Dynamics simulation of human body models has a number of applications in biomechanics and medical fields. Some algorithms have also been applied to molecular dynamics to estimate the three-dimensional structure of chemical materials such as protein. In spite of the recent progress in algorithms and processor power, realtime simulation of highly complex systems (over 100 links) is still a challenging research issue.

One of the possible ways to further improve the computation speed is to employ parallel processing. In fact, several parallel algorithms have been proposed for the forward dynamics computation of kinematic chains [1]–[4]. These algorithms have $O(N)$ complexity on fixed number of processors and $O(\log N)$ complexity if $O(N)$ processors are available. These algorithms therefore require huge number of processors to fully appreciate the power of parallel computation, and it was considered unrealistic to assume such computational resource for personal use.

The situation has changed by the recent release of multiple-core processors because they would significantly reduce the cost for constructing a parallel processing environment. In particular, the Cell™ processor [5] used in PlayStation3 has 7–8 vector processing cores which function as parallel processors with distributed memory. By optimizing the parallel forward dynamics algorithms for this processor, we would be able to considerably accelerate the computation for dynamics simulation on low-cost hardware.

Another technical barrier towards practical parallel dynamics simulation is finding the optimal scheduling, or load distribution, for a given structure and number of processors. In general, the total amount of floating-point operations increases as the number of parallel processes increases. For example, the total computational cost of a schedule that divides the computation into four processes is greater than that of two processes, and the computation time will not be reduced unless the program runs on more than four processors. Another point to be considered is that waiting time of the processors should be kept minimum to maximize the effect of parallel processing. These facts imply that the optimal schedule depends on the number of available processors as well as the target structure. For a dynamics simulator to be practical for personal use, it should be able to automatically find the optimal schedule.

In this paper, we propose an automatic scheduling method that can find the optimal schedule for any given open kinematic chain and number of available processors. A* search algorithm is applied to obtaining the best schedule. Although the scheduling process only takes place during the initialization or when the link connectivity has changed, we employ several heuristics to find the solution in a reasonable time. Our method is also applicable to fairly wide range of parallel forward dynamics algorithms. Although we have only tested our method on Assembly-Disassembly Algorithm (ADA) proposed by the authors [1], the same method can be applied to Divide-and-Conquer Algorithm (DCA) [2] and Hybrid Direct-Iterative Algorithm (HDIA) [3] with small modifications.

The rest of the paper is organized as follows. We first provide the background information of this work in section II, including a brief summary of the parallel forward dynamics algorithm ADA [1]. Section III presents the automatic scheduling method, which is the main contribution of this paper. The performance of the scheduling method will be demonstrated in section IV, followed by the concluding remarks.

II. Parallel Forward Dynamics Computation

This section provides a brief summary of the parallel forward dynamics algorithm ADA [1], [6]. We first review

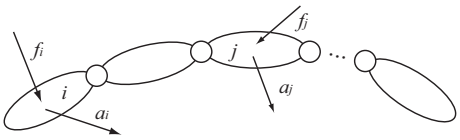

Fig. 1. The concept of Articulated Body Inertia.

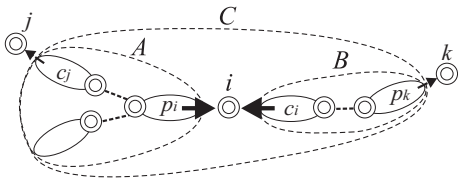

Fig. 2. Connecting partial chains A and B to assemble partial chain C.

the concept of Articulated-Body Inertia and its inverse [7] and then summarize the outline of ADA. After discussing the computational cost of the algorithm, we introduce the concept of *schedule tree* for representing a schedule to easily identify its parallelism and computation time. We also show the results of parallel processing experiments of a simple serial chain and demonstrate the importance of finding the optimal schedule.

A. Articulated-Body Inertia

The concept of Articulated-Body Inertia (ABI) was first proposed by Featherstone [7]. ABI is the apparent inertia of a collection of links connected by joints (articulated body) when a test force is applied to a link (handle) in the articulated body. The equation of motion of a kinematic chain can be written in a compact form by using ABI (see Fig. 1):

$$\boldsymbol{f}_i = \boldsymbol{I}_i^A \boldsymbol{a}_i + \boldsymbol{p}_i^A \tag{1}$$

where $\boldsymbol{I}_i^A$ is the ABI of the articulated body when link i is the handle, $\boldsymbol{f}_i$ is the test force, $\boldsymbol{p}_i^A$ is the bias force, and $\boldsymbol{a}_i$ is the acceleration of link i. Note that the ABI of a kinematic chain may change if a different link is chosen as the handle. All the equations in this paper will be represented by spatial notation [7].

