CHAPTER 8

Processor Involvement in Communication

The inclusion of communication contention in task scheduling yields more accurate and more efficient schedules compared with those produced under the classic model. Yet, the experimental results referenced in Section 7.5.4 suggest that even the contention model does not represent a parallel system accurately enough. An examination of the results (Sinnen and Sousa [172, 176]) reveals a clear relation between the increase of communication (*CCR*) and the decrease of accuracy of the produced schedules. A possible explanation is that the contention model is still unrealistic regarding communication.

In the prelude of the previous chapter, three very idealizing assumptions of the classic model (Definition 4.3) were queried. Two of them—the assumption of contention-free communication (Property 5) and the assumption of a fully connected network (Property 6)—have already been addressed in Chapter 7. Property 4, which supposes a dedicated communication subsystem to be present in the target system, is examined in this chapter. With the assumed subsystem, computation can overlap with communication, because the processor is not involved in communication.

However, many parallel systems do not possess such a subsystem (Culler and Singh [48]). Therefore, in many systems the processors are involved, in one way or another, in interprocessor communication. Furthermore, involvement of the processor also serializes communication, even if the network interfaces are capable of performing multiple message transfers at the same time, since a processor can only handle one communication at a time. For example, a processor can usually only perform one memory copy at a time. Thus, considering the processors' involvement in task scheduling is of importance as it serializes the communication and, more importantly, prevents the overlap of computation and communication.

In fact, the involvement of the processors in communication is an aspect that has been considered before. The LogP model (Section 2.1.3) dedicates the parameter *o* to describe the computational overhead inflicted on a processor for each communication. Per definition of the model, the processor cannot execute other operations while occupied with this overhead. Furthermore, the system model used by Sarkar [167], for which a clustering algorithm is proposed, includes the computational overhead of communication. Falsafi and Wood [66] compare the performance of a parallel system with and without the utilization of a dedicated communication subsystem.

This chapter investigates involvement of the processor in communication, its impact on task scheduling, and how it can be considered in the latter. A system model for scheduling that considers the involvement of the processor in communication is developed based on the contention model. As a result, the new model is general and unifies the other scheduling models. Since scheduling under the new model requires adjustment of existing techniques, it is shown how this is done for list scheduling and a simple genetic algorithm based heuristic.

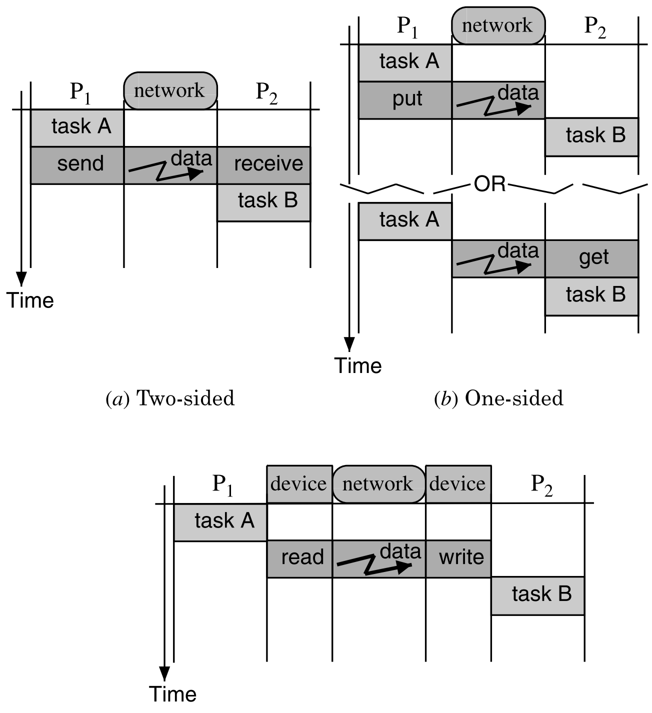
The chapter begins in Section 8.1 by classifying interprocessor communication into three types, which differ based on the involvement of participating processors. Subsequently, the classification is refined by analyzing the characteristics of the involvement. To integrate the processor involvement into scheduling, the scheduling model must be adapted, which is elaborated in Section 8.2. Section 8.3 discusses the general approaches for scheduling under the new model, which differ from those followed under the contention model. Finally, Section 8.4 puts the pieces together by formulating heuristics for scheduling under the involvement–contention model. This includes a short review of experimental results that demonstrate the fundamentally improved accuracy and the significantly reduced execution times of the schedules produced under this model.

8.1 PROCESSOR INVOLVEMENT—TYPES AND CHARACTERISTICS

8.1.1 Involvement Types

Regarding the involvement of the processor, interprocessor communication can be divided into three basic types: *two-sided*, *one-sided*, and *third-party* (Sinnen [172], Sinnen et al. [180]). For illustration of these types, consider the following situation: the output of a task A on processor P_1 is transferred via the communication network to a task B on processor P_2. Figure 8.1 visualizes the involvement of the processors in this simple situation for the three types. In the following paragraphs, the network is treated as a general and abstract entity.

Two-Sided In two-sided interprocessor communication both the source and the destination processor are involved in the communication (Figure 8.1(a)). In which kinds of parallel systems does this occur? It can be the case for message passing as well as for shared-memory architectures. On message passing systems, the involvement of the processors consists of preparing the message and sending it to and receiving it from the network. For example, in a PC cluster (Section 2.1.2), the TCP/IP based communication over the LAN involves both processors. The sending processor must divide a message into packets and wrap them into TCP/IP protocol envelopes before setting up the network card for the transfer. On the receiving side, the processor is involved in the unwrapping and assembling of the packets into the original message



(*a*) Two-sided (*b*) One-sided

(*c*) Third-party

Figure 8.1. Types of interprocessor communication: (*a*) two-sided—both processors are involved ("send" and "receive"); (*b*) one-sided—only one processor is involved, either source or destination processor; (*c*) third-party—a special communication device takes care of the communication leaving the processor uninvolved.

(Culler and Singh [48]). In shared-memory systems, communication can be two-sided, if the interprocessor communication is performed via a common buffer. The source processor writes the data into the common buffer and the destination processor reads it from there. For example, the implementation of MPI on the SGI Origin 2000 works this way (White and Bova [199]). In general, the involvement of the source and the destination processor can have different extents.

One-Sided Communication is said to be one-sided, if only one of the two participating processors is involved (Figure 8.1(*b*)). Thus, this type of communication is limited to shared-memory systems: either the source processor writes the data into its destination location (shared-memory "put") or the destination processor reads the data from its source location (shared-memory "get"). For one-sided communication it is essential that the active processor, that is, the processor that performs the communication, has direct access to the data location. Note that the shared-memory communication does not have to be transparent. Special devices or routines might be used by the active processor, but the memory access must be performed without the involvement of the passive processor. For example, Cray T3E uses special registers for remote memory access (Culler and Singh [48]).

Third-Party In third-party interprocessor communication, the data transfer is performed by dedicated communication devices, as illustrated in Figure 8.1(*c*). The communication devices must be able to directly access the local and the remote memory without involvement of the processor (i.e., it must possess some kind of a direct memory access (DMA) engine). The processor only informs the communication device of the memory area it wants transferred and the rest of the communication is performed by the device. Thus, the communication overhead on the processor takes approximately constant time, independent of the amount of data to be transferred. All devices together comprise the communication subsystem of the parallel computer, which autonomously handles all interprocessor communication. Examples for machines that possess such subsystems are the Meiko CS-2 (Alexandrov et al. [9]) or the IBM SP-2 [96]. The IBM Blue Gene/L (van der Steen and Dongarra [193]) has two identical PowerPC processors per module, but it can be configured to employ one of them as either a communication processor or a normal processor for computation. In the former case, one processor acts like a third-party communication device for the other processor.

It is important to realize that a processor engaged in communication cannot continue with computation. The charts of Figure 8.1 illustrate this by placing the involvements ("send/receive," "get/put") onto the time lines of the processors. On the contrary, in third-party communication it is the device, not the processor, that is occupied during the communication process.

Task scheduling, under both the classic model and the contention model, assumes the third-party type of interprocessor communication (Property 4 of Definitions 4.3 and 7.10). Consequently, the produced schedules overlap computation with communication; that is, a processor executes tasks while data is entering and leaving. However, this overlap can only exist on systems with a dedicated communication subsystem. On other systems, those that perform one- or two-sided communication, the produced schedules are unrealistic and thereby inappropriate. The experiments referenced in Section 7.5.4 demonstrated the negative consequences on the accuracy and the execution times of the schedules.

Among the communication types an order can be established regarding the degree of the processor involvement: two-sided, one-sided, and third-party. While the type of communication is restricted by the hardware capabilities of the target system, the software layer employed for interprocessor communication can increase the degree of involvement. For example, in a shared-memory system, communication can be one sided, but the software layer might use a common buffer, which turns it into two-sided communication (see two-sided above). This effect is not uncommon, as shown by the analysis of MPI implementations on common parallel systems by White and Bova [199]. It becomes important, since MPI is the standard for message passing and message passing is the dominant programming paradigm for parallel systems. Furthermore, almost all of today's parallel applications are written using a higher level

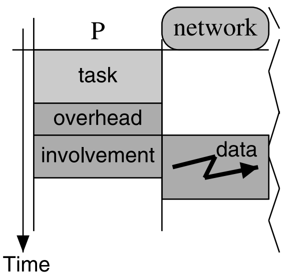


Figure 8.2. Decomposing of the processor's participation in communication into overhead and direct involvement.

programming paradigm in order to allow portability (e.g., MPI, PVM, OpenMP, HPF), which might be subject to the same effect. Task scheduling should of course reflect the effective type of involvement, taking into account the software layer employed in the code generation.

8.1.2 Involvement Characteristics

The first rough classification of interprocessor communication is established by its separation into three different types. For a more precise characterization, however, the involvement of the processor must be examined further. In the following it is shown how the processor's involvement can be decomposed into overhead and direct involvement.

Overhead The first component of the processor's involvement is the communication setup or *overhead*. From the initiation of the communication process until the first data item is put onto the network, the processor is engaged in preparing the communication. An overhead is in general also imposed on the destination processor after the data has arrived until it can be used by the receiving task.

The overhead consists mainly of the path through the communication layers (e.g., MPI [138], Active Messages (von Eicken et al. [197]), TCP/IP) and therefore becomes larger with the abstraction level of the communication environment. Typical tasks performed by these environments are packing and unpacking of the data, buffering, format conversions, error control, and so on. The communication hardware can also impose an overhead for control of the communication device, especially with third-party communication.

Note that the overhead does not arise for communication between tasks executed on the same processor. Therefore, it cannot be made part of the computation reflected by the origin and the destination tasks of the communication.

The importance of the overhead depends on the granularity of the task graph. For fine grained tasks, the overhead might have a significant influence on the program execution.

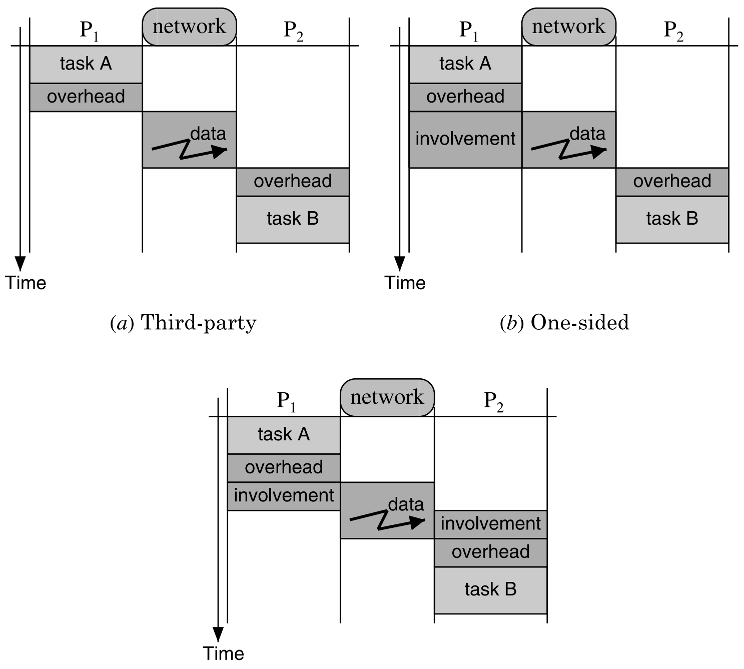
Direct Involvement After the communication has been prepared by the processor during the overhead, any further participation in communication is the direct involvement of the processor. It is characterized by the circumstance that data is already in transit on the communication network. Figure 8.2 focuses on the behavior of the origin processor in interprocessor communication, but the same principles are also valid for the destination processor. It features both the overhead and the direct involvement of the processor and thereby illustrates their differences. The direct involvement begins simultaneously with the data transit on the network, subsequent to the overhead. From now on, the term *involvement* means the direct involvement of the processor and the term *overhead* is used for the pre- and postphases discussed earlier. An example for direct involvement is the memory copy performed by a processor in shared-memory systems. The involvement only ends when the store command for the last data word has been executed. If the amount of data is small, the time of the processor involvement is short in comparison to the overhead, in particular, when only one data item is transferred.

