DOE/ER/25026--31

DE89 014169

REPORT NO. UIUCDCS-R-89-1525

ASYNCHRONOUS INTEGRATION OF ORDINARY DIFFERENTIAL EQUATIONS ON MULTIPROCESSORS

by

Sohail Aslam
C. W. Gear

July 1989

DEPARTMENT OF COMPUTER SCIENCE
1304 W. SPRINGFIELD AVENUE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, IL 61801

Supported in part by the U.S. Department of Energy under grant DOE DEFG02-87ER25026.

Asynchronous Integration of Ordinary Differential Equations on Multiprocessors

Sohail Aslam C. W. Gear

June 28, 1989

1 Introduction

Asynchronous iteration methods have been shown to offer significant speedups in multiprocessing environments for the class of iterative methods

$$\mathbf{x} = \mathbf{F}(\mathbf{x}) \tag{1}$$

For a system of equations, parallel implementation on a multiprocessor with no or minimal synchronization between cooperating processes yields the most advantage [1]. Picard or Picard-like iteration schemes fall into such a class of iterative methods. We present a preliminary report on using Picard-like integration methods for solving Initial Value Problems (IVP) from amongst the class of Ordinary Differential Equations (ODE).

2 The Class of Asynchronous Iterative Methods

We quote most of the material in this section from [1]. If $\mathbf{x}$ is a vector of $\mathbf{R}^n$, its components will be denoted by $x_i, i, 1, \ldots, n$. To avoid confusion, a sequence of vectors in $\mathbf{R}^n$ will be denoted ny $\mathbf{x}(j), j = 1, \ldots$. If $\mathbf{F}$ is an operator of $\mathbf{R}^n$ into itself, its components will be represented by $F_i(\mathbf{x})$ or by $F_i(x_1, \ldots, x_n), i = 1, \ldots, n$. Let $\mathbf{N}$ denote the set of all nonnegative integers.

Definition 1. Let $\mathbf{F}$ be an operator from $\mathbf{R}^n$ to $\mathbf{R}^n$. An *asynchronous iteration* corresponding to the operator $\mathbf{F}$ and starting with a given vector $\mathbf{x}(0)$ is a sequence of vectors $\mathbf{x}(j)$ of $\mathbf{R}^n$ defined recursively by

$$x_i(j) = \begin{cases} x_i(j-1) & \text{if } i \notin J_j, \\ F_i(x_1(s_1(j)), \ldots, x_n(s_n(j))) & \text{if } i \in J_j. \end{cases} \tag{2}$$

Define $\mathcal{J} = \{J_j | j = 1, 2, \ldots\}$ where each J_j is a non-empty subset of $\{1, \ldots, n\}$. It represents the indices of those components of $\mathbf{x}(j)$ that are updated in the j^{th} iterate. Define $\mathcal{S} = \{(s_1(j), \ldots, s_n(j)) | j = 1, 2, \ldots\}$ which a sequence of elements of $\mathbf{N}$. In addition, let $\mathcal{J}$ and $\mathcal{S}$ satisfy the following conditions for each $i = 1, \ldots, n$:

1. $s_i(j) \leq j - 1, \ j = 1, 2, \ldots;$

2. $s_i(j)$ considered a function of j tends to infinity as j tends to infinity;

3. i occurs infinitely many often in sets $J_j, \ j = 1, 2, \ldots$.

The first condition imposes the requirement that only components of previous iterates can be used in computing the new iterate. The second condition states that the values of an early iterate are not directly used in iterations far in the future; eventually, newer iterates have to be used instead. This condition is necessary to guard against the situation in which, for example, a processor crashes. The third condition guarantees that no component is abandoned forever. An asynchronous iterative scheme corresponding to $\mathbf{F}$, starting with the vector $\mathbf{x}(0)$ is denoted by $(\mathbf{F}, \mathbf{x}(0), \mathcal{J}, \mathcal{S})$. $\square$

Definition 2. An operator $\mathbf{F}$ is a *Lipchitzian operator* on a subset $\mathbf{D}$ of $\mathbf{R}^n$ if there exists a nonnegative $n \times n$ matrix A such that

$$|\mathbf{F}(\mathbf{x}) - \mathbf{F}(\mathbf{y})| \leq A|\mathbf{x} - \mathbf{y}|, \quad \forall \mathbf{x}, \mathbf{y} \in \mathbf{D}, \tag{3}$$

where, if $\mathbf{z}$ is a vector of $\mathbf{R}^n$ with components $z_i, \ i = 1, \ldots, n$, $|\mathbf{z}|$ denotes the vector with components $|z_i|, i = 1, \ldots, n$, and the inequality holds for every component. The matrix A will be termed a *Lipchitzian matrix* for the operator $\mathbf{F}$. $\square$

Definition 3. An operator $\mathbf{F}$ from $\mathbf{R}^n$ to $\mathbf{R}^n$ is a *contracting operator* on a subset $\mathbf{D}$ of $\mathbf{R}^n$ if it is a Lipchitzian operator on $\mathbf{D}$ with a Lipchitzian matrix A such that $\rho(A) < 1$ (where $\rho(A)$ is the spectral radius of A). □

Given these definitions, the following theorem establishes sufficient condition for ensuring the convergence of any asynchronous iteration corresponding to $\mathbf{F}$.

Theorem 1 *If* $\mathbf{F}$ *is a contracting operator on a closed subset* $\mathbf{D}$ *of* $\mathbf{R}^n$ *and if* $\mathbf{F}(\mathbf{D}) \subset \mathbf{D}$, *then any asynchronous iteration* $(\mathbf{F}, \mathbf{x}(0), \mathcal{J}, \mathcal{S})$ *corresponding to* $\mathbf{F}$ *and starting with a vector* $\mathbf{x}(0)$ *converges to the unique fixed point of* $\mathbf{F}$ *in* $\mathbf{D}$.

3 The Picard Method

Consider the Initial Value Problem (IVP)

$$\dot{y} = f(t, y), \qquad y(t_0) = y_0 \tag{4}$$

where $\dot{y} = dy/dt$ We can integrate the two sides with respect to t to obtain

$$y(t) = y_0 + \int_{t_0}^{t} f(s, y(s))ds. \tag{5}$$

We can employ the Picard Method to solve (5) by computing

$$y^n(t) = y_0 + \int_{t_0}^{t} f(s, y^{n-1}(s))ds. \tag{6}$$

a sequence of successive approximations $y^n(t)$ to the exact solution $y(t)$. [We are now using superscripts to refer to the iterate and parenthesised t to indicate y at a certain value of t.] The sequence has to start somewhere; the best guess is

$$y^0(t) = y(t_0).$$

Clearly if $y^n = y^{n-1}$, then $y = y^n = y^{n-1}$ solves (5) and hence (4). It can be shown that these iterates always converge, on a suitable interval, to the true solution $y(t)$ of the Initial Value Problem (4). The proof can be found in most books on Solution of Ordinary Differential Equations (ODEs). We give a brief example of the procedure for instructive purpose only. Consider the IVP

$$\dot{y} = y, \qquad y(0) = 1,$$

where $\dot{y}$ denotes dy/dt. The true solution is $y = e^t$. Picard iteration proceeds as follows: rewrite the IVP in the equivalent integral form:

$$\begin{aligned} y(t) &= y_0 + \int_{t_0}^{t} f(s, y(s))ds \\ &= 1 + \int_0^t y(s)ds \end{aligned}$$

Thus,

$$\begin{aligned} y^0(t) &= 1, \\ y^1(t) &= 1 + \int_0^t 1ds = 1 + t \\ y^2(t) &= 1 + \int_0^t y^1(s)ds = 1 + \int_0^t (1+s)ds = 1 + t + \frac{t^2}{2!} \end{aligned}$$

and in general

$$\begin{aligned} y^n(t) &= 1 + \int_0^t y^{n-1}(s)ds \\ &= 1 + t + \frac{t^2}{2!} + \ \ldots \ + \frac{t^n}{n!} \end{aligned}$$

which is the Taylor series expansion of e^t; each iteration adds the next term in the Taylor series.

The Picard Method carries over to system of equations; the quantities y, $f(t, y)$ and y_0 are vectors in this case and the integration is performed component-wise.

4 Asynchronous Generalized Picard iteration

Suppose we have a system of N ODEs:

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}), \qquad \mathbf{y}(t_0) = \mathbf{y}_0 \tag{7}$$

where $\mathbf{y}$ is composed of elements $(y_1, y_2, \ldots, y_N)$. Similarly the vector $\mathbf{f}$ consists of elements $(f_1, f_2, \ldots, f_N)$. The vector $\mathbf{y}_0$ contains the initial values of the N components $\mathbf{y}(t_0)$. An asynchronous iteration proceeds along the following sequence of computations on a multiprocessor. Assume that we have a pool of processors available. Let τ_j, $j = 1, \ldots$, be an increasing sequence of clock times. At some value of τ_j a processor P becomes available. This idle processor is assigned the evaluation of y_j^n. P proceeds to integrate the j^{th} component

$$y_j^n(t) = y_j(t_0) + \int_{t_0}^{t} f_j(\zeta, y_1^{s_1(n)}(\zeta), \ldots, y_j^n(\zeta), \ldots, y_N^{s_N(n)}(\zeta))d\zeta. \tag{8}$$

The integration of the ODE from t_0 to t requires evaluating f_j at various ζ values along the integration interval. The component y_j will most likely have coupling with other components in the system of ODEs. Evaluation of f_j will require the values of all such components at any given ζ. A natural criterion will be to pick up the most recently available values of all such components. In our proposed method however, a component does not make its most recent values available until it has finished integration. Thus the notation used in (8) is $y_i^{s_i(n)}(\zeta)$, for all y_i that y_j depends upon. According to condition 1 of the definition 1 of asynchronous iteration method, $s_i(n) \leq n - 1$. This indicates that y_j will use the most recent values of y_i if they are available, otherwise it will proceed to use earlier values. Note that the current value of y_j (y_j^n) is used. This component is readily available to the processor integrating (8) because it is computing it.