Because ABI is symmetric and positive definite [7], Eq.(1) is equivalent to

$$\boldsymbol{a}_i = \boldsymbol{\Phi}_i \boldsymbol{f}_i + \boldsymbol{b}_i \tag{2}$$

where $\boldsymbol{\Phi}_i$ is the inverse of ABI and called Inverse Articulated Body Inertia (IABI) [7] and $\boldsymbol{b}_i$ is the bias acceleration.

The major advantage of using Eq.(2) instead of Eq.(1) is that we can attach multiple handles to an articulated body. For example, if we attach a new handle to link j in Fig. 1, the accelerations of two handles i and j is written as

$$\begin{aligned} \boldsymbol{a}_i &= \boldsymbol{\Phi}_i \boldsymbol{f}_i + \boldsymbol{\Phi}_{ij} \boldsymbol{f}_j + \boldsymbol{b}_i \\ \boldsymbol{a}_j &= \boldsymbol{\Phi}_{ji} \boldsymbol{f}_i + \boldsymbol{\Phi}_j \boldsymbol{f}_j + \boldsymbol{b}_j \end{aligned}$$

where $\boldsymbol{\Phi}_{ij}$ and $\boldsymbol{\Phi}_{ji} (= \boldsymbol{\Phi}_{ij}^T)$ are the IABIs representing the coupling between the handles.

B. Outline of ADA

ADA computes the joint acceleration by the following two steps:

1) assembly: starting from the individual links, recursively add a joint one by one and compute the IABI of the partial chains, and
2) disassembly: starting from the full chain, remove a joint one by one in the reverse order of step 1) and compute the joint acceleration of each removed joint.

Consider the case of Fig. 2, where partial chains A and B are connected to form chain C through joint i. Partial chain C will be connected to other partial chains through joints j and k in the subsequent assembly operations. In general, each partial chain can have any number of such joints and we shall denote the set by $\mathcal{O}_i$ and the size of $\mathcal{O}_i$ by N_{Oi}. $\mathcal{O}_i$ is empty for the joint lastly added in step 1). In the example of Fig. 2, $\mathcal{O}_i = \{j\ k\}$ and $N_{Oi} = 2$.

In step 1), assuming that we know the IABI of chains A and B, $\boldsymbol{\Phi}_{A**}$ and $\boldsymbol{\Phi}_{B**}$ respectively, as well as the bias accelerations $\boldsymbol{b}_{A*}$ and $\boldsymbol{b}_{B*}$, we compute the IABIs and bias accelerations of chain C by the following equations:

$$\begin{aligned} \boldsymbol{\Phi}_{Cj} &= \boldsymbol{\Phi}_{Aj} - \boldsymbol{\Phi}_{Aji} \boldsymbol{W}_i \boldsymbol{\Phi}_{Aij} && (3) \\ \boldsymbol{\Phi}_{Ck} &= \boldsymbol{\Phi}_{Bk} - \boldsymbol{\Phi}_{Bki} \boldsymbol{W}_i \boldsymbol{\Phi}_{Bik} && (4) \\ \boldsymbol{\Phi}_{Cjk} &= \boldsymbol{\Phi}_{Ckj}^T = \boldsymbol{\Phi}_{Aji} \boldsymbol{W}_i \boldsymbol{\Phi}_{Bjk} && (5) \\ \boldsymbol{b}_{Cj} &= \boldsymbol{b}_{Aj} - \boldsymbol{\Phi}_{Aji} \boldsymbol{\gamma}_i && (6) \\ \boldsymbol{b}_{Ck} &= \boldsymbol{b}_{Bk} + \boldsymbol{\Phi}_{Bki} \boldsymbol{\gamma}_i && (7) \end{aligned}$$

where

$$\begin{aligned} \boldsymbol{W}_i &= \boldsymbol{N}_i^T \boldsymbol{V}_i \boldsymbol{N}_i \\ \boldsymbol{\gamma}_i &= \boldsymbol{W}_i \boldsymbol{\beta}_i + \boldsymbol{R}_i^T \boldsymbol{\tau}_i. \end{aligned}$$