With the notions of overhead and involvement, the generalized treatment of the three types of communication is possible. The overhead is a common part of interprocessor communication, independent of the type of communication. As mentioned earlier, the overhead is typically dominated by the communication layer and not by the underlying communication hardware. Thus, third-party communication is affected like two-sided and one-sided communication. In contrast, the involvement largely depends on the type of communication.

Length of Overhead and Involvement The distinction between communication types becomes obsolete if the communication is described in terms of overhead and involvement. Therefore, it is assumed that overhead and involvement are imposed on both the source and the destination processor. Then, one communication type differs from another merely by the length of the two components.

Overhead The overhead is usually of constant time on both processors, albeit of possibly different length. As explained earlier, the overhead is mainly due to the path through the communication layer, which does not change significantly for different data transfers. In other words, the overhead is usually independent of the data size. In some environments, however, data might be copied into and from a buffer, which is of course an operation taking time proportional to the data size. Examples are some MPI implementations as described by White and Bova [199].

Involvement The length of the involvement is primarily determined by the type of communication. Logically, this time is zero on some processors, namely, on both in third-party communication and on one—either the sending or the receiving processor—in one-sided communication. Figure 8.3 illustrates overhead and involvement in examples of all three types of communication. The involvement is omitted in Figure 8.3(*a*) and on P_2 in Figure 8.3(*b*), since it is zero due to the communication type.



(*a*) Third-party (*b*) One-sided

(*c*) Two-sided

Figure 8.3. Overhead and involvement in the three communication types: (*a*) no involvement achieved with DMA (zero-copy network devices); (*b*) shared-memory "put"; (*c*) only partial involvement, for example, shared-memory using a common buffer or communication devices without DMA.

The third-party communication as shown in Figure 8.3(*a*) reflects systems, where the communication subsystem possesses a DMA engine so that the processor does not need to copy the data. Nevertheless, the processor must call communication functions that initialize the network device and, on the receiving side, an interrupt must inform the processor about the incoming data, hence the overheads. Such interprocessor communication needs *zero-copy* capable devices, for example, the IBM SP-2 network adapter [96] supports zero-copy communication.

Figure 8.3(*b*) depicts one-sided communication in a shared-memory system. The source processor is the active one and copies the data into the destination location ("put"). It is involved during almost the entire communication—only the last word is transferred after the end of its involvement. For sufficiently long messages the difference between involvement and network usage can be neglected. The passive destination processor is of course not involved at all, yet it might be subject to an overhead, for example, when it becomes aware of the incoming data.

As already insinuated in Figure 8.2, involvement of the processor does not always last for the duration of the entire communication. This is the case in the example of two-sided communication shown in Figure 8.3(*c*). The example reflects shared-memory communication via a common buffer. Both processors are only involved in communication during one-half of the network activity: the source processor while writing into the common buffer and the destination processor while reading from the buffer.

Another case of partial involvement can be given on a system with a communication device or network adapter that performs the data transfer over the network, but the processors have to copy the data to and from it. The time the processor is involved is a function of the data size, but the processor is not engaged during the entire communication. Note that the communication device is considered part of the communication network.

Packet Based Communication The clear separation of overhead and involvement is not so obvious for packet based communication. For instance, consider two-sided interprocessor communication as in the last example and assume that the communication is packet based. At the beginning of a communication the sending processor goes through the communication layer, which is represented by the overhead. If the data has at most the size of one packet, it is copied to the network adapter, which then sends the message to its destination. This behavior can very well be captured with the notions of overhead and involvement. However, a long message must be split into several packets. During the transfer of each packet on the network, the processor, is not involved and can continue computation. But the processor experiences a new, although smaller, overhead and involvement for submitting the next packet to the adapter, after the transfer of the previous one finished. The situation is similar on the receiving processor, which participates in the reception and assembling of the packets.

Like the treatment of packet based communication in edge scheduling (Section 7.3.1), the packet based communication can be approximated with a single overhead and a single involvement. Figure 8.4 illustrates this for the previously described example. The various overheads and involvements (Figure 8.4(*a*)) of all packets are unified into one overhead and one involvement (Figure 8.4(*b*)). Of course, the period of network activity is identical in both views. Note that the single overhead in Figure 8.4(*b*) only corresponds to the initial overhead in the packet view (Figure 8.4(*a*)). The small incremental overheads are accredited to the involvement. This is sensible, as those overheads happen concurrently with the network activity and can thus be considered involvement. If instead they were accredited to the initial overhead, the network activity would be shifted to a later time interval, since its start would be delayed correspondingly.

The notions of overhead and involvement are very flexible as they allow the unified description of all three types of interprocessor communication, even when the processors' participation is only partial. At the same time they are intuitive and, as will be seen in the next sections, easily integrate with edge scheduling. The separation into overhead and involvement is also more general than the approach taken by the LogP model, as will be shown next. After that, the notions of overhead and involvement are employed in the task scheduling strategy that considers the involvement of the processor in communication.

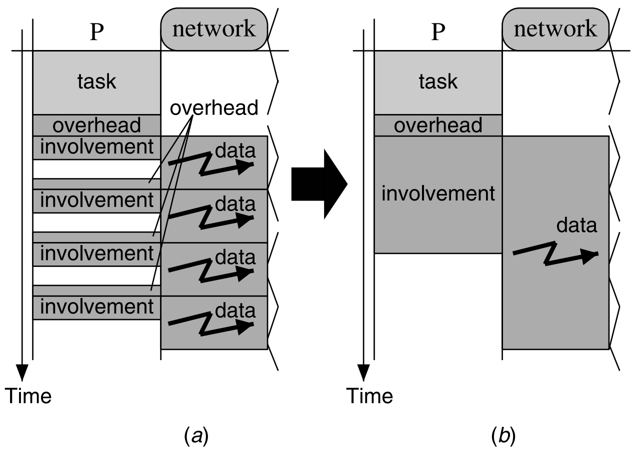


Figure 8.4. Overhead and involvement for a packet based communication: (*a*) packet view and (*b*) the approximation as one overhead and one involvement.

8.1.3 Relation to LogP and Its Variants

The LogP model (Section 2.1.3) was motivated by a problem similar to the one investigated in this chapter: the popular PRAM model for algorithm design and complexity analysis is not sufficiently realistic for modern parallel systems. Therefore, it is interesting to compare the discussed notions of overhead and scheduling with the approach of LogP, and some of its variants, since LogP also recognizes that the processor often participates in communication.

LogP LogP (Culler et al. [46, 47]) differs in one important aspect from the discussed notions of overhead and involvement. The three parameters L, o, and g are specified assuming a message of some fixed short size M (Section 2.1.3). So while LogP's approach is equivalent to the notions of overhead and involvement for such a short message M—o corresponds to the overhead and the involvement is negligible or can be considered part of the overhead—it is quite different for a long message. In LogP, a long message of size N must be modeled as a series of $\lceil N/M \rceil$ short messages. Figure 8.5 illustrates the sending of a message of size $6 \times M$; the reception on the destination processor works correspondingly. Since LogP does not distinguish between overhead and involvement, both are represented by the several o's of the long message. Thus, the communication cost imposed on the processor is modeled as being linear in the message size. Consequently, the LogP model does not capture the nature of the large overhead of communication layers such as MPI or even the smaller, but still significant, overhead (Culler et al. [47]) of communication environments such as Active Messages. This overhead is typically only paid for once for every communication, independent of the message size. This is especially an issue for third-party communication, because the overhead is typically of constant length independent of message size.

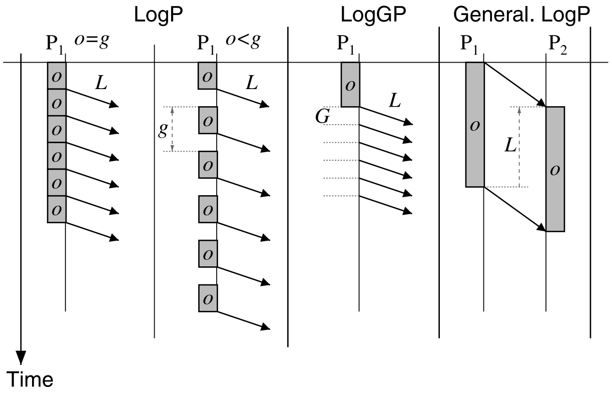


Figure 8.5. Sending a long message of size $6 \times M$ in LogP, LogGP, and generalized LogP model.

LogP was proposed as a model of parallel computing for algorithm design and evaluation. Its basis on small messages reflects that aim and in fact Culler et al. [47] argue that the overhead of high level communication layers is computation and should be modeled as such. From the task scheduling point of view, every additional computation due to communication is a cost that only has to be paid for interprocessor communication and is not inherent to the execution of tasks. A precise modeling of this overhead is therefore significant for scheduling.

A further limit of the LogP model is its basis on two-sided communication. Both processors are involved and both are imposed the cost o for every small message. That is, one-sided communication cannot be modeled accurately in LogP.

LogGP Alexandrov et al. [9] address one limitation of LogP by incorporating long messages into the model. Their new model, called LogGP, is a simple extension of LogP introducing only one additional parameter, G, which is the gap per byte for *long messages*. The reciprocal of G characterizes the available per-processor communication bandwidth in case of a long message. For a short message of the implicit fixed size M, LogGP is identical to LogP. For a long message, however, o is only paid once. After this time o, the first message M is sent into the network. Subsequent messages of size M take G time each to get out. Figure 8.5 depicts the sending of a long message in LogGP. The processor is only busy during o; the rest of the time it can perform computation, that is it can overlap computation with communication. Hence, the long message extension of LogGP represents third-party communication. LogGP is therefore not realistic for systems that do not possess a dedicated communication subsystem. Furthermore, the network capacity of LogP (at most $\lceil L/g \rceil$ messages) is not redefined for LogGP.

Generalized LogP Löwe et al. [131] and Eisenbiegler et al. [59] generalize the LogP model by making the parameters L, o, and g functions of the message size instead of constant values. This formal change also brings some changes to the interpretation of the parameters. The parameter o is no longer only the overhead of the communication, but also the involvement of the processor in communication. With a message size dependent o, it is now possible that the first data item of the message arrives at its destination before the last data item has been sent. This situation is illustrated in Figure 8.5. As a consequence, Löwe et al. [131] and Eisenbiegler et al. [59] allow the latency L to adopt negative values, as shown in the example. Generalized LogP can reflect symmetrical interprocessor communication costs, that is, the costs of two-sided and third-party communication. Its basis on functions to describe the parameters captures the overhead and the involvement of these kinds of communications accurately. One-sided communication, which has unsymmetrical costs, is not represented accurately, since the parameters are the same for both sending and receiving of a message. The inability to distinguish between overhead and involvement makes it also impossible to analyze contention for network resources. In particular, the network capacity of the LogP model—only $\lceil L/g \rceil$ messages can be concurrently in transit—is not defined for generalized LogP.

From the above comparison it follows that the notions of overhead and involvement as introduced in this section are more flexible than LogP and its variants. While LogGP and generalized LogP address some of the shortcomings of LogP, the separation of overhead and involvement is more general, for example, it also covers one-sided communication. Moreover, this separation allows the new scheduling model, introduced in the next section, to be based on the contention model. It therefore inherits the flexible and powerful modeling of end-point and network contention. In scheduling based on LogP and its variants (e.g., Boeres and Rebello [25], Kalinowski et al. [98], Löwe et al. [131]), network contention is completely ignored.

LogP is a model for algorithm design and complexity analysis. As such, a certain simplicity is essential, for example, an identical and constant o on both sides of the communication. The overhead and involvement introduced in this section are not subject to such restriction. The possible increase of conceptual complexity is justified, since these notions are used for scheduling performed by a computer algorithm and not for algorithm design performed by a human.

8.2 INVOLVEMENT SCHEDULING

The notions of overhead and involvement discussed in the last section are the key concepts to enhance task scheduling toward the awareness of processor involvement in communication (Sinnen [172], Sinnen et al. [180]). In the first step, a new target system model is defined.

