ACCURATE COMMUNICATION COST ESTIMATION IN STATIC TASK SCHEDULING

Mingfang Wang,[1] Ben Lee,[2] Behrooz Shirazi,[3] and A.R. Hurson[2]

[1]Department of Math & Computer Science
University of Central Arkansas
Conway, AR 72032

[2]Department of Electrical Engineering
Pennsylvania State University
University Park, PA 16802

[3]Department of Computer Science Engineering
University of Texas at Arlington
Arlington, TX 76019-0015

Abstract

Most of the existing static scheduling schemes either ignore the communication delay or use a very simple model to *estimate* it. The result is that when the program is actually executed, its execution timing is unpredictable due to network behavior and load. However, if we know the timing information and communication requirements for the program tasks as well as the network topology and behavior at compile time, we can compute (not estimate) the communication delays and network routes before execution. This information can then be used at execution-time for predictable network timing requirements. In this paper we show that optimal network routing is NP-complete. We then propose two heuristic algorithms, with different characteristics and complexity, to compute sub-optimal routing information for static scheduling at compile time. This information can be used at run-time for actual routing. The heuristics take into account the shortest paths from source to destination as well as the current network load on different links. The effectiveness of the proposed algorithms are demonstrated through simulation results and comparison against lower-bounds for routing.

1. Introduction

When one schedules communicating tasks in a multiprocessor, communication costs should be accounted for. In the past, very simple approaches have been taken for this purpose: Either they assumed worst-case costs, or costs were assumed to be given. A more accurate method should be devised in order to keep the schedulability of tasks as high as possible. Such a method will consider processor connectivity, amount of information transferred between tasks, contention for communication links, and scheduling of tasks themselves.

The recent task scheduling algorithms consider the communication delay in their schemes in an over-simplified manner [1-9]. The often proposed assumptions are that the time needed to send a message from one Processing Element (PE) to another is either (i) a constant amount, or (ii) a constant times the message size, or (iii) the distance between two PE's times the message size. The main reason for these assumptions is that the message routing in the interconnection network is handled by dynamic routing using computer network communication protocols at run-time [10,11]. Thus, the run-time network behavior is unpredictable and unknown at compile time.

This research is supported in part by DARPA under contract no. 5-25089-310.

The first two assumptions are obviously un-realistic. The third assumption under-estimates the communication delay since this delay depends not only on the distance between PEs, but also on the current load on the network links. Network contention often increases the communication delay. This has resulted in unpredictable program behaviors, which is unacceptable for real-time applications- the best candidates for static scheduling. However, it should be noted that given enough information, we can generate the network routes at compile time and use them at run-time for actual communication. This information includes the program task execution delays, tasks' communication requirements, message sizes, network topology, and network links' transfer rate capacity. In static scheduling all this information is known at compile time. Therefore, in this paper we propose algorithms which compute accurate communication delays and network routes at compile time. The obtained routing information is used for static scheduling and for running the program.

First, we show that optimal routing which results in the minimum communication delay in a message-passing, point-to-point, store- and-forward interconnected multiprocessor system is an NP-complete problem. Such systems include multiprocessors with a network topology of a ring, star, hypercube, linear or mesh array of processors, etc. Then, we present two heuristic algorithms, with different characteristics and complexity, for routing messages in such a system. These algorithms provide the information which indicate the links to be used to conduct the communication and the time period that each link is busy transferring the messages. According to this information, the moment that a message arrives to its destination can be computed. These algorithms can be considered as procedures to be called by the task schedulers. When a static task scheduling algorithm is selecting a PE for a particular task, the communication delay can be accurately obtained by calling these procedures. Thus, the static schedule includes network routes for messages as well. During the run-time the routes chosen at compile time are used for communication, resulting in predictable program behavior. This feature is especially useful in real-time applications in which the hardware can be configured to meet the real-time constraints. Finally, we show that due to accuracy in computing the communication delays, the proposed algorithms result in a better task schedule compared to methods which estimate the delays.

Section 2 briefly discusses our basic assumptions and shows that optimal routing is an NP-complete problem. Section 3 gives the proposed message routing algorithms. Section 4 discusses the incorporation of the communication delay in the static task scheduling. The performance evaluation and analysis of the algorithms are discussed in section 5, and finally section 6 gives a brief conclusion.

