Performance Evaluation of Self-Timed Dataflow in the ISPC-2 Hypercube

Mayez Al-Mouhamed (1) and Lubomir Bic (2)

(1) Department of Computer Engineering
King Fahd University of Petroleum and Minerals (KFUPM)
31261 Dhahran, Saudi Arabia

(2) Department of Information and Computer Science
University of California Irvine,
California 92717, USA.

Abstract: An approach for efficient execution of real-time computations (macro-dataflow) based on exploiting compile-time knowledge of computation and communication is proposed for the Intel iPSC-2 distributed-memory multiprocessor. The method is useful for synchronous dataflow computations that have predictable dataflow behavior. To reduce the run-time overhead, it is proposed to generate self-timed processor assignments based on estimating the execution and communication times and the use of a simplified model of packet exchange between the processors. At runtime, each processor executes an ordered list of computation and communication functions so that only the start is left to be decided at run-time. To further reduce the run-time overhead, each processor runs a local manager program to handle the arrival and storage of remote data in an optimized manner. The execution method does not restrict the communication order and allows remote data be received during task execution. Experimental evaluation of the iPSC-2 distributed-memory hypercube shows that: self-timed execution allows significant reduction of run-time overhead; and the predicted and run-time speed-up match at the 10% level when the ratio of communication to computation times (granularity) is below 18% for the iPSC-2. Stochastic testing shows that gross estimation of the computation parameters (within 30% error) leads to moderate degradation of potential speed-up, and degradation decreases with increasing problem size.
Performance evaluation of self-timed dataflow on the iPSC-2 hypercube

MAYEZ AL-MOUHAMED† and LUBOMIR BIC‡

An approach for efficient execution of real-time computations (macro-dataflow) based on exploiting compile-time knowledge of computation and communication is proposed for the Intel iPSC-2 distributed-memory multiprocessor. The method is useful for synchronous dataflow computations that have predictable dataflow behaviour. To reduce the run-time overhead, it is proposed to generate self-timed processor assignments based on estimating the execution and communication times and the use of a simplified model of packet exchange between the processors. At run-time, each processor executes an ordered list of computation and communication functions so that only the start is left to be decided at run-time. To further reduce the run-time overhead, each processor runs a local manager program to handle the arrival and storage of remote data in an optimized manner. The execution method does not restrict the communication order and allows remote data be received during task execution. Experimental evaluation of the iPSC-2 distributed-memory hypercube shows that: self-timed execution allows significant reduction of run-time overhead; and the predicted and run-time speed-up match at the 10% level when the ratio of communication to computation times (granularity) is below 18% for the iPSC-2. Stochastic testing shows that gross estimation of the computation parameters (within 30% error) leads to moderate degradation of potential speed-up, and degradation decreases with increasing problem size.

1. Introduction

Distributed-memory multiprocessors (DMM) are difficult to program (Gupta and Prithviraj 1992) because there is no global address space, thus the user has to handle the code and data distribution and explicitly manage the communication. Generally, a compiler for DMM systems (Chen et al. 1988, Rosing et al. 1990) searches for a compromise between minimizing communication, selecting suitable communication primitives and load balancing. Deterministic program behaviour makes it possible to perform most of the work at compile time (Lee and Bier 1990) leaving simple synchronization to run-time, thus removing most of the overhead associated with a general-purpose solution.

In dynamic dataflow (Dennis 1980) the task is assigned at run-time on any free processor provided that its operands are available. This offers the highest degree of flexibility but at the cost of significant run-time overhead, especially at the fine-grain level. For DMM systems, evaluation of dynamic and distributed allocation has been carried out (Shu and Kale 1989) for non-predictable dynamic behaviour of medium-grained processes such as trees used in AI. Using a network of workstations, dynamic demand-driven execution has been implemented (Jogannathan et al. 1989) by using, for each workstation, a set of communication programs for sequencing.
execution, propagating demands and data, and an allocator to decide where to execute a given node.

In static dataflow (Cornish et al. 1979) the compiler generates the processor assignment and the task execution time. The argument-fetching architecture proposed by Gao et al. (1988) presented a statically defined model in which the data arguments that become available fetch their segment of code. Another example of static assignment is the TI data-driven processor (Cornish et al. 1979) for which FORTRAN programs are translated by the compiler into dataflow static assignments. An example of mixing the static and dynamic approaches has been proposed by Keckler and Dally (1992) for statically scheduling coarse-grain threads. For shared-memory systems, this approach (Lee and Bier 1990) leads to the design of shared-memory that memorizes the access order of the processors.

Our approach is to extend the static assignment and self-timed execution to distributed-memory systems but without altering the hardware. The objective is to exploit knowledge of the computation and target multiprocessor in order to generate efficient self-timed execution with the least possible run-time overhead. For this, we propose static scheduling of computations and communications based on a model for packet-exchange for the iPSC-2 distributed-memory multiprocessor. At runtime, local-manager programs monitor sending/receiving of remote data and firing execution (self-timed) based on overlapping overhead with processor element (PE) computations. The approach has been extensively tested on the Intel iPSC-2 hypercube.

