可支援一致性中斷點與多執行路徑的MPI 除錯器
An MPI Debugger Supporting Consistent Breakpoints and Multiple Execution Paths

竇其仁 陳忠信 郭逸冠
Chyi-Ren Dow Jong-Shin Chen Yi-Gon Gon
逢甲大學資訊工程系
Department of Information Engineering, Feng Chia University
{crdow, jschen, yggou}@iecs.fcu.edu.tw

摘要

在本篇文章中，我們介紹一套針對 MPI 系統所設計的交談式多功能平行/分散式除錯工具。我們研究開發的系統主要包含了交談式的除錯功能， trace-based cyclical debugging 的機制。我們將平行/分散式程序相互間的執行關係轉換成以 task-based 為主的執行關係，同時我們修改 Lamport 的 timestamp 演算法[1]定出 task 的優先關係，利用這個關係我們定義出一種新的平行中斷點的設定，並提供使用者對於程式之執行路徑群組以 task-based 為主做一致性的單步追蹤。

關鍵字：平行中斷點、一致性多執行路徑、平行/分散式除錯器。

Abstract

In this paper, we present an interactive and multi-functional distributed/parallel debugger for MPI on the cluster of workstations. This system supports a range of facilities, including interactive facility and trace-based cyclical debugging facility. The execution behavior of a parallel/distributed program can be represented by a task-graph. A new partial order relation based on Lamport's timestamp algorithm [1] is developed and it can be used to set consistent breakpoints and support multiple execution paths.

Keywords: parallel breakpoints, consistent multiple execution paths, parallel/distributed and debuggers.

一、 緒論

平行/分散式程式的除錯，比傳統循序程式（sequential program）的除錯更為困難，主要的原因來自於平行/分散式程式的執行行為的不確定性(即給予相同的輸入所得的程式執行結果並不一定會相同)，因此在除錯的過程中我們往往需要一個機制來讓之前的程式執行行為重現，讓程式設計者可藉由不斷的對相同的程式執行行為做追蹤來達成除錯的最終目的。

目前國內外雖有針對分散式訊息傳遞特性而發展出來的除錯工具，但幾乎都是針對 PVM（Parallel Virtual Machine）而設計。MPI（Message Passing Interface）標準制訂於 1994 年，開始是由數個學術單位所共同訂定的一套通訊函式語法標準。制定此標準的目標在於提供一個有效率，有彈性且和程式語言、機器種類無關的標準，用以撰寫訊息傳遞的程式碼。MPI 標準提供極為完善的平行計算功能，目前雖然已有不少依據 MPI 標準而實作的系統，但因為發展時間較短，故以 MPI 系統為環境的輔助工具仍相當缺乏。有鑑於此，我們希望在 MPI 系統上開發一個整合型的平行程式除錯工具，補足傳統循序程式除錯器在平行程式上的不足，減低 MPI 程式設計者在程式開發過程中的負擔，以幫助程式設計師來有效率的發展出平行/分散式應用程式。

以下的內容我們將針對三種常用的平行程式除錯技巧[2][3]做介紹，並說明它們如何幫助程式設計者發掘問題。這些除錯技巧分別為靜態分析（static analysis）、動態執行分析（on-the-fly analysis）、與 trace-based 技巧。

在編譯過程中做資料流程（data flow）分析，可以找出可能的競爭情況（race condition）。靜態分析可以用來偵測程式中死結(deadlock)的發生，採用靜態分析的好處是不會對平行程式產生任何影響原程式執行結果的因素，也就是沒有探測效應（probe effect）。但其缺點是靜態分析需要很高的計算複雜度，而且有可能分析後回報不真實的競爭情況。

如果除錯技術是應用在程式執行時，這便稱為動態（on-the-fly or dynamic）分析除錯。它通常會提供設定中斷點的功能，要求當執行到某一行指令或某特定敘述時程式停止執行，讓程式設計師觀察此時程式的狀態。但這樣的除錯技術對平行程式會產生影響程式執行的探測效應。另外要使用者控制多個同時在執行的程序，往往會不知所措，也就是所謂的迷失效應（maze effect）[3]。