$\boldsymbol{N}_i \in \boldsymbol{R}^{6 \times (6-n_i)}$ and $\boldsymbol{R}_i \in \boldsymbol{R}^{6 \times n_i}$ are the matrices that represent the constraint and motion space of joint i respectively [8], $\boldsymbol{\tau}_i \in \boldsymbol{R}^{n_i}$ is the joint torque of joint i, n_i is the degrees of freedom of joint i, and

$$\begin{aligned} \boldsymbol{V}_i &= (\boldsymbol{N}_i (\boldsymbol{\Phi}_{Ai} + \boldsymbol{\Phi}_{Bi}) \boldsymbol{N}_i^T)^{-1} \\ \boldsymbol{\beta}_i &= \boldsymbol{b}_{Ai} - \boldsymbol{b}_{Bi} - (\boldsymbol{\Phi}_{Ai} + \boldsymbol{\Phi}_{Bi}) \boldsymbol{R}_i^T \boldsymbol{\tau}_i \end{aligned}$$

where we omit the terms including the time derivatives of $\boldsymbol{N}_i$ and $\boldsymbol{R}_i$ for clarity of representation. The initial values for IABI are the spatial inertia matrices of the rigid bodies.

In step 2), assuming that we know the forces applied joints j and k, we can compute the constraint force at joint i by

$$\boldsymbol{n}_i = \boldsymbol{V}_i \boldsymbol{N}_i (\boldsymbol{\Phi}_{Aij} \boldsymbol{f}_j - \boldsymbol{\Phi}_{Bik} \boldsymbol{f}_k + \boldsymbol{\beta}_i) \tag{8}$$

and then compute the joint acceleration by

$$\ddot{\boldsymbol{q}}_i = \boldsymbol{R}_i ((\boldsymbol{\Phi}_{Ai} + \boldsymbol{\Phi}_{Bi}) \boldsymbol{N}_i^T \boldsymbol{n}_i - \boldsymbol{\Phi}_{Aij} \boldsymbol{f}_j + \boldsymbol{\Phi}_{Bik} \boldsymbol{f}_k - \boldsymbol{\beta}_i). \tag{9}$$

Finally $\boldsymbol{f}_i$, the total joint force at joint i which will be used in the subsequent computations, is computed by

$$\boldsymbol{f}_i = \boldsymbol{N}_i^T \boldsymbol{n}_i + \boldsymbol{R}_i^T \boldsymbol{\tau}_i. \tag{10}$$

The total computational cost for assembling and disassembling joint i in Fig. 2 depends on two factors: (1) n_i, the

degrees of freedom of joint i, and (2) N_{Oi}, number of joints in $\mathcal{O}_i$. The cost can be approximated by the following formulae:

$$C_i(N_{Oi}, n_i) = \alpha N_{Oi}^2 + \beta N_{Oi} + \gamma n_i + \delta \qquad (11)$$

where the first term represents the cost for computing the N_{Oi}^2 IABIs, the second term for computing the N_{Oi} bias accelerations and one joint acceleration from N_{Oi} external forces, the third term for computing $\boldsymbol{W}_i$, the last term for computing the inertia matrices of the individual links, and α, β, γ and δ are constants. Because only N_{Oi} can be modified by changing the schedule, Eq.(11) indicates that smaller N_{Oi} would lead to less total floating-point operations.

The constants could be determined by counting the number of floating-point operations as a function of N_{Oi} and n_i; however, we chose to determine them by actually measuring the computation time for various mechanisms and schedules. The advantage of this method is that it can take into account the implementation-specific optimizations and that it would be possible to write a code to automatically determine the values for other forward dynamics algorithms.

C. Schedule Tree

ADA can be executed on multiple processes in parallel because of the following reasons:

- in the example shown in Fig. 2, the computations of the IABIs of partial chains A and B can be performed in parallel, and
- similarly, the accelerations of the joints included in partial chains A and B can be computed in parallel.

In addition, ADA allows any sequence of joints in step 1), and the parallelism and computational efficiency depends on the sequence. Scheduling can therefore be regarded as the problem of finding the best sequence of joints for step 1). However, it is difficult to identify the number of processes required to perform the schedule by looking only at the sequence.

We propose to represent a schedule by a binary tree where each node denotes a joint and every joint is included once. We refer to this tree as *schedule tree*. Figure 3 shows the three schedule trees derived from different assembly sequences (a)–(c) for a simple four-joint serial chain shown on the top of Fig. 3. The concept is similar to *assembly tree* [2] where each node of the tree represents a partial chain, while in our schedule tree it represents a joint to clarify the relationship between the joints and processors.