This paper is organized as follows: §2 presents assumptions on the computation and §3 discusses synchronizing execution on distributed-memory systems; §4 presents the proposed static scheduling and a model for the communication system of the iPSC-2; §5 presents the proposed execution scheme for the iPSC-2. Performance evaluation and comparison with other approaches are presented in §6.

2. Assumptions on the computation

Statically scheduled dataflow applies to a subclass of computation that has:

(a) static dataflow topology, for which the data producers of a given task are known at compile-time;

(b) deterministic behaviour, which requires that the semantic of the computation be preserved during execution;

(c) execution and communication times that do not vary greatly.

Even with these restrictions, many real-time applications fit into this class of synchronous dataflow. Examples of deterministic computations that fall into this class are digital signal processing (Sih and Lee 1993), robot-dynamics (Al-Mouhamed 1990a), LU decomposition and fast Fourier transforms (FFT) (Shirazi et al. 1990), vector-matrix operations, etc.

2.1. Dataflow description of computation

A program consists of a collection of tasks and each task has a set of input tokens, a task body and a set of output tokens. The inputs identify a subset of the program global tokens that are required to run the body of the task without preemption or context switching. The body of the task consists of arbitrary computation over the inputs and the local data and might include loop statements or
if-then-else constructs. Once all input tokens are received, the body can run until completion without the need of additional data. The outputs are the results generated by the task for use as input tokens by the successor tasks.

The single assignment principle ensures that any scalar or vector element is written only once and allows arbitrary different tasks to read that element in any order provided that any read occurs following the unique write. This allows the dataflow structure of the computation to be found because no global tokens may belong to the output of more than one task.

A fundamental requirement of each task is to guarantee that the operation of writing the value of all its outputs occurs regardless of the control path within its body. As a result, conditional branching is allowed within the body provided that the number and the identity of the outputs should always be unaltered by the possible control path through the task body. Conditionals may only introduce arithmetic dependence on the output. A task may call a local procedure or introduce data-dependent iteration according to the above principle. The drawback of this flexibility for a statically scheduled computation is that the estimation of the task time becomes difficult for the compiler, especially when the control path through the body becomes highly data-dependent.

3. Synchronization in message-passing systems

For shared-memory (SM) systems, static synchronization has been implemented by memorizing the order of memory accesses (Lee and Bier 1990) on the basis of accurate knowledge of the task times. For this, the bus arbiter statically memorizes the access order of the processors to shared memory. The SM is switched from one processor to another whenever an access is completed. This leads to minimum dynamic overhead but requires accurate estimates of the task times in order to preserve the potential speed-up. Another approach that relaxes the memory access order was presented by Al-Mouhamed (1990a). In this case, the gained flexibility on the static access order leads to increasing overhead associated with the use of traditional SM synchronization.

In message-passing systems where communications are more time-consuming, static synchronization based on memorizing the communication order is difficult to apply because it requires accurate estimates of task, communication and network times for a given program. Modelling the network and network delays for message-passing systems makes the static schedule less practical because of the difficulty of finding an analytical model for an arbitrary connected network and the complexity and uncertainty of a probabilistic model.

Message-passing systems do not provide efficient synchronization for fine-grain parallelism because they are inherently designed for coarse-grain computation. To implement efficient statically-scheduled computation on message-passing systems, the task execution should suffer the least from managing remote data transfer. Sending and receiving remote data requires a number of settings such as: select the best communication primitive, specify addresses of data arrays and destination PE, and receive remote data and storage in the appropriate address within the distributed memory of a given PE. Even with a statically prepared send and receive, local management of the remote data may compromise this approach unless part of the management time can overlap with the PE executing some task. In most message-passing systems, dedicated communication hardware is used to capture the remote
data before its transfer to the local memory. Therefore, the problem is to benefit from the static structure in order to maximize the overlap between computation and communication, and to reduce run-time synchronization by exploiting the available communication and synchronization functions for a given DMM.

Reducing synchronization is based on efficient local management of remote data and task execution that will be presented in §5. Maximizing the overlap between computation and communication is based on static scheduling, which will be presented in the next section.

4. Scheduling to maximize efficiency

A scheduling algorithm to minimize the overall time of precedence-constrained computation on a distributed memory system is presented (Al-Mouhamed 1990 b). The algorithm is based on minimizing the processor idle time and overlapping computation with communication while preserving the inherent data dependency among an ordered set of tasks. We present the notation used prior to defining the steps of the proposed algorithm.

A set \{T_1, \ldots, T_n\} of \( n \) tasks with their precedence-constraints and communication costs are to be scheduled on \( p \) identical processors to minimize overall execution time. The computation model (Rayward-Smith 1986, Hwang et al. 1989) is a directed-acyclic task graph where the node \( T \) is represented by its execution time \( \mu(T) \) and the edge \((T', T)\) represents the volume of data \( c(T', T) \) that should be transferred upon completion from predecessor \( T' \) to \( T \). The distributed-memory multiprocessor is denoted by \( S(P, R) \), where \( P \) is a set of processors and \( R = \{r(p', p)\} \) is the set of routing costs, where \( r(p', p) \) is the time to transfer a unit of data from processor \( p' \) to \( p \). Therefore, the time to transfer \( c(T', T) \) data operands is \( c(T', T)r(p', p) \), assuming the communication media is contention-free and \( T \) and \( T' \) are assigned to processors \( p \) and \( p' \), respectively.