Definition 8.1 (Target Parallel System—Involvement–Contention Model) *A target parallel system $M = (TG, \omega, o, i)$ consists of a set of possibly heterogeneous processors* $\mathbf{P}$ *connected by the communication network $TG = (\mathbf{N}, \mathbf{P}, \mathbf{D}, \mathbf{H}, b)$,* *according to Definition 7.5. This system has Properties 1 to 3 of the classic model of Definition 4.3:*

1. *Dedicated system*
2. *Dedicated processor*
3. *Cost-free local communication*

So in comparison with the contention model (Definition 7.10), the involvement–contention model departs from the assumption of a dedicated communication subsystem (Property 4). Instead, the role of the processors in communication is described by the new components o—for overhead—and i—for (direct) involvement.

Definition 8.2 (Overhead and Involvement) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph and $M = ((\mathbf{N}, \mathbf{P}, \mathbf{D}, \mathbf{H}, b), \omega, o, i)$ a parallel system. Let $R = \langle L_1, L_2, \dots, L_l \rangle$ be the route for the communication of edge $e \in \mathbf{E}$ from $P_{\text{src}} \in \mathbf{P}$ to $P_{\text{dst}} \in \mathbf{P}$, $P_{\text{src}} \neq P_{\text{dst}}$.*

Overhead. *$o_s : \mathbf{E} \times \mathbf{P} \to \mathbb{Q}_0^+$ and $o_r : \mathbf{E} \times \mathbf{P} \to \mathbb{Q}_0^+$ are the communication overhead functions (the subscript s stands for send and r stands for receive). $o_s(e, P_{\text{src}})$ is the computational overhead, that is, the execution time, incurred by processor P_{src} for preparing the transfer of the communication associated with edge e and $o_r(e, P_{dst})$ is the overhead incurred by processor P_{dst} after receiving e.*

Involvement. *$i_s : \mathbf{E} \times \mathbf{L}_s \to \mathbb{Q}_0^+$ and $i_r : \mathbf{E} \times \mathbf{L}_r \to \mathbb{Q}_0^+$ are the communication involvement functions, where $\mathbf{L}_s$ is the set of links that leave processors, $\mathbf{L}_s = \{H \in \mathbf{H} : H \cap \mathbf{P} \neq \emptyset\} \cup \{D_{ij} \in \mathbf{D} : N_i \in \mathbf{P}\}$, and $\mathbf{L}_r$ is the set of all links that enter processors, $\mathbf{L}_r = \{H \in \mathbf{H} : H \cap \mathbf{P} \neq \emptyset\} \cup \{D_{ij} \in \mathbf{D} : N_j \in \mathbf{P}\}$. $i_s(e, L_1)$ is the computational involvement, that is, execution time, incurred by processor P_{src} during the transfer of edge e and $i_r(e, L_l)$ is the computational involvement incurred by P_{dst} during the transfer of e.*

This is the general definition of overhead and involvement for arbitrary, possibly heterogeneous systems. Therefore, the overhead is made a function of the processor and the involvement a function of the utilized communication link. As discussed in the previous section, the overhead depends largely on the employed communication environment and is thereby normally unaffected by the utilized communication resources. In contrast, the involvement depends to a large extent on the capabilities of the utilized communication resources. Hence, the processor involvement is characterized by the outgoing or incoming link utilized for a communication. Figure 8.6 illustrates this for a parallel system, where each processor is incident on several links. A communication between the two depicted processors inflicts the involvement associated with the undirected link between them, while a communication into and from the not shown rest of the network inflicts another involvement, associated with the directed edges that are utilized for these communications.

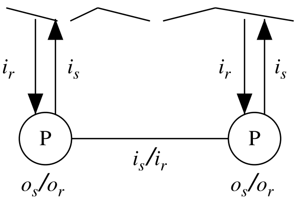


Figure 8.6. Extract of a parallel system represented by the involvement–contention model.

Of course, if the overhead changes with the link employed for a communication, it can also be defined as a function of the link, as done for the involvement.

With the distinction between the sending (o_s, i_s) and the receiving side (o_r, i_r) of communication, all three types of communication—third-party, one-sided, and two-sided—can be precisely represented. The corresponding functions are simply defined accordingly, for example, $i_s(e, L) = i_r(e, L) = 0$ for involvement-free third-party communication.

Homogeneous Systems For homogeneous systems or systems that have homogeneous parts, the definition of overhead and involvement can be simplified.

In the case where the entire system has only one link per processor (that encompasses full duplex links represented by two counterdirected edges), i_s and i_r can be defined as a function of the processor, instead of as a function of the link, that is, $i_{s,r}(e, L) \Rightarrow i_{s,r}(e, P)$.

If interprocessor communication is symmetric, that is, the overhead and the involvement on both sides of the communication have the same extent (which is possible in third-party and two-sided communications), the distinction between the sending and receiving side can be abolished, resulting in only one function o and one function i, that is, $i_s(e, L) = i_r(e, L) = i(e, L)$ and $o_s(e, P) = o_r(e, P) = o(e, P)$.

Furthermore, in homogeneous systems and considering communication links, the involvement functions can be defined globally, that is, $i_{s,r}(e, L) = i_{s,r}(e)$. The same is true for the overhead functions in the case of homogeneous processors, that is, $o_{s,r}(e, P) = o_{s,r}(e)$.

8.2.1 Scheduling Edges on the Processors

In contention aware scheduling (Section 7.4), the edges of the task graph are scheduled onto the links of the communication network, like the nodes are scheduled on the processor. Incorporating overhead and involvement into contention aware task scheduling is accomplished by extending edge scheduling so that edges are also scheduled on the processors. To do so, the first step is to define the start and finish times of an edge on a processor.

Definition 8.3 (Edge Start and Finish Times on Processor) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph and $M = ((\mathbf{N}, \mathbf{P}, \mathbf{D}, \mathbf{H}, b), \omega, o, i)$ a parallel system.*

The start time $t_s(e,P)$ of an edge $e \in \mathbf{E}$ on a processor $P \in \mathbf{P}$ is the function $t_s : \mathbf{E} \times \mathbf{P} \to \mathbb{Q}^+$.

Let $R = \langle L_1, L_2, \ldots, L_l \rangle$ be the route for the communication of edge $e \in \mathbf{E}$ from $P_{\text{src}} \in \mathbf{P}$ to $P_{\text{dst}} \in \mathbf{P}$, $P_{\text{src}} \neq P_{\text{dst}}$. The finish time of e on P_{src} is

$$t_f(e, P_{\text{src}}) = t_s(e, P_{\text{src}}) + o_s(e, P_{\text{src}}) + i_s(e, L_1) \tag{8.1}$$

and on P_{dst} is

$$t_f(e, P_{\text{dst}}) = t_s(e, P_{\text{dst}}) + o_r(e, P_{\text{dst}}) + i_r(e, L_l). \tag{8.2}$$

Figure 8.7 illustrates scheduling under the involvement–contention model. The edge e_{AB} of the simple task graph is not only scheduled on the two links between the communicating processors, but also on both processors in order to reflect the incurred overhead and their involvement. Consequently, the execution time of an edge on a processor is the sum of the overhead and the involvement (see Eqs. (8.1) and (8.2)).

Clearly, for a sensible scheduling of the edges on the processors, some conditions must be formulated.

Like a node, an edge scheduled on a processor represents computation, precisely the computation necessary for the communication of the edge. Thus, the exclusive processor allocation Condition 4.1 also applies to edges scheduled on the processors. Remember, this is a requirement imposed by Property 2 of the system model (Definition 8.1).

An edge is only scheduled once on a processor, from which it follows that a separation between the overhead and the involvement is not possible. This is already ensured by the definition of the finish time of an edge on a processor (Eqs. (8.1) and (8.2)). Even for systems where the separation of overhead and involvement might

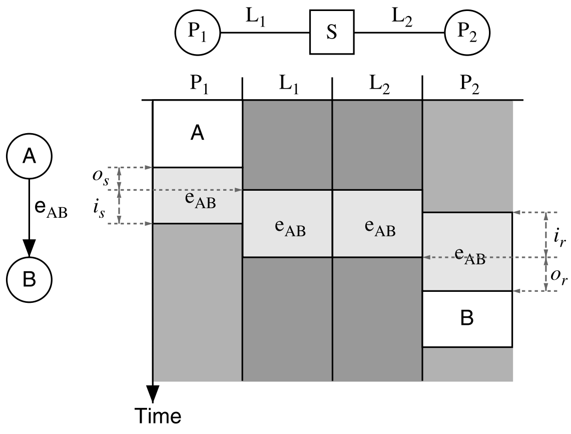


Figure 8.7. Scheduling under the involvement–contention model: edges are also scheduled on the processors. S is a switch or other processor.

occur, the improvement on accuracy by allowing the separation would presumably not outweigh the more complicated scheduling.

For a meaningful and feasible schedule, the scheduling of the edges on the processors must obey Condition 8.1.

Condition 8.1 (Causality in Involvement Scheduling) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $M = ((\mathbf{N}, \mathbf{P}, \mathbf{D}, \mathbf{H}, b), \omega, o, i)$ *a parallel system. Let* $R = \langle L_1, L_2, \ldots, L_l \rangle$ *be the route for the communication of edge* $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, *from* $P_{\text{src}} \in \mathbf{P}$ *to* $P_{\text{dst}} \in \mathbf{P}$, $P_{\text{src}} \neq P_{\text{dst}}$.

To assure the node strictness (Definition 3.8) of n_i

$$t_s(e_{ij}, P_{\text{src}}) \geq t_f(n_i, P_{\text{src}}). \tag{8.3}$$

Edge e_{ij} *can be transferred on the first link* L_1 *only after the overhead is completed on the source processor* P_{src}:

$$t_s(e_{ij}, L_1) \geq t_s(e_{ij}, P_{\text{src}}) + o_s(e_{ij}, P_{\text{src}}). \tag{8.4}$$

To assure the causality of the direct involvement on the destination processor P_{dst},

$$t_s(e_{ij}, P_{\text{dst}}) \geq t_f(e_{ij}, L_l) - i_r(e_{ij}, L_l). \tag{8.5}$$

The three inequalities can be observed in effect in Figure 8.7. Edge e_{AB} starts on P_1 after the origin node A finishes (inequality (8.3)). On the first link L_1, e_{AB} starts after the overhead finishes on P_1 (inequality (8.4)), at which time the involvement of P_1 begins. And last, e_{AB} starts on P_2 so that the involvement finishes at the same time as e_{ij} on L_2 (inequality (8.5)).

Discussion of Condition 8.1

Order of Overhead–Involvement For simplicity, Condition 8.1 fixes the temporal order between overhead and involvement in the way it has been assumed so far: first the overhead then the involvement on the source processor (established by Eq. (8.4)) and the other way around on the destination processor (established by Eq. (8.5)).

Freedom of Node–Edge Order It is not required by Condition 8.1 that an edge e_{ij} is scheduled on the source processor immediately after the termination of the origin node n_i. If n_i has multiple leaving edges, this is not possible anyway—only one edge can start immediately after n_i—but other edges, not leaving n_i, or even nodes might be scheduled between n_i and e_{ij}. The same holds for e_{ij} and the receiving node n_j, whereby e_{ij} must of course finish before n_j starts. This freedom in the scheduling order of the nodes and edges on the processors grants a high degree of flexibility to scheduling algorithms. Communications can be delayed, while executing other nodes

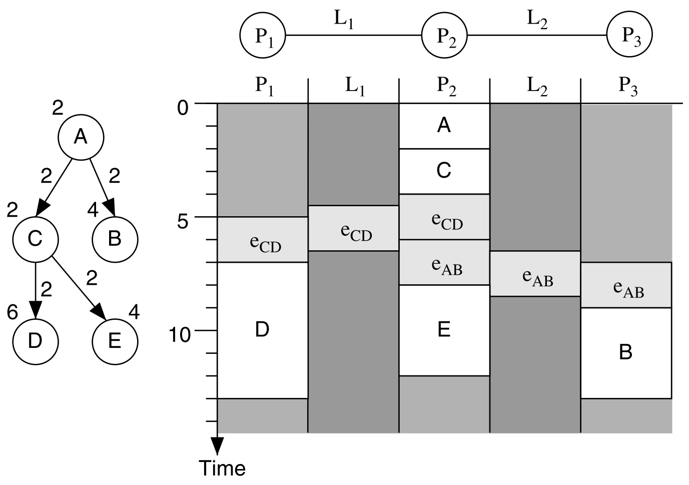


Figure 8.8. Example for a schedule under the involvement–contention model, where a delayed communication (e_{AB}) results in a shorter schedule length. $O_{s,r}(e,P) = 0.5, i_{s,r}(e,L) = 0.75\varsigma(e,L)$.

in the meantime, to obtain more efficient schedules. Figure 8.8 shows an example where the delayed scheduling of edge e_{AB} reduces the schedule length. If it were scheduled before C on P_2, the start of the communication e_{CD} would be delayed by two time units and in turn the start of D.