2. Basic Assumptions and Static Task Scheduling Framework

Multiprocessor systems can be in many different configurations. Here we focus on the loosely coupled, distributed memory multiprocessor systems. It is assumed that identical PEs are connected by a message-passing, point-to-point, store-and-forward interconnection network (e.g. meshes, hypercubes, etc.) Furthermore, we assume that the interconnection network is symmetric. Formally, we can use a graph representation, GP(VP,EP), to represent a multiprocessor system. A node $vp \in VP$ in the graph represents a PE and an undirected edge $ep \in EP$ represents a link connecting two PEs. Let $P_{i1}, P_{i2}, \ldots, P_{ik}$ be k subsets of the nodes that can be reached from node vp_i, $i \leq |VP|$, with distance 1, 2, ..., k, respectively. Here, k is the diameter of the graph. Obviously $P_{ia} \bigcap P_{ib} = \phi, a \neq b$. For a symmetric graph we have:

$$| P_{i1} | = | P_{j1} |, | P_{i2} | = | P_{j2} |, \ldots, | P_{ik} | = | P_{jk} |$$

for any i and j such that $i, j \leq |VP|$. Hypercube, ring, and wrapped-around mesh are some popular symmetric configurations of multiprocessor systems [12]. One of the advantages of a symmetric configuration is that during the initial steps of task distribution, any PE is a suitable PE for an entry task. In addition, there is no need for special treatment of network boundary conditions for routing analysis.

A program can be represented by a directed graph, GC(VC,EC.) Each node $vc \in VC$ represents a task which is a portion of the operations of a program. A task is a side-effect free function defined in a functional programming language, such as SISAL. Equivalently, a task can be a procedure or a block in the dataflow graph representation of the program. Further, we assume a task must receive all its input messages before it is enabled for execution. Upon completion of a task, it may send several messages to other tasks.

When a program is to be run on a multiprocessor system, a weight function $WT: VC \rightarrow Z^+$ is used to indicate the amount of time required to execute each task by a PE. Here, Z^+ is the set of integers and, for simplicity, we measure the execution times in time units. The directed edges in the graph give the precedence among the nodes and show the flow of data among the dependent tasks. The amount of data transferred through an edge is measured by the weight function $WE: EC \rightarrow Z^+$. Again, for simplicity, the amount of data is measured in data units.

Before presenting the proposed algorithms, we can briefly show that optimal routing is an NP-complete problem. This can be trivially proved by mapping the routing problem into the partitioning problem [15]. The partitioning problem is as follows. Given a finite set A and a "size" $s(a) \in Z^+$ for each $a \in A$, find a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a) .$$

To see how the mapping can be achieved, consider a very simple case. Figure 1 shows a multiprocessor system with only two PE's connected in a ring structure. Note that there are two bi-directional links between the PEs. Assume that PE1 is sending n messages with different sizes to PE2. PE1 can send messages through either link. To achieve the minimum communication time, a decision must be made as to which message should be assigned to which link so that the loads on the two links are balanced. This is nothing but the above mentioned partitioning problem.

3. Proposed Algorithms

In this section, we present two heuristics that can be used to route the messages in our assumed multiprocessor system. We assume that the bandwidth of the links is the same and a PE can execute a task and perform data communication simultaneously.

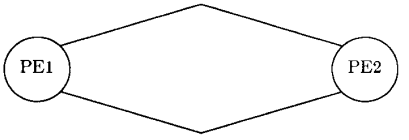


Figure 1. A 2-PE ring connected system.

Thus, the network interface operations are not influenced by the work-load of a PE. Finally, it is assumed that a link is dedicated to transfer one message at a time. Only after the transfer of the current message is completed, can this link be used to transfer another message. There is no preemption on any transfer. If a link is free, a message can be sent through the link with a communication time proportional to the message size. Without loss of generality, we assume the communication time on **one** link (not the entire source-destination path) to be equal to the message size. In practice, one can appropriately scale this time by a constant factor. If a link is not free, the upcoming message must wait until the current message completes its transfer. Thus, the communication time for the upcoming message, on **one** link, is the waiting time plus the message size. If a message is traveling through several links, the total communication time is the summation of the communication time spent on each link and the waiting delays.

A good message routing strategy attempts to avoid possible link blocking which can cause additional waiting delays for a message during its transfer. An example of blocking is shown in Figure 2. Here, both PE3 and PE2 are sending messages to PE1. Assume that message1 is produced by PE3 at moment 0 and message size is 5. Message2 is produced by PE2 at moment 3 and is also of size 5. With the interconnections given in Figure 2, if message1 is given higher priority, then the link between PE2 and PE1 will be busy during the period from 5 to 10 to conduct message1. Thus, message2 can arrive at PE1 no earlier than moment 15. In such a case, message2 is blocked for 7 time units. If message2 is sent out first, the link between PE2 and PE1 is busy from moment 3 to 8 for message2. When message1 arrives at PE2, it will be blocked until moment 8 and thus, it can arrive at PE1 at moment 13. Therefore, the second strategy provides an overall gain of 2 time units over the first strategy.