This scheme does not require any synchronization amongst the processors integrating various components of the system of ODEs. At some time τ_k, $k > j$, processor P will finish integrating component y_j and will be assigned another component y_k. Note that unlike the traditional Picard method, the processor P does not carry out further iterations after reaching the end of the integration interval. The processor is relinquished after having computed y_j^n. Later, when any y_k on which y_j depends upon, computes its new iterate and makes it available, component y_j will have to compute yet another iterate.

This scheme is not restricted to any particular multiprocessor architecture. Shared memory multiprocessors offer an advantage when it comes to movement of data but even in a network of processors, e.g., hypercubes or distributed network of computers, the scheme is well suited as long as the cost of communication amongst the elements of the network is not too expensive compared to the cost of computation itself.

5 The Integration Procedure

The method allows for various choices to be made for the integration procedure employed for various components of the system of ODEs. The system of ODEs can be split into a set of sub-systems. For a particular sub-system, a one-step method may be selected while for another, a multi-step integration method may be chosen. The method selected can be fixed step size or variable step size. For the variable step size case, different local error tolerances can be specified for each sub-system to control step size. The integration method maybe explicit for one sub-system and implicit for another.

In a variable step size numerical integration procedure, the step size is partially controlled by stability constraints. For example, for the model problem $\dot{y} = \lambda y$, the stability condition for *forward Euler* method is $|1 + h\lambda| \leq 1$. If an implicit method or a predictor-corrector scheme, additional conditions have to be placed on step size to ensure convergence of asynchronous integration. We illustrate the need for such constraint with the following discussion. Consider the following system of ODEs:

$$\begin{aligned}\dot{y} &= \lambda y + \mu z,\\ \dot{z} &= \mu y + \lambda z.\end{aligned}$$

where $\dot{y}$ and $\dot{z}$ represent dy/dt and dz/dt respectively. Assume that $y_0 = z_0 = 0$. Suppose that we use *backward Euler* to compute the m^{th} iterate for y; we'll thus compute a series of y values using the recurrence relation

$$y_n^m = y_{n-1}^m + h_n \lambda y_n^m + h_n \mu z_n^{m-1}.$$

where h_n is the step size such that $t_n = t_{n-1} + h_n$. The superscript m in y_n^m refers to the m^{th} iterate while the subscript n refers to the value of y at t_n. We assume that the most recent z iterate, z^{m-1}, is available. Define a shift operator E as $y_n = Ey_{n-1}$, thus $y_{n-1} = E^{-1}y_n$. We can rewrite the recurrence relation in terms of this shift operator as

$$y_n^m = E^{-1}y_n^m + h_n \lambda y_n^m + h_n \mu z_n^{m-1},$$

or

$$[1 - h\lambda - E^{-1}]y_n^m = h\mu z_n^{m-1}.$$

The iterate y^m will emerge as a vector of y values computed at various values of t. We can represent the computation in matrix-vector form:

$$\begin{pmatrix} (1-\lambda h_1) & & & \\ -1 & (1-\lambda h_2) & & \\ & \ddots & \ddots & \\ & & -1 & (1-\lambda h_n) \end{pmatrix} \begin{pmatrix} y_1^m \\ y_2^m \\ \vdots \\ y_n^m \end{pmatrix} = \begin{pmatrix} \mu h_1 & & & \\ & \mu h_2 & & \\ & & \ddots & \\ & & & \mu h_n \end{pmatrix} \begin{pmatrix} z_1^{m-1} \\ z_2^{m-1} \\ \vdots \\ z_n^{m-1} \end{pmatrix},$$

or

$$\begin{pmatrix} y_1^m \\ y_2^m \\ \vdots \\ y_n^m \end{pmatrix} = \begin{pmatrix} (1-\lambda h_1) & & & \\ -1 & (1-\lambda h_2) & & \\ & \ddots & \ddots & \\ & & -1 & (1-\lambda h_n) \end{pmatrix}^{-1} \begin{pmatrix} \mu h_1 & & & \\ & \mu h_2 & & \\ & & \ddots & \\ & & & \mu h_n \end{pmatrix} \begin{pmatrix} z_1^{m-1} \\ z_2^{m-1} \\ \vdots \\ z_n^{m-1} \end{pmatrix}.$$

The A matrix required by the Lipchitzian condition of the iteration operator is then

$$A = \begin{pmatrix} (1-\lambda h_1) & & & \\ -1 & (1-\lambda h_2) & & \\ & \ddots & \ddots & \\ & & -1 & (1-\lambda h_n) \end{pmatrix}^{-1} \begin{pmatrix} \mu h_1 & & & \\ & \mu h_2 & & \\ & & \ddots & \\ & & & \mu h_n \end{pmatrix}.$$

We make the simplifying assumption that $h_1 = \ldots = h_n = h$. Then,

$$\rho(A) = \frac{\mu h}{1 - \lambda h}.$$

In order to satisfying the contracting property, we must enforce

$$\left| \frac{\mu h}{1 - \lambda h} \right| \leq 1.$$

Although backward Euler is an A-stable method, this condition places a restriction on the step size chosen for computing any of y_n^m. We do not yet know of a suitable method to enforce this condition. The condition would most likely be quite complex for a system of ODEs that is not as simple as the one above. If this condition is not satisfied, the asynchronous iteration will diverge.

Any such restriction is not required if an explicit method is chosen. We illustrate this by chosing *forward Euler* to compute the m^{th} iterate for y. The recurrence relation for forward Euler is

$$y_n^m = y_{n-1}^m + h_n\lambda y_{n-1}^m + h_n\mu z_{n-1}^{m-1}.$$

Rewrite in terms of the shift operator E:

$$y_n^m = E^{-1}y_n^m + h_n\lambda E^{-1}y_n^m + h_n\mu E^{-1}z_n^{m-1},$$

or

$$[1 - E^{-1}(1+h\lambda)]y_n^m = h\mu E^{-1}z_n^{m-1}.$$

Computation of the vector y^m can be represented in matrix-vector form:

$$\begin{pmatrix} 1 & & & \\ (-1-\lambda h_2) & 1 & & \\ & \ddots & \ddots & \\ & & (-1-\lambda h_n) & 1 \end{pmatrix} \begin{pmatrix} y_1^m \\ y_2^m \\ \vdots \\ y_n^m \end{pmatrix} = \begin{pmatrix} 0 & & & \\ \mu h_2 & 0 & & \\ & \ddots & \ddots & \\ & & \mu h_n & 0 \end{pmatrix} \begin{pmatrix} z_1^{m-1} \\ z_2^{m-1} \\ \vdots \\ z_n^{m-1} \end{pmatrix}.$$

Or,

$$\begin{pmatrix} y_1^m \\ y_2^m \\ \vdots \\ y_n^m \end{pmatrix} = \begin{pmatrix} 1 & & & \\ (-1-\lambda h_2) & 1 & & \\ & \ddots & \ddots & \\ & & (-1-\lambda h_n) & 1 \end{pmatrix}^{-1} \begin{pmatrix} 0 & & & \\ \mu h_2 & 0 & & \\ & \ddots & \ddots & \\ & & \mu h_n & 0 \end{pmatrix} \begin{pmatrix} z_1^{m-1} \\ z_2^{m-1} \\ \vdots \\ z_n^{m-1} \end{pmatrix}.$$

The A matrix required by the Lipchitzian condition of the operator $\mathbf{F}$ is then

$$A = \begin{pmatrix} 1 & & & \\ (-1-\lambda h_2) & 1 & & \\ & \ddots & \ddots & \\ & & (-1-\lambda h_n) & 1 \end{pmatrix}^{-1} \begin{pmatrix} 0 & & & \\ \mu h_2 & 0 & & \\ & \ddots & \ddots & \\ & & \mu h_n & 0 \end{pmatrix}.$$

Clearly, $\rho(A) = 0$ so that asynchronous iteration doesn't impose any additional requirements on the choice of step size. Only stability requirements will restrict the step size. This is intuitively obvious because for the forward Euler method, only the previous values are used. The direction of flow of information is in the forward direction.

Similarly, for a predictor-corrector method with forward Euler predictor and backward Euler corrector, it can be shown that contracting property requirement will impose the condition $|h\mu| < 1$. For general Backward Difference Formulae methods (BDF), the condition is $|\beta_0 h\mu| < 1$.

For larger and more complex systems of ODEs, things get more complex. For the simple example above, μ associated with component z is known but for more complex coupling and larger system of ODEs, μ will not so readily available. Consider a system of ODEs with a component equation

$$\dot{y} = f(t, y, z_1, \ldots, z_i, \ldots, z_k).$$

Let the $z_1, \ldots z_k$ be elements of vector $\mathbf{z}$. Here μ will be estimated by computing the norm $f_{\mathbf{z}}$. More than often it would not be possible to evaluate $||f_{\mathbf{z}}||$ analytically; it would have to be evaluated numerically.

6 The Asynchronous Integration Algorithm

The components of the system of ODEs are split into sub-systems. Possible splitting can range from one component per sub-system to only one sub-system with all of the components. Let $\mathcal{Y}_i$ denote the sub-system i. Its members will be components of the system of ODEs (7). Thus, for example, a system of 8 ($N = 8$) ODEs could be split into 3 sub-systems with the assignment

$$\mathcal{Y}_1 : \left\{ \begin{array}{lcl} \dot{y}_1 & = & f_1(t, \mathbf{y}) \\ \dot{y}_2 & = & f_2(t, \mathbf{y}) \\ \dot{y}_3 & = & f_3(t, \mathbf{y}) \end{array} \right.$$

$$\mathcal{Y}_2 : \left\{ \begin{array}{lcl} \dot{y}_4 & = & f_4(t, \mathbf{y}) \\ \dot{y}_5 & = & f_5(t, \mathbf{y}) \\ \dot{y}_6 & = & f_6(t, \mathbf{y}) \end{array} \right.$$

$$\mathcal{Y}_3 : \left\{ \begin{array}{lcl} \dot{y}_7 & = & f_7(t, \mathbf{y}) \\ \dot{y}_8 & = & f_8(t, \mathbf{y}) \end{array} \right.$$

We now rewrite the system of N ODEs in terms of such sub-systems. Suppose we have split the N ODEs into K sub-systems. The system is then represented as

$$\begin{aligned}
\dot{\mathcal{Y}}_1 &= \mathcal{F}_1(t, \mathbf{Y}) \\
\dot{\mathcal{Y}}_2 &= \mathcal{F}_2(t, \mathbf{Y}) \\
&\dots \\
\dot{\mathcal{Y}}_K &= \mathcal{F}_K(t, \mathbf{Y})
\end{aligned}$$

where $\mathbf{Y} = (\mathcal{Y}_1, \mathcal{Y}_2, \dots, \mathcal{Y}_K)$. The choice of splitting will dictate the nature of this coupling. We then apply the procedure (8) to the sub-systems.