It is essential that the code generated from such a schedule respects the defined order of tasks and communications. For comparison, under the classic model it is assumed that all communications are sent immediately after the origin node finishes (Section 4.1). Even though the order of communications is established in contention aware scheduling, the relation to the node finish times is not clearly defined, as nodes are executed concurrently with the communications. Thus, only scheduling under the involvement–contention model admits the precise definition of the task and communication order.

Of course, the data to be sent must remain unmodified until it is sent. This might require additional buffering, for example, if an array holding the data is reutilized by other tasks on the sending processor. Yet, for nonblocking communications additional buffering is necessary anyway. In nonblocking communication the processor that initiates a communication does not have to actively wait (blocking wait) until its communication partner becomes available; that is, it can execute other tasks in the meanwhile.

Nonaligned Scheduling Condition 8.1 does not strictly adhere to the notion of the direct processor involvement introduced in the previous section. There, it is stated that a processor is involved in communication at the same time the communication is sent to or received from the network. But inequalities (8.4) and (8.5) also allow a later start

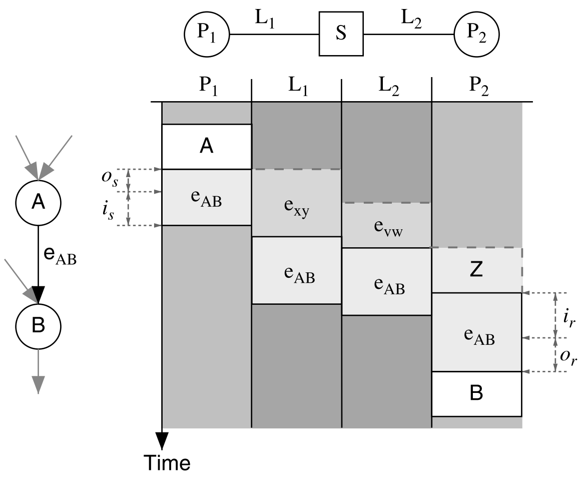


Figure 8.9. Nonaligned scheduling of the edges on the processors and links; chart and task graph are only extracts.

of the edge on the first link and an earlier start on the last link, so that the involvement is scheduled earlier (on the source processor) or later (on the destination processor) than the network activity. An extract of a corresponding schedule is depicted in Figure 8.9. Due to contention, the start of edge e_{AB} is delayed on the links and on P_2 so that the alignment with the involvement on the processors is not given.

This relaxation corresponds to the nonaligned scheduling approach in edge scheduling (Section 7.3.1). An enforced aligned scheduling, where the involvement part of the edge on the processors is aligned with the edge on the links, would result in the same issues as for the aligned edge scheduling approach (Section 7.3.1). Many idle time periods would be created on the processors, and the scheduling of an edge, on the processors and links, became more complex. The discussion of scheduling algorithms for the involvement–contention model in the next section makes this clearer.

Scheduling As for the edge scheduling on the links (Section 7.3.2), a scheduling condition is formulated for the correct choice of an idle time interval into which an edge can be scheduled on a processor, with either the end or the insertion technique (Sections 5.1 and 6.1).

Condition 8.2 (Edge Scheduling Condition on a Processor) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $M = ((\mathbf{N}, \mathbf{P}, \mathbf{D}, \mathbf{H}, b), \omega, o, i)$ *a parallel system. Let* $R = \langle L_1, L_2, \ldots, L_l \rangle$ *be the route for the communication of edge* $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, *from* $P_{\text{src}} \in \mathbf{P}$ *to* $P_{\text{dst}} \in \mathbf{P}$, $P_{\text{src}} \neq P_{\text{dst}}$. *Let* $[A, B]$, $A, B \in [0, \infty]$, *be an idle time interval on P,* $P \in \{P_{\text{src}}, P_{\text{dst}}\}$, *that is, an interval in which no other edge or node is scheduled* *on P. Edge e_{ij} can be scheduled on P within $[A, B]$ if*

$$B - A \geq \begin{cases} o_s(e_{ij}, P_{\mathrm{src}}) + i_s(e_{ij}, L_1) & \mathit{if}\, P = P_{\mathrm{src}} \\ o_r(e_{ij}, P_{\mathrm{dst}}) + i_r(e_{ij}, L_l) & \mathit{if}\, P = P_{\mathrm{dst}} \end{cases}, \tag{8.6}$$

$$B \geq \begin{cases} t_f(n_i, P_{\mathrm{src}}) + o_s(e_{ij}, P_{\mathrm{src}}) + i_s(e_{ij}, L_1) & \mathit{if}\, P = P_{\mathrm{src}} \\ t_f(e_{ij}, L_l) + o_r(e_{ij}, P_{\mathrm{dst}}) & \mathit{if}\, P = P_{\mathrm{dst}} \end{cases}. \tag{8.7}$$

This condition corresponds to Condition 7.3 for edge scheduling on the links. It ensures that the time interval $[A, B]$ adheres to the inequalities (8.3) and (8.5) of the causality Condition 8.1. For a given time interval, the start time of e_{ij} on P_{src} and P_{dst} is determined as follows.

Definition 8.4 (Edge Scheduling on a Processor) *Let $[A, B]$ be an idle time interval on P adhering to Condition 8.2. The start time of e_{ij} on P is*

$$t_s(e_{ij}, P) = \begin{cases} \max\{A, t_f(n_i)\} & \mathit{if}\, P = P_{\mathrm{src}} \\ \max\{A, t_f(e_{ij}, L_l) - i_r(e_{ij}, L_l)\} & \mathit{if}\, P = P_{\mathrm{dst}} \end{cases}. \tag{8.8}$$

So, as with the scheduling on the links (Definition 7.9), the edge is scheduled as early as possible within the limits of the interval. Of course, the choice of the interval should follow the same policy on the links and on the processors; that is, either the end or insertion technique should be used.

The Overhead and Involvement Functions In all definitions and conditions related to scheduling under the involvement–contention model, overhead and involvement are simply treated as functions for the sake of generality. Nevertheless, these functions have a certain nature for most parallel systems, which is studied next. Generally, the overhead and the involvement functions are at most linear in the communication volume for probably all parallel systems.

The overhead is of constant time for many parallel systems and their communication environment. As mentioned before, the overhead is normally the path through the communication layer. Thus,

$$o_s(e, P) = o_s(P), \; o_r(e, P) = o_r(P), \tag{8.9}$$

that is, the overhead merely depends on the processor speed, so it is constant for homogeneous systems. Sometimes, however, the data to be transferred is copied between buffers during the overhead by which the overhead becomes a linear function of the communication volume.

The involvement is by definition not longer than the network activity. On the sending side, the involvement stops at the latest when all data has been sent across the first link, and on the receiving side, the involvement does not start before the communication begins on the last link. Hence, the involvement must be smaller than the communication time on the respective link:

$$i_s(e, L_1) \leq \varsigma(e, L_1), \tag{8.10}$$

$$i_r(e, L_l) \leq \varsigma(e, L_l). \tag{8.11}$$

Since the involvement represents the actual data transfer performed by the processor, this time is typically proportional to the communication time of the first/last link:

$$i_s(e, L_1) = C_s \bullet \varsigma(e, L_1), \tag{8.12}$$

$$i_r(e, L_l) = C_r \bullet \varsigma(e, L_l), \tag{8.13}$$

where C_s, $C_r \in [0, 1]$. For example, $C_s = C_r = 0.5$ for the two-sided memory copy as shown in Figure 8.3(c).

Consequently, in many common parallel systems, the overhead and the involvement are simply characterized by constants. Furthermore, these constants are independent of the task graph and they only need to be determined once to describe the target system for the scheduling of any task graph.

On the other hand, the functions might also be used to describe in detail the behavior of interprocessor communication. For instance, consider a parallel system, where the overhead of third-party communication is quite high. For few data words, other means of communication (e.g., memory copy by the processor) might be cheaper than setting up the communication device. In this case, there is a threshold data size for which it is worthwhile to employ the communication device. The overhead and involvement functions can be formulated accordingly. Similar things can happen on the protocol level of the communication environment, where an optimized procedure is employed for short messages (White and Bova [199]).

Note that the size of the involvement does not depend on the contention in the network. The assumption is that if the processor has to wait to send or receive a communication due to contention, this wait is passive or nonblocking, which means it can perform other operations in the meantime.

8.2.2 Node and Edge Scheduling

Few alterations are imposed by the new model on the edge scheduling on the links and on the scheduling of the nodes.

Edge Scheduling on Links While inequalities (8.3) and (8.5) of Condition 8.1 are newly introduced for the scheduling of the edges on processors, Eq. (8.4) substitutes a constraint of edge scheduling. The edge scheduling Condition 7.3 assures that an edge starts on the first link of a route after the origin node has finished. With Eq.

(8.4) the start of the edge on the first link must be further delayed by at least the time of the communication overhead. Hence, the first case ($k = 1$) of inequality (7.7) of Condition 7.3, $B \geq t_f(n_i) + \varsigma(e_{ij}, L_1)$, becomes

$$B \geq t_s(e_{ij}, P_{\text{src}}) + o_s(e_{ij}, P_{\text{src}}) + \varsigma(e_{ij}, L_1). \tag{8.14}$$

The calculation of the edge's start time on the first link (Definition 7.9) must be modified accordingly. Thus, the first case ($k = 1$) of Eq. (7.8), $t_s(e_{ij}, L_1) = \max\{A, t_f(n_i)\}$, becomes

$$t_s(e_{ij}, L_1) = \max\{A, t_s(e_{ij}, P_{\text{src}}) + o_s(e_{ij}, P_{\text{src}})\} \tag{8.15}$$

The rest of the edge scheduling procedure is completely unaffected by the scheduling of the edges on the processors and remains unmodified.

Node Scheduling To adapt the scheduling of the nodes to the new model, it is only necessary to redefine the total finish time of the edge, which was originally defined for the classic model in Definition 4.6 and redefined for the contention model in Definition 7.11.

Definition 8.5 (Edge Finish Time) *Let $G = (\mathbf{V}, \mathbf{E}, w, c)$ be a task graph and $M = ((\mathbf{N}, \mathbf{P}, \mathbf{D}, \mathbf{H}, b), \omega, o, i)$ a parallel system. The finish time of $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, communicated from processor P_{src} to P_{dst}, $P_{\text{src}}, P_{\text{dst}} \in \mathbf{P}$, is*

$$t_f(e_{ij}, P_{\text{src}}, P_{\text{dst}}) = \begin{cases} t_f(n_i, P_{\text{src}}) & \textit{if } P_{\text{src}} = P_{\text{dst}} \\ t_f(e_{ij}, P_{\text{dst}}) & \textit{otherwise} \end{cases}. \tag{8.16}$$

As with the edge scheduling discussed earlier, the rest of the node scheduling procedure is completely unaffected by the scheduling of the edges on the processors and remains unmodified.

8.2.3 Task Graph

For scheduling under the contention model, the communication cost (i.e., the edge weight) was defined to be the average time a link is occupied with the communication associated with the edge (Definition 7.7). As elaborated in Section 7.4.1, this definition is compatible with the edge weight under the classic model, which represents the communication delay. Scheduling under the involvement–contention model is based on edge scheduling and adopts the communication cost definition of edge scheduling. Thus, it remains unmodified.

Yet, in practice, some caution is indicated. Under the contention model, the time an edge spends on a link corresponds to the communication delay. In consequence, pre- and postprocessing parts are either ignored or accredited to the communication cost represented by the edge weight. Under the involvement–contention model, the communication delay is also composed of the overheads inflicted on the communicating processors; hence, these overheads are not included in the edge weights (see below). Thus, task graph weights determined for different models are not necessarily identical.

Path Length Most metrics used for analyzing a task graph and its nodes are based on the path length, for example, node levels or the critical path. As stated in Section 4.4, the length of a path can be interpreted as its execution time, when all communications are interprocessor communications. Under the involvement–contention model, the total communication time, or communication delay, does not merely consist of the edge weight, but also of the overheads incurred by the sending and receiving processors. Thus, the path length must be adapted in order to still represent the execution time of the path.

Definition 8.6 (Path Length with Overhead) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $M = ((\mathbf{N}, \mathbf{P}, \mathbf{D}, \mathbf{H}, b), \omega, o, i)$ *a parallel system. Let* $\bar{o}_{s,r}(e) = \sum_{P \in \mathbf{P}} o_{s,r}(e, P)/|\mathbf{P}|$ *be the average overhead. The length of a path p in G is*

$$len(p) = \sum_{n \in p, \mathbf{V}} w(n) + \sum_{e \in p, \mathbf{E}} (\bar{o}_s(e) + c(e) + \bar{o}_r(e)). \tag{8.17}$$

Of course, nothing changes for the computation path length, where all communications are assumed to be local. All metrics based on path lengths (e.g., node levels) are used as before with the new definition. Thus, to establish node priorities, the schemes from the classic model and the contention model can be used directly.

