Optimized Buffer Management for Sequence Comparison in Molecular Biology Databases

Marco Antonio Casanova

Pontifícia Universidade Católica
Rua Marquês de S. Vicente, 225
Rio de Janeiro
Brazil
casanova@inf.puc-rio.br

Melissa Lemos

Pontifícia Universidade Católica
Rua Marquês de S. Vicente, 225
Rio de Janeiro
Brazil
melissa@inf.puc-rio.br

PUC-RioInf.MCC01/01 February, 2001.

Abstract

Comparing sequences - nucleotide, aminoacid - is one of the basic operations every Molecular Biology database must support. There are two well known sequence comparison algorithms, FAST and BLAST, the latter being more popular (and many variations thereof).
We propose in this paper a buffer management strategy to optimize the simultaneous execution of a set *P* of BLAST processes. The essence of the strategy is to partition *P* into subsets P_1 , ..., P_n and to allocate a separate set B_i of buffers to each partition P_i. All sequences in the database will then be cycled through the buffers in B_i so that all BLAST processes in P_i will perform their sequence comparison synchronously.
We first show that finding the partition that minimizes overall process delay is NP-complete. We then describe a heuristics that is computationally feasible and yet obtains a near optimal solution.

Keywords: Molecular Biology databases, buffer management, sequence comparison, BLAST.

Resumo

Todos os bancos de dados de biologia molecular precisam ter a operação básica de comparação de sequências de nucleotídeos e aminoácidos. Os algoritmos de comparação FAST e BLAST são os mais conhecidos, sendo o último o mais popular.
Neste trabalho propomos uma estratégia de gerenciamento de buffers para otimizar a execução de um conjunto P de processos BLAST. A essência desta estratégia está em particionar P em subconjuntos P_1,...,P_n e alocar conjuntos separados de buffers B_i para cada partição P_i. Todas as sequências do banco de dados serão circuladas nos buffers em B_i e todos os processos em P_i farão suas comparações sincronamente.
Primeiro mostramos que descobrir a partição que minimiza o atraso de todos os processos é NP-completo e, em seguida, descrevemos uma heurística que é computacionalmente aceitável e que se aproxima de uma solução ótima.

Palavras-chave: Bancos de dados de Biologia Molecular, gerenciamento de buffers, comparação de sequências, BLAST.

1. Introduction

Molecular Biology databases typically store nucleotide or aminoacid sequences, and annotations thereof, and offer tools to search and analyze the data. The growth in size and complexity of such databases just reflects the rapid progress of Molecular Biology [Doo90]. From the point of view of database technology, they demand a closer look due to the peculiarities of some of basic operations, such as nucleotide or aminoacid sequence comparison [MS97].

There are two well known comparison algorithms, FAST and BLAST, the latter being more popular (and many variations thereof). Very briefly, given a sequence *s*, BLAST performs an exhaustive search of the database trying to locate the sequence that best matches *s*, according to a well defined criteria. The cost of the search is naturally dependent of the size of the database, as well as on the size of the input sequence *s*.

The crucial remark is that BLAST must perform an exhaustive search of the database to find the best match.

We then propose in this paper a buffer management strategy to optimize the simultaneous execution of a set P of BLAST processes. The essence of the strategy is to partition P into subsets $P_1, \dots, P_n$ and to allocate a separate set B_i of buffers to each partition P_i. All sequences in the database will then be cycled through the buffers in B_i so that all BLAST processes in P_i will perform their comparison synchronously. By allocating several processes to the same set of buffers, we reduce resource consumption, but we introduce delays, since some processes may run faster than others.

We first show that finding the partition that minimizes overall processes delay is NP-complete. We then describe a heuristics that is computationally feasible and yet achieves good performance. We also outline simulation results that help understand our buffer management strategy.

Molecular Biology databases adopt a variety of approaches to store their data. Several Molecular Biology databases, such as GenBank [BB99], Genome DataBase (GDB) [CF93] and PIR [BG99], are based on relational technology. Some databases that used to store and distribute their data as text files, such as Protein Data Bank (PDB), Swiss-Prott, and OMIM, moved to DBMSs. Others, such as GDB, Flybase, Genome Sequence DataBase (GSDB), use commercial or specially designed DBMSs, such as ACeDB [DT92]. Yet others, such as LabBase and MapBase are implemented on top of OO-DBMSs, such as ObjectStore [GR94, MR95].

The FAST algorithm is described in [Pea90, Pea91] and the BLAST algorithm in [AGM+90, AMS+97, WU00]. The buffer management strategy proposed in this paper is somewhat similar to that described in [MNÖ+96] for serving a video stream to a set of processes. A detailed account of the results in Section 3 can be found in [Le00]. In special, [Le00] presents a comprehensive set of simulations results that help understand the buffer management strategy proposed.

This paper is organized as follows. Section 2 summarizes just the essential concepts of sequence comparison. Section 3 describes the buffer management strategy. Finally, Section 4 contains the conclusions.

2. Nucleotide / Aminoacid Sequence Comparison

Nucleotide (DNA or RNA) and aminoacid sequences are just finite sequences over the alphabets [Le00]:

$\Sigma_D = \{A,C,G,T\}$	(DNA)
$\Sigma_R = \{A,C,G,U\}$	(RNA)
$\Sigma_A = \{A, C, D, E, \ldots, W, Y\}$	(Aminoacid)

One of the first peculiarities one encounters when studying Molecular Biology databases is that (nucleotide or aminoacid) sequence comparison is not exact pattern matching.

For example, consider the following nucleotide sequences:

s_1 = GACGGATT
s_2 = GATCGGAAT

Figure 1 shows the best way to match s_1 and s_2, which is:

1. to introduce a "hole", represented by the symbol "-", in the third position of s_1 , thereby increasing the length of s_1 to 9 symbols; and
2. to allow a mismatch in the eighth position, where s_1 now has a "T" and s_2 has an "A".

```
s1 : G A - C G G A T T
s2 : G A T C G G A A T
     1 2 3 4 5 6 7 8 9
```

Figure 1. Comparing two nucleotide sequences

In the context of Molecular Biology databases, sequence comparison is indeed an optimization process that allows mismatches and "holes", as in the example of Figure 1.

Since it will be irrelevant to the rest of the discussion which type of sequence we are talking about, we will refer to them simply as sequences. Likewise, it will not be necessary to distinguish between Σ_D, Σ_R or Σ_A .

Therefore, let

Σ	denote a fixed alphabet
"$-$"	denote a symbol not in Σ, called a *hole*
$\Sigma^+ = \Sigma \cup \{-\}$	
λ	denote the empty string.

We will use the term *sequence* to mean any finite sequence over Σ. A *Molecular Biology database* is a set of sequences over Σ.