Trace-based 的平行程式除錯，就是記錄

程式的執行過程，並於事後將所得資料，利用輔助工具把程式執行過程中訊息傳遞的情形以視覺化的方式表現出來以提供程式設計者判斷是否有非預期的程式行為發生，並進而檢測競爭（race）的情況。採用 trace-based 技術的除錯工具有其基本的限制：它每次的分析只能在一種情形下運作；在 trace 的階段它仍無法避免對程式產生探測效應。Trace-based 的技巧通常利用一個 sensor (為內嵌於程式中的一段外加程式碼)[4]，而每當執行時則同時記錄想要的資料，因此這種方法較適用於各種特定資訊的取得。而我們的系統也正是採用這種技巧。

平行/分散式程式的另一特性就是缺少一個 global clock ，但是當我們分析程式的執行行為時卻必須為各個程序中的事件定義出它們之間的先後發生關係，因此 Lamport 發展出 logical time 的演算法[1]。Lamport 的演算法為每一個事件定義出一個唯一的 timestamp 而利用這個 timestamp 可以比較出各個事件發生的關係。Lamport 的 timestamp 雖然可以解決缺少 global clock 的問題，但是 Lamport 的 timestamp 必須被夾帶於訊息(message)之間傳遞，而夾帶的 timestamp 的大小與程序的數目成正比，這個方法並不太適用於當程序的數目太大時。因此我們發展 timestamp 對於平行中斷點的設定以及如何自動維持使用者所追蹤的多條路徑上的各個事件的一致性非常有幫助，此外我們的 timestamp 只需要一個 byte 的資料量。

二、 相關研究

國內外已發展出來和正在進行研究開發的平行/分散式除錯工具有許多，下面我們針對一些除錯系統做進一步的介紹，並依據它們的功能特色與本系統做比較。

p2d2 （Portable Parallel/Distributed Debugger）[5]是美國 NASA AMES 研究中心正在發展的一套平行與分散式除錯器。它採用主從式（client-server）的架構，在使用者界面與遠端服務（remote server）之間和遠端客戶（remote client）與除錯器之間定義協定（protocol），而使用者界面和除錯器之間，便以遠端服務和遠端客戶做為二者之間互動的介面。IVD（Interactive Visualization Debugger）[6]是在 Hewlett-Packard Research Labs 發展的除錯系統。它希望在 PVM 的程式發展環境中提供線上（on-line）除錯、效能分析及資料視覺化等多功能的整合性機制，其實作方法是利用在循序語言已存在的除錯工具，擴充其功能並整合至 PVM 的環境中。Mantis[7]是 U.C.Berkeley 從 1994 年開始針對 CM-5 機器上的 Split-C 這套平行語言所發展的除錯器，它以達成四個目標：rapid focus、scalability、economy of presentation 和 portability 為其設計的方針。中山大學所發展的分散式除錯器[8]是建立在 PVM 上，它提供圖形化的使用者界面。並利用高階抽象的表示方式表現程式行為。

表一 是我們針對上述的幾個除錯系統依其適用環境及功能所做的比較： Platform 這欄是指除錯器所適用的工作平台；Message Passing Environment 欄位是指除錯器所適用的訊息傳遞環境；On the fly 這欄是表示除錯器是否可提供使用者在程式執行過程中，動態的對程式做分析除錯；Replay 這欄表示除錯器是否提供 execution replay 的功能；Visualization 這欄表示除錯器是否提供視覺化的功能；Checkpoint 這欄表示除錯器是否提供 checkpoint 和 rollback 的機制；這機制可以使程式在 replay 的過程中，由先前做 checkpoint 的地方執行，而不需從頭執行。這部份的功能我們的除錯器目前已近於完成階段，而其它的除錯器則都未提供這樣的機制。