Consider a hypercube interconnection network of 8 nodes. Figure 3 contains 9 example messages sent by 4 different PE's, PE1,

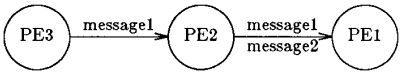


Figure 2. An example of message blocking.

message:	0	1	2	3	4	5	6	7	8
start at:	2	3	5	3	5	5	8	4	6
location:	7	7	7	5	5	6	6	1	1
size:	2	2	2	2	2	3	1	2	2

Figure 3. An example of a set of messages.

PE5, PE6, and PE7 in this network. PE1 sends messages 7 and 8. PE5 sends messages 3 and 4. PE6 sends messages 5 and 6. PE7 sends messages 0, 1, and 2. These PE's constitute the ***source PE set*** . PE0 is chosen as the ***destination PE*** . This example will be used to show the results from the heuristic algorithms introduced in the later part of this section.

We use two quantities to measure the quality of a message routing: the total waiting time and the message-passing completion time. Total waiting time is the summation of all the waiting times incurred during the message transfers. The completion time is the moment that all the messages have arrived at the destination PEs.

Heuristic Algorithm 1:

Consider a path of nodes for a message routing. The message is sent from the source to destination node by going through the intermediate nodes, one link at a time. An ***ideal path*** for a message is defined to be the path with the shortest length and the Least Waiting Time (LWT) on the links of the path. Heuristic algorithm 1 is greedy in the following sense: At each intermediate node on the path, the message is sent through a neighboring link which yields the least waiting time among the links that a PE can choose to send the message to the next intermediate node. A node on a path from the source to the destination will send the message to one of its neighbors which is closest to the destination PE. We say PE_i is closer than PE_j to PE_d if the shortest path from PE_i to PE_d is shorter than the shortest path form PE_j to PE_d. In symmetric point-to-point networks, such as ring and hypercube, the length of the shortest path from a source to a destination can be easily figured out by a special numbering of the network nodes. For example, in the case of hypercube, the length of the shortest path between nodes A and B is determined by the number of 1's in the binary exclusive-or of the node numbers for A and B. The algorithm is given below:

Initially, let PE_SET1 be the set of all the PE's which are source PE's. A PE is a source PE if either one or more tasks executed by that PE have generated some messages to be sent out or it is on the path of other messages, requiring it to receive and send a message between two of its neighbors. MS_i is used to represent the set of messages in the message queue of PE_i. PE_d is the destination PE. PE_SET2 is a temporary set variable. Different values are assigned to it in the algorithm.

We use S to represent a message. Let MOMENT be a function. MOMENT(S, PE_i) is the moment that message S is generated or received (to be passed to another PE) by PE_i. Normally, a message S is put in the MS_i queue at MOMENT(S, PE_i). Let DISTANCE be a function. DISTANCE(PE_i, PE_j) gives the length of shortest path between the two PE's. Let NEIGHBOR be a function. NEIGHBOR(PE_i) gives a set of PE's which are the neighbors of PE_i; i.e. they are connected via one link.

Let SE_i be a message at the head of the MS_i queue. SE_i represents the next message to be processed by PE_i.

Arrays ARRIVAL and PATH are two dimensional arrays used to record messages arrival times and paths, respectively. ARRIVAL[PE_j][S] gives the moment that S is generated or received (to be passed to another PE) by PE_j. PATH[PE_j][S] gives the PE which has sent S to PE_j.

```
For every message, indicated as S_i, do
    For every PE, indicated as PE_j, do
        If PE_j holds S_i Then
        ARRIVAL[PE_j][S_i] = MOMENT(S_i, PE_j)
        Else ARRIVAL[PE_j][S_i] = ∞;
    End For;
End For;
While PE_SET1 <> Φ do
    Let PE_a be the PE such that PE_a is in PE_SET1 and
    DISTANCE(PE_a, PE_d) =
        MAX(DISTANCE(PE_i, PE_d)), PE_i is in PE_SET1;
    While MS_a <> Φ do
        PE_SET2 := {PE | PE ∈ NEIGHBOR( PE_a ) and
            DISTANCE(PE, PE_d) < DISTANCE(PE_a,
            PE_d)};
        IF PE_SET2 <> Φ Then
            Find the PE, say PE_j, such that
                If SE_a is sent from PE_a to PE_j then
                MOMENT( SE_a, PE_j ) is minimal
                among PE_j's in PE_SET2;
                /* Finding the link with shorter
                waiting queue */
            M := MOMENT( SE_a, PE_j);
            Send SE_a from PE_a to PE_j;
            MS_j := MS_j ∪ {SE_a};
            ARRIVAL[PE_j][SE_a] := M;
            PATH[PE_j][SE_a] := PE_a;
            If ({PE_j} ∩ PE_SET1 ) = Φ Then
                PE_SET1 := PE_SET1 ∪ {PE_j};
            MS_a := MS_a - {SE_a};
    End While;
    PE_SET1 := PE_SET1 - {PE_a};
End While;
```

When the algorithm terminates, the routing information is kept in array PATH and the timing information is kept in array ARRIVAL. For example, PATH[PE_d][S_i] gives the last intermediate PE which sends message S_i to the destination PE, PE_d. Let this PE be PE_i. We can further trace that message S_i is sent to PE_i by PE_j if PATH[PE_i][S_i] = PE_j. In this way, the whole path to route message S_i to destination PE_d can be found. In the same manner, the time when a message arrived at a certain PE can be found in the array ARRIVAL.