In a schedule tree, each node has zero to two direct descendants. If a node has two descendants as node 2 in schedules (b) and (c) of Fig. 3, the joint connects two partial chains composed of all the subsequent descendants of each direct descendant. A node with one descendant connects a rigid link and a partial chain, and a node without a descendant connects two rigid links. The recursive process to compute the IABIs starts from the leaves towards the root, while the process to compute the joint accelerations propagates from the root towards the leaves.

A schedule tree is useful for identifying which joints can be processed in parallel and estimating the total computation

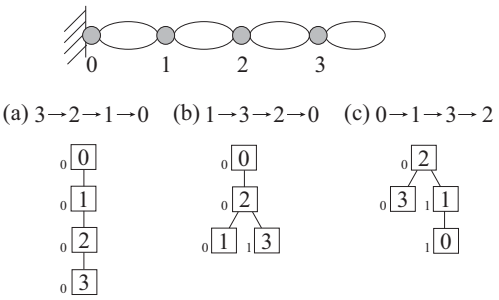

Fig. 3. Three schedule tree examples for different assembly orders of the four-joint serial chain on the top.

time. The two descendants of a node in a schedule tree can obviously be processed in parallel. For example, the numbers at the lower-left corner of each node in Fig. 3 indicates the process in which the node is handled when two processors numbered 0 and 1 are available. If the number of available processors is greater than the number of the leaves of the tree, the height of the schedule tree gives a rough estimate of the computation time. For example, the computation time for schedule (a) in Fig. 3 would be four times longer than that of processing a single joint, while those for schedules (b) and (c) would be approximately equivalent to processing three joints if more than two processors are available.

D. Parallel Processing Experiments

We implemented ADA for parallel processing using C++ programming language and MPICH2 [9] for inter-process communication. The codes were compiled by gcc version 4.0.2. All the examples, including the ones in section IV, were executed on a cluster consisting of dual Xeon 3.8GHz servers running Linux operating system.

Figure 4 shows the performance of parallel forward dynamics computation of a 200-joint serial chain with fixed root link. The solid line represents the computation times on two processors with various schedules. The schedule trees were constructed manually by first choosing a joint as the root, whose index is indicated by the horizontal axis of Fig. 4, and then sequentially appending the joints of each of the partial chains divided by the chosen joint, the joints next to the root being the direct descendants. The assembly process starts by assembling the joints at the both ends in parallel, and ends at the root of the schedule tree. Schedule (c) in Fig. 3 is an example of such schedule for the four-joint chain with root index 2.

The horizontal dashed line in Fig. 4 represents the computation time on single processor when the links were assembled sequentially from the end link to the root, which results in the minimum number of floating-point operation and therefore is the best schedule for serial computation.

As intuitively expected, parallel computation shows the best performance when the computational load is equally distributed to the two processors, which reduces the computation time by 33%. However, the performance degrades if inappro-

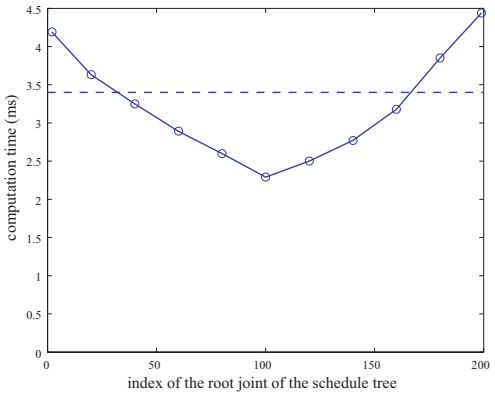

Fig. 4. Computation time of the forward dynamics of a 200-joint serial chain on two processors with various schedules. The dashed horizontal line represents the computation time on single processor.

priate schedule is used and becomes even worse than serial computation due to the communication cost. This experiment demonstrates the importance of selecting appropriate schedule to fully extract the advantage of parallel processing. We also need an algorithm for automatically finding the optimal schedule for more general cases, e.g. for branched chains or when more processors are available, because the optimal schedule is not trivial any more.

III. Automatic Scheduling

A. Overview

The purpose of the automatic scheduling process is to automatically find the best schedule, or the best assembly order, that minimizes the computation time for the given mechanism and number of available processes. As described in the previous section, the forward dynamics algorithms we consider in this paper allow any assembly order. The number of all possible orders therefore becomes as large as $N!$ for a mechanism with N joints. We employ several heuristics to avoid the search in such a huge space.