三、系統架構

我們的平行/分散式除錯系統提供了一個圖形化的操作界面。它是以交談式的除錯方式為主。交談式除錯模組是提供傳統循序語言中常見的除錯方式，trace/replay 機制是為了解決平行程式中 non-determinism 的問題。而 checkpointing 模組是針對需長時間執行的程式所提供的功能。

整個交談式平行/分散式除錯系統的架構如圖一所示，當程式被執行以後，它會在每個節點（node）上各產生（spawn）一個程序（process），這時我們也在每個節點上都各執行一個除錯器，並把產生除錯器的程序視為被偵錯的對象。

在 master node 中的 execution control manager、data visualization manager、mutli-level process controller 和每個 node 上的 debugger 是屬於交談式除錯的功能模組。master node 中的 collector、dispatcher 和每個 node 上的 agent 是屬於 trace/replay 功能模組。以下我們分別針對系統中的每個模組所扮演的角色及提供的功能做更詳盡的說明。

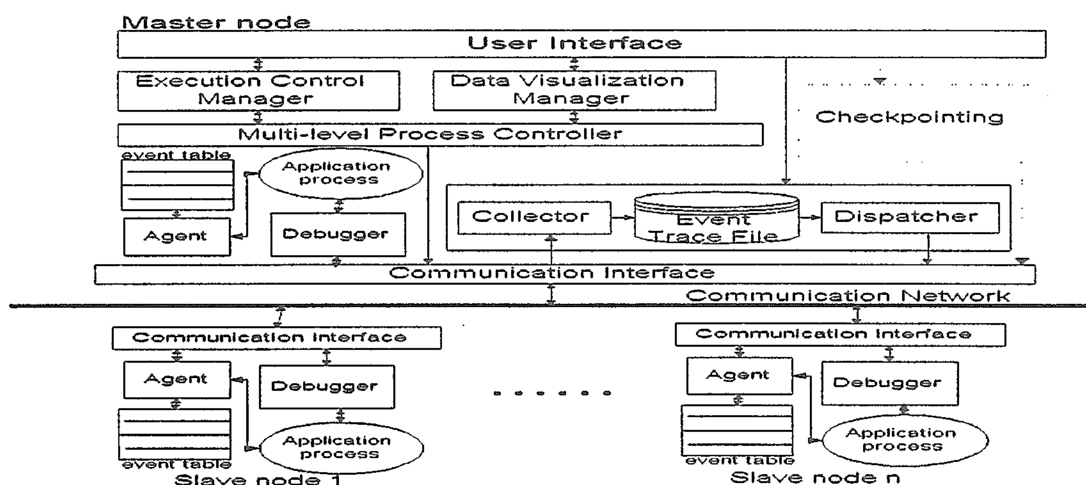
- 交談式除錯器人機界面：圖形化的使用者操作界面(graphical user interface)的提供，是為了取代傳統操作複雜且需背誦指令的命令列模式。因為所使用的是 source level 交談式除錯器，因此我們在視窗界面提供了程式原始碼，並藉以輔助下達除錯命令。為系統賦予易學、易操作的使用界面。
- 程式執行控制管理者：程式執行控制管理者（execution control manager），提供傳統除錯器常使用的除錯功能，如設定中斷點、單步追蹤等。因為純粹循序性錯誤也會出現在平行處理的過程中，當程式發生錯誤後，使用者常需要檢視更進一步的資料，控制程式執行的流程，瞭解其狀態之變化，逐步找出問題的癥結。
- 資料視覺化管理者：資料視覺化管理者（data visualization manager）在視窗界面提供了程式原始碼，並可藉以輔助下達除錯命令。資料視覺化管理者並且負責將除錯過程