Consider the hypercube interconnection network such as that of the iPSC-2 (Intel 1988 a and b) for which the communication cost \( r(p', p) \) between \( p' \) and \( p \) is proportional to the Hamming distance \( h(p', p) \). More precisely, we have identified a simple model for the time to transfer \( c(T', T) \) bytes between arbitrary processors of the iPSC-2:

\[
c(T', T)r(p', p) = \begin{cases} 0, & \text{if } p = p' \\ F(c(T', T)) + t_{\text{hop}}(h(p', p) - 1), & \text{otherwise} \end{cases}
\]

(1)

where, \( F(c(T', T)) \) is a function of the volume of data and the set-up time, \( t_{\text{hop}} \) is the time to hop from one processor to another, and \( (h(p', p) - 1) \) is the number of hops between \( p' \) and \( p \). The iPSC-2 has different packet formations depending on the volume of data. Depending on whether the packet is small (less than 100 byte) or large, the cost function \( F(c(T', T)) \) is identified by the following expression:

\[
F(c(T', T)) = \begin{cases} t_s + t_{\text{min}}, & \text{if } c(T', T) \leq 100 \text{ byte} \\ t_1 + t_u \cdot c(T', T), & \text{otherwise} \end{cases}
\]

(2)

where \( t_s \) and \( t_1 \) are the set-up times for small and large packets, respectively, \( t_{\text{min}} \) is the time to transfer a small packet, and \( t_u \) is the transfer time per byte for large packets. The values of the various parameters have been identified as follows: \( t_s = 100 \mu s \), \( t_{\text{min}} = 250 \mu s \), \( t_1 = 625 \mu s \) and \( t_u = 0\cdot36 \mu s \text{ byte}^{-1} \). These parameters assume
that the communication media is contention-free, which is found to be acceptable when at any given time the overall communication does not exceed 70% of the available bandwidth (100 Mb s⁻¹). The costs of referencing scalars and arrays and performing arithmetic operations has been separately estimated. The above model of the iPSC-2 enables the effects of computation and communication that are needed in the following static scheduling to be modelled.

Let T be a task and denote by Pred(T) the set of predecessors of T. The earliest starting time est(T, p) of T on some processor p depends on the finish time f(T') of each predecessor T' ∈ Pred(T), the number of messages c(T', T) sent from T' to its successor T, and the routing costs between p(T') and p:

\[
est(T, p) = \max_{T' \in \text{Pred}(T)} \{f(T') + c(T', T) \cdot r(p(T'), p)\}
\]

(3)

where \(r(p(T'), p) = 0\) whenever \(p = p(T')\) because no data transfer is needed within \(p\). In the event \(T\) has no predecessors (Pred(T) = 0), then \(est(T, p) = 0\) for every processor \(p\). As \(est(T, p)\) depends on the routeing cost \(r(p(T'), p)\), then there exists a processor \(p^*\), that is free at time \(t(p^*)\), for which task \(T\) can start at the earliest time \(est(T, p^*)\) among all processors:

\[
est(T, p^*) = \min_p \{\max \{est(T, p), t(p)\}\}
\]

(4)

The notion of ‘est’ is useful in the design of scheduling algorithms when the objective is to maximize the processor efficiency as the basis for minimizing the overall finish time. Selecting tasks based on earliest-startable-task first is one useful approach to overlapping communication with computation. The proposed algorithm uses set \(B\) to store the tasks that have been started on some processors and set \(A\) to store the tasks that have no predecessors or whose predecessors belong to \(B\). It is assumed function \(\lambda(T)\) is initialized to the number of predecessors of \(T\). The scheduling algorithm is as follows.

**Algorithm: earliest-startable-task (EST) first**

1. **Step 1.** Initialize: \(A\leftarrow \{T: \text{Pred}(T) = 0\}\), \(B\leftarrow 0\), for each task and each processor:
   \(est(T, p) = 0\), \(t(p) = 0\)