In general, communication is more important in the involvement-contention model than in the other models. In particular, the involvement of the processors is a crucial aspect of communication. But the involvement does not “lie” on the path and is therefore not considered in its length.

8.2.4 NP-Completeness

As can be expected, scheduling under the involvement–contention model remains an NP-hard problem. This is easy to see, as the involvement model is based on the contention model, which is NP-hard. Therefore, the proof of the following NP-completeness theorem employs the straightforward reduction from the decision problem C-SCHED(G, M_{TG}) associated with the scheduling under the contention model (Theorem 7.1).

Theorem 8.1 (NP-Completeness—Involvement-Contention Model) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $M = ((\mathbf{N}, \mathbf{P}, \mathbf{D}, \mathbf{H}, b), \omega, o, i)$ *a parallel system.*

The decision problem INVOLVEMENT-CONTENTION-SCHED *(G, M), shortened to* IC-SCHED*, associated with the scheduling problem is as follows. Is there a schedule $\mathcal{S}$ for G on M with length $sl(\mathcal{S}) \leq T, T \in \mathbb{Q}^+$? IC-SCHED (G, M) is NP-complete.*

Proof. First, it is argued that IC-SCHED belongs to NP; then it is shown that IC-SCHED is NP-hard by reducing the NP-complete problem IC-SCHED (G_C, M_{TG}) (Theorem 7.1) in polynomial time to IC-SCHED.

Clearly, for any given solution $\mathcal{S}$ of IC-SCHED it can be verified in polynomial time that $\mathcal{S}$ is feasible (Algorithm 5, adapted for edge scheduling on links and processors) and $sl(\mathcal{S}) \leq T$; hence, IC-SCHED $\in$ NP.

For any instance of C-SCHED, an instance of IC-SCHED is constructed by simply setting $G = G_C$, $M = (M_{TG}, o, i)$, $o_{s,r} = 0$, $i_{s,r} = 0$, and $T = T_{\text{C-SCHED}}$; thus, the overhead and the involvement take no time. Obviously, this construction is polynomial in the size of the instance of C-SCHED. Furthermore, if and only if there is a schedule for the instance of IC-SCHED that meets the bound T, is there a schedule for the instance C-SCHED meeting the bound $T_{\text{C-SCHED}}$. □

$\alpha|\beta|\gamma$ Notation In terms of the $\alpha|\beta|\gamma$ notation (Section 6.4.1), the problem of scheduling under the involvement–contention model can be characterized with a simple extension of the notation used for the contention model (Section 7.4.2).

The essential aspect of the contention model is the fact that edges are scheduled. In Section 7.4.2 it was suggested to denote this by *-sched* in the β field, immediately after the specification of the communication costs (e.g., c_{ij}*-sched*). To indicate on which network the edges need to be scheduled, the α field was extended with a third subfield describing the network topology (e.g., *net* or *star*).

In the involvement–contention model, the edges are also scheduled and the β field is therefore left unchanged. What changes is the fact that edges also need to be scheduled on the processors. To indicate this, the α field is extended, with fields that indicate the presence of overhead and direct involvement.

The fourth subfield relates to the overhead and can take the following values:

- $\circ$. Communication inflicts no computational overhead on the sending or receiving processor.
- $o_{s,r}$. Communication inflicts computational overhead on the sending and on the receiving processor. Restrictions might be specified for the overhead in an obvious way as in the following examples.
  - $\{o_s, o_r\}$. Communication inflicts computational overhead only on the sending (o_s) or on the receiving processor (o_r).
  - o. Communication inflicts identical computational overhead on the sending and receiving processors, that is, $o_s = o_r = o$.

The fifth subfield relates to the (direct) involvement and can take the following values:

- $\circ$. Communication inflicts no computational involvement on the sending or receiving processor. This implies that there is a dedicated communication subsystem.
- $i_{s,r}$. Communication inflicts computational involvement on the sending and receiving processors. Restrictions might be specified for the involvement in an obvious way as in the following examples.

  $\{i_s, i_r\}$. Communication inflicts computational involvement only on the sending (i_s) or on the receiving processor (i_r). In other words, the communication type is one-sided.

  i. Communication inflicts identical computational involvement on the sending and receiving processors, that is, $i_s = i_r = i$.

So, unless both the fourth and the fifth field are empty, edges are also scheduled on the processors. With these notations, the general problem of scheduling under the involvement–contention model is specified by $P, net, o_{s,r}, i_{s,r} | prec, c_{ij}\text{-}sched | C_{\max}$ for a limited number (α's second subfield is empty) of homogeneous processors (P).

8.3 ALGORITHMIC APPROACHES

The integration of edge scheduling into task scheduling in order to achieve contention awareness had little to no effect on the fundamental scheduling methods, as was shown in Chapter 7. In contrast to scheduling on the links, the scheduling of the edges on the processors, which seems at first a simple extension, has a strong impact on the operating mode of scheduling algorithms (Sinnen [172], Sinnen et al. [180]).

Essentially, the problem is that at the time a free node n is scheduled, it is generally unknown to where its successor nodes will be scheduled. It is not even known if the corresponding outgoing communications will be local or remote. Thus, no decision can be taken whether to schedule n's leaving edges on its processor or not. Later, at the time a successor is scheduled, the period of time directly after node n might have been occupied with other nodes. Hence, there is no space left for the scheduling of the corresponding edge.

The general issue behind the described problem is not specific to a certain heuristic, for example, list scheduling, but it applies to all scheduling algorithms under the involvement–contention model. Also, scheduling under the LogP model faces the same problem with the scheduling of o for each communication (Kalinowski et al. [98]). Ideally, the processor allocations of the nodes are known *before* the scheduling.

This problem does not arise in contention scheduling, because communication can overlap with computation; that is, an edge can be scheduled on a link at the same time a node is executed on the incident processor.

Two different approaches to handle the described issue in scheduling under the involvement-contention model, which have reasonable complexity, can be distinguished: (1) direct scheduling and (2) scheduling based on a given processor allocation (Sinnen [172], Sinnen et al. [180]).

8.3.1 Direct Scheduling

Direct scheduling means that the processor allocation and the start/finish time attribution of a node are done in one single step. This is the dominant technique of most scheduling heuristics discussed so far. Thus, the direct scheduling approach is basically an attempt to adapt the known scheduling techniques to the involvement–contention model.

The essential question is *when* to schedule the edges on the processors. It is clear that in direct scheduling (i.e., a one-pass scheduling), the nodes considered for scheduling must be free. Otherwise, the start time of a node cannot be determined and it has to be set in a later step of the algorithm.

In the first attempt, the approach of list scheduling under the contention model is considered; that is, edges are scheduled at the same time as their destination nodes.

Scheduling Entering Edges Scheduling an edge together with its free destination node is surely the easiest approach. At the time an edge is scheduled, the source and the destination processor are known and the route can be determined. An edge can be scheduled in one go on the source processor, the links, and the destination processor. Unfortunately, in many cases, this approach generates inefficient schedules.

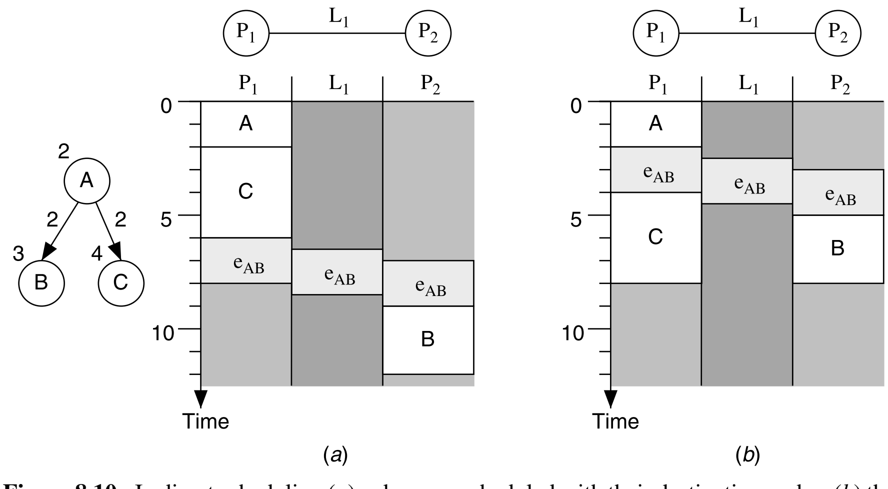


Figure 8.10. In direct scheduling (*a*), edges are scheduled with their destination nodes; (*b*) the optimal schedule. $o_{s,r}(e, P) = 0.5$, $i_{s,r}(e, L) = 0.75\varsigma(e, L)$.

Consider the example of Figure 8.10(a), where the nodes of the depicted task graph were scheduled in the order A, C, B, which is their bottom-level order. As communication e_{AB} was scheduled together with node B, the time period after node A was already occupied by C. A much shorter schedule length is achieved, when e_{AB} starts immediately after A as shown in Figure 8.10(b).

Even though there are cases where the delayed scheduling of communication can be of benefit (see Figure 8.8), in most situations the parallelization of the task graph is hindered with this approach. The start time of a node is delayed not only by the total communication time of an entering edge but also by the execution times of the nodes scheduled between the origin node and the entering edge on the origin node's processor (in the example of Figure 8.10(a) this is node C between A and e_{AB} on P_1). In fact, a list scheduling with start time minimization based on this approach would schedule all nodes in Figure 8.10 on P_1, as the parallelization is never beneficial: the start/finish time of each considered node is always earliest on P_1. Moreover, this even generalizes to any fork task graph: list scheduling with start time minimization would produce a sequential schedule!

Provisionally Scheduling Leaving Edges The direct application of the scheduling method from contention scheduling is inadequate under the new model. Consequently, it is necessary to investigate how edges can be scheduled earlier. Of course, the problem remains that at the time a free node is scheduled, it is not known to where its successors will be scheduled. Nevertheless, the scheduling of the leaving edges must be prepared in some way.

The most viable solution is to reserve an appropriate time interval after a node for the later scheduling of the leaving edges. This must be done in a worst case fashion, which means the interval must be large enough to accommodate all leaving edges. A straightforward manner is to schedule all leaving edges on the source processor, directly after the origin node. The scheduling of the edges on the links and the destination processors is not possible at that time, since the destination processors, and with them the routes, are undetermined. Fortunately, this is also not necessary, as the scheduling on the links and the destination processor can take place when the destination node is scheduled, in the way it is done under the contention model. If the destination node is scheduled on the same processor as the origin node, the corresponding edge, which was provisionally scheduled with the origin node, is simply removed from that processor.

As an example, Figure 8.11 depicts three Gantt charts illustrating the scheduling process of the task graph on the left side. First, A is scheduled on P_1, together with its three leaving edges (shown in Figure 8.11(a)); hence, the worst case that B, C, and D are going to be scheduled on P_2 is assumed. Indeed, node B is scheduled on P_2, which includes the preceding scheduling of e_{AB} on the link and on P_2. Next, C is scheduled on P_1. So the communication between A and C is local and e_{AC} is removed from P_2. The situation at this point is shown in Figure 8.11(b). Finally, D is scheduled on P_2 with the respective scheduling of e_{AD} on the link and P_2 (Figure 8.11(c)).

On heterogeneous systems, the described provisional scheduling of an edge on its source processor must consider that the involvement depends on the first link of the

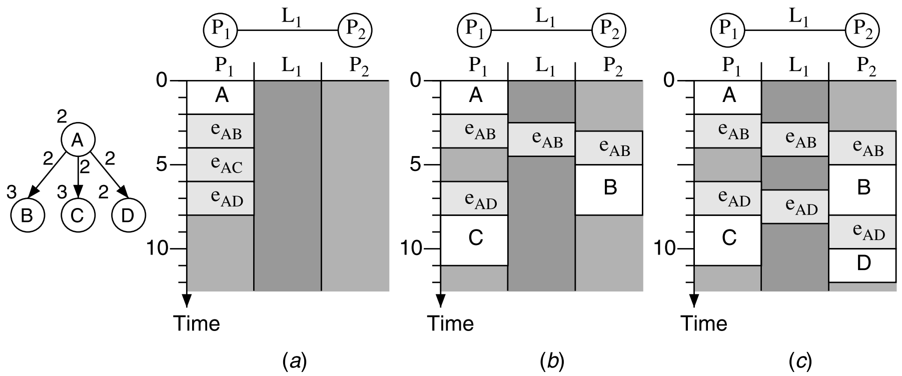


Figure 8.11. Direct scheduling: edges are scheduled on source processor together with their origin node; (a, b) charts illustrate intermediate schedules, while (c) shows the final schedule of the depicted task graph. $o_{s,r}(e, P) = 0.5$, $i_{s,r}(e, L) = 0.75\varsigma(e, L)$.

utilized route. Again, as the route is unknown at the time of the scheduling, the worst case must be assumed and the finish time is calculated as defined in the following.