中，查詢的相關資料及除錯器的狀態訊息顯示在界面上。

- 多階層程序控制器：如何對多個正在執行的程序作控制，是屬於操作界面考慮的重點之一，在本系統中，透過多階層程序控制器（mutli-level process controller）的控制，決定將使用者下的指令，送給那些節點上的除錯器。使用者可選擇對單一、所有或被選擇的某幾個 node 上的程序下命令。
- Trace/Replay 機制：在 trace 階段，collector 一開始便會開啟檔案準備記錄事件，當某兩個程序間有訊息傳遞的事件發生時，資料送出者把資料送到後，接收端並不馬上接收資料，而會由代理者（agent）先將資料的來源作記錄，存放在各個節點的記憶體中，等程式執行結束時 agent 會將 event table 的資料送到 collector，collector 收集每個 agent 上的資料，並將它儲存在檔案中。在 replay 階段，程式一開始執行 dispatcher 便從檔案中讀取資料，並將資料放在每個 node 上的 event table，當 application process 有接收訊息的事件發生時，再由 agent 從 event table 中讀取資料，決定所要接收的資料來源。
- Checkpointing：因為需要利用分散式處理的運算程式，通常都需要很長的執行時間，因此我們希望能提供 checkpointing 技術，如果程式在執行過程中受外在因素發生錯誤導致系統當機，使用者只需從最後一次做 checkpoint 的地方開始除錯，而不需再重頭執行，這對除錯工作將有很大幫助。

	Panorama	SD^2	p2d2	IVD	Mantis	NSYSU	Ours
Platform	nCube/2 iPSC/860 Paragon	NOW	NOW	NOW	CM5	NOW	NOW
Message Passing Environment	message-passing multi-computer	socket system call	PVM, MPI	PVM	Split-C	PVM	MPI
On the fly	Yes	No	Yes	Yes	Yes	Yes	Yes
Replay	No	Yes	No	Yes	No	Yes	Yes
Visualization	Yes	Yes	No	Yes	No	Yes	Yes
Checkpoint	No	No	No	No	No	No	Yes

表一、 平行除錯器功能之比較



圖一、 除錯系統架構圖

四、平行/分散式中斷點與多條執行路徑的追蹤

一般為了瞭解平行/分散程式的執行行為，常將程式的執行行為以 time-space diagram 表示， 在這一節中我們將以 time-space diagram 代表一個平行/分散程式的執行行為。在文章中我們將介紹如何將 time-space diagram 轉換成 task graph，以及如何以 task graph 來完成平行/分散式中斷點的設定與多條執行路徑的一致性行為的追蹤。

- **Task graph**

在這一小節中我們將介紹如何將特定的 time-space diagram 轉換成 task graph 並且給每一個 task 一個標籤做為 task 被重新執行 (replay) 時的先後關係，以及利用 task graph

和 task 的標籤所推導出的原理，來應用在平行中斷點設定以及數條被追蹤的執行路徑間一致性的控制。

定義一： Time-space diagram 到 task graph 的轉換；下面的規則可用來將一個特定的 time-space diagram 轉換成 task graph。

1. First task：對於每一個在 time-space diagram 的程序的第一個 task 是由程序的起始區段開始到第一個事件(send or receive event) 發生為止(包含此事件)或一直到程序結束為止，形成一個 task。
2. Recursive definition of tasks：對於一個未位於任一 task 之 time-space diagram 的程序區段由先前 task 之後開始然後到另一個事件(send or receive event)發生為止或是一直到這個程序的終止，這樣一個區段形成一個 task。
3. Edge：保留原先的 message-passing 的關係；同時為發生於同一個程序間的 task 循序的加上 edge。

由定義一中我們得知，每一個程序中的 task 除了最後一個 task 不包含任何事件(send or receive event)外，其它 task 都包含一個且僅有一個事件。

由前面 task graph 的定義我們可以推導出下面的定理：

定理一：每一個 time-space diagram 只能對映到唯一的一個 task graph。(證明省略)

定理二：Task graph 中並不存在迴圈。(證明省略)