The first observation is that different assembly orders may lead to the same schedule tree. This fact can be easily illustrated by using the examples shown in Fig. 5 for the same serial chain as in Fig. 3. In Fig. 5, the two schedules (a) and (b) result in the conceptually equivalent schedule trees. The order of adding joints 1 and 3 obviously does not affect the schedule tree because the resulting partial chains are independent of each other. We can reduce the search space by eliminating such duplicated assembly orders.

The second observation is that, as described in section III-D, the best schedule can be determined without further running the search process if the number of processes assigned to a partial chain becomes one. This fact implies that the cost for the search depends more on the number of processors rather than the number of joints. We can considerably reduce the search time because practical parallel processing environments usually have far less processors than the number of joints.

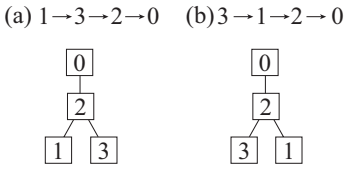

Fig. 5. Different assembly orders resulting in the same schedule tree.

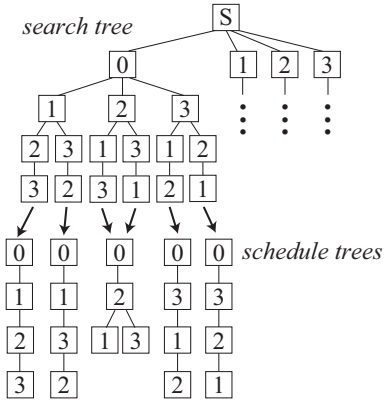

Fig. 6. Search and schedule trees.

B. Search Algorithm

The problem of finding the optimal schedule can be viewed as a travelling salesman problem where the joints are the cities to visit and the cost function is the time to cover all cities. The difference in multiple-processor case is that more than one salesmen are working in parallel and each city is visited by only one of them. We also have the constraint that a city should be visited later than some other cities due to the dependencies between the IABI of partial chains.

We apply A* search [10] to our problem of finding the optimal schedule. The general form of A* search is summarized in the Appendix. The following three functions for a node x have to be customized for our problem:

- $x.NextNodes()$: returns the set of descendant nodes of x
- $x.Cost()$: returns the cost to add x to the search tree
- $x.AstarCost()$: returns an underestimation of the cost from x to a goal

One point to note here is the relationship between the schedule tree and the tree constructed during the search (called the *search tree* hereafter). Each path from the root node to a leaf node of a search tree is associated with a schedule tree. We depict an example of the search tree and associated schedule trees in Fig. 6 for the simple serial chain in Fig. 3, where the node marked "S" is a dummy start node.

We describe the three functions customized for our problem in the rest of this subsection.

1) NextNodes(): A naive way to implement this function is to add all unvisited joints as the descendants. However, this approach may yield duplicated schedules as observed in the

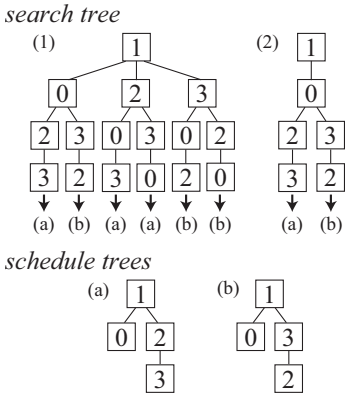

Fig. 7. Search tree by naive version of *NextNodes()* (1) and improved version (2).

previous subsection.

In order to reduce the number of nodes in the search tree, we select the descendants such that the corresponding schedule tree can be effectively extended. When we try to extend the search tree by adding descendants to node i associated with joint i, we construct the (incomplete) schedule tree derived from the joint sequence up to node i, and look for nodes which have only 0–1 descendants. Let us denote the index of such joint in the highest layer of the schedule tree by j. Cutting joint j generates two partial chains: one on the root side and the other on the end link side. If joint j has no descendants, the joints on the root side will be added as direct descendants of joint i in the search tree, which will eventually add the first descendant of joint j in the schedule tree. If joint j has one descendant, which means that the joints on the root side have already been added, we add the joints on the end link side.

Figure 7 shows the comparison between the naive version of *NextNode()* and our improved version, using the part of the search tree in Fig. 6 under node 1 directly below the start node. Although the naive version generates the large search tree (1), each of them results in one of the schedule trees (a) and (b). Using the improved version (2), on the other hand, the tree only includes necessary nodes.