The algorithm to integrate a system of N initial value ODEs is rather straight forward. Each sub-system $\mathcal{Y}_i$ is assigned a parallel thread of execution. We prefer to use the term *thread* because in a multiprocessor environment the number of processors available will likely be different than the number of sub-systems we have at hand. If there are less processors available than the number of of sub-systems then these threads will have to share the available pool of processors. Load balancing will become an important issue in such a case. The notion of a thread of execution will allow us to defer the issue of processor assignment while we outline our procedure. We will address it when we discuss how the procedure has been implemented as a computer code.

The first item on the agenda for all sub-systems is to set up the initial values vector $\mathbf{Y}(t_0)$. Once this is done, each sub-system makes it initial values vector available to all sub-systems that depend upon it. Then each sub-system must then wait until all sub-systems it depends upon have initialized.

In the discussion that follows, we will use the terms *client* sub-systems and *server* sub-systems. For a given sub-system, the sub-systems that it depends upon will be termed its *servers* while those that depend upon it will be termed *clients*. For example, if the coupling in the 3-sub-systems example presented previously is as follows:

$$\begin{aligned}
\dot{\mathcal{Y}}_1 &= \mathcal{F}_1(t, \mathcal{Y}_1, \mathcal{Y}_2) \\
\dot{\mathcal{Y}}_2 &= \mathcal{F}_2(t, \mathcal{Y}_1, \mathcal{Y}_2, \mathcal{Y}_3) \\
\dot{\mathcal{Y}}_3 &= \mathcal{F}_3(t, \mathcal{Y}_1, \mathcal{Y}_3)
\end{aligned}$$

$\mathcal{Y}_2$ is a server of $\mathcal{Y}_1$, $\mathcal{Y}_1$ and $\mathcal{Y}_3$ are servers of $\mathcal{Y}_2$, and $\mathcal{Y}_3$ is a server of $\mathcal{Y}_2$. On the clients side, $\mathcal{Y}_2$ and $\mathcal{Y}_3$ are clients of $\mathcal{Y}_1$, $\mathcal{Y}_1$ is a client of $\mathcal{Y}_2$, and $\mathcal{Y}_3$ is a client of $\mathcal{Y}_2$.

We will present the subsequent events from the point of view of sub-system 2 in the 3-sub-systems example we have chosen. Sub-system 2 integrates

$$\dot{\mathcal{Y}}_2 = \mathcal{F}_2(t, \mathcal{Y}_1^0, \mathcal{Y}_2(t_0), \mathcal{Y}_3^0) \tag{9}$$

using any one-step or multi-step method to obtain $\mathcal{Y}_2^1$, i.e., the second iterate. It then notifies its client-sub-system 2 that it has a new iterate and makes $\mathcal{Y}_2^1$ available to it. Sub-system 2 then waits to be notified by one or both of its server sub-systems 1 and 3. Sub-systems 1 and 3 would have performed a similar integration, perhaps at a faster or slower pace than sub-system 2. They would also send notifications to their clients. Suppose sub-system 2 receives a notification from its server sub-system 1 while its other server sub-system, i.e., sub-system 3, is still integrating. Sub-system 2 now proceeds with its second iteration:

$$\dot{\mathcal{Y}}_2 = \mathcal{F}_2(t, \mathcal{Y}_1^1, \mathcal{Y}_2(t), \mathcal{Y}_3^0) \tag{10}$$

to compute $\mathcal{Y}_2^2$. Note that it re-uses $\mathcal{Y}_3^0$ because that is the most recent information it has at hand from sub-system 3. Sub-system 2 then notifies its client, sub-system number 1, and ships $\mathcal{Y}_2^2$ over to it. Next time around, it may receive notification from both of its clients, sub-system 1 and 3. It then proceeds with the next iteration.

When a notification from a server sub-system arrives, the notified sub-system could either be in the middle of iteration or it could be waiting for a notification. In the latter state, the sub-system performs the next iteration. In the former state, it can either abort the current iteration and restart again because a more recent iterate is available from another sub-system, or it can continue the present iteration and attend to the notification only after the current iteration has completed. Our algorithm is designed to operate using the latter strategy, that is, sub-systems continue the current iteration and attend to notifications only after the current iteration has completed. We want to reduce the number of occasions a sub-system has to synchronize with other sub-systems. Moreover, more than one notification could arrive while a sub-system is computing an iterate. Each notification tells the sub-system how far back to go and compute another iterate. By waiting until the current iteration is done, the sub-system can attend to all of the outstanding notifications, determine the earliest time among the integration times that are part of the notifications and compute the next iterate.

The goal for sub-system 2 is to integrate $\dot{\mathcal{Y}}_2 = \mathcal{F}_2(t, \mathbf{Y})$ out to t_{final}. Whenever it receives a notification from one or both of its servers, it can integrate out to t_{final}. But it decides to forgo t_{final} as the end point of integration in favor of the following: upon receiving a notification, sub-system 2 begins computing a new iterate $\mathcal{Y}_2^n$. At each time step during the numerical integration of $\dot{\mathcal{Y}}_i^n = \mathcal{F}_i(t, \mathbf{Y})$, sub-system 2 checks the difference between the previous iterate $\mathcal{Y}_2^{n-1}(t)$ and newly computed $\mathcal{Y}_2^n(t)$. While this difference is less than a user specified tolerance ϵ_c, the integration continues, but the sub-system retains the last iterate. This is called the *validate* phase and the sub-system checks that convergence has been achieved. When the difference $|\mathcal{Y}_2^n(t) - \mathcal{Y}_2^{n-1}(t)|$ exceeds a user specified tolerance ϵ_c at some point t_c, then sub-system 2 decides to integrate $\mathcal{Y}_2^n(t)$ to some point t_w where $t_w = \min(t_c + w, t_{final})$. This is called the *update* phase. Beyond, t_w, sub-system 2 sets $\mathcal{Y}_2^n(t) = \mathcal{Y}_2^n(t_w)$ for $t_w < t \leq t_{final}$. The interval $[t_c, t_w]$ is termed a *window*. The window size, that is, w can be constant, or more interesting, computed dynamically. Our current algorithm uses a constant window size; the user can specify a window size for each sub-system. Efficiency dictates that this window should be less than the entire time interval. But it should not be so much less that a large number of integrations over windows are required since each integration has an overhead. On the other hand, if the window is too large, too many iterations are required, increasing the cost. A dynamic window sizing algorithm would be ideal but we do not know a suitable algorithm yet.

Once $\mathcal{Y}_2^n$ is at hand, sub-system will send a notification to its clients that they need to re-iterate beyond t_c. Recall that t_c is where sub-system 2 found a change larger than ϵ_c. Sub-system 2 will also make it new iterate $\mathcal{Y}_2^n$ available to its clients. In a similar manner, the servers of sub-system 2 will send notifications along with their t_c's. Sub-system 2 takes the minimum of these t_c's and begins it next iteration at that minimum.

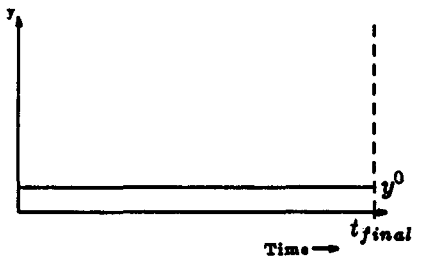
As notifications arrive, sub-system 2 iterates to compute $\mathcal{Y}_2^n(t)$, $\mathcal{Y}_2^n(t)$ and so on, ultimately reaching t_{final}. And, hopefully, so do the other sub-systems. When the entire system reaches t_{final}, and there are no notifications pending, the iteration activity stops because the all sub-systems have converged.

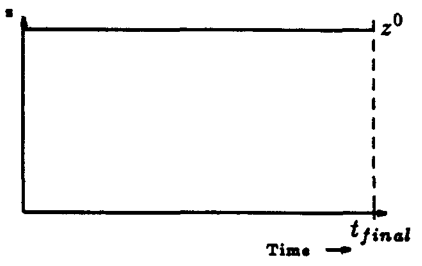
If a sub-system does not have any servers, that is, it does not depend upon any other sub-system then it will continue with its next iteration starting at the end of the previous window. This would happen for example if all of the ODEs in a system were lumped into just one sub-system.