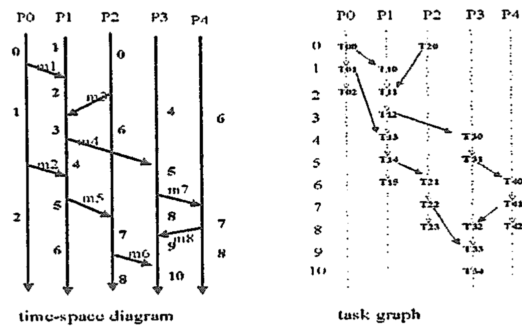
以下我們介紹如何給每一個 task 一個標籤(timestamp)，並利用此標籤定義出一個 task 的優先關係。在定義每個 task 的標籤之前我們先做一些假設：

1. 一個 send event 會將目前所在 task 的標籤隨著 message-passing 傳遞到其它的事件中。
2. T(i, j) 表示 process i 的第(j+1)個 task。
3. Priotity_T(i, j) 表示 process i 的第(j+1)個 task 的標籤。

定義二：標籤的設定。對於一個執行中程式的每一個 process i 我們做以下的設定。

1. 第一個 task 標籤的設定：若 T(i, 0) 中包含著 send event 則 Priority_T(i,0) 設定為 0；若 T(i, 0) 中不包含任何事件(即 T(i, 0)代表整個 process i)則 Priority_T(i, 0) 設定為 0；若 T(i, 0) 中包含著 receive event 則 Priority_T(i, 0) 等於 t+1，其中 t 為此 receive event 接受到的訊息中所夾帶的標籤值。
2. 中間 task T(i, j)標籤的設定：若 T(i, j) 中包含著 send event，Priority_T(i, j) 等於 Priority_T(i, j-1)+1，而此 send event 會將 Priority_T(i, j)夾帶於訊息中送出；若 T(i, j)中包含著 receive event，Priority_T(i,j) 等於 [Priority_T(i,j-1)+1]和(t+1)的最大值，其中 t 為此 receive event 接受到的訊息中所夾帶的標籤值。
3. 最後一個 task T(i, k)標籤的設定：最後一個 task 的 Priority_T(i, k) 等於 Priority_T(i, k-1)+1

圖二為一個將 time-space diagram 轉換成 task graph 的例子，其中 P_i 代表 process i ，i = 0 到 4；m_j 代表一 message j，j = 1 到 8；T_{ab} 代表 T(a, b)； time-space diagram 中各個程序區段旁的整數代表此相對應 task 的標籤 ；而在 task graph 中其右邊的 0 到 10 為各個 task 所對應的標籤值，例如：Priority_T(0,2) = Priority_T(1,1) = 2；由圖中我們可觀察出若 Priority_T(a, b)等於 Prioity_T(c, d)時，則 T(a,b) 與 T(c,d) 是互相平行的兩個 tasks 亦即沒有 happened-before 關係存在 【happened-before relation 是由 Lamport [1] 定義】。



圖二、time-space diagram 與相對應的 task graph

定理三：對於任一程式 P 由定義一中所訂出的任意兩個 tasks T(i, a)、T(j, b)若 Priority_T(i, a)小於或等於 Priority_T(j, b) 時，則 T(i, a) 與 T(j, b) 不是互相平行的兩個 tasks 就是 T(i, a) happened-before T(j, b)。(證明省略)

● 平行/分散式中斷點的設定

中斷點的設定主要是用來讓程式設計者觀看程式的執行行為，但是對於平行/分散式程式而言因為缺乏 global time 的特性無法分析出各個程序上的各個事件的先後發生關係導致如何設定平行中斷點成為平行/分散除錯器中不易掌握的一環，以下將介紹我們的系統中如何設定平行中斷點。