Given two sequences s_1 and s_2, an *alignment* of s_1 and s_2 is a pair of sequences t_1 and t_2 created by inserting holes at arbitrary points of s_1 and s_2 such that t_1 and t_2 have the same length and t_1 and t_2 do not have holes at the same position. We also recursively define the function $| t_1 , t_2 |$ as follows:

$$
\begin{array}{ll}
| t_1 , t_2 | = 0 & \text{if } t_1 = t_2 = \lambda \\
| t_1 , t_2 | = 1 + | tail(t_1), tail(t_2) | & \text{if } head(t_1) = head(t_2) \\
| t_1 , t_2 | = -1 + | tail(t_1), tail(t_2) | & \text{if } head(t_1) \neq head(t_2) \neq \text{"}-\text{"} \\
| t_1 , t_2 | = -2 + | tail(t_1), tail(t_2) | & \text{if } head(t_1) = \text{"}-\text{"} \text{ or } head(t_2) = \text{"}-\text{"}
\end{array}
$$

We are now in a position to define our problem.

Sequence Comparison Problem:

Given a sequence *s* and a Molecular Biology database *S*, find *s'* in *S* and an alignment *t* and *t'* of *s* and *s'* such that, for any *s"* in *S* and any alignment *u* and *u"* of *s* and *s"*, we have $| t, t' | \geq | u, u" |$.

One of the most popular heuristics to address the sequence comparison problem is the BLAST algorithm (and variations thereof) [AGM+90, AMS+97, WU00]. Very briefly, given a sequence *s*, BLAST performs an exhaustive search of a Molecular Biology database *S* trying to locate the sequence that best matches *s*, in the above sense. The cost of the search is dependent on the size of *S*, as well as on the size of the input sequence *s* [Gis00]. The crucial remarks are:

(1) BLAST performs an exhaustive search of *S* to try to find the best match. Furthermore, the search order is irrelevant.
(2) Because the comparison process allows holes and mismatches, there is no obvious (if one at all) access method that helps speed up BLAST.
(3) As a rule, there will be a large number of BLAST processes simultaneously searching *S*.

In view of these remarks, we investigate in the next section a buffer management schema that optimizes the simultaneous execution of a set of BLAST processes.

3. Optimized Buffer Management for BLAST Processes

We describe in this section a buffer management strategy to optimize the processing of a set of BLAST processes. We first introduce the strategy informally, then we prove that the general problem is NP-complete and, finally, we describe a practical heuristics to schedule sets of BLAST processes. We also outline simulation results that help understand the buffer management strategy.

3.1 Alternative buffer management strategies

We will use the consumer/producer paradigm [Tan92] to address buffer management. Let $p_1, p_2, \ldots, p_n$ be a set of BLAST processes - the consumer processes. Assume that the processes simultaneously access a Molecular Biology database *S*, start at different times and execute on a single processor *M* - the producer process. Assume also that it is not feasible to retrieve all sequences stored in *S* into main memory.

To handle this scenario, we may first adopt a strategy, which we call the *private-ring strategy*, defined as follows:

1. Allocate a set of buffers $\boldsymbol{b}_i$ to each p_i, organized as a ring (circular list).
2. Manage the buffer rings as follows:
   a. Divide the producer process time into cycles of fixed length.
   b. During a cycle:
      i. Each consumer process p_i will consume the non-empty buffers in $\boldsymbol{b}_i$.
      ii. The producer process M will cycle through the buffer rings, loading data from S into the empty buffers.

This strategy is similar to the scheduling algorithm for continuous data proposed in [MNÖ+96].

The size of each $\boldsymbol{b}_i$ and the length of the cycle must be chosen so that buffer underflow never occurs. At this point, we offer the following brief intuitive explanation, leaving to Section 3.2 a more detailed analysis. We first observe that the producer I/O rate must be no less than the aggregate processing rate of the BLAST processes and that the total amount of buffer space must be no less than the aggregate buffer space required by the rings. Moreover, the size of each $\boldsymbol{b}_i$ must be chosen so that, during cycle T_k, while process p_i consumes half of the buffers in $\boldsymbol{b}_i$, roughly speaking, processor M must fill in the other half of the buffers (which p_i will consume during the next cycle T_{k+1}).

This strategy is not very interesting, though, because it may easily exhaust buffer space or processor I/O capacity.

An alternative strategy, which we call *public-ring*, would be to:

1. Allocate a single (public) buffer ring $\boldsymbol{b}$ to all processes.
2. Regulate buffer consumption by the slowest process.
3. Continuously bring all sequences in S into the buffers in $\boldsymbol{b}$ (sequences in S are considered to be ordered).
4. Signal to a process when it completes reading all sequences in S (with the help of auxiliary structures).

Figure 2 illustrates this situation.

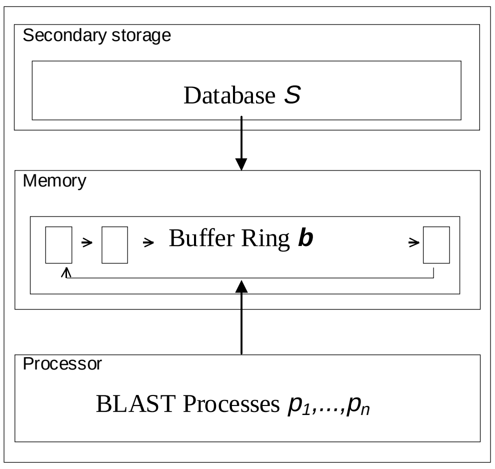
Secondary storage
Database S
Memory
Buffer Ring $\boldsymbol{b}$
Processor
BLAST Processes $p_1,\ldots,p_n$

Figure 2. Public-ring strategy.

Step 3 is the interesting part of the public-ring strategy. Indeed, first observe that, since processes start at different times, they will join the public ring when buffers have different sequences. This is handled by continuously bringing all sequences in *S* into ***b***, creating *database reading cycles* (step 3 of the strategy). For example, suppose that a process p_i starts when s_k is the sequence in ***b*** with smallest *k*. Then, p_i will not have to wait until s_1 , the first sequence in *S*, is brought into ***b***. Process p_i will start reading s_k and will stop when sequence s_{k-1} is brought again into ***b***, on the next database reading cycle (step 4 of the strategy).

The public-ring strategy reduces resource consumption since all BLAST processes share the same buffers. However, recall that the performance of a BLAST process is proportional to the size of its input sequence. Therefore, the rate at which processes consume the sequences will vary. In other words, the slowest process in the buffer ring will delay the other processes.

We therefore propose a *multi-ring strategy* that:

1. Maintain multiple buffer rings.
2. Possibly allocate several processes to the same ring, trying to minimize delays.
3. For each ring, regulate buffer consumption by the slowest process allocated to that ring.
4. Manage the multiple buffer rings as in the private-ring strategy.
5. Signal to a process when it completes reading all sequences in *S* (with the help of auxiliary structures).

Naturally, the private-ring and the public-ring strategies are extreme cases of the multi-ring strategy.