We present a pictorial example of how such an iteration proceeds. Consider a system of two ODEs:

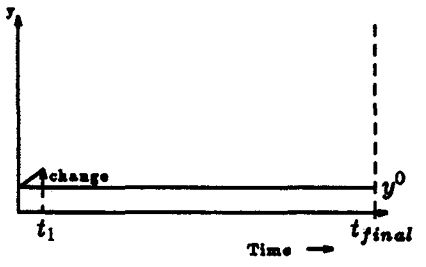
$$\begin{aligned} \dot{y} &= f(t, y, z) \\ \dot{z} &= g(t, y, z) \\ y(0) &= y_0,\ z(0) = z_0,\ 0 \le t \le t_{final}. \end{aligned}$$

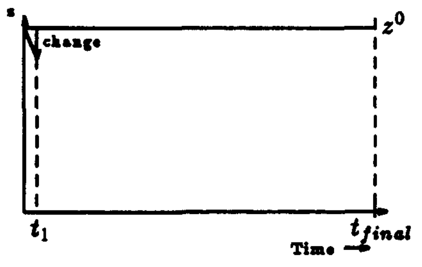
We set up two sub-systems and start the integration for each in parallel. Because it is such a simple example, we will use y and z instead of sub-system 1 and sub-system 2. Both notify each other that the initial iterates are as follows:





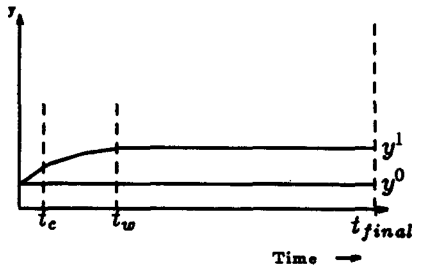
The first iteration begins; for brevity we will assume that threads of execution for y and z, each have a processor so they begin without any holdups. Thread y takes a step, computes $y^1(t_1)$ and compares it to $y^0(t_1)$. It is very likely that the difference would exceed the specified tolerance. Thread z would go through a similar stage. A freeze-frame of the two will look like the following:



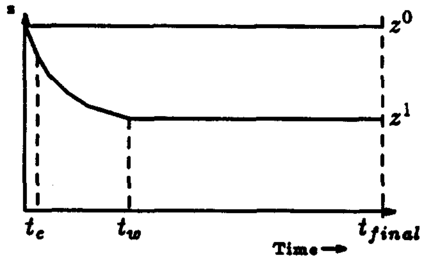


Thread y will set t_c equal to t_1, the time at which it detected change. It will create a window from t_c to t_w, where $t_w = t_c + w$ integrate over the window without performing any difference tests along the way. Beyond t_w, y^1 will be set to $y^1(t_w)$.

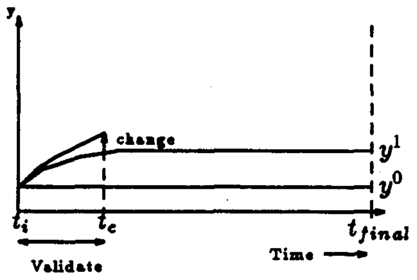
The profile of the first iterate will then be:

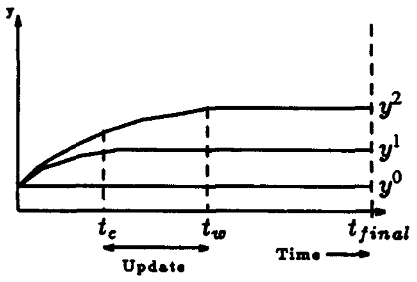


Similarly, thread z will end up with its first iterate with the following profile:

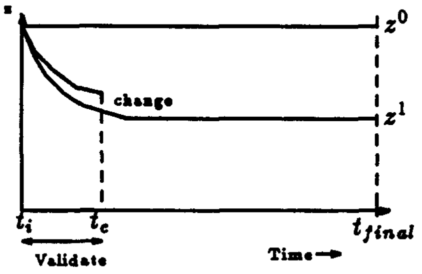


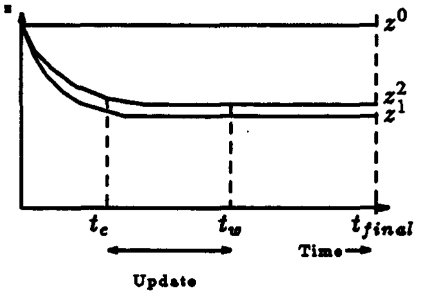
As soon as y has the first iterate, it will notify its client z and make the new iterate available. The notification will also include y's t_c; z will have compute new values for t including and beyond t_c. Component y now awaits notification from z. When the notification arrives, y will take t_c^z, i.e., the value of t at which z has revised its iterate, and look through y^1, its most recent solution vector. The goal of the search would be to find the largest t_k such that $t_k < t_c \leq t_{k+1}$. Such a t_k will be taken as the starting point t_i for the second iteration. Again, at each time step, y will check to see if $y^2(t)$ exceeds $y^1(t)$ a specified tolerance. When it does, y, will create a window in time, integrate over it, and send a notification to z along with y^2. Pictorially, the following will occur:



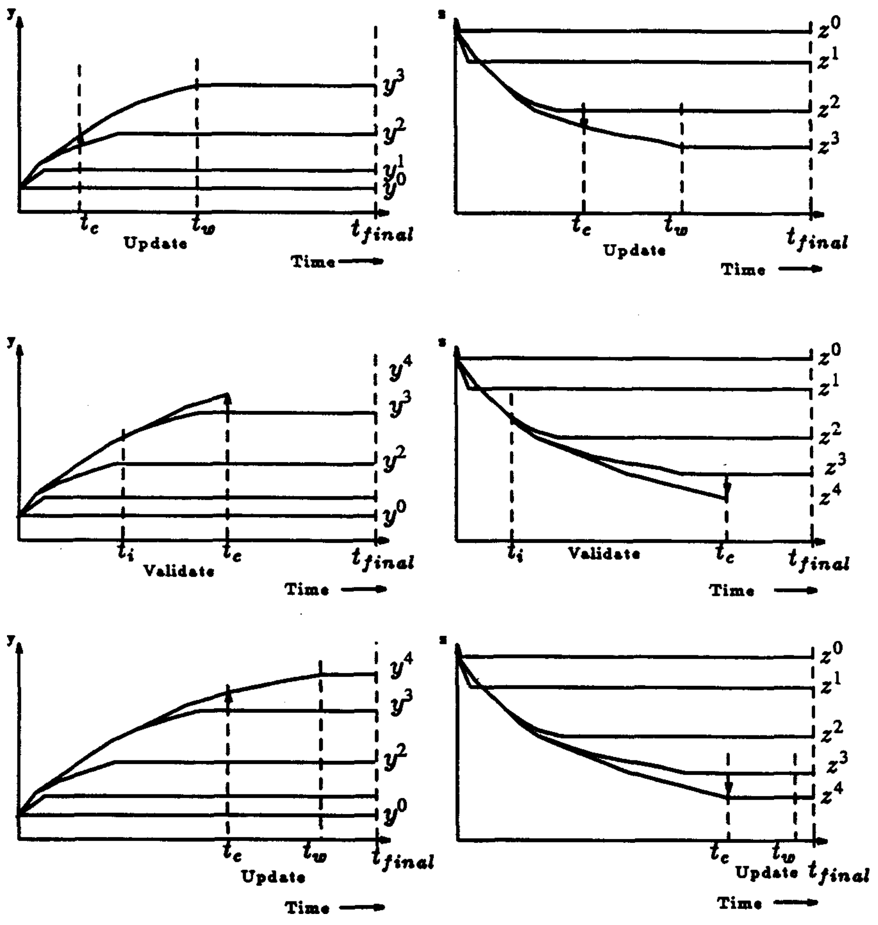


As indicated earlier, the iteration stage in which the difference check is performed at each time step is called *validate* phase, and the integration over the window $[t_c, t_w]$ is termed *update* phase. Similarly, z will go through a similar phase upon receiving a notification from y:





The process continues, progressing through stages represented by the following frames:



Ultimately, for some iterate m, the difference $|y^m - y^{m-1}|$ will be within the tolerance. The integration will span out to t_{final}. Component y will not sent any notification to z. Similarly, z will end up in a similar state. With no notifications pending and both components sitting at t_{final}, the system would have reached convergence. The iteration activity will cease.

Figure ?? presents a state diagram which describes the transitions through various stages of a single iteration for a particular sub-system.

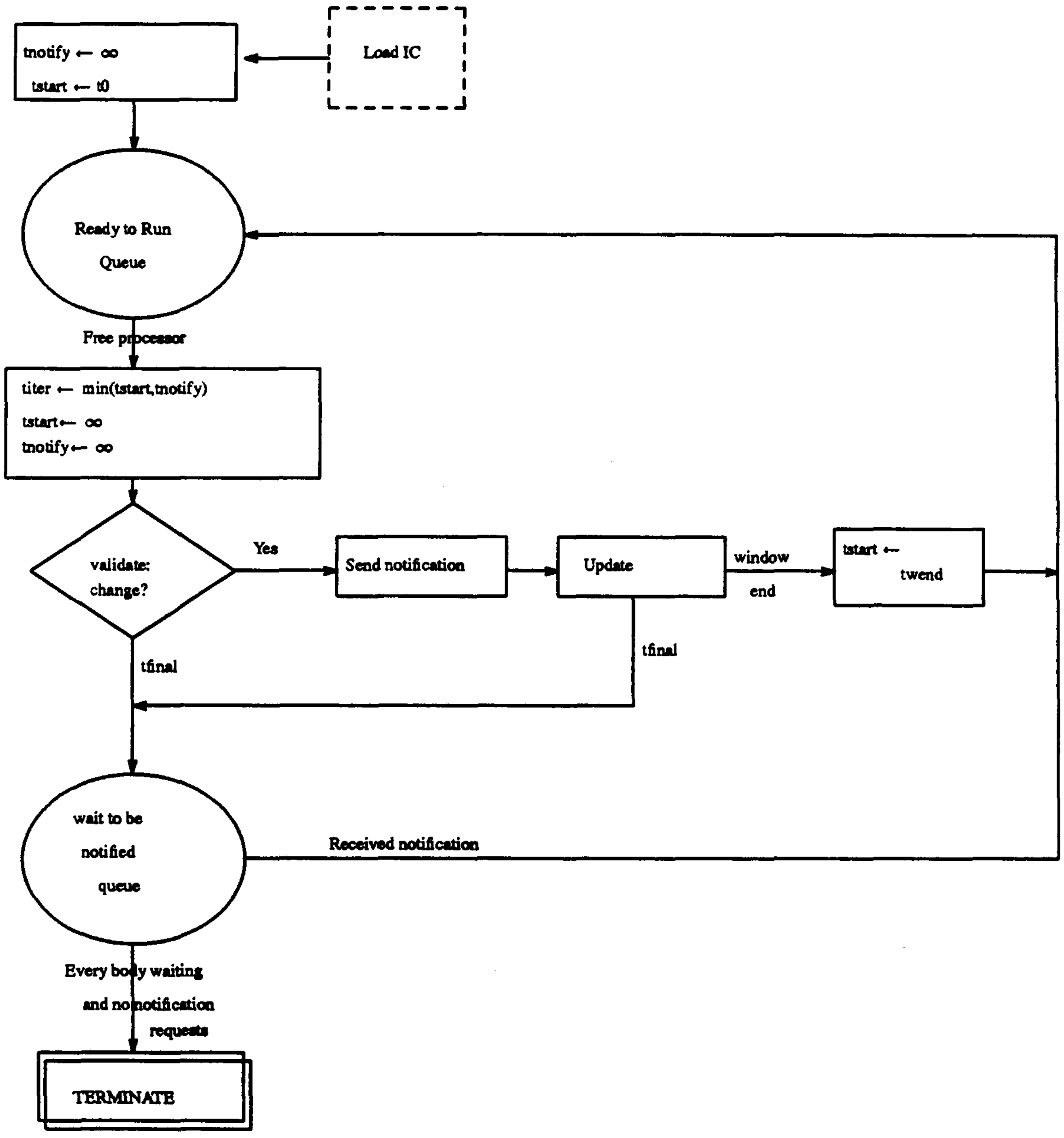


Figure 1: State diagram for transitions a sub-system goes through during asynchronous iteration

