Preventing Deadlock with Dynamic Message Scheduling

Ernesto Gomez, Yasha Karant, Keith Schubert

Department of Computer Science, California State University San Bernardino

Abstract

Although deadlock is not completely avoidable in distributed and parallel programming, we here describe theory and practice of a system that allows us to limit deadlock to situations in which there are true circular data dependences or failure of processes that compute data needed at other processes. This allows us to guarantee absence of deadlock in SPMD computations absent process failure. Our system guarantees optimal ordering of communication statements.

We gratefully acknowledge the support of the US National Science Foundation under Award CISE EIA 9810708 without which this work would not have been possible.

1: Previous work

Standard systems that deal with deadlock in database and operating systems, based on the conditions expressed by [5], rely on detecting deadlock cycles and pre-empting processes to break the cycle [1, 2, 3, 4, 5, 7, 8]. These methods are not applicable to parallel or distributed message passing systems, because pre-empting a process destroys any messages it holds; any other processes needing those messages will still wait forever. Other approaches include trying to make deadlock structurally impossible, which we find restricts the expression of parallelism [10, 8, 25]. Remaining alternatives are detecting deadlock communication patterns through static analysis of code, or simply relying on the individual programmer to not write code that will deadlock[9].

We here limit ourselves to the analysis of SPMD execution (introduced in [12]), in which multiple processes each execute the same code on different data. SPMD has become a common style of parallel program, implemented for example by the Planguages [25] and by a host of message passing systems, including MPI [9], PVM [15] and many vendor-specific implementations. This paradigm allows us to relate particular sends and receives at different process, and permits an analysis of data dependences.

In the SPMD model it is possible for each process to follow a different execution path through the code, if the execution path depends on data which may be different at each process. It is therefore possible that processes needing to communicate with each other will execute different sequences of communication statements, if execution is repeated with different data. Thus, static analysis of deadlock must take into account all possible combinations of execution sequences at every process; this is a problem of exponential complexity which is impractical to solve in all but the simplest cases. Further, it is necessary to write
communication statements in some order; blocking communication statements may prevent execution of later statements, even when those later statements have no data dependences. Although compiler analysis may allow re-ordering of communication statements without data dependences [17], we cannot in general guarantee that the rewritten code will in fact not deadlock for all possible execution paths.

We propose to re-order communication statements dynamically, through a combination of static analysis at compile time and a runtime system. Our methods are based on the overlapping protocol first described by one of us and Scott in 1998[26]. Overlapping was originally designed to dynamically schedule messages to compensate for load imbalances in irregular problems by opportunistic scheduling.

2: SPMD

An SPMD execution is a collection of multiple serial executions of the same code. We represent this code as a control flow graph (CFG) \( F = \{V, A, s, E\} \), where \( V \) is a set of nodes representing basic blocks, \( A \) is the set of directed arcs connecting them, \( s \) is the start node, and \( E \) is a set of end nodes [17]. Each basic block is composed in turn of a set of statements which must be executed together or not at all. We model serial execution of a program as a path in \( F \) from \( s \) to an end node \( e \in E \), stepping through the set of statements in each block. Program execution progresses through a series of transitions (steps) from one statement to the next, with each transition numbered sequentially starting from 0. Execution continues until the program reaches an end state or diverges. We define serial state in terms of statement in a CFG node and data in memory when that statement is executed:

Definition 2.1. Serial state and transition: The \( j \)th state of a serial execution is a tuple: \( \sigma_j = (x_j, D_j) \), where \( x_j \) denotes execution of statement \( x \) at step \( j \), and \( D_j \) is the set of data values stored in variables by the process when execution reaches \( x \) after performing \( j \) transitions from the start of execution. We denote a transition from a serial state to the next state by: \( \sigma_j \rightarrow \sigma_{j+1} \)

Every state implies execution of one statement, and every transition either remains inside a node of the CFG or follows an arc of the CFG. Basic blocks and statements inside them appear in an order determined by the CFG, however the state number \( j \) is a simple count of steps or transitions and always increases. A parallel SPMD state is then defined as a collection of serial states, and a parallel SPMD execution is a set of serial processes, each of which executes the same program text, and all of which begin execution (logically) together. We will assume a condition of serial progress: processes will not fail or halt until they reach an end state, unless they are waiting for communications, in order to confine our attention to communication problems.