A detailed description of the multi-ring strategy must deal with issues such as: (1) when to create, destroy and combine buffer rings; (2) what ring should be allocated to a new process. These and other issues are discussed in the next section.

Figure 3 illustrates the multi-ring strategy for the case of 2 rings and 5 BLAST processes, p_1, p_2 , p_3 , p_4 and p_5 , partitioned into two sets {p_1, p_3 , p_5} and {p_2 , p_4}.

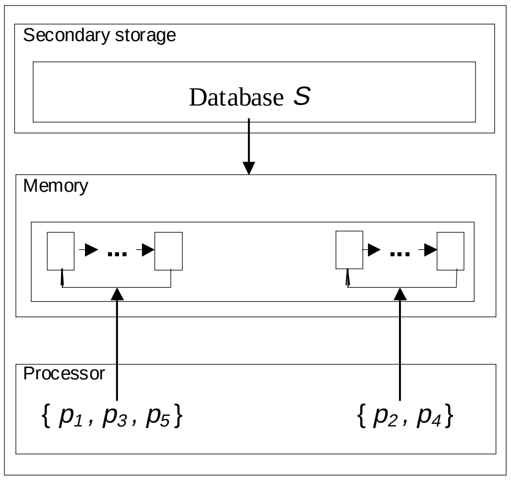

Figure 3. Multi-ring strategy.

3.2 An analysis of the multi-ring strategy

In this section, we first refine the concepts pertaining to the multi-ring strategy and describe an (exponential) algorithm that finds an optimal multi-ring schedule for a set of processes. Then, we indicate that the general problem is NP-complete.

We will consider the following parameters:

s	is the Molecular Biology Database
S	is the size of ***s*** (in bits)
B	is the *total buffer space* (in bits)
R	is the *producer rate* (in bits per second)
T	is the *producer cycle* (in seconds)
P	is a finite set of *consumer processes*

$\tau : P \rightarrow Z^+$ is the *consumer rate function* such that $\tau(p)$ is the *consumer rate* of $p \in P$ (in bits per second)

Intuitively, *B* is the total amount of main memory that can be allocated to buffers, *R* is the rate at which the producer process brings data from secondary storage into main memory, and *T* is the time the processor takes to cycle through empty buffers, filling them with data retrieved from ***s***. Moreover, $\tau(p)$ is the rate at which process *p* analyses data brought into main memory, which is proportional to the size of the input sequence of *p* [Gis00] (the larger $\tau(p)$ is, the faster the process is).

A *multi-ring schedule* for *P* and τ is a partition $\Pi = \{\Pi_1, \Pi_2, ..., \Pi_k\}$ of *P*.

Consider the following additional parameter and additional definitions:

B_i is the *total buffer space* (in bits) allocated to partition Π_i

$\mathrm{T}(\Pi_i) = \max(\tau(p) / p \in \Pi_i)$

$\mu(\Pi_i) = \min(\tau(p) / p \in \Pi_i)$

Intuitively, all processes in each Π_i will share the same buffer ring and the slowest process in Π_i will dictate the rate at which buffers will be consumed, which is $\mu(\Pi_i)$, by definition of μ.

We now discuss the relationships between these parameters. The analysis follows [MNÖ+96], but it ignores seek, rotational and settle time delays for simplicity.

To avoid buffer underflow, the following conditions must be met:

(1) $R \geq \sum_{i=1}^{i=k} \mu(\Pi_i)$

(2) $B \geq \sum_{i=1}^{i=k} B_i$

(3) $\frac{B_i}{2} \geq T.\mu(\Pi_i)$

(4) $\frac{B}{2} \geq T.R$

Equations (1) and (2) say that the producer rate must be no less than the aggregate consumer rate and that the total amount of buffer space must be no less than the aggregate buffer space required by the rings. Now, recall from Section 3.1 that, intuitively, in a cycle *T,* while the consumer processes in Π_i consume half of the buffer in B_i , the producer process fills in the other half of the buffers. Therefore, (3) and (4) capture the relationships between buffer space, process rates and the cycle *T* to guarantee correct buffer utilization.

From (4), we can take the producer cycle *T* as a function of *B* and *R*:

(5) $T = \frac{B}{2R}$

and, from (3) and (5), we can take the buffer space allocated for Π_i as a function of *B*, *R* and $\mu(\Pi_i)$:

(6) $B_i = B.\frac{\mu(\Pi_i)}{R}$

Then, (2) will follow from (1) and (6). Therefore, by assuming (5) and (6), we are left with (1) and the parameters, *R*, *B* and τ, since τ is used to define μ .

Moreover, note that *B* has no lower bound. This limit can be obtained if we take into account the seek, rotational and settle time delays of the producer disk, or if we want to achieve minimum performance for a population of consumer processes, defined by classes of processes, each with an average consumer rate. We will continue to simplify the discussion by not taking into account these considerations.

We then define that a multi-ring schedule Π for *P* and τ is *feasible* for *R* iff $R \geq \sum_{i=1}^{i=k} \mu(\Pi_i)$ is satisfied (with *T* and B_i computed as in (5) and (6) from *R*, τ and an arbitrated total buffer space *B*).

The *delay* of a process *p* in Π is $\Delta_\Pi(p) = S.\left(\frac{1}{\mu(\Pi_i)} - \frac{1}{\tau(p)}\right)$, where Π_i is the partition in Π that *p* belongs to.

The *total delay* of Π is $\Delta(\Pi) = \sum_{p \in P} \Delta_\Pi(p)$.

A multi-ring schedule Π for *P* and τ is *optimal* iff, for any other multi-ring schedule Ω for *P* and τ, we have $|\Pi| \leq |\Omega|$ and $\Delta(\Pi) \leq \Delta(\Omega)$. The first condition says that Π has at most as many rings as Ω and the second, that the total delay of Π is less than or equal to the total delay of Ω.

It is possible to show that:

Theorem 1:

Let *P* be a set of *n* processes,

$p_1,\dots,p_n$ be the elements of *P* ordered in increasing process rate, that is, $\tau(p_1) < \dots < \tau(p_n)$,

Π be a multi-ring schedule for *P* and τ, with *k* elements.

If Π is optimal, then there are integers $r_1,\dots,r_{k-1}$ in $[1,n]$ such that

$$\Pi = \{\{p_1,\dots,p_{r_1}\},\{p_{r_1+1},\dots,p_{r_2}\},\dots,\{p_{r_{k-2}+1},\dots,p_{r_{k-1}}\},\{p_{r_{k-1}+1},\dots,p_n\}\}$$

That is, Π is obtained by cutting $p_1,\dots,p_n$ at the *k-1* points $p_{r_1}, p_{r_2},\dots,p_{r_{k-1}}$.

(See the appendix for a proof).