由定理三中我們得知若同一個程式中的任意兩個 tasks T(a, b)、T(c, d) 若 Priority_T(a, b) = Priority_T(c, d)則我們可以斷定 T(a, b)與 T(c, d) 為互相平行的兩個 tasks，而在 T(a, b) 與 T(c, d)分別設 BR_i，BR_j 中斷點，則 BR_i 與 BR_j 會形成一個一致性的中斷點。基於這個方法我們推導出定義三：一致性平行/分散式中斷點的設定。

定義三：一致性平行/分散中斷點的設定

若 G 為一個 task graph 而且 task graph 中每一個 task 都已經利用定義二的方法設定了標籤，並假設我們欲以 process i 中的一點 br 為基準點設定一個一致性的平行/分散中斷點，其中 br 位於 T(i, a)中且 Priority_T(i, a)= k，以下的規則為設定的方法。

對於 G 中的每一個 process j ，其中 j 不等於 i ：

1. 假若 Priority_T(j,0) >k ，則在 T(j,0)包含的事件(send or receive event)之前設定一中斷

點。

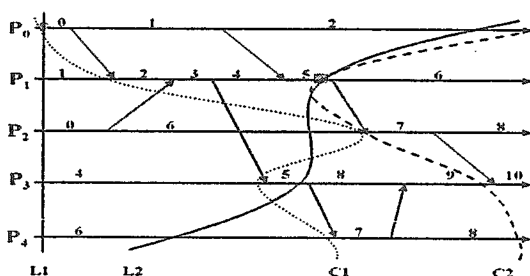
2. 假若存在 T(j, a)、T(j, a+1) 屬於 process j 而且 Priority_ T(j, a)<k<Priority_ T(j, a+1)，則在 T(j, a) 、T(j, a+1) 之間設定一中斷點。

3. 假若 Priority_T(j, α) < k ，其中 T(j, α) 為 process j 的最後一個 task ，則在 T(j, α) 之間設定一中斷點。

4. 假若存在 T(j, a)=k，則在 T(j, a)之中設定一中斷點。

此時，所有的中斷點為形成一個一致性的中斷平面。

圖三是以定義三的方法設定出的中斷平面與一般由因果關係所形成的中斷平面上的比較，由圖三中我們可知我們的方法在相同的條件下會比由因果關係所形成的中斷平面較早呈現出整體程式的一致性行為，亦即為較快的反映出使用者所想觀看的程式執行行為，因為我們的方法是以一個中斷點為基準點然後找出其它和這個點可以被同時執行的點來形成一個一致性的平行/分散式中斷平面；因果關係所形成的中斷平面是以一點為中斷點而導致其它程序因為這一點的影響而無法繼續執行。



L1 是以◎為基準點由我們的演算法所設定中斷點平面；

C1 是以◎為基準點的因果關係中斷平面；

L2 是以▣為基準點由我們的演算法所設定中斷點平面；

C2 是以▣為基準點的因果關係中斷平面。

圖三、我們的平行中斷點與因果關係中斷點

在除錯的過程中程式設計者往往會追蹤某些程式的執行路徑，但是因為沒有好的方法來維護所有路徑上各個事件的一致性，常常需要要求程式設計者不斷回到其它並不是目前所觀測的路徑上執行某些事件來保持所有被追蹤路徑的一致性[9]。而我們的方法為程式的執行行為重新定出了一種新的 task graph 而且找出各個事件的因果關係，所以當程式設計者同時追蹤數條執行路徑時我們可以很輕易的達成各被追蹤路徑一致性。

我們達成各被追蹤路徑一致性的敘述如下。假設 G 是一個已經被追蹤後程式的 task graph，而 path 1 到 path n 是由 G 中選取的 n 條可達成的路徑，路徑追蹤的過程中，在同一個時間中我們只允許使用者選取某一路徑中的單一個 task 做觀察。當使用者選取一 task T (在此假設 T 的標籤為 δ)做為觀察時，每一個程序自動往前執行直到所有程序中小於或等於標籤值為 δ 的 task 全部被完成後停止。而系統只回應位於 path 1 到 path n 等 n 條路徑中的 task 狀況給使用者，觀看完 task T 後假設使用者又選取位於 n 條路徑中的另一個 task $\underline{T}$ (在此假設 $\underline{T}$ 的標籤為 $\underline{\delta}$)作為觀察時，這時系統的做法為下：