The search tree (2) is constructed as follows. Suppose we are extending the search tree by adding descendant(s) to node 1. Because the corresponding schedule tree also has only one node associated with joint 1, we try to add the descendants to this node. Joint 1 divides the chain into two partial chains composed of the joints $\{0\}$ (towards the root) and $\{2, 3\}$ (towards the end). We therefore add joint 0 as the descendant of node 1 of the search tree. In the next step, because node 1 in the schedule tree has only one descendant, we try to find the second, which should be either joint 2 or 3. Two branches are therefore added to node 0 in the search tree.

2) Cost(): The cost of a node is defined as the total computation time increased by adding the joint. At each node in the search tree, we maintain the total active time of each process. The computation time for processing a node is added to the total active times of all the processes assigned to the node. The actual computation time is obtained by looking for the maximum active time among the processes. The cost of a node is then obtained by subtracting the total cost of its ascendant from its own total cost.

3) AstarCost(): We first compute the minimum costs to assemble the two partial chains generated by cutting the corresponding partial chain at the joint, using the method described in section III-D. We then take the larger cost and divide it by the number of processors available at the node. This is guaranteed to be an underestimate because a schedule for multiple processors results in more total floating-point operations than the one for single processor.

C. Assigning Processes

Once we have a schedule tree, we then assign the processes to each node. The point to be considered here is that the time for communication between the processes (in shared-memory environments, the time for waiting for the permission to access the memory) should be kept minimum. Strictly speaking, the amount of data to be passed also affects the communication time and should be considered in the scheduling process. However, we ignore the data size because it is usually small (288 bytes for each IABI) and the delay for invoking the communication is a more serious issue in most communication devices.

Let us denote the number of all available processes by N_P, which are numbered $0 \ldots N_P - 1$. Here we assume that N_P is a power of 2. The basic approach is to assign a group of processes to a node in the schedule tree, and divide the group into two if the node has two descendants. We denote the range of processes assigned to node i of the schedule tree by $[a_{Pi}, b_{Pi})$ which means that processes $a_{Pi}, a_{Pi}+1, \ldots b_{Pi}-1$ are assigned to node i. The actual computations for assembling and disassembling joint i take place at process a_{Pi}.

The procedure for assigning processes to the nodes is described as follows:

1) Initialize $a_{Pr} = 0$ and $b_{Pr} = N_P$ where r is the index of the root node.
2) At node i
   a) If $b_{Pi} - a_{Pi} = 1$, or if node i has one descendant, set the same range for the descendant(s).
   b) If node i has two descendants m and n, and $b_{Pi} - a_{Pi} > 1$, set $a_{Pm} = a_{Pi}, b_{Pm} = a_{Pn} = b_{Pi} + (b_{Pi} - a_{Pi})/2, b_{Pn} = b_{Pi}$.
3) Recursively call step 2) for all descendants.

The communication cost is minimized by ensuring that one of the descendants is handled at the same process, therefore requiring no communication.

D. Optimal Schedule for Single Process

The optimal schedule for single process is the one that minimizes the number of floating-point operations. As Eq.(11) and the subsequent discussion imply, we have to keep N_{Oi} at every joint as small as possible. To realize this, for each joint we divide the joints in $\mathcal{O}_i$ into two groups by checking whether they exists towards the root side or end side of the

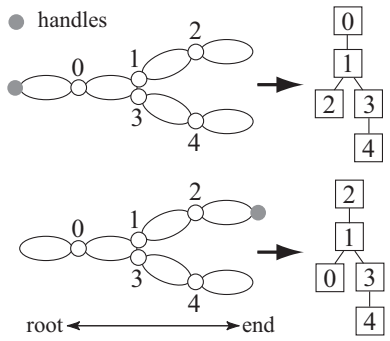

Fig. 8. Examples of partial chains and their optimal schedule for single process, with different $\mathcal{O}_i$.

joint. Let us denote the number of joints in the two groups by N^r_{Oi} and N^e_{Oi} respectively. If $N^r_{Oi} \leq N^e_{Oi}$ we put the joint at the lower layer of the schedule tree, i.e. processed earlier in the assembly step, because we can keep N_{Oi} smaller than in the reverse order. This principle defines the sequence of adding joints where there is no branches. At branches, we count the number of joints included in $\mathcal{O}_i$ in each branch and then process the branches in the ascending order of the number.