This simple remark induces the following algorithm to compute a feasible and optimal multi-ring schedule for *P:*

Algorithm 1: Multi-ring optimal scheduling

```
1. Let $L = p_1 ,..., p_n$ be the list of elements in $P$ sorted by increasing consumer process rate.
2. Cut $L$ in all possible ways, generating candidate multi-ring schedules for $P$ and $\tau$, as in Theorem 1.
3. For each multi-ring schedule $\Pi$ generated in Step 2:
    a. If $\Pi$ is feasible, then compute the total delay of $\Pi$.
    b. Otherwise discard $\Pi$.
4. Return the multi-ring schedule with the smallest total delay among those that passed the test in Step 3.
```

Note that there are 2^{n-1} ways to cut L in step 2. Indeed, there are C_{n-1}^{k} ways to select k cut points out of the *n-1* possible cut points, for k=0,...,*n-1*. Hence, the total number of possible ways to cut L is:

$$C_{n-1}^{0} + C_{n-1}^{1} + ... + C_{n-1}^{n-1} = 2^{n-1}$$

In other words, Algorithm 1 is exponential.

As a simple example, consider 5 consumer processes, sorted in a list L by increasing process rate:

$$L = p_1 ,..., p_5 \text{ with } \tau(p_1) < ... < \tau(p_5)$$

Algorithm 1 will then generate the following schedules:

(a) C_4^0 = 1 multi-ring schedule

Cuts	Partitions
-	$\Pi_1 = \{ p_1, p_2, p_3, p_4, p_5\}$

(b) C_4^1 = 4 multi-ring schedules

Cuts	Partitions
p_1	$\Pi_1 = \{ p_1 \}$, $\Pi_2 = \{ p_2 , p_3 , p_4 , p_5\}$
p_2	$\Pi_1 = \{ p_1 , p_2 \}$, $\Pi_2 = \{ p_3 , p_4 , p_5\}$
p_3	$\Pi_1 = \{ p_1 , p_2 , p_3 \}$, $\Pi_2 = \{ p_4 , p_5 \}$
p_4	$\Pi_1 = \{ p_1 , p_2 , p_3 , p_4 \}$, $\Pi_2 = \{p_5 \}$

(c) C_4^2 = 6 multi-ring schedules

Cuts	Partitions
p_1 , p_2	$\Pi_1 = \{p_1 \}$, $\Pi_2 = \{p_2 \}$, $\Pi_3 = \{p_3 , p_4 , p_5 \}$
p_1 , p_3	$\Pi_1 = \{p_1 \}$, $\Pi_2 = \{p_2 , p_3 \}$, $\Pi_3 = \{p_4 , p_5 \}$
p_1 , p_4	$\Pi_1 = \{p_1 \}$, $\Pi_2 = \{p_2 , p_3 , p_4 \}$, $\Pi_3 = \{p_5 \}$
p_2 , p_3	$\Pi_1 = \{p_1 , p_2 \}$, $\Pi_2 = \{p_3 \}$, $\Pi_3 = \{p_4 , p_5 \}$
p_2 , p_4	$\Pi_1 = \{p_1 , p_2 \}$, $\Pi_2 = \{p_3 , p_4 \}$, $\Pi_3 = \{p_5 \}$

p_3, p_4	$\Pi_1 = \{p_1, p_2, p_3\}$, $\Pi_2 = \{p_4\}$, $\Pi_3 = \{p_5\}$

(d) $C_4^3 = 4$ multi-ring schedules

Cuts	Partitions
p_1, p_2, p_3	$\Pi_1 = \{p_1\}, \Pi_2 = \{p_2\}$, $\Pi_3 = \{p_3\}, \Pi_4 = \{p_4, p_5\}$
p_1, p_2, p_4	$\Pi_1 = \{p_1\}$, $\Pi_2 = \{p_2\}$, $\Pi_3 = \{p_3, p_4\}, \Pi_4 = \{p_5\}$
p_2, p_3, p_4	$\Pi_1 = \{p_1, p_2\}, \Pi_2 = \{p_3\}$, $\Pi_3 = \{p_4\}, \Pi_4 = \{p_5\}$
p_1, p_2, p_4	$\Pi_1 = \{p_1\}$, $\Pi_2 = \{p_2, p_3\}$, $\Pi_3 = \{p_4\}, \Pi_4 = \{p_5\}$

(e) $C_4^4 = 1$ multi-ring schedule

Cuts	Partitions
$p_1, p_2,$ p_3, p_4	$\Pi_1 = \{p_1\}, \Pi_2 = \{p_2\}, \Pi_3 = \{p_3\}$, $\Pi_4 = \{p_4\}, \Pi_5 = \{p_5\}$

Note that Algorithm 1 creates and tests 2^4 multi-ring schedules. Indeed, we have

$$C_4^0 + C_4^1 + C_4^2 + C_4^3 + C_4^4 = 2^4$$

We now proceed to investigate the complexity of the multi-ring scheduling problem, defined as follows:

Instance: A tuple (P, τ, R, N), where P, τ and R are as before and N is a positive integer.

Question: Is there a multi-ring schedule Π of P and τ that is feasible for R and is such that at least N processes do not suffer delays in Π ?

Note that this formulation of the problem simply asks if there are N processes that do not suffer delays. Yet, we can prove that:

Theorem 2: The multi-ring scheduling problem is NP-Complete.

(See the appendix for a proof).

3.3 An heuristic approach to implementing the multi-ring strategy

In view of Theorem 2, we introduce in this section Algorithm 2 that implements the multi-ring strategy using an heuristics to reduce the delay of a process when choosing the ring it is allocated to.

A state of the algorithm is characterized by the following state variables:

R is the producer rate

$P = \{p_1,...,p_n\}$ is the current set of consumer processes

$\tau(p)$ is the consumer rate of $p \in P$

$\Pi = \{\Pi_1, \Pi_2, \ldots \Pi_k\}$ is the current multi-ring schedule

Recall that $\mu(\Pi_r)$, the rate of the slowest process in Π_r, dictates the rate at which buffers will be consumed by all processes in Π_r.

Figure 4 abstracts a state of the algorithm, where the consumer rates are plotted in the horizontal axis and the boxes represent the partitions in Π.

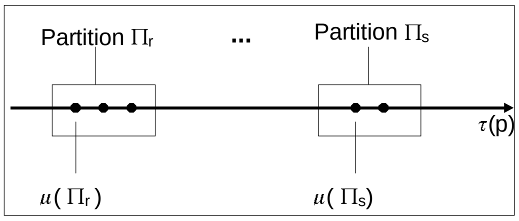

Figure 4. Schematic representation of the state.

Algorithm 2: Heuristic Multi-ring scheduling

Case A: p_{new} is a new BLAST process, with consumer rate $\tau(p_{new})$, that must be scheduled.