3: Communication

We consider a typical semantics of reliable message passing to be given by blocking synchronous communication (Dongarra, et. al. [15]), characterized by a requirement that all participating processes wait at the communication statement, and must receive confirmation of success before being allowed to continue. This is the semantics of Hoare’s CSP
rendezvous communications \[13\], and has the added advantage that no buffers are required.

Blocking communications makes a message transmission into an atomic action. However, many systems (some examples are described in Silbershatz and Galvin\[1\], Tannenbaum \[4\], Coffman et. al.\[5\], MPI \[9\]) implement less restrictive modes through buffering or other protocols, and these modes may allow execution of more parallelism than strict blocking communications would permit. However, all message passing systems must eventually enforce some form of blocking to ensure determinism; it is at a minimum necessary for a process to wait for arrival of a message before it can use the received data.

Given the condition of serial progress 2, the only way an execution can fail is if some communication statement fails to complete. Let \(s_{j,p}\) be a statement executed at step \(j\) in process \(p\). Since \(s_{j,p}\) is blocking, for it to complete there must be a matching statement \(s_{k,q}\), \(p \neq q\) in another process. We denote the communication by \(s_{j,p} \iff s_{j,q}\), where the double arrow indicates both data and schedule dependences.

Assume that \(s_{j,p}\) is a receive statement which matches a send statement \(s_{j+d,p}\), \(d > 0\). Assume further, that \(s_{k,q}\) is a send statement, preceded by a matching receive statement \(s_{k-f,q}\), \(f > 0\). If these four statements are part of the same execution, then progress is not possible. Denote a schedule dependence in which one statement follows another in execution by \(\rightarrow\), the schedule involving these four statements is a cycle: \(s_{k,q} \iff s_{j,p} \rightarrow s_{j+d,p} \iff s_{k-f,q} \rightarrow s_{k,q}\).

This cycle is produced by a program text in which both \(p\) and \(q\) execute receive statements from each other first, followed by send statements; it is a classic deadlock in which each process is waiting for the other to send. In fact, neither \(s_{j+d,p}\) nor \(s_{k,q}\) would execute in this case, so their step numbers could only be computed by counting, for example, the number \(d\) of statements in the program text between the receive statement \(s_{j,p}\) and the send statement. If there are no data dependences between \(s_{j,p}\) and \(s_{k,q}\) at process \(p\) (or \(s_{k-f,q}\) and \(s_{k,q}\) at \(q\)) the order of either pair of statements can be reversed, breaking the cycle.

Note, however, that with blocking communications \(\iff\) we cannot reverse the order of both pairs of statements, since a send statement cannot complete before its matching receive does. The optimum solution would be to reverse the order of send and receive at the faster of the two processes \(p, q\), because a send logically initiates the synchronous send-receive action \(\iff\). However, process speed information is not available to the compiler, and in fact which of \(p, q\) is faster may be different in different parallel executions of the same program.

The problem is compounded by consideration of more complicated cycles involving more than two processes. It may even be the case that cycles such as this will appear in some executions of a program, but not in other executions of the same code on different data. For instance, if the pair of statements \(s_{k,q} \iff s_{j,p}\) were each enclosed in a conditional clause depending on some predicate \(x\) that was the same at both processes \(p, q\), deadlock would occur only if \(x\) was true.

4: Dynamic scheduling and static analysis

We formalize our notion of communication statements: Let \(c_{i,p}(D, P, T)\) be a statement of type \(T\) (e.g., send, receive, collective communication) involving variable(s) \(D\) and process(es) \(P\), executed at process \(p \in P\) in step \(i\). Note that, in SPMD, this information is local to each process, but in particular the variable names \(D\) are replicated at every process.

Static analysis gives us the following:
Lemma 4.1. Given a pair of communication statements $c_i$ and $c_j$ without any data dependences; let there be a statement $s_k$, $i < k < p$ such that it is on an execution path from $c_i$ to $c_j$. Static analysis allows us to move $s_k$ so it is not executed between $c_i$ and $c_j$.

Proof. Since $c_i$ and $c_j$ have no data dependences, $s_k$ can at most have a dependence with one of $c_i$ and $c_j$. Therefore $s_k$ can be moved so its order is interchanged with the communication statement with which it has no dependence.