7 Implementation on a Multiprocessor

As we mentioned in the introduction, we have implemented the asynchronous integration procedure on a multiprocessor, shared memory system. In our case, it is an Encore Corporation's Multimax computer with 20 National Semiconductor NS32000 series processors. We use the Encore's Parallel Threads Package [4] which is based upon the Brown University's Threads Package [2]. The computer code is written in C. It follows the traditional environment provided by existing ODEs solvers; the user supplies the main driver, a routine to supply the initial conditions and a routine that evaluates f in $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$. Splitting of the system of ODEs into sub-systems places another requirement to be fulfilled by the user: the user has to specify the number of sub-systems and the coupling amongst them.

7.1 Processor Allocation

For a given set of sub-systems, the code spawns off threads of execution to run on the available processors. The pool of threads then perform iterations on behalf of the sub-systems. As noted earlier, the number of sub-systems, and thus the number of threads, may not be the same as the number of physical processors available. This immediately leads to the issue of processor assignment. At the simplest level, we can let the underlying operating system (UNIX in our case) manage the thread to processor assignment. More desirable, however, is to schedule threads for execution based upon need for iteration. Consider that we have a large set of sub-systems that are carrying out iteration. When a sub-system finishes computing an iterate it would send notifications to its clients, that is, the sub-systems that depend upon it. At any one moment there would be a number of such pending notifications. As these get serviced, the notified sub-systems will want to carry out another iteration. An appropriate scheduling strategy would be that an idle processor be assigned to the sub-system which has the earlier iteration restart time. This time would be the minimum of the all the t_e in the notifications that were serviced. There are two ways such a scheduling decision can be made. There can either be a manager thread or a pool of threads servicing sub-systems.

If a manager thread is spawned along with threads for each sub-system then all notifications will be routed to the manager. When a sub-system wants to notify its clients, it will place a request with the manager. After placing the request, the sub-system will suspend itself in a wait queue. The manager will take these requests and inform the targeted clients. The manager will then assign processors in order of the values t_c that are part of the notification request; the sub-system which is notified with the earliest t_c will be the first one to get a processor.

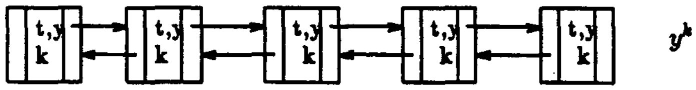
The other approach is to have a pool of threads that service the sub-system. A thread from the pool carries out an iteration on behalf of a sub-system. Upon reaching the end, the sub-system will either send a notification to its clients or would reach t_{final}. The notified sub-systems will be tagged to indicate that they need to re-iterate. After generating the notification and making the new iterate available, the notifying sub-system will relinquish the thread. The thread will now look through the all of the tagged sub-systems for the sub-system with the minimum t_c value. When one is found, the thread will execute the next iteration on behalf of the sub-system.

At present, we are not using either of these strategy. We are relying on the underlying operating system to scheduling. However, we are in the process of modifying the code for the two scheduling schemes outlined above.

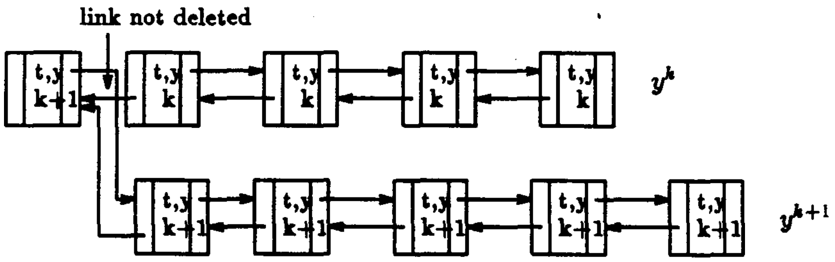
7.2 Data Structures for Storing and Managing Iterates

Each iterate is actually a series of values at various steps over the integration interval. When a sub-system computes a new iterate, part of the previous iterate will have to replaced. The issue of storing, updating and making newer iterates available to client sub-systems will be largely decided based upon the underlying multiprocessor architecture. In a distributed or networked set of processors, each sub-system will likely keep a local database of iterates of sub-systems that it needs to carry out the integration. As newer iterates arrive as part of a notification message, the local database would be updated. In shared memory system, an obvious choice is to have a database of iterates that is shared amongst all of the sub-systems. Whenever a sub-system needs to update its iterate it will either go into the database itself or request a manager process to do so on its behalf. In either case, the entry in the database for the sub-system's iterate will require synchronized access. Our goal, however, is to carry out asynchronous iteration with the least amount of synchronization. Our present implementation is on a shared memory multiprocessor; we had to choose a data structure for the shared database that would require the least amount of synchronization, e.g., locking etc.

It is not known at the outset how many values each iterate will have so the data structures used to store the iterates have to be dynamic. A frequent operation on the iterates is interpolation; a linked list is thus a suitable choice. It is also convenient to store and delete nodes in a linked list. Our code uses a doubly linked list to store the iterates. When a new iterate is computed, its is linked into the existing linked list at the the appropriate point. For example, suppose the k^{th} iterate of some component y looks like the following:



When y^{k+1} has been computed, it is linked in based upon the value t:

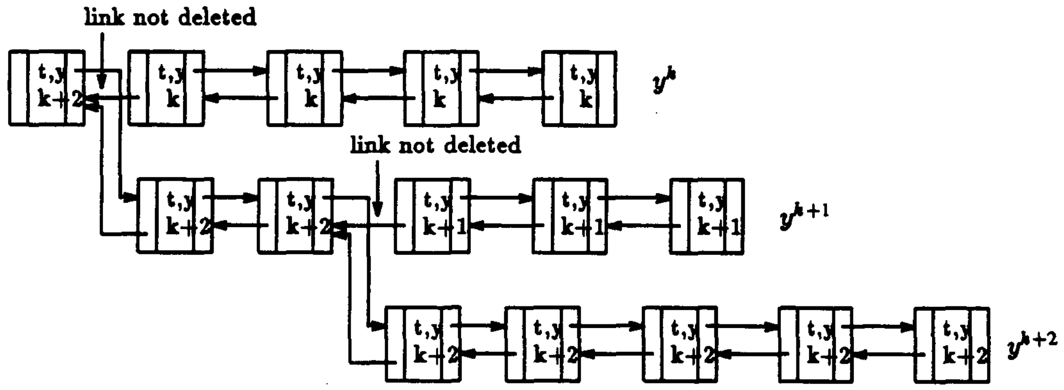


Note however, that the forward pointer of the node after which the new list is inserted is changed but the reverse pointer from the first outdated node is left in place. This way if a client using the old list backs up, it will end up in the new segment. With such an arrangement looking up values in the list does not require a lock which otherwise would impose a major synchronization overhead. This also allows the sub-system managing the iterate to insert a new iterate without locking. The only time any synchronization is needed is when an outdated iterate is to be deleted and storage reclaimed.

The outdated nodes are not deleted immediately to reclaim the memory. The reason is that there could be a client sub-system using y^k. There is only one database being shared amongst all sub-systems. We can not delete the outdated position until y is sure that no one is using the old portion. This is determined by keeping a reference count in the database for each outdated branch. When an outdated iterate branch is created, the reference count for the branch is set to the number of clients of the sub-system, y in this example. When a client of y comes into the database to use the value of y's iterate, it decrements the reference count for all of the outdated branches.

When the count becomes zero for a branch, y can can delete the branch and reclaim the storage.

When y^{k+2} is computed, it is linked in as follows:



8 Preliminary Results

To illustrate the code, we integrated the following system of ODEs which are the equations for a two-body problem [3], page 236-238:

$$\begin{aligned}
\dot{y}_1 &= y_3 \\
\dot{y}_2 &= y_4 \\
\dot{y}_3 &= -y_1/(y_1^2+y_2^2)^{\frac{3}{2}} \\
\dot{y}_4 &= -y_2/(y_1^2+y_2^2)^{\frac{3}{2}}
\end{aligned}$$

Time range is $0 \le t \le 7.5$. The initial conditions are $y_1(0) = 0.5$, $y_2(0) = 0$, $y_3(0) = 0$, and $y_4(0) = \sqrt{3}$. We grouped all four into one sub-system, then into two sub-systems with the split y_1, y_3, and y_2, y_4, i.e.,

$$\mathcal{Y}_1 : \left\{ \begin{aligned} \dot{y}_1 &= y_3 \\ \dot{y}_3 &= -y_1/(y_1^2+y_2^2)^{\frac{3}{2}} \end{aligned} \right.$$

$$\mathcal{Y}_2 : \left\{ \begin{aligned} \dot{y}_2 &= y_4 \\ \dot{y}_4 &= -y_2/(y_1^2+y_2^2)^{\frac{3}{2}} \end{aligned} \right.$$

and finally, into four sub-systems with one component per sub-system. The sub-systems generate a trace of their progress which can displayed on any workstation running the X Windows System [5]. The figures we present were generated by doing a dump of the graphics window as a bitmap and then included within the body of this report.