1. If it is feasible, add a new partition $\Pi_{new} = \{\ p_{new}\ \}$ to the current multi-ring schedule and return.
2. If $\tau(p_{new})$ is equal to the processing rate of some existing process p_i, then allocate p_{new} to the same partition as p_i and return.
3. Otherwise, try to allocate p_{new} to an already existing partition, which implies that p_{new} will share buffer with existing processes. There are two alternatives:
   a. Let Π_r be the partition with the largest $\mu(\Pi_r)$ such that $\mu(\Pi_r) < \tau(p_{new})$. If we allocate p_{new} to Π_r, then p_{new} will be delayed by:

   (*) $\Delta_r = S.\left(\frac{1}{\mu(\Pi_r)} - \frac{1}{\tau(p_{new})}\right)$

   b. Let Π_s be the partition with the smallest $\mu(\Pi_s)$ such that $\mu(\Pi_s) > \tau(p_{new})$. If we allocate p_{new} to Π_s, then p_{new} will be the new slowest process in Π_s and will therefore delay all processes originally in Π_s. Assuming that Π_s has k processes, the total increase in the delay incurred by processes in Π_s is

   (**) $\Delta_s = k.S.\left(\frac{1}{\tau(p_{new})} - \frac{1}{\mu(\Pi_s)}\right)$

   c. If $\Delta_r < \Delta_s$ then allocate p_{new} to Π_r and return.
   d. Otherwise, allocate p_{new} to Π_s and return.

Case B: *p* is a BLAST process, belonging to Π_r , that finishes.

1. Remove *p* from Π_r .
2. If *p* is not the only process in Π_r , then:
   a. If *p* is not the slowest process in Π_r then return.
   b. Otherwise, let *q* be second slowest process in Π_r and assume that *q* has consumer rate $\tau(q)$. Increase the consumer rate of Π_r up to $\tau(q)$ until the new schedule is feasible and return.
3. If *p* is the only process in Π_r , then Π_r becomes empty.
   a. Move to Π_r as many processes as possible, starting with the slowest process, then the second slowest process, and so on.
   b. Continue the process as long as the new multi-ring schedule is feasible and return.

For the sake of simplicity, the above description ignored several boundary cases, some of which we now briefly discuss. Suppose that *p* is the first BLAST process to start. If Step A.1 fails, *p* is faster than the producer process. In this case, all consumer processes will be allocated to the same (public) buffer ring and will run at the producer rate. Suppose that *p* is a new process and that *p* has a consumer rate larger than any of the current processes. Then, if the test in Step A.1 fails, we fall directly into the case of Step A.3.a. Symmetrically, suppose that *p* is a new process and that *p* has a consumer rate smaller than any of the current processes. Then, if the test in Step A.1 fails, we fall directly into the case of Step A.3.b.

Note that Algorithm 2 produces feasible schedules, but it may not return an optimal schedule.

We may also sophisticate Algorithm 2 by incorporating other heuristics, for example, that try to combine two or more buffer rings. Indeed, it may be worthwhile to combine buffer rings that: (1) have approximately the same consumer rate; and (2) point to approximately the same position in the database. Note that combining buffer rings that violate condition (2) is not desirable since it will force processes in one ring to wait for the processes in the other ring to reach the same position in the database.

3.4 Comparing the buffer management strategies through simulation

We implemented an environment to compare the buffer management strategies through simulation. The environment features a small, 10MB test database, created by borrowing sequences from GenBank [BML+00], EMBL [BBC+00], DDBJ [TMO+00] and PDB [BWF+00], obtained from [NCBI00]. The environment also implements a buffer pool with approximately 10% of the size of the database.

The simulation results corroborated the intuition that the multi-ring strategy outperforms the private-ring strategy. They also helped understand how the size of the buffer pool influences process performance. The details can be found in [Le00].

As a sample result, consider a set of 6 processes, partitioned into two sets: P_X, with 4 processes with rate X, and P_Y, with 2 processes with rate Y. We selected X to be much greater than Y, that is, processes in P_X are faster. We simulated two different schedules:

Schedule 1: a public-ring schedule with 600 buffers

Schedule 2: a 2-ring schedule $\Pi=\{P_X,P_Y\}$, with 400 buffers for P_X and 200 buffers for P_Y

Figure 5 compares Schedules 1 and 2. The results show that the mean processing time of the processes in P_X dropped from, approximately, 800K ms in Schedule 1 to less than 100K ms in Schedule 2. Indeed, in Schedule 1, processes in P_X were being delayed by processes in P_Y. The mean processing time of the processes in P_Y remained approximately the same. This means that Schedule 1 is indeed a poor choice, when compared to Schedule 2.

Figures 6 and 7 show the results for two other schedules, again maintaining the total number of buffers equal to 600:

Schedule 3: a 2-ring schedule $\Pi=\{P_X,P_Y\}$, with 200 buffers for P_X and 400 buffers for P_Y

Schedule 4: a 2-ring schedule $\Pi=\{P_X,P_Y\}$, with 50 buffers for P_X and 550 buffers for P_Y

As expected, these simulations show that the larger the number of buffers, the smaller the processing time (up to saturation).

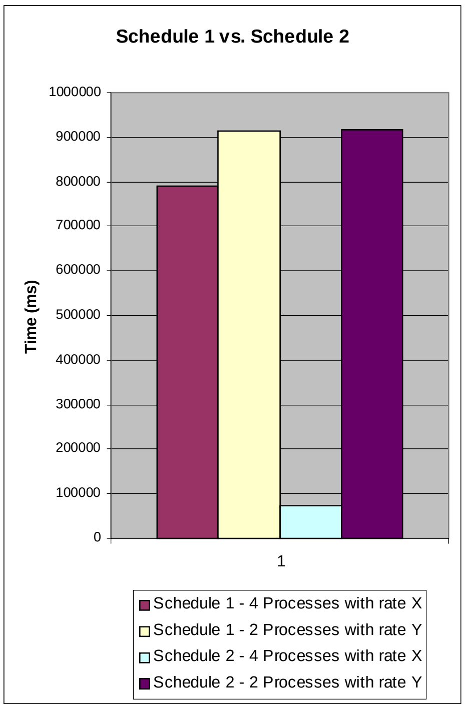

Figure 5. Comparing the public-ring and the multi-ring strategies.

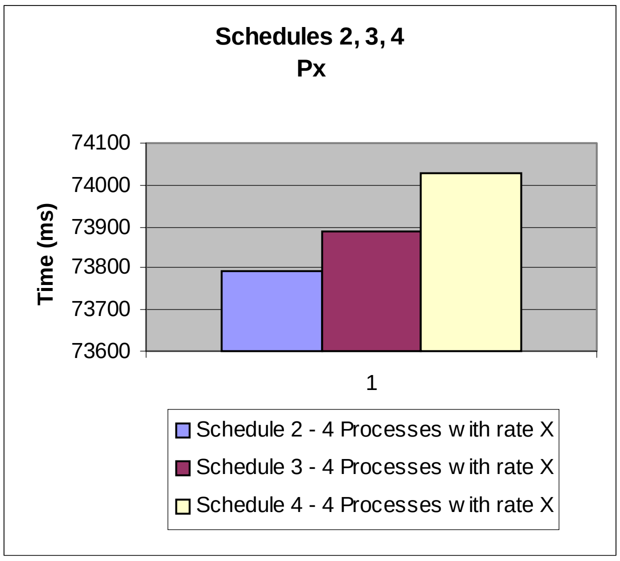

Figure 6. Comparing two multi-ring schedules.