2. **Step 2.** While \(|B| < n\) Do
   Begin
   (2.1) Select \(T^* \in A\) and \(p^*\) that satisfy: \(est(T^*, p^*) = \min_{T \in A} \{\min_p \{est(T, p)\}\}\)
   (2.2) Assign \(T^* \in A\) and \(p^*\) and \(f(T^*) = est(T^*, p^*) + \mu(T^*)\), \(t(p^*) = f(T^*)\),
   Remove \(T^*\) from \(A\), Add \(T^*\) to \(B\), For each \(T \in A\), update: \(est(T, p^*) = \max \{est(T, p^*), t(p^*)\}\)
   (2.3) Repeat for each task \(T \in \text{Succ}(T^*): \lambda(T) = \lambda(T) - 1\),
   If \(\lambda(T) = 0\) then Update \(A: A\leftarrow A + \{T\}\),
   Evaluate \(est(T, p)\) for each \(p\):
   \(est(T, p) = \max \{\max_{T' \in \text{Pred}(T)} \{f(T') + c(T', T) \cdot r(p(T'), p)\}, t(p)\}\)
   End.
Step 1 places all tasks that can run without remote data in set A. Step 2.1 selects task \( T^* \) that can run at the earliest time, and Step 2.2 assigns it to its best-suited processor \( p^* \). Also, Step 2.2 updates the values of \( \text{est}(T, p^*) \) for each ready task, as a result of updating the time \( \lambda(p^*) \) at which \( p^* \) becomes free. Step 2.3 searches whether any successor task \( T \in \text{Succ}(T^*) \) of \( T^* \) is ready to run (\( \lambda(T) = 0 \)). It also evaluates the \( \text{est}(T) \) times for those newly ready tasks. Each task requires at most \( pn \) steps to evaluate its \( \text{est}(T, p) \) time, and \( n \) tasks require \( O(pn^2) \), that is, the time complexity of the scheduling algorithm.

The next section investigates the implementation of the proposed statically scheduled dataflow scheme by using the iPSC-2 message-passing system.

5. Implementing statically scheduled dataflow on the iPSC-2

The iPSC-2 (Intel 1988 a and b) is a distributed-memory multiprocessor with a hypercube interconnection network that allows the host and 32 nodes to communicate by using synchronous and asynchronous message packets. The theoretical bandwidth of the network is 100 Mb s\(^{-1}\).

To implement our statically scheduled computation, the host broadcasts (Load) the node program to all active PEs prior to sending (isend) the global program code and prompt for an acknowledge from every PE. The host requests all PEs to start computation by sending an synchronous start (csend) and executes an asynchronous receive in order to collect reporting results from each PE upon its completion. A special host processor-node is used to: generate graph inputs according to some description; schedule the graph inputs and obtain the processor assignments; broadcast the assignments to the processors and synchronize starting execution; collect results from the processors.

5.1. Model and generation

A graph generator is used to control the statistical and topological characteristics of loop programs. The model is a directed acyclic graph that consists of a number of levels. Each graph level corresponds to a loop that is distributed over a number of processors by strip-mining its outermost parallel iterator. A level consists of a number of nodes such that each node represents the computation time of a set of iterations. The node execution time results from estimating the time needed to reference local or remote arrays that have been transferred prior to running that node.

The edges can be of two types: semantic, and communication-dependent. Semantic edges are used to preserve the ordering of write-after-read operations in the program to enforce deterministic program behaviour. The use of semantic edges removes the need to implement barrier synchronization between the nodes of different levels. This guarantees that the scheduler generates the correct solution (deterministic) that preserves the ordering of the loops and the iterations. The communication edges represent the dependence of a node over its data-producer node, i.e. explicitly read-after-write dependence. These edges carry array communication that might result from imperfect array partitioning of the original problem. Each node is assumed to fully or partially (stride effect) reference the array elements of its communication edges in order to perform its arithmetic operations.

The graph generator enables control of the statistical characteristics of the simulated problem by setting the distribution parameters, such as the average
execution time of the nodes, the average communication time carried by the edges, the deviation from the above values, and the topological characteristics of the problem.

Each node can start execution following the completion of all remote data transfer and the satisfaction of the semantic edges. On completion, the task may include initiating data transfer towards its successor nodes. It is assumed that each processor has dedicated communication hardware so that computation need not interrupt transfer of data between the hardware buffer and local memory. This assumption is justified for the iPSC-2 because each processor has a dedicated node unit that moves each recently received array from its local memory to processor memory by using DMA. In this case, the processor should provide this unit the source/destination address within its local memory.

5.2. Scheduling the input computation

The scheduler operates on the input graph and transforms it to a set of sequential nodes that can run in parallel. The scheduler uses the heuristic EST in order to minimize overall execution time by locally maximizing the overlap between execution and communication. Outgoing arcs from each task are replaced by asynchronous communication primitives such as: sending data from one processor to another (one-to-one), multicasting data from one processor to many others (one-to-many), and one-to-all data broadcasting. These primitives are used to transfer data \( e(T', T) \) from task \( T' \) to its successor \( T \) whenever tasks \( T' \) and \( T \) are assigned distinct processors. Terminal tasks have reporting missions to the host processor.

5.3. Self-timed execution

The host program begins by sending to each processor a replicated copy of a local manager (LM) program whose mission is to supervise execution of each processor assignment. Next, each processor receives its own ordered set of tasks. Execution starts by simultaneously running all LMs.

The processor program is an ordered set of tasks to be sequentially executed upon data availability. This is referred to as self-timed execution because firing tasks is conditioned by testing the availability of all its remote data within the distributed-memory of the PE. The execution order of the tasks is maintained by every LM. Only the execution order of the tasks is known by LM, however; firing of the next task depends on the completion of the remote data transfer. On completion of every task, the LM checks whether all the remote data transfers have completed for the next task. Each task \( T \) has a variable \( \lambda(T) \) that is initially set to the number of data packets that should be received from the predecessor's tasks (different processors only) prior to starting \( T \). The variable \( \lambda(T) \) is decremented every time a data transfer is completed and made available in the processor memory. Task \( T \) becomes ready to run when \( \lambda(T) \) becomes zero.