Figure 1 presents the results for the one sub-system case. There are three view ports in the graph. The upper most presents the solution profiles of the four components. The middle frame has a series of five horizontal lines, each at a higher level than the previous. These lines represent the windows t_c through t_w. The lower most frame presents a profile of the step sizes that were used during integration. As there is only one sub-system, the solution profiled was not computed by iteration. The integration however, was carried out over a set of windows such that each subsequent window starts off where the previous ended. For such one sub-system cases, our code reduces to a typical ODEs solver. The sharp downward pointing spikes in the step sizes plot are due to the fact that for every new iteration, the sub-system starts off with a default value of step size. A much better choice would be the average of step sizes used in the previous iteration.

Figure 2 and 3 present similar frames for the two sub-systems case. The iterates y_1^j and y_3^j are plotted in the top most frame in Figure 2. Note how the iterates go off the track and then fall back on track. The windows now show overlap, or backtracking, caused when each of the two sub-systems notifies each other of a revised value of its iterate. The progression of windows is towards the right indicating progress along t towards t_{final}. The profiles for step sizes show larger variation especially in the regions where the solution curves have steeper slopes. Figure 3 presents the results for the sub-system composed of y_2 and y_4.

Figures 4 through 7 present graphics frames for the 4 sub-systems case. Each component of the system of ODEs is being integrated asynchronously. Many more iterations were required. The iterates show significant deviation before settling down on the correct path.

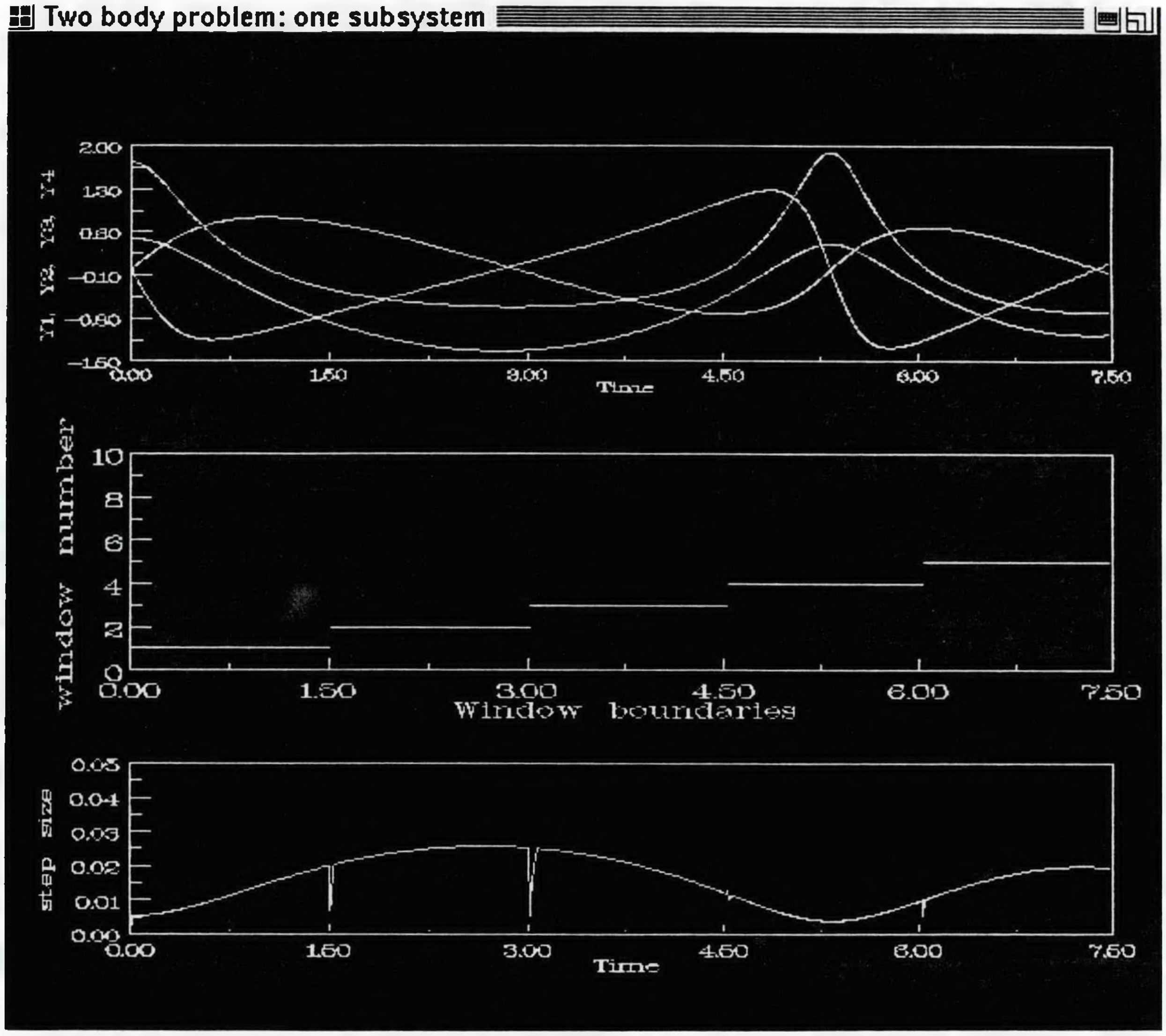


Figure 2: Two body problem, one sub-system: profile of solution iterates, step sizes and windows.

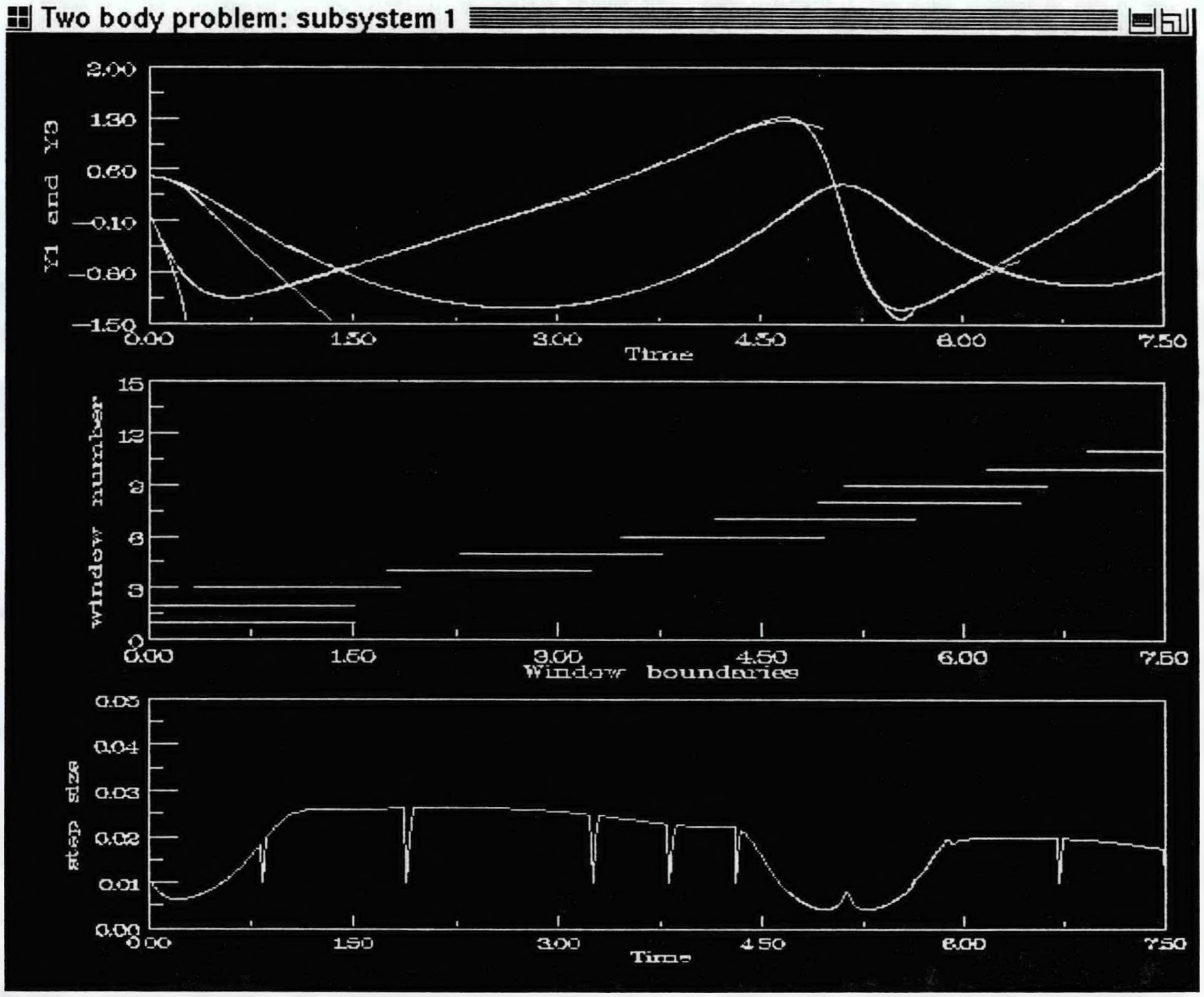


Figure 3: Two body problem, two sub-systems: profile of solution iterates, step sizes and windows for sub-system 1.