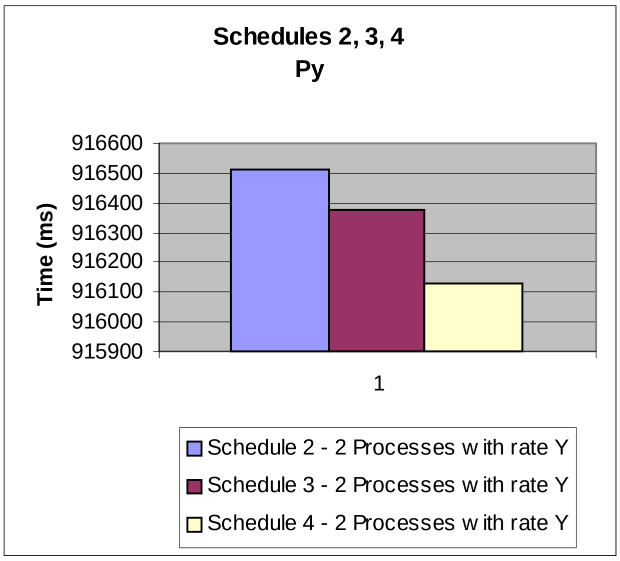

Figure 7. Comparing two multi-ring schedules.

4. Conclusions

In this paper, we addressed sequence comparison, one of the basic operations every Molecular Biology database must support.

We addressed the problem in the context of the BLAST algorithm. Since BLAST performs an exhaustive search of the database to find the best match and there is no obvious (if one at all) access method that helps speed up BLAST, we focused on a buffer management strategy that optimizes the simultaneous execution of a set of BLAST processes.

We described an (exponential) algorithm that finds an optimal set of buffer rings for a set of processes and indicated that the problem is NP-complete. In view of this result, we outlined a second algorithm that implements the multi-ring strategy using an efficient heuristics to reduce the delay of a process when choosing the ring it is allocated to. We also outline simulation results that help understand the buffer management strategy.

Acknowledgement

We wish to thank Antonio Basílio, from the FIOCRUZ Foundation, for his insights and continuos clarification of the concepts pertaining to Molecular Biology.

References

[AGM+90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, e D. J. Lipman. "A basic local alignment search tool". *J. of Molecular Biology* 215, pp. 403-410 (1990).

[AMS+97] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, e D. J. Lipman. "Gapped blast and psi-blast: a new generation of protein database search programs". *Nucleic Acids Research,* Vol. 25, No. 17, pp. 3389-3402 (1997).

[BB99] D.A. Benson, M.S. Boguski, D.J. Lipman, J. Ostell, B.F. Francis Ouellette, B.A. Rapp, D.L. Wheeler. "GenBank". *Nucleic Acids Research*, Vol. 27, pp. 12-17 (Jan. 1999).

[BBC+00] W. Baker, A. van den Broek, E. Camon, P. Hingamp, P. Sterk, G. Stoesser, M. Ann Tuli. "The EMBL Nucleotide Sequence Database". *Nucleic Acids Research,* Vol. 28, No. 1, pp. 19-23 (2000).

[BG99] W.C. Barker, J.S. Garavelli, P.B. McGarvey, C. R. Marzec, B.C. Orcutt, G.Y. Srinivasarao, Lai-Su L. Yeh, R.S. Ledley, H.-W. Mewes, F. Pfeiffer, A. Tsugita, C. Wu. "The PIR-International Protein Sequence Database". *Nucleic Acids Research*, Vol. 27, pp. 39-43 (Jan. 1999).

[BML+00] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, B.A. Rapp, D.L. Wheeler. "GenBank". *Nucleic Acids Research*, Vol. 28, No. 1, pp. 15-18 (2000).

[BWF+00] H.M. Berman, J. Westbrook, Z. Feng, G. Gillil, T.N. Bhat, H. Weissig, I.N. Shindyalov, P.E. Bourne. "The Protein Data Bank". *Nucleic Acids Research*, Vol. 28, No. 1, pp. 235-242 (2000).

[CF93] A.J. Cuticchia, K.H. Fasman, D.T. Kingsbury, R.J. Robbins, P.L. Pearson. "The GDB human genome data base anno 1993". *Nucleic Acids Research,* Vol. 21, pp. 3003-3006 (1993).

[Chu96] S.M. Chung (Editor). *Multimedia Information Storage and Management.* Kluwer Academic Publisher (1996).

[Doo90] R.F. Doolittle (editor). "Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences." *Methods in Enzymology*. Academic Press 183 (1990).

[DT92] R. Durbin, J. Thierry-Mieg. "*Syntactic Definitions for the ACeDB Data Base Manager*". in http://probe.nalusda.gov:8000/acedocs/.

[Gis00] W. Gish, personal communication (2000).

[GJ78] M. Garey and D. Johnson, *Computers and Intractability - A Guide to the Theory of NP-Completeness,* W.H. Freeman and Co., San Francisco (1978).

[GR94] N. Goodman, S. Rozen, L. Stein. "Building a Laboratory Information System Around a C++ Based Object-Oriented DBMS". *Proceedings of the Twentieth International Conference on Very Large Databases*, pp. 722-729 (1994).

[Le00] M. Lemos. "Gerenciamento de Memória para Comparação de Biossequências", M.Sc. Dissertation, Departamento de Informática, Puc-Rio (Sept. 2000).

[MNÖ+96] C. Martin, P.S. Narayanan, B. Özden, R. Rastogi, A. Silberschatz. "The Fellini Multimedia Storage Server". in [Chu96], pp. 117-146.

[MR95] V.M. Markowitz, O. Ritter. "Characterizing Heterogeneous Molecular Biology Database Systems". *Journal of Computational Biology*, Vol.2, No.4 (1995).

[MS97] J. Meidanis, J.C. Setúbal. *Introduction to Computacional Molecular Biology.* PWS Publishing Company (1997).

[NCBI00] National Center for Biotechnology Information.*"BLAST"*. in http://www.ncbi.nlm.nih.gov/BLAST/ (2000).

[Pea90] W.R. Pearson. "Rapid and sensitive sequence comparison with FASTP and FASTA" in [Doo90], pp.63-98.

[Pea91] W.R. Pearson. "Searching Protein Sequence Libraries: Comparison of the Sensitivity and Selectivity of the Smith-Waterman and FASTA algorithms". *Genomics* 11, pp.635-650 (1991).

[Ram98] R. Ramakrishnan. *Database Management Systems.* WCB/McGrawHill (1998).

[Tan92] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Inc. (1992).