Definition 8.7 (Provisional Finish Time of Edge on Source Processor) *Let* $G = (\mathbf{V}, \mathbf{E}, w, c)$ *be a task graph and* $M = ((\mathbf{N}, \mathbf{P}, \mathbf{D}, \mathbf{H}, b), \omega, o, i)$ *a parallel system. The provisional finish time of edge* $e_{ij} \in \mathbf{E}$ *on its source processor* $P = proc(n_i)$, $P \in \mathbf{P}$, *is*

$$t_f(e_{ij}, P) = t_s(e_{ij}, P) + o_s(e_{ij}, P) + i_{s,\max}(e_{ij}, P), \tag{8.18}$$

where

$$i_{s,\max}(e_{ij}, P) = \max_{L \in \mathbf{L}: L \text{ leaving } P} \{i_s(e_{ij}, L)\}. \tag{8.19}$$

When the destination node n_j is scheduled, the finish time must be reduced, if applicable, to the correct value.

With the reservation of a time interval for the outgoing edges on the processor, the rest of scheduling can be performed as under the contention model. The clear disadvantage of this approach are the gaps left behind by removed edges, which make a schedule less efficient. In order to relieve this shortcoming, two techniques can help to eliminate or even avoid the gaps.

Gap Elimination—Schedule Compaction In a completed schedule, gaps can be eliminated by repeating the scheduling procedure. The nodes and their edges must be scheduled in the exact same order as in the first run and the processor allocation is taken from the completed schedule. This makes the provisional scheduling of edges needless and the gaps are avoided.

It is important to realize that this technique can reduce the schedule length, but it has no impact on the execution time of the schedule. All nodes and edges are scheduled in exactly the same order, before and after the elimination; hence, the execution time cannot change. The only benefit of rescheduling is a better estimation of the execution time with a more accurate schedule length.

Note the difference from Theorem 5.1 for classic scheduling, which states that the rescheduling of any schedule with a list scheduling algorithm results in a schedule at least as short as the original schedule. There, it suffices to schedule the nodes in their start time order of the original schedule. Under the involvement–contention model, the edge order on the links (Sections 7.4.1 and 7.5.1) as well as on the processors (Section 8.2.1, see also later) is relevant for the schedule length. Generally, it is therefore necessary to repeat the scheduling with the same heuristic employed for the original schedule to ensure the exact same scheduling order of the nodes and edges. Otherwise, the schedule length might increase.

Inserting into Gaps The insertion technique can be applied to use the emerging gaps during the scheduling. This has the advantage that scheduling decisions might be different when a node or an edge can be inserted into a gap and therefore start earlier. Note that use of the insertion technique in a straightforward manner is only possible because the causality Condition 8.1 for involvement scheduling gives the freedom of the node and edge order (Section 8.2.1). Inserting a node or an edge into a gap is very likely to separate edges from their origin or destination nodes, as, for example, in Figure 8.8.

Scheduling Order of Edges As in contention scheduling (Sections 7.4.1 and 7.5.1), the order in which the edges are scheduled is relevant and might lead to different schedule lengths. Here, the scheduling of the edges is divided into the scheduling of the entering edges on the links and the destination processors and the leaving edges provisionally on the source processors. Scheduling the entering edges is very similar to the scheduling under the contention model and the same considerations apply to their order (Sections 7.4.1 and 7.5.1). A natural order of the leaving edges is the scheduling order of their destination nodes, as it follows the same rational as the minimization of the node's start and finish times (Section 5.1). Of course, the ordering of the edges slightly increases the complexity of a scheduling algorithm. It is very important for most algorithms that the scheduling order of the edges is deterministic (Section 7.5.1).

In Section 8.4.1, a list scheduling algorithm will be introduced, employing the technique of reserving time intervals for the leaving edges. With list scheduling, utilization of the end or insertion technique is no problem at all.

8.3.2 Scheduling with Given Processor Allocation

The second approach to involvement scheduling assumes a processor allocation, or mapping, to be given at the time the actual scheduling procedure starts. Since it is known for every node to where its successors will be scheduled, there is no need for the provisional scheduling of the edges on the processors.

Scheduling an Edge The scheduling of an edge e_{ij} can be divided into three parts: scheduling on the source processor P_{src}, on the links of the route R, and on the destination processor P_{dst}. On the source processor, an edge must be scheduled together with its origin node n_i, as the foregoing considerations in the context of the direct scheduling showed. Of course, when n_i is scheduled, only those leaving edges, whose destination nodes will be executed on a processor other than P_{src}, are scheduled on P_{src}. The scheduling on the links and on the destination processor can take place with either the origin node n_i or the destination node n_j. Hence, there are three alternatives for the scheduling of an edge e_{ij}.

1. *Schedule e_{ij} on P_{src} with n_i, on R, P_{dst} with n_j.* This alternative is identical to the approach of direct scheduling, where the edge e_{ij} is scheduled on the links of R and on the destination processor P_{dst}, when its destination node n_j is scheduled. Figure 8.12 illustrates this alternative, where the nodes A and C of the depicted task graph are allocated to processor P_1, and B and D are allocated to P_2, and they are scheduled in the order A, B, C, D. Figure 8.12(a) shows the intermediate schedule after scheduling B and Figure 8.12(b) shows the final schedule.
2. *Schedule e_{ij} on P_{src}, R with n_i, on P_{dst} with n_j.* Here, e_{ij} is scheduled not only on P_{src}, but also on the links, when n_i is scheduled. This way, the edges are scheduled on the links of R in the scheduling order of their origin nodes, while in the first alternative the edges are scheduled on the links in the order of their destination nodes. Figure 8.13 demonstrates this difference, by repeating the example of Figure 8.12, now scheduling e_{ij} on the links with the origin node n_i. So e_{AD} starts earlier than e_{BC}; in Figure 8.12 it is the other way around.
3. *Schedule e_{ij} on P_{src}, R, P_{dst} with n_i.* Edge e_{ij} is completely scheduled together with its origin node n_i. This alternative is likely to produce the best scheduling

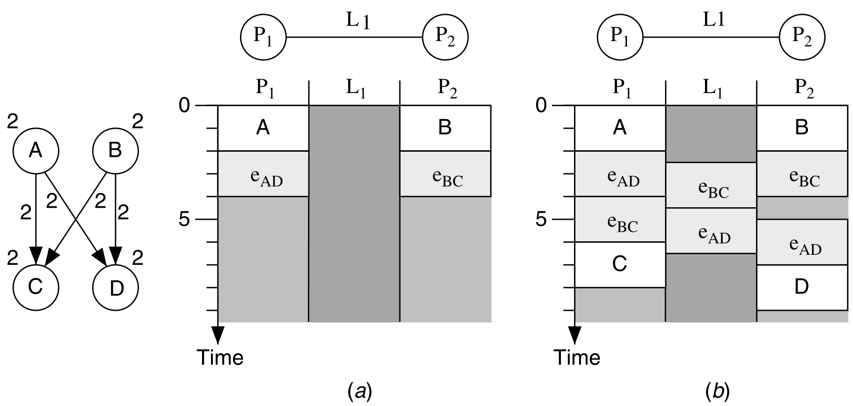


Figure 8.12. Scheduling with given allocation: edge e_{ij} scheduled on P_{src} with n_i, on R, P_{dst} with n_j, as in direct scheduling. Node order is A, B, C, D; $o_{s,r}(e, P) = 0.5$, $i_{s,r}(e, L) = 0.75\varsigma(e, L)$.

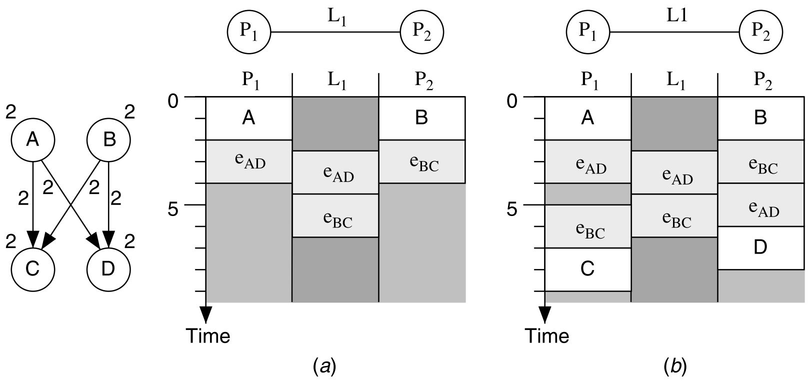


Figure 8.13. Scheduling with given allocation: edge e_{ij} scheduled on $P_{\rm src}$, R with n_i, on $P_{\rm dst}$ with n_j. Node order is A, B, C, D; $o_{s,r}(e,P) = 0.5$, $i_{s,r}(e,L) = 0.75\varsigma(e,L)$.

alignment of the edge on the source processor, the links and the destination processor, as the scheduling is done in a single step. Note that the scheduling of an edge on the processors and links does not require alignment; therefore, it is called the nonaligned approach, but it is obvious that large differences between the scheduling times on the links and the processors are not realistic. Hence, in terms of the scheduling of the edge, this alternative is probably the most accurate. Unfortunately, it has a similar disadvantage as the direct scheduling of the entering edges: the scheduling of the edges on their destination processors might prevent the efficient scheduling of the nodes. Figure 8.14 illustrates the previous example, now with the complete scheduling of the edges on all resources with their origin nodes. As the edge e_{AD} is scheduled before

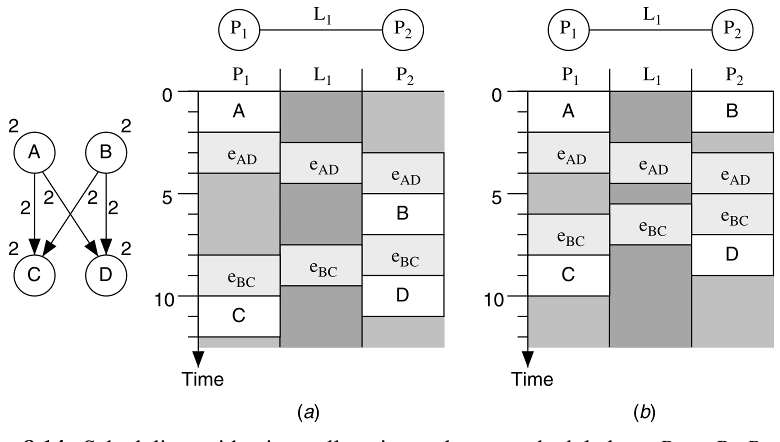


Figure 8.14. Scheduling with given allocation: edge e_{ij} scheduled on $P_{\rm src}$, R, $P_{\rm dst}$ with n_i; (*a*) without insertion; (*b*) with insertion. Node order is A, B, C, D; $o_{s,r}(e,P) = 0.5$, $i_{s,r}(e,L) = 0.75\varsigma(e,L)$.

node B, B can only start after e_{AD} on P_2 (Figure 8.14(a)), unless the insertion technique is employed (Figure 8.14(b)). The early scheduling of the edges on their destination processors rapidly increases their finish times, leaving large idle gaps. So the conjoint scheduling of an edge on all resources is only sensible with the insertion technique.

There is no clear advantage of the first over the second alternative or vice versa. Which one is better (i.e., more realistic) depends on the way the communication is realized in the target parallel system, whether it is initiated by the receiving (first alternative) or by the sending side (second alternative).

In all three alternatives, the scheduling order of the edges is relevant and the same considerations apply as for direct scheduling (Section 8.3.1).

Section 8.4.2 looks at two-phase scheduling heuristics, where the first phase establishes the processor allocation and the second phase constructs the actual schedule with a simple list scheduling heuristic, based on the foregoing considerations.

8.4 HEURISTICS

This section formulates heuristics for scheduling under the involvement–contention model. It starts with the venerable list scheduling and then discusses two-phase heuristics, where the scheduling in the second phase disposes of a processor allocation determined in the first phase.

8.4.1 List Scheduling

As under the contention model (Section 7.5.1), it suffices to reformulate the determination of the DRT and the scheduling procedure of a node in order to adapt list scheduling for the involvement–contention model. The two procedures are modified using the direct scheduling approach discussed in Section 8.3.1.