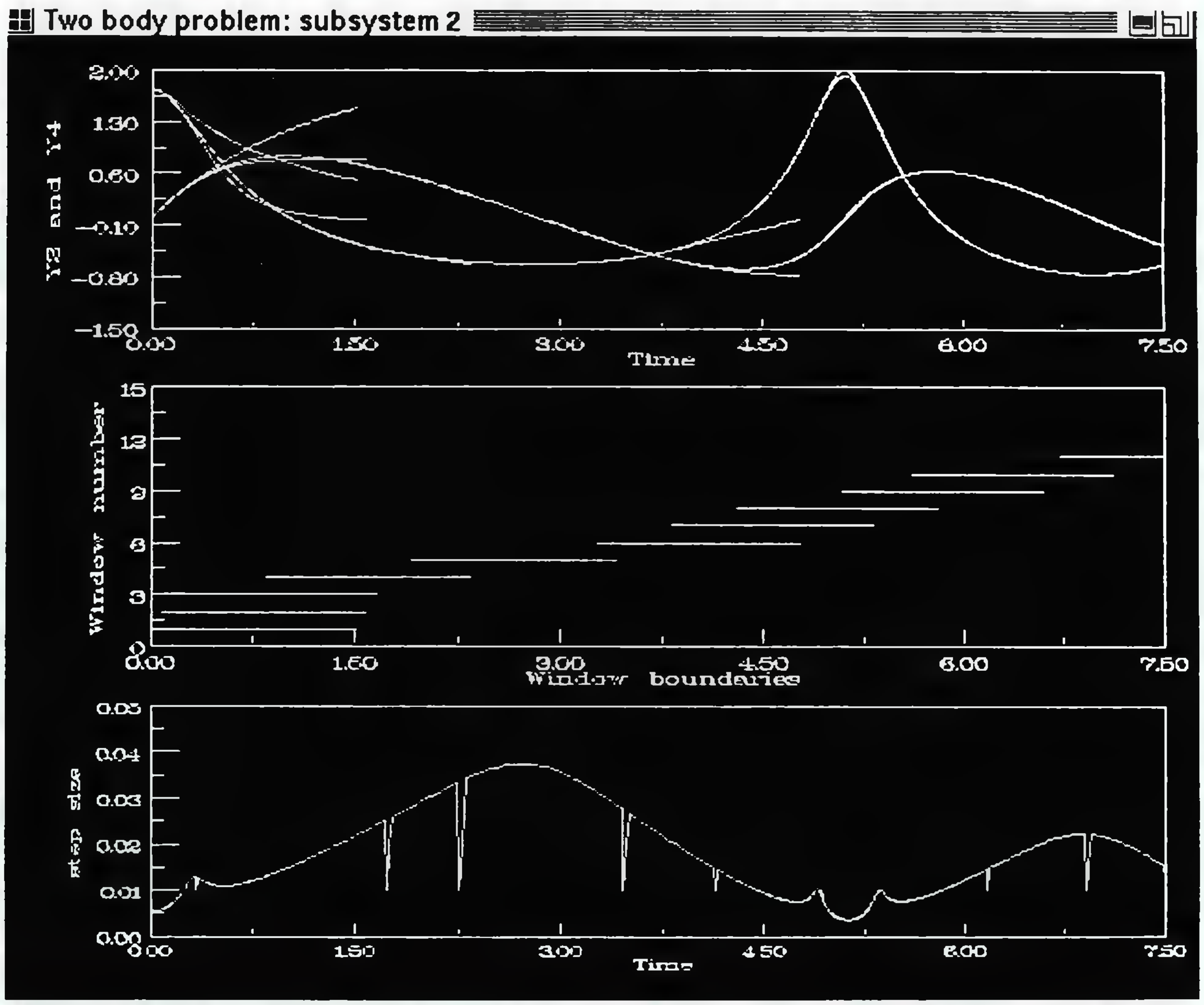


Figure 4: Two body problem, two sub-systems: profile of solution iterates, step sizes and windows for sub-system 2.

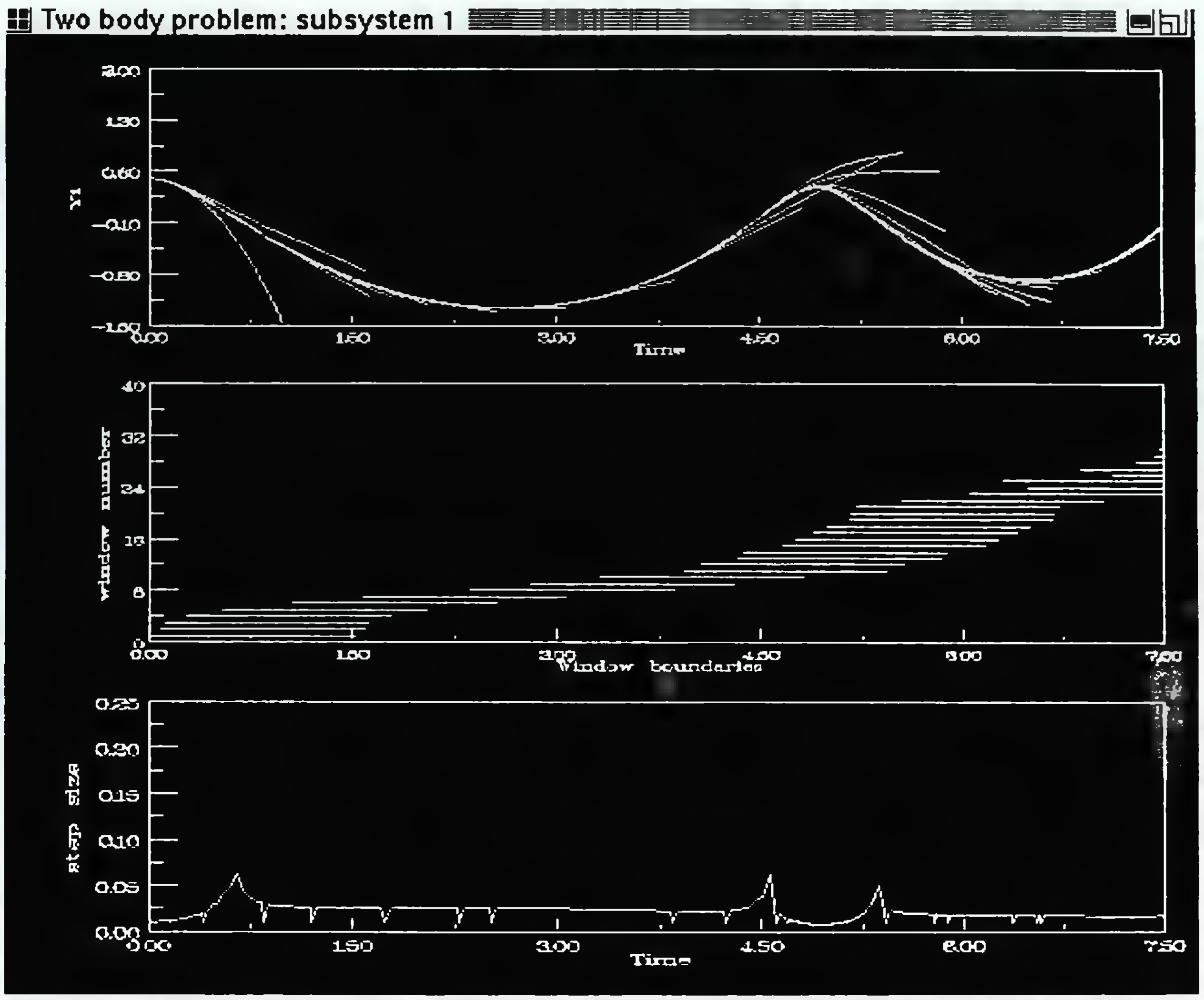


Figure 5: Two body problem, four sub-systems: profile of solution iterates, step sizes and windows for sub-system 1.

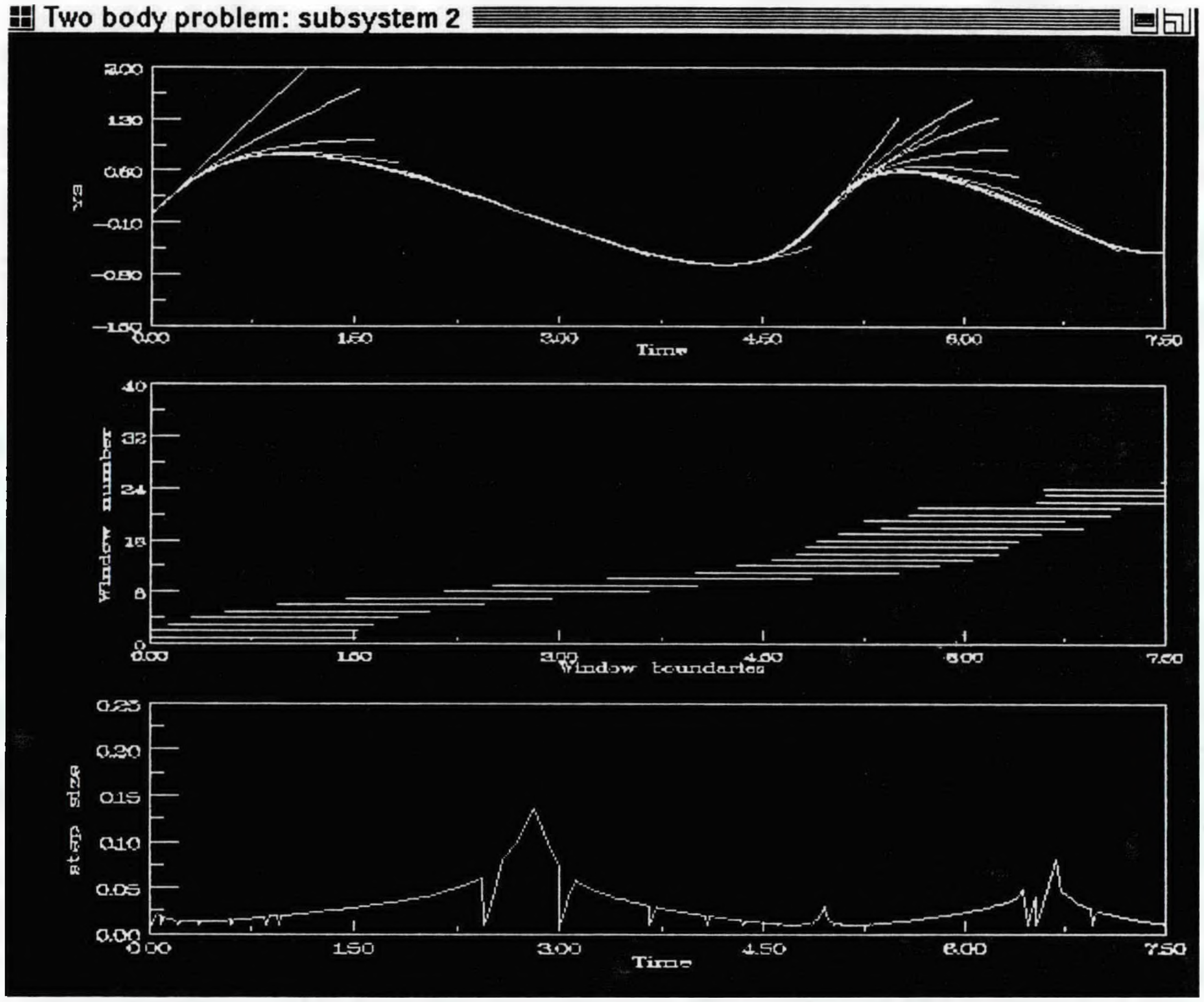


Figure 6: Two body problem, four sub-systems: profile of solution iterates, step sizes and windows for sub-system 2.