To avoid adding overhead on the current task execution, the mission of the LM is limited to one byte decrementation. By checking variable \( \lambda(T) \) of the next task, the LM can rapidly detect whether \( T \) can start. The processor utilization is maximum when all the remote data arrive earlier than the completion of the previously running task. Otherwise, the processor remains idle as long as \( \lambda(T) \neq 0 \). The LM processes messages for all the remaining tasks; the destination task of a current transfer need not necessarily be the task that should run next.
5.4. Synchronizing communication

On the iPSC-2, the PEs may communicate using asynchronous send and receive as shown in Figs 1 and 2. The sending primitive should include: the identifier of the message (called type), the source data buffer in local memory, the length of the array, and the ID of the destination PE. Note that message identifiers (types) are global constants that are allocated to the edges of the dataflow graph following the scheduling.

Once a send is issued by a PE, that PE can continue processing without modifying the source buffer. A message-wait primitive msg_wait ensures that the source buffer has been read by the communication hardware but has not necessarily completed its transfer. Processing continues once message-wait returns and it becomes possible to perform write operations to that buffer. The receive primitive (Fig. 2) operates in a similar manner to the send. It is important that an initiated send is preceded or followed by a posted receive in the destination PE. Sending and receiving data operates as follows: the receive causes a physical path to be established; wait until destination is ready to receive the data; perform the physical transfer to the destination PE through the network; and, use the DMA to store the message in the destination address in the local memory. Note that significant overhead might be incurred with this approach if receiving communication packets cannot overlap in time with PE execution.

In our case, the synchronization of the communication between tasks is performed as follows. The scheduling generates static assignment for each PE so that each assignment is a time-ordered sequence of communicating tasks. Each task consumes a number of data elements that have been produced by its predecessor tasks. The predecessor tasks can be part of the static assignment of arbitrary PEs. Therefore, the need is to transfer the data from the producer PE to the consumer PE. Normally, each task starts by receiving the data needed and ends by sending its produced data to remote PEs. In this case, the task processing will be delayed by the time to transfer the remote data to the appropriate buffers as a result of executing receive primitives just before starting. This approach was used by Wu and Gajski (1990). The scheduling assigns the task to run at its EST, which generally depends on the last data transfer, but other transfers complete much earlier than the starting time. Therefore, the order of receiving data by a PE should not be constrained by the order of the tasks. In our approach, remote data can be received in any order, thus

\[
\text{msg-id = isend(type, buf.ptr, len, nod.pid)}
\]

- Perform processing that does not modify the output buffer (buf)

\[
\text{msg-wait(msg-id)}
\]

- Perform processing that might modify buf.

Figure 1. Asynchronous send (isend) from one node to another.

\[
\text{msg-id = irecv(type, buf.ptr, len)}
\]

- Perform processing that does not modify the input buffer (buf)

\[
\text{msg-wait(msg-id)}
\]

- Perform processing that might use buf.

Figure 2. Asynchronous receive.
While \{tasks\} \neq \emptyset do

begin

msg_id=0

\text{Until}(type \in \text{Type(Next\_task)} \text{ OR } \text{Type(Next\_task)} = \emptyset)

\text{type}=\text{infotype}; \text{End}

\text{If } type \neq 0 \text{ Then}

\text{msg}_\text{id} = \text{irecv}(type, \text{buf}, \text{ptr}, \text{len}); \text{c\_task}=\text{task}(type); \text{endif}

\text{If } \lambda (\text{next\_task}) = 0 \text{ Then}

\text{Run next\_task}; \text{endif}

\text{If } \text{msg\_id} \neq 0 \text{ Then}

\text{msg\_wait}(\text{msg\_id})

\lambda (c\_task) = \lambda (c\_task) - 1; \text{endif}

end

Figure 3. Local manager routine.

increasing the flexibility of the approach and also offering some opportunity to overlap computation with communication, as will be explained below.

The main concern is to overlap communication with computation and use a non-blocking message exchange scheme. For this, the LM initiates a receive operation prior to starting a task. This ensures that one receive operation, if there is one, can be performed prior to running a ready task. The LM routine is shown in Fig. 3. First it probes the communication hardware and gets the type of the pending message to be received (Statement Until). In the event more remote data is needed (Type(Next\_task) \neq 0) to run the next task, it checks whether any send is pending for the next task. That send will then be given priority over other sends. In either case, it returns the type of the pending send if there is any (type \neq 0). If there is some message waiting to be received, then a receive is initiated (irecv) and the running of the next task is attempted.

The next task can run (see Fig. 4) until completion. In the event the next task cannot run (\lambda (next\_task) \neq 0) because the remote data required by it is not available, then the system waits (msg\_wait) until the current receive is complete. The function \lambda(task) gives the number of remote receives (also the number of remote predecessors) that should complete prior to running the task. Next, the task for which the last receive was transferring data will get its \lambda decremented by one as a result of the completion of the previous receive.