Figure 4 shows the routes generated by algorithm 1, using the example of Figure 3. The total waiting time of this algorithm is 8 and the completion time is 13.

In the worst case, this algorithm examines all the PEs on a path from the source to destination. Therefore, the complexity of algorithm 1 is O(n) for each message, where n is the number of PEs.

Heuristic Algorithm 2:

There are two type of blockings: a later message blocking an earlier message (LBE blocking) and an earlier message blocking a later message (EBL blocking). Here, "later" and "earlier" refer to the time the message was originally generated. Consider example of Figure 2 in which message1 is generated by PE3 at time 0 (earlier) and message2 is produced by PE2 at time 3 (later). Both messages have a 5 unit size. If we send message1 from PE3 to PE1 first, message2 will be blocked for 7 time units. Such a blocking is EBL blocking. If message2 is send out immediately after its generation, it

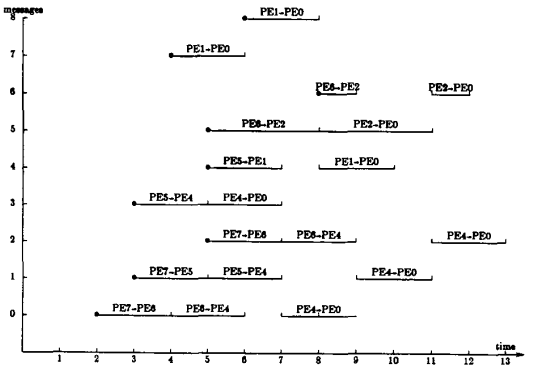


Figure 4. Network routing based on algorithm 1 using messages of Figure 3.

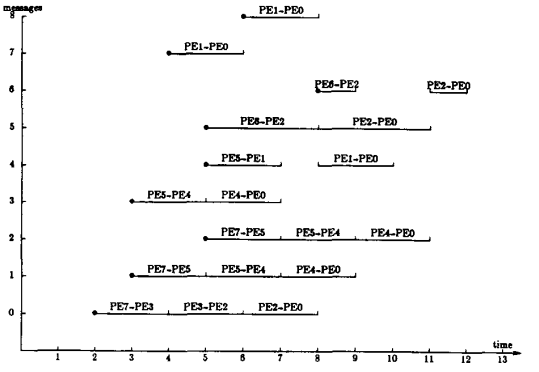


Figure 5. Network routing based on algorithm 2 using example of Figure 3.

will block message1 for 3 time units. This is an LBE blocking. In this example and many others we tried, the EBL blockings resulted in longer wait times for the messages. We have used this a heuristic principle for message routing. In the case that several messages are competing for a link (thus, blocking becomes unavoidable), we avoid EBL blockings. The second proposed heuristic algorithm is called the least blocking algorithm which tries to avoid EBL blocking:

The identifiers used in this algorithm are the same as those defined in algorithm 1.

For every message, indicated as S_i, do
    For every PE, indicated as PE_j, do
        If PE_j holds S_i Then
        ARRIVAL[PE_j][S_i] := MOMENT(S_j, PE_i)
        Else ARRIVAL[PE_j][S_i] := ∞;
    End For
End for
While PE_SET1 <> Φ do
    Let PE_a be the PE such that PE_a in PE_SET1 and
        $MOMENT(SE_a, PE_a)$=$MIN(MOMENT(SE_b, PE_b))$,
            for $1 \le b \le$ | PE_SET1 |;
    PE_SET2 := {PE | PE *in* $NEIGHBOR(PE_a)$ and
    $DISTANCE(PE, PE_d) < DISTANCE(PE_a, PE_d)$};
    If PE_SET2 <> Φ Then
    For each PE in PE_SET2, indicated as PE_i, do
        Send SE_a from PE_a to PE_i;
        M := MOMENT(SE_a, PE_i);
        If ARRIVAL[PE_i][SE_a] > M Then
            PATH[PE_i][SE_a] := PE_a;
            ARRIVAL[PE_i][SE_a] := M;
            If ({PE_i} $\cap$ PE_SET1) = Φ Then
                PE_SET1 := PE_SET1 $\cup$ {PE_i};
    End For;
    MS_a := MS_a;
    IF MS_a = Φ Then PE_SET1 := PE_SET1 - {PE_a};
End While;

Again, the example of Figure 3 is used to show the message routing produced by this heuristic algorithm. Figure 5 gives the result.