[TMO+00] Y. Tateno, S. Miyazaki, M. Ota, H. Sugawara, T. Gojobori. "DNA Data Bank of Japan (DDBJ) in collaboration with mass sequencing teams". Nucleic Acids Research, Vol. 28, No. 1, pp. 24-26 (2000).

[WU00] Washington University BLAST Archives. in http://blast.wustl.edu (2000).

Appendix

First recall that

$$\mathrm{T}(\Pi_i) = \max(\tau(p) / p \in \Pi_i)$$
$$\mu(\Pi_i) = \min(\tau(p) / p \in \Pi_i).$$

A *canonical ordering* for Π is any ordering $\Pi_1,...,\Pi_k$ for the elements of Π such that, for any $i, j \in [1,k]$, if $i < j$ then $\mu(\Pi_i) \leq \mu(\Pi_j)$. Note that a canonical ordering for Π always exists.

Lemma 1:

Let *P* be a set of processes and Π be a multi-ring schedule for *P* and μ with *k* elements.

Let $\Pi_1,...,\Pi_k$ be a canonical ordering for Π.

If Π is optimal, then for any $i, j \in [1,k]$, with $i \neq j$, if $i < j$ then $\mu(\Pi_i) < \mu(\Pi_j)$ and $\mathrm{T}(\Pi_i) < \mu(\Pi_j)$.

Proof

Assume that Π is optimal, but there are $i, j \in [1,k]$, with $i \neq j$, such that $i < j$ and $\mu(\Pi_i) \geq \mu(\Pi_j)$ or $\mathrm{T}(\Pi_i) \geq \mu(\Pi_j)$.

Case 1: Assume that

(1) $\mu(\Pi_i) \geq \mu(\Pi_j)$

First observe that, since $\Pi_1,...,\Pi_k$ is a canonical ordering for Π, we have

(2) $\mu(\Pi_i) \leq \mu(\Pi_j)$

Hence, by (1) and (2), we must have

(3) $\mu(\Pi_i) = \mu(\Pi_j)$

Now, construct a new schedule Ω by collapsing Π_i and Π_j into a single partition Ω_i and leaving the rest unchanged. Since $\mu(\Pi_i) = \mu(\Pi_j)$, we have that $\mu(\Omega_i) = \mu(\Pi_i) = \mu(\Pi_j)$, which implies that

(4) for any $p \in \Pi_i \cup \Pi_j$, $\Delta_\Pi(p) = S.\left(\frac{1}{\mu(\Pi_i)} - \frac{1}{\tau(p)}\right) = S.\left(\frac{1}{\mu(\Omega_i)} - \frac{1}{\tau(p)}\right) = \Delta_\Omega(p)$

Thus, by (4), we conclude that $\Delta(\Pi) = \Delta(\Omega)$. But Ω has one less ring than Π. Therefore, Ω has the same total delay as Π and one less ring. Thus, by definition of optimal schedule, Π is not optimal. Contradiction.

Case 2: Assume that

(5) $\mathrm{T}(\Pi_i) \geq \mu(\Pi_j)$

By case 1, we may assume that:

(6) $\mu(\Pi_i) < \mu(\Pi_j)$

Let $p_i \in \Pi_i$ be such that:

(7) $\tau(p_i) = \mu(\Pi_i)$

Now, construct a second scheduling Ω such that

(8) $\Omega_r = \Pi_r$ if $r \neq i$ and $r \neq j$

$\Omega_i = \Pi_i - \{p_i\}$

$\Omega_j = \Pi_j \cup \{p_i\}$

Note that, since $\tau(p_i) = \mathrm{T}(\Pi_i) \geq \mu(\Pi_j)$, moving p_i to Π_j does not affect $\mu(\Pi_j)$. That is

(9) $\mu(\Omega_j) = \mu(\Pi_j)$

The delays of p_i in Π and in Ω are

(10) $\Delta_\Pi(p_i) = S.\left(\frac{1}{\mu(\Pi_i)} - \frac{1}{\tau(p_i)}\right)$ since $p_i \in \Pi_i$

(11) $\Delta_\Omega(p_i) = S.\left(\frac{1}{\mu(\Omega_j)} - \frac{1}{\tau(p_i)}\right)$ since $p_i \in \Omega_j$

But

(12) $\Delta_\Omega(p_i) = S.\left(\frac{1}{\mu(\Omega_j)} - \frac{1}{\tau(p_i)}\right)$ by (11)

$= S.\left(\frac{1}{\mu(\Pi_j)} - \frac{1}{\tau(p_i)}\right)$ by (9)

$< S.\left(\frac{1}{\mu(\Pi_i)} - \frac{1}{\tau(p_i)}\right)$ by (6)

$= \Delta_\Pi(p_i)$ by (10)

But (12) implies that $\Delta(\Omega) < \Delta(\Pi)$, that is, Π is not optimal. Contradiction.

Theorem 1:

Let P be a set of *n* processes,

$p_1,...,p_n$ be the elements of *P* ordered in increasing process rate, that is, $\tau(p_1) < ... < \tau(p_n)$,

Π be a multi-ring schedule for *P* and τ, with *k* elements.

If Π is optimal, then there are integers $r_1,...,r_{k-1}$ in $[1,n]$, with $r_1 < ... < r_{k-1}$, such that

$\Pi = \{\{p_1,..., p_{r_1}\},\{p_{r_1+1},..., p_{r_2}\},...,\{p_{r_{k-2}+1},..., p_{r_{k-1}}\},\{p_{r_{k-1}+1},..., p_n\}\}$

That is, Π is obtained by cutting $p_1,...,p_n$ at the *k-1* points $p_{r_1}, p_{r_2},..., p_{r_{k-1}}$.

Proof

Let Π be a multi-ring schedule for *P* and τ, with *k* elements. Let $\Pi_1,...,\Pi_k$ be a canonical ordering for Π. Assume that Π is optimal. By Lemma 1, we have $\mathrm{T}(\Pi_{i-1}) < \mu(\Pi_i) \leq \mathrm{T}(\Pi_i) < \mu(\Pi_{i+1})$. By definition of T and μ, this implies that each process in Π_i has a process rate greater than the process rate of any process in Π_{i-1} and process rate smaller than the process rate of any process in Π_{i+1}. Therefore, we can select $r_1 < ... < r_{k-1}$, with the desired property.

Lemma 2:

Let $I' = (U', s', v', L', K')$ be an instance of the knapsack problem [GJ78].

Let $I = (U, s, v, L, K)$ be another instance of the knapsack problem constructed as follows:

$u_{\max}$ is an element not in U'

$U = U' \cup \{u_{\max}\}$

$s(u) = s'(u)$ for each $u \in U'$

$s(u_{\max}) = 1$

$v(u) = v'(u)$ for each $u \in U'$

$v(u_{max}) = \sum_{u \in U'} v(u) + 1$

$L = L' + 1$

$K = K' + v(u_{\max})$.

Then, I' has a solution iff I has a solution that contains $u_{\max}$.