During the waiting time (msg\_wait), the local bus of the PE will be busy performing DMA transfer into the local memory. There is no benefit from issuing more than one receive at a time. The next task is not delayed during execution of msg\_wait because either: the current data transfer is for one of its operands, the next task is run simultaneously with the receive of data for another task, or the next task cannot run anyway because its remote data is not yet available. Clearly, the next task waits only when there is still some data transfer for it that is pending. But in all

Task Body:

\text{task processing}

Repeat for each produced data type:

isend(type, buf, ptr, len, nod, pid); \text{end}

Figure 4. Task processing and transfer of produced data.
<table>
<thead>
<tr>
<th>Case</th>
<th>Tasks</th>
<th>Levels</th>
<th>Arcs</th>
<th>Comm</th>
<th>Comm/Comp</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>200</td>
<td>10</td>
<td>200–300</td>
<td>1–8 kbyte</td>
<td>0.11</td>
<td>17</td>
</tr>
<tr>
<td>1.2</td>
<td>160</td>
<td>10</td>
<td>150–250</td>
<td>1–8 kbyte</td>
<td>0.14</td>
<td>14</td>
</tr>
<tr>
<td>1.3</td>
<td>100</td>
<td>10</td>
<td>100–200</td>
<td>1–8 kbyte</td>
<td>0.18</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>10</td>
<td>200–300</td>
<td>1–2 kbyte</td>
<td>0.15</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>160</td>
<td>10</td>
<td>160–320</td>
<td>1–100 byte</td>
<td>0.25</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>10</td>
<td>80–240</td>
<td>1–100 byte</td>
<td>0.15</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>10</td>
<td>40–160</td>
<td>1–100 byte</td>
<td>0.10</td>
<td>2.5</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>8</td>
<td>16–64</td>
<td>1–100 byte</td>
<td>0.07</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Summary of test cases.

cases, the next task runs after the LM initiates some data receive, whenever there is some pending send, in order to maximize the overlap between task processing and message communication.

6. Performance evaluation

Evaluation of this work was carried out by comparing static and run-time results versus the change in communication and parallelism, analysing the effect of uncertainties on the computation parameters, and comparing this approach with other proposals.

6.1. Comparison of static and run-time results

Many test cases were executed on the hypercube and the results of a representative set are analysed in this section. The Table summarizes the representative cases. The number of tasks in a given problem is listed under the column Tasks, Levels denotes the number of levels in the dataflow graph, Arcs is the range of the number of edges, Comm is the range of communication carried by each edge, Comm/Comp is the average communication to computation, and Parallel is the average number of tasks that can run in parallel at any time. By changing the characteristics of the problem generation, the communication and parallelism can be increased or decreased in order to study the effect of the proposed self-timed execution.

Cases 1.1–1.3 and case 2 represent applications requiring significant data transfer (large packets) such as the matrix multiply and the N-D convolution. Cases 3–6 are typical of moderate communication requirements (small packets) such as robot dynamics, FFT and other DSP applications.

Each case consisted of ten problems that were generated, statically scheduled, and their corresponding assignment run on the iPSC-2 hypercube with an incremental number of processors. Ten run times, within the same test case, for each problem were then averaged. These points were used to plot speed-up versus the number of processors. The plots in Fig. 5 (case 1.2), Fig. 6 (case 2) and Fig. 7 (cases 3–6) compare the speed-up obtained from the static scheduler with that obtained from the run-time.

It can be observed that in all cases shown the run-time speed-up matches the speed-up obtained with the scheduler at the 10% level (except for case 3 (Fig. 7), which will be discussed later). This close matching is due to three reasons:

(a) the medium-to-coarse granularity of these problems (Comm/Comp ≤ 0.18);
(b) an accurate model of communication ((1) and (2)) that has been used as the cost function in the evaluation of the EST;

(c) the use of an efficient execution approach (LM) that has a low run-time overhead, does not restrict the communication order, and has the ability to overlap execution with receiving of remote data whenever possible.

However, finer task granularity does not allow the same deviation on the speed-up between run-time and scheduler to be maintained. For example, case 3 with Comm/Comp = 0.25 caused the run-time speed-up to decrease by nearly 12%, and this deviation increases as finer-grain tasks are run. The interesting point is to find the limit of task-granularity for each type of packet (small or large) for which this execution approach gives efficient speed-up or no more than 10% degradation.

As expected, the speed-up was dependent on several factors: the inherent parallelism of the problems, the average communication-to-computation ratio, and the number of PEs used. The results from case 1 (Fig. 5) exhibited almost linear speed-up to about eight PEs, then dropped off to approximately 80% from linear using 16 PEs. In the case of a large number of data transfers, the performance of the scheduler depends on the number of processors and how execution can be overlapped with communication. Any misplacement of the communication strongly affects the overall finish time of the static schedule. The run-time follows the static assignment and was affected by these fluctuations as shown in Fig. 5, and with lesser effect in Fig. 6.