The total waiting time is 2 and the completion time is 12. For this particular example, these results are optimal. This algorithm is a variation of Dijkstra's shortest path procedure. It also achieves the least blocking. Thus, similar to Dijkstra's algorithm, its complexity is O(elogn) [16] for each message, where n is the number of processors and e is the number of links.

Theorem 1 below shows that both algorithms guarantee that any message is sent through a shortest path from the source to the destination.

Theorem 1: In Algorithm1 and Algorithm2, each message is sent through a shortest path form its source to its destination.

Proof: Assume that a message is sent through a path, PE_s, PE_{k-1}, PE_{k-2}, ..., PE_1, PE_d, where PE_s is the source PE and PE_d is the destination PE. The length of this path is k. Since at each node the messages are sent to the PE's that are closer (closeness was previously defined) to PE_d, PE_{k-1} must be closer to PE_d than PE_s, PE_{k-2} must be closer to PE_d than PE_{k-1}, and so on. If there is a shorter path between PE_s and PE_d, then let its length be j, where j<k, and i=k-j. Then since the distance between PE_s and PE_d is j, not k, the neighbors of the PE_s which are chosen to receive the message cannot include PE_{k-1}. Instead, there should be some neighbor of the source PE which is i links closer to PE_d. This implies that the message would never be sent to PE_{k-1}, PE_{k-2}, ..., PE_{k-i}. In other words, the path PE_s, PE_{k-1}, ..., PE_1, PE_k would never be chosen as a path to carry the message. This is a contradiction to the original assumption. Thus, there is no other path between PE_s and PE_d that is shorter than k. □

4. Incorporating Communication in Static Task Scheduling

The routing algorithms presented are particularly suitable for list scheduling algorithms. A task schedule is a mapping from a program computation graph GC to a system configuration graph GP. A list scheduling algorithm repeatedly carries out the following steps [13]:

1. The tasks that are ready to be assigned are put into a priority queue according to a priority criteria. A task becomes ready for assignment when all of its predecessors are already assigned for execution.
2. Select a PE with the least amount of work already assigned to it.
3. Assign the task at the head of the priority queue to this PE.

There are several methods to generate the priority number for each task. The most common version of the priority scheme is called Critical Path Method (CPM) [14] which gives each task a priority number according to the length of its exit path to a terminating task. The idea is that the tasks with a longer critical path length should be assigned first since many other tasks are dependent on them. A simpler method called Heavy Node First (HNF) [14] first orders the tasks level by level, giving higher priority to the entry level nodes. Entry nodes are the nodes with no predecessor. It will then order the nodes at each level from heaviest (requiring most

execution time) to lightest. The level-by-level ordering preserves the order of precedence and load balancing is achieved by first assigning heavier tasks to PEs with the least amount of work already assigned to them. In our simulation experiments, we have used the HNF algorithm. In any case, the proposed algorithms are used as subroutines for these static scheduling methods to provide communication delays among the tasks as they are assigned to different PEs.

Consider task T at the head of the priority queue. If T is an entry task, it would simply be assigned to the PE with the least amount of work already assigned to it. However, if T has some predecessors, then the best PE to assign T to would be the PE which can execute it at earliest possible moment, taking into account the communication cost from the predecessors to T. A task cannot be executed until all the communications between this task and its parents are completed. We call the time that a task receives all of its input messages the Desirable Starting Moment (DSM) of that task. For different PEs, the DSM of a task can be different. We will use $DSM_{i,T}$ to indicate the DSM of task T on PE_i. The $DSM_{i,T}$ can simply be computed by calling algorithm 1 or 2 and using PE_i as destination. Naturally, if a task has several inputs, the DSM would be computed according to the time the last input is arrived at PE_i. To find a PE to assign T to, we need to compute MIN($DSM_{i,T}$) for all i. There are two problems. First, PE_i with the smallest DSM for task T may not necessarily be able to execute T at its earliest possible time. Second, computing $DSM_{i,T}$ for all i is too time-consuming.

The first problem stems from the fact that a processor with smallest DSM for a task may already be heavily loaded with other tasks. Thus, a PE which yields minimum DSM for a certain task may not be the ideal PE for a task. We define the Actual Starting Moment (ASM) of a task (T) assigned to a processor (i) as:

$$ASM_i = MAX(LOAD(PE_i),\ DSM_{i,T}), \tag{1}$$

where $LOAD(PE_i)$ is the summation of the execution time of the tasks already assigned to PE_i and the ideal times scheduled on this PE so far. To select the PE for a task T at the head of the priority queue, the actual starting moment of each PE should be computed and compared. The PE with minimum ASM:

$$MIN(ASM_i) \qquad \text{for } i = 1, 2, \ldots, n, \tag{2}$$

is chosen. Here, n is the number of PEs.