1. 假如 $\delta \leq \underline{\delta}$，則每個程序由之前所停在的位置繼續往前執行，直到每一個程序所有程序中小於或等於標籤值為 $\underline{\delta}$ 的 task 全部被完成後停止。相同的，系統只回應位於 path 1 到 path n 等 n 條路徑中的 task 狀況給使用者。

2. 假如 $\delta > \underline{\delta}$，則系統將整個程式由頭開始再重新執行，直到每一個程序所有程序中小於或等於標籤值為 $\underline{\delta}$ 的 task 全部被完成後停止。同樣的，而系統只回應位於 path 1 到 path n 等 n 條路徑中的 task 狀況給使用者。

如果一直重複前面的步驟，即可達成同時追蹤數條執行路徑並保持各被追蹤路徑一致性的目的。

五、 系統實作

為了能使具有非決定性(non-deterministic)分散/平行程式的除錯工作能順利進行，反覆執行的過程必須能夠保證原本執行行為的重新出現外，我們提供了傳統式trace/replay機制，來達成上述的目標。我們亦利用MPICH中的 wrapper 功能，修改 MPICH 中的啟始（MPI_Init）、結束（MPI_Finalize）及可能導致非決定性的所有的通訊函式。因為這些被修改的函式都是經過特殊的包裝（wrapped），故具有透通性，換句話說，不必因為做trace/replay而需要更動某些原始程式。

在trace階段，我們先追蹤程式執行行為所得到的訊息資訊，依其發生的先後順序存放在各個節點的區域記憶體(local memory)中，等程式執行結束後，再從各節點中收集這些資料，然後存到同一檔案中。如此，可避免在程式執行時增加因收集這些資訊而增加通訊上的負擔。由於trace的資料只會記錄和訊息來源的相關資料，並不包含訊息本身的內容，所以儲存資料時，並不會對記憶體及磁碟空間造成太大的負荷。

在replay階段，每個節點會從先前所記錄的trace檔案中讀取與其有關的資料，並將它以表單的形式儲存於記憶體中。當程式執行過程中，如果通訊所造成race condition 成立時，便去查詢此一表單，以決定該程式中每個訊息發生的先後順序。藉此我們便可確保在replay階段中，每一次程式執行的行為是決定性的(deterministic)，而使一般在傳統循序式程式偵錯時，最常採用的反覆除錯(cyclical debugging)的策略，得以應用於除錯分散式/平行程式上。

以下我們將針對系統實作上的一些方法與技巧加以說明：

- **Profiling Interface** ： MPI profiling interface 的目的，是為了使撰寫 profiling tools

的工作變得更容易。因為 MPI 是與機器無關(machine independent)的標準，所以若對 MPI 設計 profiling tools 需要對實作在某些特定機器上的 MPI 系統的原始碼做修改是很不合理的。因此為了方便使用者對 MPI 函式做包裝，MPICH 提供了包裝產生器(wrapper generator) 的輔助工具，稱為 "wrappergen"。在 MPI 的函式重新包裝的過程中不需對下層的實作系統做修改，大大的增加了系統的通透性與 MPI 程式的發展空間。

- 交談式除錯功能：當 MPI 程式執行到呼叫除錯器的函式時，它會在每個節點上都分支（spawn）出一個除錯器，這個除錯器並把分支出它的程序視為被偵錯的對象。我們採用 gdb（GNU debugger）除錯器做為在每個節點上的基本除錯器，而 gdb 原來是由 GNU 為傳統循序語言所發展的 source-level 交談式除錯器，它是目前在 UNIX 中非常普遍的除錯工具。若使用者熟悉 gdb 則對本系統必可輕鬆上手。交談式除錯系統的使用者界面是用 Tcl/Tk 及 expect 這個 Tcl extension language 而設計的。