Figure 8 shows two examples of optimal schedules for a partial chain composed of joints 0–4, with different $\mathcal{O}_i$. In both cases, $\mathcal{O}_i$ includes one joint represented by the gray circle. The branch neighboring the joint in $\mathcal{O}_i$ should be processed later in the assembly process regardless of the hierarchy in the structure. The joints in that branch are therefore placed near the top of the schedule tree.

IV. Experiments

A. Setup

We used the same computer environment as in section II-D and determined the constants in Eq.(11) as $\alpha = 1.6, \beta = 1.0, \gamma = -1.0$, and $\delta = 14.4$, which gives the total computation time in µs for assembling and disassembling a joint. The negative value for γ is because the computation of $\boldsymbol{V}_i$ involves the inversion of a $(6 - n_i) \times (6 - n_i)$ matrix. The coefficients for n_i^3 and n_i^2 turned out to be too small to be identified by this method, and therefore practically negligible.

B. Serial Chains

We applied the method to finding the optimal schedule for handling serial chains on two- and four-processor environments. In the two-processor case, the optimal schedule is trivial as shown in Fig. 4: the root of the schedule tree is the joint at the middle of the chain, and other joints are added sequentially towards the ends. We therefore only confirm that the method can find the trivial solution. The four-processor case is no longer trivial. Dividing the chain into four equal-length partial chains and assigning one process to each partial chain is not the optimal schedule because the two partial chains in the middle requires more computations than those at the ends because of larger $N_{Oi}(= 2)$.

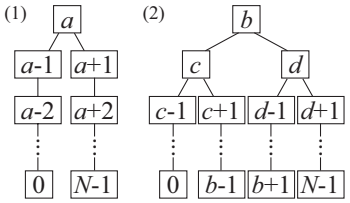

Fig. 9. Optimal schedules for handling N-joint serial chain on two processors (1) and four processors (2).

TABLE I

The values of a, b, c and d for the schedule trees in Fig. 9.

DOF	2 processes	4 processes		
N	a	b	c	d
16	7	7	3	11
32	15	15	8	22
64	31	31	17	45

The scheduling algorithm derived the schedule trees for handling N-joint serial chain on two and four processors shown in Fig. 9, where the actual indices a, b, c and d were the values shown in Table I for $N = 16, 32$ and 64. The schedules for two processors match the trivial solutions. Those for four processors also reasonable because the internal two partial chains are slightly shorter than the others.

C. Branched Chains

We applied the method to finding the optimal schedule for handling branched chains on two processors. The scheduling is not trivial as in the case of serial chains. We used three human figures with different complexities (Fig. 10): 40 DOF, 52 DOF, and 161 DOF composed of 1 DOF rotational, 3 DOF spherical, and 6 DOF free joints. In all models, the method selected a joint in the backbone near the neck as the root of the schedule tree. Table II shows the computation time on one to four processors, and the ratios of speedup. Parallel computation reduced the computation time by 38–43% for two processors and 39–54% for four processors. The speedup gain is comparable to that of Fig. 4 which is the ideal load balance case. The effect of parallel computation was more prominent in complex models.

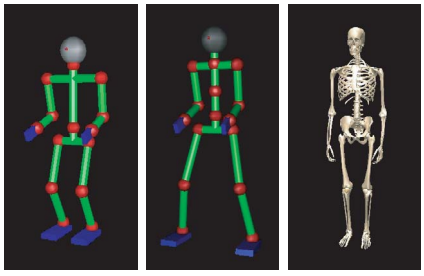

Fig. 10. Three human character models for test; from left to right: 40 DOF, 52 DOF, and 161 DOF.

TABLE II

COMPUTATION TIMES FOR SERIAL AND PARALLEL COMPUTATIONS (MS) AND RATIO OF SPEEDUP.

DOF	40	52	161
♯ of joints	15	19	53
1 process	0.249	0.301	0.773
2 processes (speedup)	0.155 (38%)	0.186 (38%)	0.443 (43%)
4 processes (speedup)	0.153 (39%)	0.177 (41%)	0.356 (54%)

V. CONCLUSION

In this paper, we proposed a method for automatically scheduling the parallel forward dynamics computation of open kinematic chains. The conclusion of the paper is summarized as follows:

1) We proposed an efficient method for applying A* search algorithm to the scheduling problem.
2) We proposed a systematic method for assigning processors to the partial chains considering the communication cost.
3) The method was applied to three human character models with different complexity and reduced the computation time by up to 43% on two processors and 54% on four processors.