Scheduling of Node Algorithm 23 shows the procedure for the scheduling of a node in accordance with the direct scheduling approach. As under the contention model, the actual scheduling of a node n is preceded by the scheduling of its entering edges on the links and the destination processor P, including the possibly necessary correction of the edges' finish times on the source processors (lines 6–13). If the origin node of an edge is scheduled on P, the provisionally scheduled edge is removed from P first (lines 1–5), since the communication is then local. It is important that those edges are removed before the scheduling of the entering edges coming from remote processors, as the latter might already use the freed time intervals. After the scheduling of n on P (line 14), each of its leaving edges is scheduled on P in order to reserve the corresponding time periods (lines 15–17). As discussed in Section 8.3.1, the finish time of each edge is calculated for the worst case, that is, the longest possible involvement (Definition 8.7).

Algorithm 23 Scheduling of Node n_j on Processor P in Involvement–Contention Model

```
Require: $n_j$ is a free node
 1: for each $n_i \in \mathbf{pred}(n_j)$ do
 2:    if $proc(n_i) = P$ then
 3:       remove $e_{ij}$ from $P$
 4:    end if
 5: end for
 6: for each $n_i \in \mathbf{pred}(n_j)$ in a deterministic order do
 7:    if $proc(n_i) \neq P$ then
 8:       determine route $R = \langle L_1, L_2, \ldots, L_l \rangle$ from $proc(n_i)$ to $P$
 9:       correct $t_f(e_{ij}, proc(n_i))$
10:       schedule $e_{ij}$ on $R$ (Definition 7.9)
11:       schedule $e_{ij}$ on $P$ (Definition 8.3 and 8.4)
12:    end if
13: end for
14: schedule $n_j$ on $P$
15: for each $n_k \in \mathbf{succ}(n_j)$ in a deterministic order do    ▷ reserve space for leaving edges
16:    schedule $e_{jk}$ on $P$ with worst case finish time (Definition 8.7)
17: end for
```

Determining DRT The calculation of the DRT, formulated by Algorithm 24, is performed in the same fashion as the scheduling of a node. It is noteworthy that provisionally scheduled edges, whose origin nodes are scheduled on the processor P for which the DRT is calculated (i.e., communication is local), are first removed (lines 1–5) and then replaced at the end (line 19). Also, the possibly necessary correction of the incoming edges' finish times on the source processors is calculated, but the schedule is not changed (line 11).

End and Insertion Technique As before, both techniques, end and insertion, can be employed with list scheduling. Under the involvement–contention model, the insertion technique is more indicated, since the removing of provisionally scheduled edges leaves gaps, which should be filled by other nodes or edges. However, inserting a node or an edge into a gap is very likely to separate edges from their origin or destination nodes. If this is not supported by the code generation or is simply not desired, the end technique should be employed, where the order of nodes and edges is strict.

The scheduling order of the edges is required to be deterministic (see the corresponding `for` loops in Algorithms 23 and 24), just as under the contention model (Section 7.5.1).

Algorithm 24 Determine $t_{\mathrm{dr}}(n_j, P)$ in Involvement–Contention Model

```
Require: $n_j$ is a free node
 1: for each $n_i \in \mathbf{pred}(n_j)$ do
 2:    if $proc(n_i) = P$ then
 3:       remove $e_{ij}$ from $P$
 4:    end if
 5: end for
 6: $t_{\mathrm{dr}} \leftarrow 0$
 7: for each $n_i \in \mathbf{pred}(n_j)$ in a deterministic order do
 8:    $t_f(e_{ij}) \leftarrow t_f(n_i)$
 9:    if $proc(n_i) \neq P$ then
10:       determine route $R = \langle L_1, L_2, \ldots, L_l \rangle$ from $proc(n_i)$ to $P$
11:       calculate $t_f(e_{ij}, proc(n_i))$ for route $R$
12:       schedule $e_{ij}$ on $R$ (Definition 7.9)
13:       schedule $e_{ij}$ on $P$ (Definitions 8.3 and 8.4)
14:       $t_f(e_{ij}) \leftarrow t_f(e_{ij}, P)$
15:    end if
16:    $t_{\mathrm{dr}} \leftarrow \max\{t_{\mathrm{dr}}, t_f(e_{ij})\}$
17: end for
18: remove edges $\{e_{ij} \in \mathbf{E} : n_i \in \mathbf{pred}(n_j) \wedge proc(n_i) \neq P\}$ from links and $P$
19: replace provisionally scheduled edges $\{e_{ij} \in \mathbf{E} : n_i \in \mathbf{pred}(n_j) \wedge proc(n_i) \neq P\}$
    on $P$
20: return $t_{\mathrm{dr}}$
```

An Example As in previous chapters, the list scheduling for the involvement–contention model is illustrated with an example. Since scheduling the sample task graph of Figure 3.15 under the involvement–contention model is too extensive to be instructive, the smaller task graph displayed in Figure 8.15 is utilized for the example.

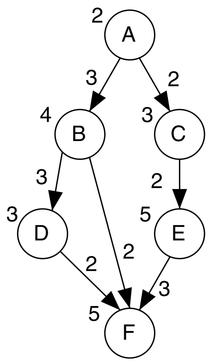


Figure 8.15. Task graph for list scheduling example in Figure 8.16.

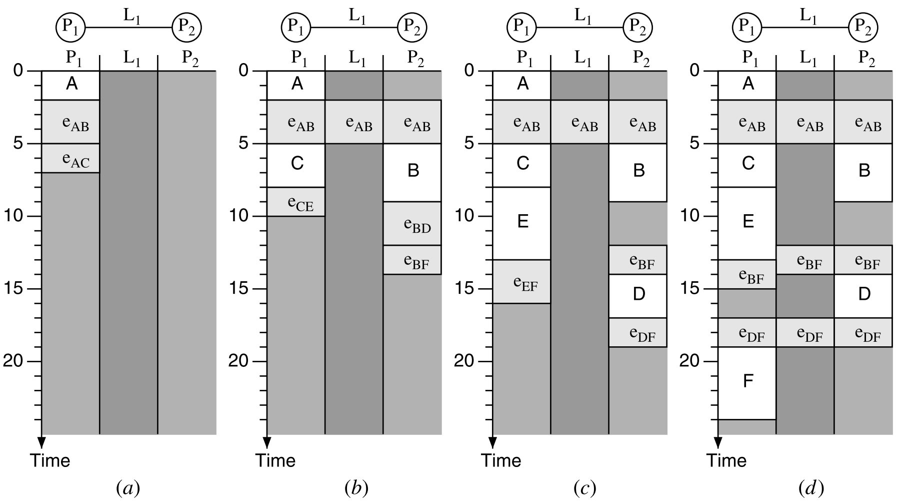


Figure 8.16. Example of list scheduling under involvement–contention model; (*a*)–(*c*) intermediate schedules, (*d*) final schedule. Task graph of Figure 8.15 is scheduled in the order A, B, C, E, D, F; $o_{s,r}(e, P) = 0$, $i_s(e, L_1) = \varsigma(e, L_1)$, $i_r(e, L_l) = \varsigma(e, L_l)$.

Its nodes are scheduled in the order A, B, C, E, D, F by a list scheduling heuristic with start time minimization and the end technique on two identical processors connected by a half duplex link. For simplicity, the overhead is assumed to be zero, $o_{s,r}(e, P) = 0$, and both processors, the source and the destination, are 100% involved in communication, $i_s(e, L_1) = \varsigma(e, L_1)$, $i_r(e, L_l) = \varsigma(e, L_l)$.

Figure 8.16(*a*) displays the initial schedule after assigning A to P_1, with its leaving edges also scheduled on P_1. Next, node B is scheduled earliest on P_2 together with the corresponding entering edge e_{AB} on the link and P_2. The scheduling of C on P_1 is preceded by the removal of its entering edge e_{AC}, because the communication is local. In consequence, C can benefit from the freed space and starts at time unit 5 (Figure 8.16(*b*)). The scheduling of E on P_1 is straightforward including the removal of e_{CE}. After the scheduling of E on P_1, D starts earliest on P_2, however, its scheduling leaves a gap of 3 time units (Figure 8.16(*c*)). If the insertion technique were used, D could have been inserted into this gap, though this would separate D and its leaving edge e_{DF}. The final schedule is shown in Figure 8.16(*d*), after node F has been scheduled on P_1, preceded by the scheduling of its entering edges on L_1 and on P_1. Note that the schedule length (24) is above the sequential time (22), yet after the elimination of the gap between B and e_{BF} (see Section 8.3.1) it reduces to 22 time units. Nevertheless, the high involvement of the processors impedes an efficient parallelization.

Priority Schemes Due to the fact that the involvement–contention model is an extension of the contention model, it appears sensible to utilize the same priority schemes as under the contention model. In particular, it is proposed to employ the *bottom-level* order for list scheduling under the involvement–contention model.

Complexity In comparison to contention aware list scheduling, the time complexity under the involvement–contention model does not increase. The additional scheduling of the edge on the processors increases the effort to implement the algorithm, but it does not modify its time complexity. For example, the complexity of the second part of simple list scheduling (Algorithm 9) with start/finish time minimization and the end technique is $O(\mathbf{P}(\mathbf{V} + \mathbf{E}O(\mathit{routing})))$ (see Section 7.5.1).

Also, the complexity does not increase for the insertion technique (Section 6.1), even though the individual scheduling of a node and an edge on a processor is more complex. In the involvement–contention model, there are more objects on the processors, namely, nodes *and* edges, which must be searched to find an idle time interval. On all processors there are at most $O(\mathbf{V} + \mathbf{E})$ nodes and edges; hence, the total complexity for the scheduling of the nodes increases from $O(\mathbf{V}^2)$, under the contention model, to $O(\mathbf{V}(\mathbf{V} + \mathbf{E}))$. The scheduling of the edges on the processors is $O(\mathbf{E}(\mathbf{V} + \mathbf{E}))$, compared to $O(\mathbf{PE})$ with the end technique, and on the links it remains $O(\mathbf{PE}^2 O(\mathit{routing}))$. Hence, the total complexity of the second part of simple list scheduling with start/finish time minimization and the insertion technique is $O(\mathbf{V}^2 + \mathbf{VE} + \mathbf{PE}^2 O(\mathit{routing}))$, that is, $O(\mathbf{V}^2 + \mathbf{PE}^2 O(\mathit{routing}))$, as under the contention model (Section 7.5.1).

8.4.2 Two-Phase Heuristics

The generic form of a two-phase scheduling algorithm was already outlined in Algorithm 13. The first phase establishes the processor allocation and the second phase constructs the actual schedule with a simple list scheduling heuristic, as described in Section 5.2. The next paragraphs discuss how such an algorithm is applied under the involvement–contention model.

Phase 1—Processor Allocation The processor allocation can originate from any heuristic or can be extracted from a given schedule. For example, a schedule produced under the classic or contention model might serve as the input (see also Section 5.2).

Using the first two steps of a clustering based heuristic, that is, the clustering itself and the cluster-to-processor mapping (Algorithm 14, Section 5.3), is very promising, because clustering tries to minimize communication costs, which is even more important under the involvement–contention model. The clustering itself can be performed under the classic or, as described in Section 7.5.3, under the contention model. Even the involvement–contention model can be used with a straightforward extension of the considerations made in Section 7.5.3.

Genetic Algorithm In Sinnen [172] and Sinnen et al. [180], a genetic algorithm, called GICS (genetic involvement–contention scheduling), is proposed for the determination of the processor allocation. The simple idea is that the genetic algorithm searches for an efficient processor allocation, while the actual scheduling is performed with a list scheduling heuristic as discussed later for phase 2.

GICS follows the outline of Algorithm 20, Section 6.5. Naturally, the chromosome encodes the processor allocation, which is an indirect representation (Section 6.5.2). Hence, the construction of the schedule corresponding to each chromosome, which is necessary for the evaluation of its fitness, requires the application of a heuristic. As already mentioned, this is performed by a list scheduling heuristic under the involvement–contention model (see later discussion). Strictly speaking, this means that a two-phase heuristic is applied multiple times in GICS—once for each new chromosome.

To reduce the running time of the evaluation, the node order is determined only once at the beginning of the algorithm, namely, according to their bottom levels. Like most GAs, GICS starts with a random initial population, enhanced with a chromosome representing a sequential schedule (all nodes are allocated to a single processor). The pool is completed by one allocation extracted from a schedule produced with a list scheduling heuristic (Section 8.4.1). Most other components are fairly standard. In fact, the algorithm can even be employed for scheduling under the classic or the contention model, using a modified evaluation (i.e., scheduling heuristic).

Phase 2—List Scheduling From Section 5.2 it is known that the second phase can be performed with a simple list scheduling algorithm. The processor choice of list scheduling is simply a lookup from the given mapping $\mathcal{A}$ established in the first phase. Even though list scheduling under the involvement–contention model was already studied in Section 8.4.1, it must be revisited as there are three alternatives for the scheduling of the edges when the processor allocation is already given (Section 8.3.2).