For the second problem, it should be noted that if the number of PEs is large, one can use a heuristic to only consider a subset of the processors. One such heuristic can select the K nearest neighbors of the source tasks as possible candidates for assigning the task to.

Once a task T is assigned to PE_i, the routing information for each input message to T can be obtained from the ARRIVAL and PATH arrays generated by either algorithm 1 or 2. The route for each message can then be stored into a 2-dimensional table. This table is similar in both organization and functionality to the array PATH which was discussed in detail in algorithm 1. By copying this table in each processor, a simple table look-up mechanism can be used at run-time for actual routing of the messages.

It should be noted that as tasks are assigned to the PEs, we keep track of the load on the links. Even though algorithm 1 or 2 is called for each task independently, the routes for a new message are generated according to the current load on the network as generated by previous messages. The algorithms do not consider future network loads.

5. Performance Evaluation and Analysis

In this section we we compare the two algorithms against a lower-bound time for routing the messages. The proposed algorithms are then incorporated into the HNF static scheduling method. The resulting scheduler is compared against comparable schedulers which use other routing algorithms, including dynamic network routes.

In our experiments, the network configuration is assumed to be that of a 16-node hypercube. The following parameters are randomly generated:

1. The size of each message (between 1 and 5).
2. The total number of messages (between 1 and 80).
3. The PEs which produce the messages.
4. The destination PE.
5. The number of messages a PE produces.
6. The moment each message is produced. This moment is confined to a given range. In our tests, we tried 2 time ranges: 10 and 30 time units.

We assume that a message consists of several bytes (e.g. in the range of 40 to 200 bytes). We do not consider very small (few bits) synchronization or very large (KBytes) messages. Thus, we use a range of 1 to 5 units for messages. The choice of the total number of messages and message generation ranges were dictated by the space and time limitations of our VAX 11/780 system. However, they are wide enough ranges to represent most cases.

The algorithms are compared against optimal using the RAI measure, where RAI stands for the Rate of Actual message transfer time over the Ideal message transfer time. The time needed to send a message through a link includes WAIT, the waiting time, and SIZE, the time that the link is dedicated to transferring the message. Thus, the rate of actual transfer time on a link j for message i is $SIZE_i + WAIT_{i,j}$. In the ideal case, however, there should not be any waiting time.

The ideal case provides a lower-bound on message transfer time. Note that this is a loose lower-bound in the sense that the actual optimal route may have to include some wait times as well. Thus, RAI gives an approximate measure for comparison against the optimal case. The higher the RAI for a message, the worse the selected route is compared to the ideal case. On the other hand, an RAI close to 1 indicates that the route is near optimal.

The RAI for a link is given as:

$$RAI = \frac{SIZE_i + WAIT_{i,j}}{SIZE_i} = 1 + \frac{WAIT_{i,j}}{SIZE_i} \tag{3}$$

We use p_i to indicate the path for message i. p_i is a set of links. Let $L_i = |p_i|$. Then the RAI for a path is given as:

$$RAI = \sum_{j \in p_i} \frac{SIZE_i + WAIT_{i,j}}{L_i \times SIZE_i}, \tag{4}$$

or,

$$RAI = \frac{1}{L_i} + \frac{1}{L_i \times SIZE_i} \times \sum_{j \in p_i} WAIT_{i,j}. \tag{5}$$

Now consider multiple messages. Let k be the number of messages sent through the network and R be the set of all paths. The average RAI for the set of messages is defined as:

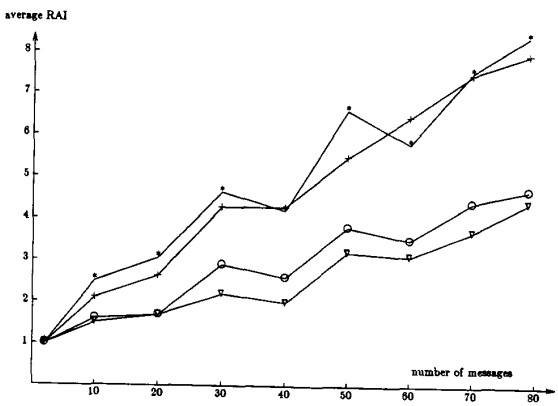
$$average\ RAI = \frac{1}{k} \times \left(\sum_{p_i \in R} \left(\frac{1}{L_i} + \frac{\sum_{j \in p_i} WAIT_{i,j}}{L_i \times SIZE_i} \right) \right) \tag{6}$$

For a given time range and a given number of PE's, the algorithms are applied to 500 different instances of message distributions. The RAI presented in the following figures represent the average measure. In addition, for the same set of parameters, we measured the RAI for hypercube's dynamic routes as well as randomly chosen (among the possible shortest paths) routes. Figures 6 and 7 present the average RAI measure vs. the number of messages for time ranges of 10 and 30, respectively. Note that the time range defines a range for the time of generation of messages. A time range of 10 represents a heavy load on the network, i.e. many messages are generated in a short time. Similarly, a time range of 30 represents a medium load on the network.