Depending on the inherent parallelism, the number of PEs used lowered the completion times significantly (for high parallelism) or slightly (for lower parallelism). In general, increasing the number of PEs decreased the execution times proportionally (linearly) up to a point and then tapered off to approximately 60% of linear as the number of PEs exceeded the available parallelism. Examples of this effect are shown in Figs 6 and 7. Adding more PEs to problems with the maximum

Figure 5. Case 1, medium-grain (Comm/Comp = 0.14, Comm = 1.8 kbyte) using large packets.
Figure 6. Case 2, medium-grain (Comm/Comp=0.15, Comm=1–2 kbyte) using large packets.

Figure 7. Cases 3–6, medium-to-large grain (0.07 ≤ Comm/Comp ≤ 0.2, Comm = 1–100 byte) using small packets.
Figure 8. Percentage decrease in run-time speedup versus increasing communication.

parallelism already extracted would simply cause the added PEs to idle. The more inherent parallelism a program has, the greater the speed-up expected. Hopefully, this speed-up grows linearly as with parallelism.

The ratio of average communication to computation does not have the same effect when using small and large data packets in communicating between the processors. Small data packets have relatively large overhead, thus, strong degradation is observed when moving towards finer-grain computation. For example, the communication time for case 3 (1–100 byte) placed a greater percentage of overhead relative to the communication time of case 2 (1–2 kbyte) because of the finer granularity in case 3 (Comm/Comp = 0.25). In case 3 (Fig. 7) the inherent parallelism is 14 but that achieved is only about 8. Even with a low number of data transfers (case 3) the computation should be large enough to cover the communication needed in order to produce significant speed-up. This explains why there is less deviation between the scheduler and the run-time in case 2 than in case 3. On the hypercube, the cost to send 10 byte is about 350 μs while the cost to send 1 kbyte is about 1000 μs. As can be seen, the overhead per byte to send 10 byte is much more than for 1000 byte.

Increasing Comm/Comp leads to increasing the amount of remote data transfer, which in turn reduces the available parallelism and increases the deviation between the static and run-time speed-up. In another experiment it was noticed that setting Comm/Comp to 0.3 for small and large packets leads the run-time speed-up to drop by 68% and 43%, respectively, compared with the speed-up obtained from the scheduler. Figure 8 shows the average percentage degradation of the run-time speed-up compared with the speed-up generated by the static scheduler versus increasing communication. Small packets are only useful for coarse-grain tasks where the volume of data for each transfer is about 100 byte. Smaller data transfers implicitly increase the amount of overhead and give unacceptable degradation on the run-time speed-up. Large packets support nearly the same observation but the degradation of
the speed-up is less because the overhead is uniformly distributed per byte. This indicates that one limitation of this approach (and that of the iPSC-2) is its applicability to coarse-grain computations for which Comm/Comp should be below 0.18.

To test the efficiency of our execution model the structure of the LM was modified so that it posts only receives (if any) for the next task and no receives are performed for the subsequent tasks. This approach (large packets) leads to significant degradation of the run-time speed-up, as shown in Fig. 8 under (Receive for next), because some data receives for subsequent tasks (other than the next) are delayed until the LM attempts to run that task. This shows the benefit of this approach in minimizing the run-time overhead by reducing the run-time overhead through overlapping execution with communication whenever possible, and by proposing an efficient execution model (LM) to implement the self-timed task sequencing.

6.2. Stochastic analysis and effect of uncertainties

This section studies the effect of uncertainties in estimating the problem parameters on the proposed self-timed static execution. For this a static assignment is generated based on gross estimation of the task times and the volume of communication and comparing its run-time performance with that corresponding to exact knowledge of the problem. Loops are represented by fork-and-join graphs and their corresponding static assignments are generated to run on the iPSC-2 according to the execution approach.

As a first step, the static assignment is generated based on perfect knowledge of the iteration times and the amount of incoming and outgoing communication. The perfect static assignment and its running time on the iPSC-2 are denoted by $S_{ref}$ and $t_{ref}$, respectively. Next, uncertainty is added to the iteration times and the volume of communication according to a centred and uniform distribution with a given standard deviation. To account for the uncertainties, the iteration times and the amount of incoming and outgoing communication in the perfect assignment $S_{ref}$ are altered by the uncertainties but the time ordering of the tasks and their communication primitives (isend) are kept identical to that of $S_{ref}$. The resulting static assignment and its running time are denoted by $S_{unc}$ and $t_{unc}$, respectively. Function $(t_{ref}/t_{unc} - 1) \times 100$ enables analysis of the percentage drop of the run-time speed-up compared with the speed-up obtained by the static scheduler.

For each number of iterations, 250 fork-and-join graphs were generated and run in order to find the statistical distribution of the ratio $t_{unc}/t_{ref}$. Six experiments were performed by setting the number of iterations in the fork-and-join to 32, 64, 128, 192, 256 and 320, respectively. To find the effect of increasing the accuracy on the loop parameters and its impact on the speed-up, the standard deviation ($\sigma$) was set to $\sigma_1 = 10\%$, $\sigma_2 = 20\%$ and $\sigma_3 = 30\%$ of the original loop parameters. In all the experiments, the iPSC-2 was configured with 16 processors.