Scheduling of the Edges The first alternative corresponds to direct scheduling, and the procedures for the scheduling of a node and the calculation of the DRT are presented in Algorithms 23 and 24. The scheduling of an edge on the source processor must be modified (line 16 in Algorithm 23), since it is no longer necessary to assume the worst case. Lines 1–5 (removing local edges) and line 9 (correcting the finish time) can be dropped completely, since the provisional scheduling of the edges is not necessary with the given processor allocation. The other two alternatives are only distinguished from this procedure through the place where the edges are scheduled on the links and the destination processor. So, for the second alternative, where the edge is scheduled on the links as a leaving edge, lines 8 and 10 of Algorithm 23 move to the `for` loop on lines 15–17. In the third alternative, the edge is also scheduled on the destination processor within this loop, and the `for` loop for the entering edges (lines 6–13) is completely dropped.

Of course, in the determination of the DRT the entering edges do not need to be tentatively scheduled, as the processor allocation is already given.

Complexity The complexity of list scheduling with a given processor allocation decreases slightly in comparison to list scheduling with start/finish time minimization. It is not necessary to tentatively schedule a node and its edges on every processor. Thus, with the end technique the complexity is $O(\mathbf{V} + \mathbf{E}O(routing))$ (instead of $O(\mathbf{P}(\mathbf{V} + \mathbf{E}O(routing))))$ and with the insertion technique it is $O(\mathbf{V}^2 + \mathbf{E}^2 O(routing))$ (instead of $O(\mathbf{V}^2 + \mathbf{PE}^2 O(routing)))$. Remember, the third alternative of edge scheduling, where the leaving edges of a node are scheduled on the processors and links, is only meaningful with the insertion technique (Section 8.3.2).

8.4.3 Experimental Results

As for the contention model (Section 7.5.4), the question arises as to how much the accuracy of scheduling and the execution times of the produced schedules benefit from the involvement–contention model. As argued in Section 7.5.4, only an experimental evaluation on real parallel systems can help answer this question.

Such an experimental evaluation is performed in Sinnen [172] and Sinnen et al. [180]. The employed methodology is based on the one described in Section 7.5.4: a large set of graphs is generated and scheduled by algorithms under the different models to several target systems. Because the focus is on the comparison of the different scheduling models, the same list scheduling algorithm is employed under each model. From the produced schedules, code is generated—using C and MPI—and executed on the real parallel systems.

Modeling of the target machines as topology graphs is relatively simple, as discussed in Chapter 7. For the involvement–contention model it is additionally necessary to specify for each target system the overhead and involvement of the processors in communication. Due to the lack of a deep insight into the target systems' communication mechanisms and their MPI implementations, 100% involvement is assumed, that is, the source and destination processors are involved during the entire communication time on the first and last link, respectively:

$$i_s(e, L_1) = \varsigma(e, L_1) \quad \text{and} \quad i_r(e, L_l) = \varsigma(e, L_l). \tag{8.20}$$

The overhead is intuitively set to an experimentally measured setup time:

$$o_s(e, P) = o_r(e, P) = setup\_time. \tag{8.21}$$

While it is clear that this definition of the overhead and the involvement is probably not an accurate description of the target systems' communication behavior, it is very simple. The idea is to demonstrate that accuracy and efficiency of scheduling can be improved even with a rough but simple estimate of the overhead and involvement functions.

Results The experiments demonstrated that the involvement–contention model achieves profoundly more accurate schedules and significantly shorter execution times.

Thus, considering processor involvement in communication further improves the already improved accuracy under the contention model. The length of a schedule is now in a region where it can be seriously considered an estimate of the real execution time. Under the classic model and for some topology graphs under the contention model, the schedule lengths are only small fractions of the real execution times, in particular, for medium ($CCR = 1$) and high communication ($CCR = 10$), which can hardly be considered execution time estimations. For example, on the Cray T3E-900 (Section 7.5.4), the estimation error under the involvement–contention model was on average below 20%, while the estimation error under the classic model goes up to 1850%.

Still, the scheduling accuracy under the involvement–contention model is not perfect, especially for low communication ($CCR = 0.1$). A possible explanation might be the blocking communication mechanisms used in MPI implementations (White and Bova [199]), which does not match the assumption of nonblocking communication made in the involvement–contention model. Furthermore, the employed overhead and involvement functions are very rough estimates; a better approximation of these functions has the potential to further increase the accuracy. In any case, it is in the nature of any model that there is a difference between prediction and reality.

The profoundly improved accuracy under the involvement–contention model allows more than just the reduction of execution times: it also allows one to evaluate algorithms and their schedules without execution of the schedules. Hence, new algorithms can be developed and evaluated without the large efforts connected with an evaluation on real parallel systems.

In the experiments conducted, the involvement–contention model also clearly demonstrated its ability to produce schedules with significantly reduced execution times. The benefit of the high accuracy is apparent in the significantly improved execution times with speedup improvements of up to 82%.

Despite the very good results, the efficiency improvement lags behind the accuracy improvement. A possible explanation lies in the heuristic employed in the experiments. List scheduling is a greedy algorithm, which tries to reduce the finish time of each node to be scheduled. Thus, it does not consider the leaving communications of a node, which may impede an early start of following nodes. The high importance of communication under the involvement–contention model seems to demand research of more sophisticated algorithms in order to exploit the full potential of this new model.

8.5 CONCLUDING REMARKS

This chapter investigated processor involvement in communication and its integration into task scheduling. The motivation originated from the unsatisfying accuracy results obtained under the classic and the contention model as referenced in Section 7.5.4, which indicated a shortcoming of these scheduling models.

Thus, the chapter began by thoroughly analyzing the various types of processor participation in communication and their characteristics. Based on this analysis, another scheduling model was introduced that abandons the idealizing assumption of a dedicated communication subsystem and instead integrates the modeling of all types of processor involvement in communication. This model, referred to as the involvement–contention model, is a generalization of the contention model analyzed in Chapter 7. It thereby adopts all properties of the contention model, for example, the contention awareness and the ability to reflect heterogeneous systems.

Scheduling under the new model requires some modifications of the scheduling techniques, since edges are now also scheduled on the processors to represent the processors' involvement in communication. This chapter discussed the general issue and its possible solutions. The necessary alterations to list scheduling were investigated in order to employ it, in all its forms, under the involvement–contention model. Another strategy for scheduling under the new model is based on the initial determination of the processor allocation.

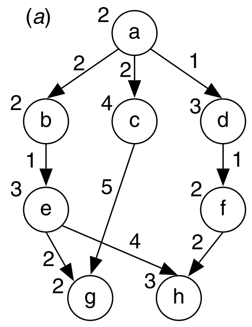
It will depend on the target system and its communication mechanisms whether the benefits of the involvement–contention model outweigh its higher conceptual complexity and additional effort. The experimental results referenced in Section 8.4.3 indicate that for various parallel systems the accuracy for schedules produced under the involvement–contention model fundamentally improved. This allows more than just the reduction of execution times: it also allows one to evaluate algorithms and their schedules without execution of the schedules.

8.6 EXERCISES

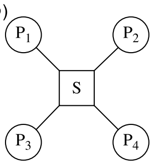
8.1 Visit the online *Overview of Recent Supercomputers* by van der Steen and Dongarra [193] on the site of the TOP500 Supercomputer Sites [186]. Find examples for systems that use different types of processor involvement in communication. (*Hint*: The information provided in Ref. [186] might not be sufficient and you will need to aquire more information from the manufacturer's site.)

8.2 Describe in your own words what the essential challenges are of task scheduling when processor involvement is considered.

8.3 Schedule the following task graph



under the involvement–contention model on four processors connected via a central switch, employing half duplex links:

(b)


The nodes are preallocated to the processors as specified in the following table.

Node	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
Processor	1	3	2	4	1	4	1	4

Schedule the nodes in alphabetic order using the end technique. For the scheduling of the edges on the processors and links use the first alternative (Section 8.3.2), that is, edge e_{ij} is scheduled on P_{src} with n_i, on R, P_{dst} with n_j. The overhead is identical on the sending and receiving sides, namely, $o_{s,r}(e, P) = 0.5$, and the processors on both sides are involved 50% in communication, that is, $i_{s,r}(e, L) = 0.50\varsigma(e, L)$.

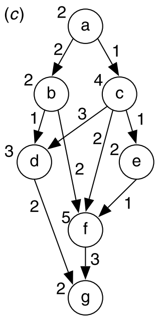
8.4 Perform Exercise 8.3 with the following modifications:

(a) The processor involvement is one-sided (sending side), with 75% involvement, that is, $i_s(e, L) = 0.75\varsigma(e, L)$, $i_r(e, L) = 0$, and overhead $o_s(e, P) = 0.5$, $o_r(e, P) = 0$.

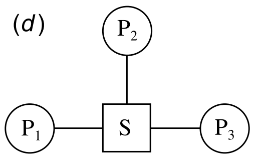
(b) The processor involvement is one-sided (receiving side), with 75% involvement, that is, $i_s(e, L) = 0$, $i_r(e, L) = 0.75\varsigma(e, L)$, and $o_s(e, P) = 0$, $o_r(e, P) = 0.5$.

8.5 Perform Exercise 8.3 with the third alternative (Section 8.3.2) for the scheduling of the edges on the processors and links, that is, edge e_{ij} is scheduled on P_{src}, R, P_{dst} with n_i. Remember that this implies that the insertion technique must be used for the scheduling of the nodes and edges.

8.6 Use list scheduling with start time minimization and the end technique to schedule the following task graph

(c)


under the involvement–contention model on three processors connected via a central switch, employing half duplex links:



The overhead is negligible, hence $o_{s,r}(e,P) = 0$. The processors on both sides are 100% involved in communication, that is, $i_{s,r}(e,L) = \varsigma(e,L)$.

8.7 Perform Exercise 8.6 with the following modifications:

(a) The processor involvement is one-sided (sending side), with 50% involvement, that is, $i_s(e,L) = 0.50\varsigma(e,L)$, $i_r(e,L) = 0$.

(b) The processor involvement is one-sided (receiving side), with 50% involvement, that is, $i_s(e,L) = 0$, $i_r(e,L) = 0.50\varsigma(e,L)$.

(c) Why is one-sided involvement on the receiving side different from other involvement types in what relates to the scheduling approach?

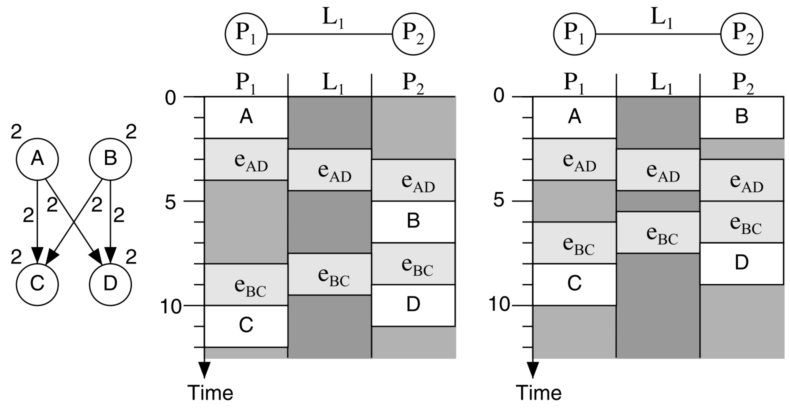
8.8 In Section 8.2.3 the length of a path in a task graph was analyzed under the light of the overhead of interprocessor communication. Determine the bottom levels of the nodes of Exercise 8.6's task graph, assuming an overhead of $o_{s,r}(e,P) = 2$.

Is the decreasing bottom-level order of the nodes different without the consideration of the overhead?

8.9 To generate code from a given task graph and schedule assume that a node A is translated into `function(A)`. Remote communication is performed with the two nonblocking directives `send(to_node, p_dst)` and `receive(from_nodes, p_src)`. The parameters `from_node` and `to_node` stand for the origin and destination nodes, respectively, while `p_src` and `p_dst` stand for the source and destination processors, respectively. Local communication is performed through main memory and does not require explicit directives.

Using these simple statements, generate code for the following task graphs and schedules:

(a)



(b) The schedule obtained in Exercise 8.3.

How does this differ from code generation for schedules produced under the classic mode?

8.10 Research: The experimental results referenced in Section 8.4.3 suggest that list scheduling might not be the most adequate heuristic for scheduling under the involvement–contention model. As discussed in Section 8.4.2, an alternative is to use a two-phase scheduling heuristic, where the processor allocation is determined in the first phase. It was argued that using clustering in the first phase has the potential to produce good results, as it tries to reduce the cost of communication.

Implement a two-phase scheduling algorithm under the involvement–contention model, where clustering is used in the first phase to obtain a good processor allocation. Experimentally compare this algorithm to list scheduling.