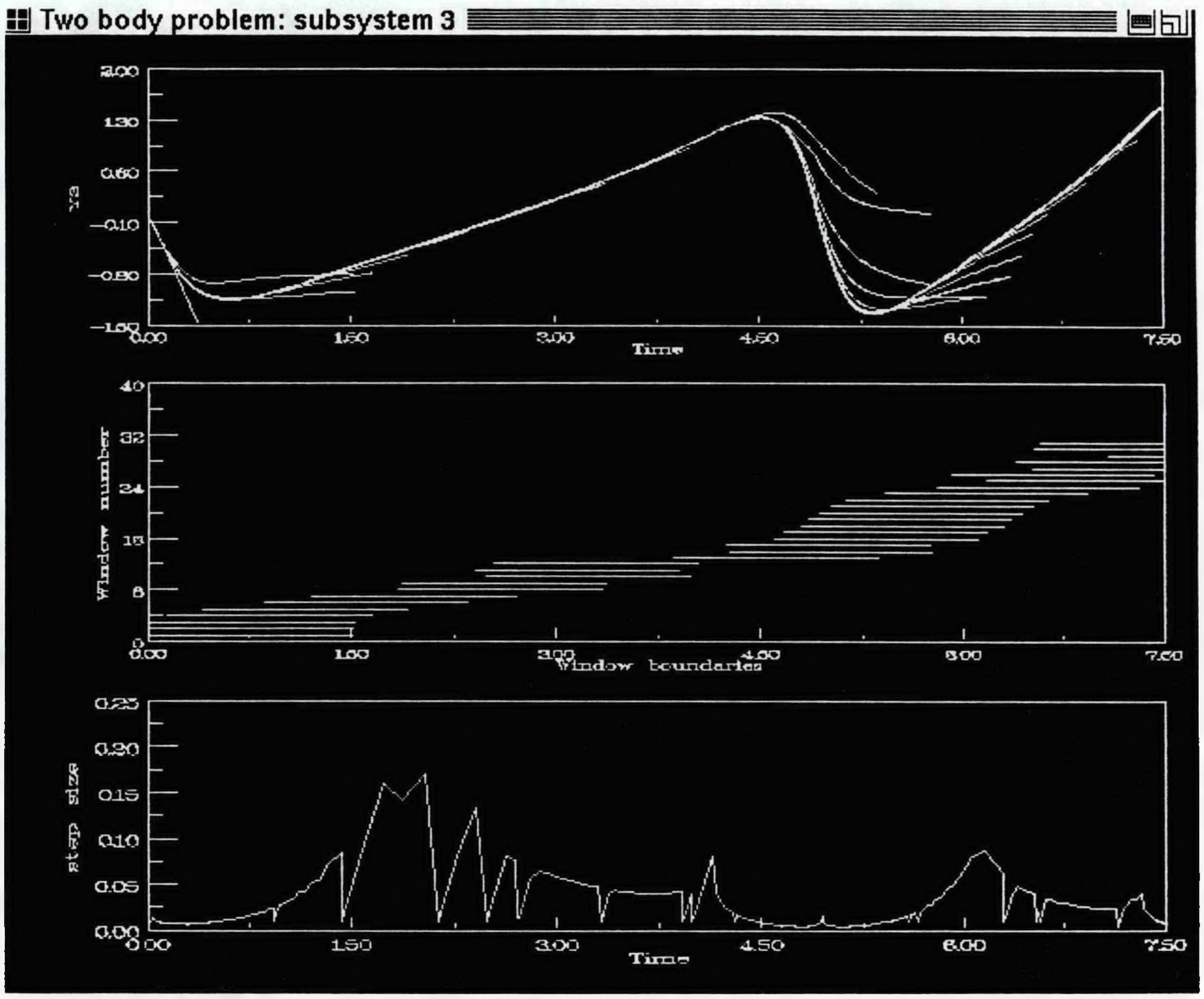


Figure 7: Two body problem, four sub-systems: profile of solution iterates, step sizes and windows for sub-system 3.

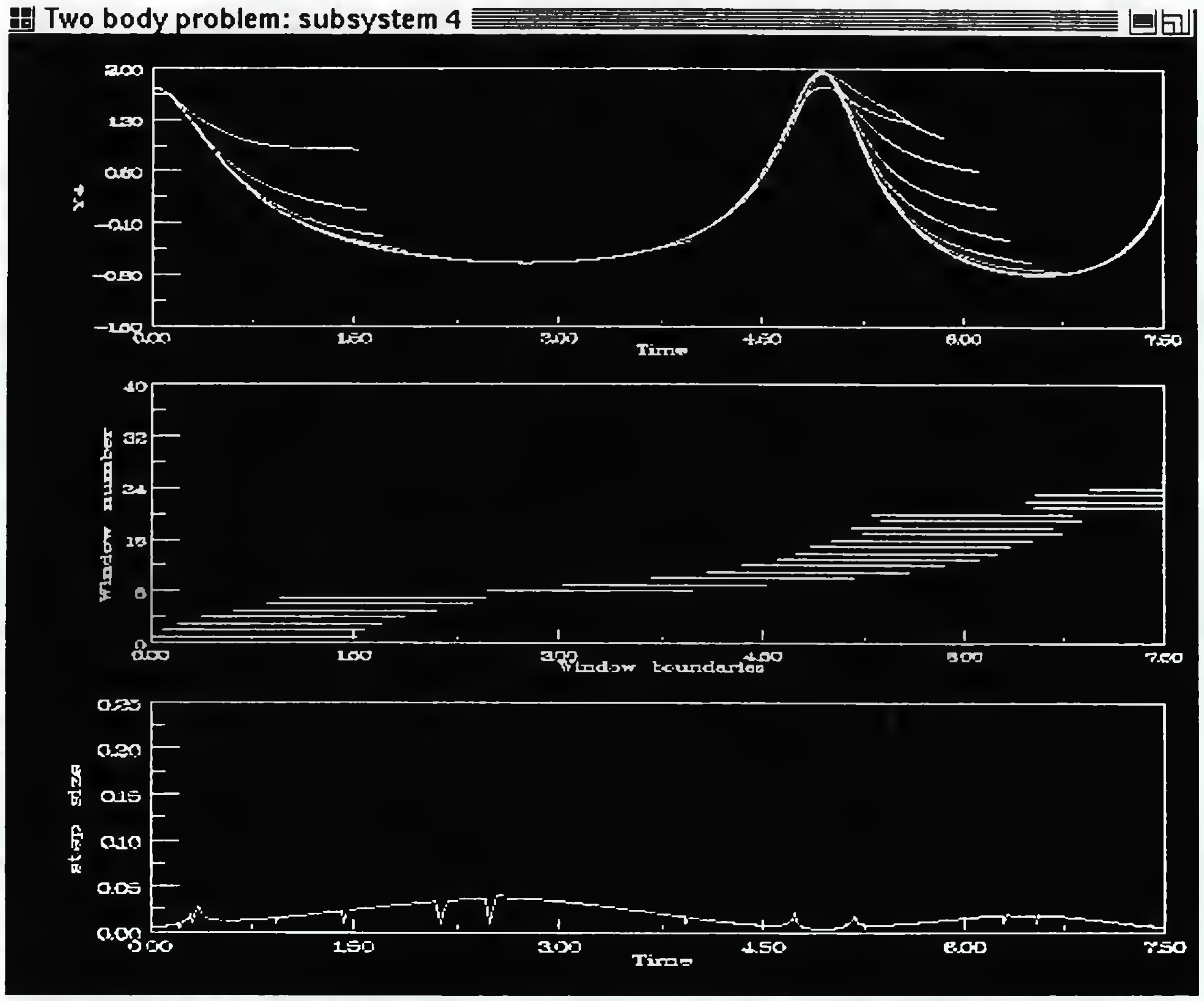


Figure 8: Two body problem, four sub-systems: profile of solution iterates, step sizes and windows for sub-system 4.

9 Future Work

We have hinted at a number of issues that need to be resolved. We gather them, and a few more, in this section.

We need to determine an effective procedure to ensure the contracting property of the iteration matrix. Recall that if implicit methods are selected for integration of the sub-systems then asynchronous iteration places a restriction on the step sizes chosen.

We are in the process of implementing the two processor allocation algorithm we discussed, that is, having a manager process or a pool of processors serving sub-systems.

Large systems will require large amounts of storage space for storing the iterates. As portions of the iterates become outdated, the storage is reclaimed but if the integration interval is large and a large number of intermediate steps are taken then the linked lists for the iterate could beyond the storage capacity available. One obvious solution is that once all of the sub-systems have converged up to a certain point in the integration interval, the iterates up to this point can be written off to an external device and the freed storage used for future nodes. We are in the process of devising code to do so.

Once the code is in a form that we try large problems, we'll try out our code against well known ODE solvers. We envision that only then we'll see a gain in using asynchronous integration over the traditional sequential codes.

We could also like to implement the code on hypercube in order to address and subsequently attempt to resolve, issues raised in a networked multiprocessing environment.

References

[1] Gérard M. Baudet, *Asynchronous Iterative Methods for Multiprocessors*, J. ACM, Vol. 25, No. 2, (April 1978), pp. 226-244.

[2] Thomas W. Doeppner, *Threads: A System for the Support of Concurrent Programming*, Department of Computer Science, Brown University, Technical Report CS-87-11, June 16, 1987.

[3] E. Hairer, S. P. Nørsett, G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, Springer-Verlag, 1987.

[4] *Encore Parallel Threads manual*, Encore Computer Corporation, 1988.

[5] *X Window System*, Massachusetts Institute of Technology, 1988.