我們的除錯系統中提供的除錯功能包含了基本除錯功能與進階除錯功能兩種：(1) 基本除錯功能方面包括了幾個比較常用的功能，屬於執行控制的有 continue、step 和 next。(2)進階除錯功能方面，我們在視窗界面提供了程式原始碼並加上行號，這除了可以方便使用者觀察原始程式，並藉以輔助下達除錯命令。這可使 gdb 這個 source level 的除錯器使用起來更得心應手。

在平行程式操作界面另外還需考慮的一個重點是如何同時控制多個程序，在我們的除錯器中可以允許使用者分別對單一程序下命令；對所有的程序下命令或從使用者界面中選擇要那些程序接收命令，透過這樣不同層級的指令選擇，可為系統賦予易學、易操作的使用界面。

六、結論與未來工作

我們的平行/分散式除錯系統是針對 MPI 的環境，而所完成的包含了交談式的功能和 execution replay 的功能。使用者對於有問題的平行分散式程式，可選擇使用最常見的交談式除錯方法。如果需要做 cyclical debugging 且為了避免平行/分散式程式中 non-deterministic 的問題，使用者可以利用 execution replay 的功能。

目前除了繼續的實現文章中提出的平行中斷點、多條執行路徑的同時追蹤等方法外，我們亦同時在著手於 checkpointing[10-11]機制的整合以及程序間因為競跑現象(race condition)所可能產生的隱藏性行為。當然，如何盡量減低追蹤程式對程式產生的探測效應(probe effect)也一直是我們所持續研究的主題之一。

參考文獻

[1] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of ACM, 21(7), pp.558-565, July 1978.

[2] S. K. Damodaran-Kama and J. S. Brown, "Towards Heterogeneous Distributed Debugging," Technical Report, LAUR-95-906, Los Alamos National Lab., Los Alamos, NM-87545, USA, 1995.

[3] S. K. Damodaran-Kama, "Testing and Debugging Nondeterministic Message Passage Parallel Programs," Ph.D. Dissertation, Department of Computer Science, University of Southwestern Louisiana, Spring 1994.

[4] David M. Ogle, Karsten Schwan, and Richard Snodgress, "The Dynamic Monitoring of Distributed and Parallel System," IEEE Transactions on Parallel and Distributed Systems, 4(7), pp.762-778, July 1993.

[5] R. T. Hood, "The p2d2 Project: Building a Portable Distributed Debugger," Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools, May 1996.

[6] M. C. Hao, A. H. Karp, M. Mackey, V. Singh and J. Chien, "On-the-Fly Visualization and Debugging of Parallel Programs," Technical Report, Hewlett-Packard Research Labs, Palo Alto, CA, 1994.

[7] S. S. Lumetta and D. E. Culler, "Mantis: a Graphical Debugger for the Split-C Language," Technical Report, UC Berkeley, August 1994.

[8] Z. C. Huang, M. Z. Li and Z. X. Yang, "A Debugger for Distributed Programs," Proceedings of the Second Workshop on Compiler Techniques for High-Performance Computing, CTHPC'96, pp.195-203, Taiwan, R.O.C, March 1996.

[9] T. Kamada and A. Yonezawa, "A Debugging Scheme for Fine-Grain Threads with a Small Amount of Log Information," The University of Tokyo, Technical Report, 1995.

[10] G. M. Chiu, C. R. Young, "Efficient Rollback-Recovery Techniques in Distributed Computing Systems," IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 6, June 1996.

[11] R. Baldoni, J. M. Helary, A. Mostefaoui, M. Raynal, "Consistent Checkpointing in Message Passing Distributed Systems," Technical Report, IRISA Campus de Beaulieu, 35042 Rennes Cedex, France, June 1995.