The method is applicable to parallel forward dynamics algorithms which can be physically interpreted as successive connection of two partial chains described as schedule trees, e.g. [1]–[3]. The coefficients of Eq.(11) should be modified accordingly.

Future work includes extension to closed kinematic chains. This problem can be partially solved by the following procedure: (1) cut the loops by removing some joints and apply the method described in this paper to the resulting open chain, and (2) insert the removed joints to the root of the schedule tree, although the resulting schedule may not be optimal. Another issue is the relationship between the schedule and numerical accuracy. If the schedule affects the accuracy, it would be possible and beneficial to optimize the accuracy as well in the automatic scheduling process.

ACKNOWLEDGEMENTS

This work was supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT) and the New Energy and Industrial Technology Development Organization (NEDO), Japan.

REFERENCES

[1] K. Yamane and Y. Nakamura, "Efficient Parallel Dynamics Computation of Human Figures," in *Proceedings of the IEEE International Conference on Robotics and Automation*, May 2002, pp. 530–537.

[2] R. Featherstone, "A Divide-and-Conquer Articulated-Body Algorithm for Parallel $O(\log(n))$ Calculation of Rigid-Body Dynamics. Part1: Basic Algorithm," *International Journal of Robotics Research*, vol. 18, no. 9, pp. 867–875, September 1999.

[3] K. Anderson and S. Duan, "Highly Parallelizable Low-Order Dynamics Simulation Algorithm for Multi-Rigid-Body Systems," *AIAA Journal on Guidance, Control and Dynamics*, vol. 23, no. 2, pp. 355–364, March-April 2000.

[4] A. Fijany, I. Sharf, and G. D'Eleuterio, "Parallel $O(\log N)$ Algorithms for Computation of Manipulator Forward Dynamics," *IEEE Transactions on Robotics and Automation*, vol. 11, no. 3, pp. 389–400, 1995.

[5] IBM Research, "The Cell Project at IBM Research," online, http://www.research.ibm.com/cell/.

[6] K. Yamane and Y. Nakamura, "$O(N)$ Forward Dynamics Computation of Open Kinematic Chains Based on the Principle of Virtual Work," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2001, pp. 2824–2831.

[7] R. Featherstone, *Robot Dynamics Algorithm*. Boston, MA: Kluwer Academic Publishers, 1987.

[8] K. Yamane and Y. Nakamura, "Parallel $O(\log N)$ Algorithm for Dynamics Simulation of Humanoid Robots," in *Proceedings of IEEE-RAS International Conference on Humanoid Robotics*, Genoa, Italy, December 2006, pp. 554–559.

[9] Mathematics and Computer Science Division, Argonne National Laboratory, "MPICH2 Home Page," online, http://www.mcs.anl.gov/mpi/mpich2/.

[10] Steven M. LaValle, *Planning Algorithms*. New York, NY: Cambridge University Press, 2006.

APPENDIX

Algorithm 1 shows the general form of A^* search [10], where T is the search tree, Q is a list of nodes sorted in the ascending order of $priority\_cost$ of each node. The following operations are predefined for list Q and search tree T:

- $Q.GetFirst()$: extract the first node in Q
- $Q.Insert(x)$: insert node x to Q such that the nodes are aligned in the ascending order of $x.priority\_cost$
- $T.SetRoot(x)$: set node x as the root of T
- $T.AddDescendant(x, x')$: add x' to T as a descendant of x

while the actions of the following methods for node x should be customized to fit the particular problem:

- $x.NextNodes()$: returns the set of descendant nodes of x
- $x.Cost()$: returns the cost to add x to the search tree
- $x.AstarCost()$: returns an underestimation of the cost from x to a goal

Algorithm 1 General A* Search

Require: the initial node x_I and set of goal nodes X_G

$x_I.total\_cost \leftarrow 0$
$x_I.priority\_cost \leftarrow 0$
$Q.Insert(x_I)$
$T.SetRoot(x_I)$
while Q not empty **do**
  $x \leftarrow Q.GetFirst()$
  if $x \in X_G$ **then**
    return x
  end if
  for all $x' \in x.NextNodes()$ **do**
    $T.AddDescendant(x, x')$
    $x'.total\_cost \leftarrow x.total\_cost + x'.Cost()$
    $x'.priority\_cost \leftarrow x'.total\_cost + x'.AstarCost()$
    $Q.Insert(x')$
  end for
end while
return NULL