Analysis of the results shows that increasing the number of iterations, or the granule size of the CPU times, reduces the variances of the speed-up caused by gross estimation of the loop parameters. The reason is that executing more sequential iterations tends to reduce the effect of the degradation caused by the gross estimate of their parameters. For example, the highest deviation is 30% against only 13% when forking 32 and 320 iterations when both fork-and-join have been altered by a
deviation of $\sigma_3 = 30\%$ from the original. Figures 9 and 10 show the average and worst-case drop in the run-time speed-up compared with the speed-up obtained by the scheduler for different values of uncertainties in the computation and communication parameters.

For a small number of iterations, the benefit of this approach is not clear because the speed-up may degrade by more than 30\% as a result of gross estimation of the loop parameters, as shown in Fig. 10 for 32 iterations. It is also noted that increasing the number of iterations reduces the deviation of the speed up between the distributions obtained for different values of $\sigma$. For example, the difference between the plotted worst case drops in Fig. 6 is about 5\% which tends to decrease with an increasing number of iterations. When the number of iterations is large enough, we conclude that estimating the loop parameters with a deviation $\sigma \leq 30\%$ and applying the static assignment approach gives acceptable degradation on the speed-up compared with the speed-up of perfectly known loops.

6.3. Comparison with other approaches

The proposed static scheduling uses the principle of earliest-startable-first, which has been experimentally evaluated by Al-Mouhamed and Al-Maasarani (1994). In El-Rewini and Lewis (1990) and Hwang et al. (1989), scheduling computation with communication was proposed based on the use of a global time to track the processor completion times so that only the successors of newly completing tasks may become ready to run. In our approach, there is no global time and the successors of newly assigned tasks may become ready. Extended testing demonstrated that the EST algorithm outperforms (Al-Mouhamed and Al-Maasarani 1994) the algorithm (processor-driven/ETF) that was proposed by Hwang et al. (1988). Both scheduling methods have the same time complexity ($O(pa^2)$) but our approach (algorithm EST) is faster due to the absence of global time and the updating of the est($T,p$) that is shown in Step 2.2 of the EST algorithm. The

![Figure 9](image)

Figure 9. Average percentage drop of run-time speed-up compared with the speed-up obtained from the scheduler due to uncertainties.
scheduling proposed by El-Rewini and Lewis (1990) is based on the principle of earliest-finishable-task. All these scheduling methods, including our heuristic, attempt to locally maximize the processor efficiency by using \(O(nm^2)\) heuristics.

The static assignment and self-timed execution are compared with other approaches. First note that executing synchronous dataflow computation using the approach of Shu and Kale (1989) or that of Jogannathan et al. (1989) would lead to degradation of the speed-up because of their significant dynamic overhead in coordinating the decision to schedule a task on a given processor and propagating status information to all the processors. These approaches are suitable for computations whose dataflow topology cannot be predicted at compile-time.

The method that was proposed by Lee and Bier (1990) is based on memorizing the order in which the processors access the shared memory. This gives the least run-time overhead but requires accurate knowledge of the task times and dedicated hardware. Our approach applies to general-purpose DMMs and tolerates more variation in task and communication times because the communication order is relaxed compared with Lee and Bier (1990). Only the execution order is enforced at run-time. Computation and communication may overlap. More importantly, most of the overhead associated with receiving remote data is done using DMA operations that may overlap with arithmetic execution of the next task. Results indicate that static (no overhead) and dynamic execution times match at the 10% level provided that the fluctuations in task and communication times do not exceed 30%, on average, and that the ratio of communication to computation is below 0.18 for the iPSC-2 hypercube.

7. Conclusion

This work has been concerned with a class of real-time computations that can be modelled by means of synchronous dataflow. According to this deterministic model, the data producers of any task can be identified at compile-time, as can the computation and communication requirements.
An execution approach has been proposed for distributed-memory systems based on exploiting the knowledge of the computation and target multiprocessor in order to generate efficient self-timed execution. A communication model for small and large packet exchange between the processors was also proposed. The approach has been extensively tested on the Intel iPSC-2 hypercube multiprocessor. The execution model uses a simple local manager that efficiently monitors receiving remote data and self-timed execution. While the processor is running arithmetic tasks, DMA operations are used to receive the remote data in local memory. The static assignment, self-timed execution and hidden local transfer allow significant reduction of the run-time overhead.

The result is that run-time speed-up matches that predicted by the scheduler (no overhead) at the 10% level within our assumption of the computation granularity and the linearity of the communication model. The proposed self-timed execution is useful for implementing large-scale real-time computations on scalable distributed-memory multiprocessors.

ACKNOWLEDGMENTS

The authors thank Greg Kwan and Steven Wallace for their help with the iPSC-2. They also thank Veronica L. Weiss for her contribution in the stochastic analysis.

REFERENCES


Rayward-Smith, V. J., 1986, Uet scheduling with interprocessor communication delays. Report SYS-C86-6, University of East Anglia, Norwich, U.K.