When the number of messages is small (up to about 20 messages), both algorithms 1 and 2 performed close to optimal. In other words the average RAI was less than 2. Note that an optimal routing delay is greater than or equal to the ideal case delay. This indicates that the generated routes are in the worst case twice as long as the optimal routes. However, as the number of messages are increased, the RAI is also increased indicating that the generated routes have become lengthier than the ideal case. It should be noted that as the number of messages is increased, it is more likely that the optimal routes are also lengthier than the ideal case. This is because the probability of blocking at different links is increased. Therefore, it is our conjecture that even for large number of messages, the generated routing delays are close to optimal.

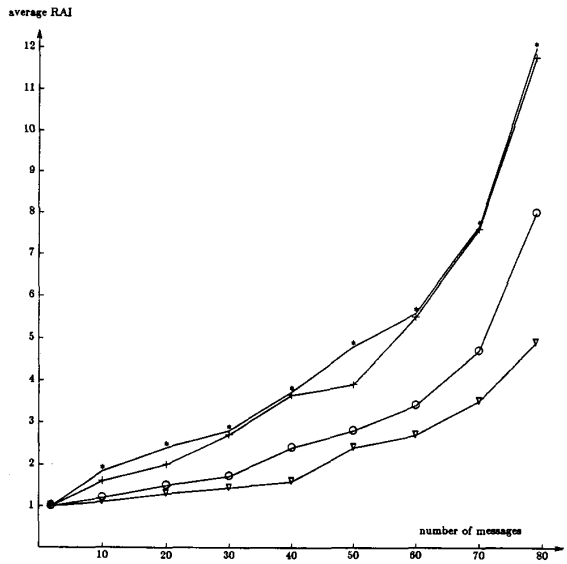
By comparison, the dynamic and random routes are several times worse than the ideal case. It is also interesting to note that algorithm 2 consistently has a lower RAI measure compared to the other methods. This indicates that by comparison, the routes generated by this algorithm cause the least amount of waiting time on the links.

In the second study, we evaluated the impact of the proposed algorithms on the quality of static schedules. We used HNF algorithm (discussed in section 4) for static scheduling. The execution of the applications scheduled by HNF using the proposed algorithms and other methods which estimate the communication delay are compared. A practical application, namely FFT is considered.



O: Heuristic algorithm1 ∇: Heuristic algorithm2 *: Dynamic routes +: Random routes

Figure 6. RAI measure for Time range = 10 and Number of PE = 16.

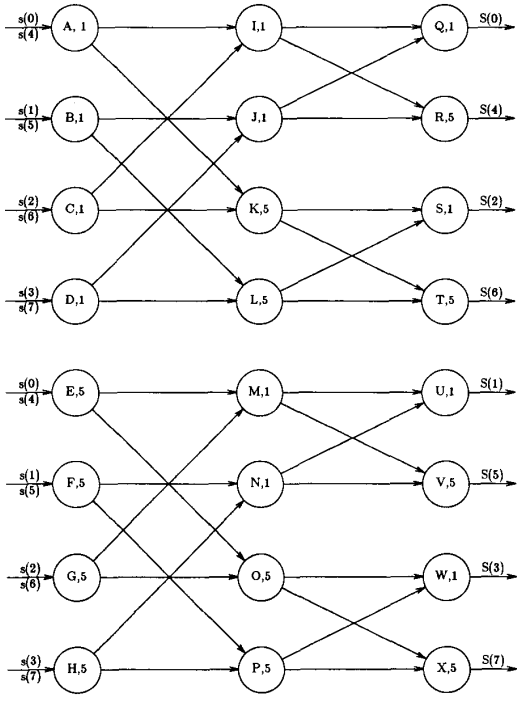


O: Heuristic algorithm1 ∇: Heuristic algorithm2 *: Dynamic routes +: Random routes

Figure 7. RAI measure for Time range = 30 and Number of PE = 16.

In our studies, we chose to use algorithm 1 for network route generations because it is simpler and has a lower time complexity compared to algorithm 2. This algorithm is compared against two of the existing communication delay estimation methods. The first scheme assumes no communication delay in static scheduling. In other words, the assumption is that the computation time far outweighs the communication time. Thus, the PEs are selected for task allocation solely based on their current load. Since each time the PE with the Smallest Load is selected, we call this method SL algorithm. The second existing scheme estimates the communication cost as product of message size and source-destination PE distance. We call this method Estimated Communication Delay (ECD) algorithm.