We introduce a dynamic scheduling runtime system $R$ with the following properties:

1. A communication statement $c_{i,p}(D,P,T)$ is a declaration to $R$ that the statement is ready to be executed.
2. In an SPMD execution, $c_{i,p}(D,P,T,p)$ is matched by a known $c'_{j,q}(D',P,T')$; since communication is blocking, execution of such statements must occur at every process in $p$ for the statements at each process to complete.
3. At each process $p$, $R$ is provided with information on usage of variables in $D$. If the actions defined by $T$ require that $c$ be executed to ensure correctness (for example, if $T$ is a receive of $D$ and the process needs to read $D$), $R$ blocks $p$ until the communication completes.
4. $R$ may hold multiple uncompleted communication statements at each process $p$; execution of $R$ checks every pending statement and executes those that are matched at other processes in any order.

Property 2 is needed to guarantee program execution, since we need to ensure that every communication statement will be matched by a statement at some other process; otherwise individual statements may not complete.

Property 3 ensures correctness of code, by blocking program execution where data constraints would be violated.

We can now show:

Theorem 4.2. Given a sequence of communication statements $S_p = <c_{i,p}(D,P,T)>$ that have matching communication statements $S_q = <c'_{j,q}(D',P,T')>$ at processes $q \neq p$, and such that there are no data dependences $c_{i,p}(D,P,T,) \Rightarrow c_{j>1,p}(D,P,T)$ between each pair of statements in $S_p$, then a combination of static transformation by Lemma 4.1 and dynamic scheduling by $R$ is capable of rearranging the execution of $S_p$ to match the actual order of $S_q$. Furthermore, $R$ can execute a schedule which can not be produced by static, compile time analysis.

Proof. If $R$ does not block process $p$ by property 3, then the theorem is true by property 4. By Lemma 4.1, if the appropriate static analysis has taken place, there are no statements in $R$ other than communication statements that can cause $R$ to block execution. However, since $S_p$ has no data dependences, property 1 requires that each communication statement be enqueued without halting the process. Once communication systems are held by $R$, they are executed (property 4) in the order in which matching communication statements occur at other processes.

Static analysis alone can not always match the schedule executed by $R$ because it must take into account all possible execution paths. This will in general introduce additional data dependences and prevent re-ordering.

Corollary 4.3. Absent data dependences, $R$ will not deadlock.
5: Implementation and conclusions

SOS, the runtime system described in [26, 6] has the properties of $R$ here given and therefore is subject to theorem 4.2 and its corollary. The deadlock avoidance properties of SOS were first described, giving experimental results, in [27]. In the present system, there is substantial communication overhead as compared to raw communication costs, when tests are carried out on programs that perform no computation; delays are 10 to 100 times greater due to the extra messages needed to match sends with receives. When used in programs that perform substantial computation, the opportunistic scheduling performed by the runtime system minimizes synchronization wait times. Although direct communication costs are still higher, this has resulted in reduction of total program runtime to as little as one third of the time required using static scheduling.

It should be possible to prove that a combination of static and dynamic scheduling, as outlined here, will produce an optimal schedule for a particular SPMD execution, by minimizing wait times. This will be addressed in followup work to this paper.

We have developed the foregoing for SPMD execution because we can talk about states at different processes using the same notation and underlying CFG. Since all processes are executing the same program text, it is possible to analyze this text and verify that, given some set of communication statements $S_p$ at process $p$, that there is in fact a matching set of statements $S_q$ at other processes. The extension of our treatment to situations in which some processes are executing different sections of code within the same program (task parallelism) is treated in [6]. It should be possible to further extend the system beyond SPMD, but we may then be unable to guarantee the existence of $S_q$. Although the present analysis considers only synchronous messaging, extension to buffered message passing systems can be done by considering the buffer as a separate communicating entity, as is done in [16].

References


[27] Ernesto Gomez, Yasha Karant and Keith Schubert, ”Preventing Deadlock with Dynamic Message Scheduling”, in: ”Proceedings of the Hawaii International Conference on Computer Science”, pages 122-140, Honolulu, Hawaii, 2004. This is a precursor of this paper, under the same title but presenting experimental results and lacking proof.