We compare algorithm 1 against SL and ECD algorithms using an 8-point fast fourier transform computation DAG which is shown in Figure 8. Each task is identified by a letter and number pair, where the letter is the task ID and the number is the execution delay of the task. The message size is fixed at 2 units. A multiprocessor with four PE's connected by a hypercube interconnection network is assumed. When algorithm 1 is used, the run-time communication is carried out according to routing information generated by algorithm 1. This schedule takes 21 time units to complete. When SL alogrithm is used, it takes 25 time units to complete the same program. Finally, when ECD algorithm is used, an execution delay of 22 time units resulted for the same program. For the cases of SL and ECD, dynamic hypercube network routing is used since these methods do not generate specific routes. It is notable that SL and ECD methods resulted in inferior schedules, since delays due to network load on different links were ignored at scheduling time.



Each link represents a message with size equal to 2.

Figure 8. Program DAG of FFT.

6. Conclusion

Precise management of data communication is an important topic in static task scheduling in order to obtain realistic schedules. Optimal routing is an NP-Complete problem. Thus, two heuristic algorithms were introduced with the goal of finding sub-optimal routes for messages in a point-to-point, store-and-forward interconnection network. These routes are used to determine the best task assignment at compile time. In addition, they can be used to set up the required routing tables for dynamic routing at run-time. The simulation results show that: i) for small number of messages, the generated routes are no longer than twice a defined loose lower-bound for the communication delay, and ii)the proposed algorithms can outperform simple methods which estimate the communication cost during static scheduling.

7. References

[1] E. G. Coffman, J. L. Bruno, G. L. Graham, W. H. Kohler, R. Sethi, K. Steiglitz and J. D. Ullman, "Computer and Job-shop Scheduling Theory, Wiley-Interscience Publication, 1976.

[2] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEETrans.Comput.* vol. c-33, pp 1023-1029, November 1984.

[3] R. L. Graham "Bounds on Multiprocessing Anomalies and Related Packing Algorithms," *SIAM J. Appl. Math*, vol. 17 pp 416-429, March 1969.

[4] B. Kruatrachue "Optimal Grain Determination for Parallel Processing System" Report 87-60-5, Oregon State University Computer Science Department, Corvallis, Oregon 97331.

[5] S. J. Kim, "A General Approach to Multiprocessor Scheduling", TR-88-04, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712-1188.

[6] V. Sarkar and J. Hennessy, "Compile-time Partitioning and Scheduling of Parallel Programs," Proc. of the *SIGPLAN86Symp.onCompilerConstruction*, 1986, p 17.

[7] T. Ravi M. D. Ercgovac, T. Lang and R. R Muntz, "Static Allocation for a Data Flow Multiprocessor System," 2nd Int'l Conf. on Supercomputing, May 1987.

[8] R. Tsai, "A Heuristic for Static Task-to-processor Assignment Using A Constraint Satisfaction Approach", Ph. D. Dissertation, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75275.

[9] B. Lint and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," *IEEETrans.SoftwareEngineering*, Vol SE-7, p 174, March 1981.

[10] Andrew S.Tanenbaum "Computer Network" Prentice-Hall, Inc. 1981.

[11] "iPSC System Overview Manual" Intel Corporation November 1986, Order Number: 310610-001.

[12] Hwang, K. and Briggs, A. F. "Computer Architecture and Parallel Processing", McGraw-Hill, Inc., 1984.

[13] C. D. Polychronopoulos and U. Banerjee, "Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds", *IEEETrans.Comput.* April 1987, p 410.

[14] B. Shirazi and M. Wang, " Design and Analysis of Heuristic Functions for Static Task Distribution", Proceeding of Workshop on the Future Trends of Distributed Computing System in the 1990s. p 124.

[15] M. R. Garey and D. S. Johnson, "Computers and Intractability A Guide to the Theory of NP-Completeness" San Francisco Freeman, 1979.

[16] A. Aho, J.E. Hopcroft, and J.D. Ullman, "Data Structures and Algorithms," Addison-Wesley, 1985.

[17] M.F. Wang and Shirazi, B., "Static Network Routing Scheduling," Tech. Report: 88-CSE-37, Dept. of Computer Science and Eng., SMU, Dec. 1988.

[18] R. G. Babb II, "Programming Parallel Processors" Addison Wesley, 1988.

[19] A.R. Hurson, B. Lee, B. Shirazi, and M. Wang, "A Program Allocation Scheme for Dataflow Computers," Int'l Conf. on Parallel Processing, pp 415-423, Aug. 1990.