Proof

$(\Rightarrow)$ Suppose that I' has a solution.

Then, there is $U'' \subseteq U'$ such that $\sum_{u \in U''} s'(u) \le L'$ and $\sum_{u \in U''} v'(u) \ge K'$.

Let $U''' = U'' \cup \{u_{\max}\}$. Then, $U''' \subseteq U$ and

(1) $\sum_{u \in U'''} s(u) = \sum_{u \in U''} s(u) + s(u_{\max})$

$= \sum_{u \in U''} s'(u) + 1 \le L' + 1 = L$

(2) $\sum_{u \in U'''} v(u) = \sum_{u \in U''} v(u) + v(u_{\max})$

$= \sum_{u \in U''} v'(u) + v(u_{\max})$

$\ge K' + v(u_{\max}) = K$

Therefore, I has a solution.

$(\Leftarrow)$ Suppose I has a solution.

Then, there is $U''' \subseteq U$ such that $\sum_{u \in U'''} s(u) \le L$ and $\sum_{u \in U'''} v(u) \ge K$.

Recall that $K' \in \mathrm{N}^+$, $K = K' + v(u_{\max})$ and $v(u_{max}) = \sum_{u \in U'} v(u) + 1$. Hence

(3) $\sum_{u \in U'''} v(u) \ge K = K' + v(u_{max}) = K' + \sum_{u \in U'} v(u) + 1 \ge \sum_{u \in U'} v(u) + 1$

that is

(4) $\sum_{u \in U'''} v(u) \ge \sum_{u \in U'} v(u) + 1$

Recall that $U'''\subseteq U$ and $U = U' \cup \{u_{max}\}$, that is

(5) $U'''\subseteq U' \cup \{u_{\max}\}$

But (4) is possible in the presence of (5) only if $u_{\max} \in U'''$ since $v(u_{\max}) = \sum_{u\in U'} v(u) + 1$.

Let $U'' = U''' - \{u_{\max}\}$. Then, $U''\subseteq U'$ and

(6) $\sum_{u\in U''} s'(u) = \sum_{u\in U''} s(u)$

$$= \sum_{u\in U'''} s(u) - s(u_{\max})$$

$$\leq L - s(u_{\max}) = L - 1 = L'$$

(7) $\sum_{u\in U''} v'(u) = \sum_{u\in U''} v(u)$

$$= \sum_{u\in U'''} v(u) - v(u_{\max})$$

$$\geq K - v(u_{\max}) = K'.$$

Therefore, I' has a solution.

Theorem 2: The multi-ring schedule problem is NP-complete.

Proof

Let $I' = (U', s', v', L', K')$ be an instance of the knapsack problem [GJ78]. By Lemma 2, I' has a solution iff $I = (U, s, v, L, K)$ has a solution $U''\subseteq U'$ such that $u_{\max} \in U''$, where I and $u_{\max}$ are constructed as in Lemma 2:

(1) $u_{\max} \notin U'$

$U = U' \cup \{u_{\max}\}$

$s(u) = s'(u)$ for each $u \in U'$

$s(u_{\max}) = 1$

$v(u) = v'(u)$ for each $u \in U'$

$v(u_{\max}) = \sum_{u\in U'} v(u) + 1$

$L = L' + 1$

$K = K' + v(u_{\max}) = K' + \sum_{u\in U'} v'(u) + 1$.

Assume, without loss of generality, that

(2) for all $u, u' \in U$, if $u \neq u'$ then $s(u) \neq s(u')$

Construct an instance (P, τ, R, N) of the multi-ring schedule problem as follows:

(3) $P[u]$ is a set of $v(u)$ processes, for each $u \in U$

$\tau(u) = s(u)$ for each $u \in U$ and $p \in P[u]$

$R = L$

$N = K$

We call $u \in U$ the *generator* of each $p \in P[u]$ and also say that $p, p' \in P[u]$ are *siblings*.

We show that the knapsack instance (U, s, v, L, K) has a solution iff the instance (P, τ, R, N) of the multi-ring schedule problem has.

$(\Rightarrow)$ Suppose the knapsack instance (U, s, v, L, K) has a solution.

Then,

(4) there is $U'' \subseteq U$ such that $\sum_{u \in U''} s(u) \leq L$ and $\sum_{u \in U''} v(u) \geq K$

By Lemma 1, we may assume that $u_{\max} \in U''$.

By reordering U and by (1), we may assume that

(5) $U'' = \{u_1, \ldots, u_k\}$, $u_k = u_{\max}$ and $s(u_i) < s(u_{i+1})$ for $i \in [1, k-1]$.

Construct a partition $\Pi = \{\Pi_1, \ldots, \Pi_k\}$ of P as follows:

(6) $p \in \Pi_i$ iff $s(u_i) \leq \tau(p) < s(u_{i+1})$ for each $i \in [1, k-1]$

$p \in \Pi_k$ iff $s(u_k) \leq \tau(p)$

Then, all processes in $P[u_i]$ belong to Π_i, by construction of Π_i and since $\tau(p) = s(u_i)$, for all $p \in P[u_i]$.

Moreover, these are the slowest processes in Π_i, since $\tau(p) = s(u_i)$. Hence, they do not suffer delays.

Now, by construction, $|P[u_i]| = v(u_i)$. Therefore

(7) $\sum_{i=1}^{k} |P[u_i]| = \sum_{i=1}^{k} v(u_i) > K = N$

That is

(8) $\sum_{i=1}^{k} |P[u_i]| > N$

Thus, at least N processes in Π do not suffer delays.

Now, by construction of Π, we have

(9) $\mu(\Pi_i) = s(u_i)$ for each $i \in [1, k]$.

Hence

(10) $\sum_{i=1}^{k} \mu(\Pi_i) = \sum_{i=1}^{k} s(u_i) \leq L = R$

That is

(11) $\sum_{i=1}^{k} \mu(\Pi_i) \leq R$

which implies that Π is feasible for *R*.

Therefore, we may conclude that the instance (P, τ, R, N) of the multi-ring schedule problem has a solution.

$(\Leftarrow)$ Suppose that the instance (P, τ, R, N) of the multi-ring schedule problem has a solution.

Then

(12) there is a partition $\Pi = \{\Pi_1, \ldots, \Pi_k\}$ of P such that $\sum_{i=1}^{k} \mu(\Pi_i) \leq R$ and N processes do not suffer delays.

Let p_i be the slowest process in Π_i. Let u_i be the element of U corresponding to p_i. Let $U' = \{u_1, \ldots, u_k\}$.

We shall show that

(13) $\sum_{i=1}^{k} s(u_i) \leq L$ and $\sum_{i=1}^{k} v(u_i) \geq K$.

Recall that, by construction, $s(u_i) = \tau(p_i) = \mu(\Pi_i)$, for each $i \in [1,k]$. Hence, we have:

(14) $\sum_{i=1}^{k} s(u_i) = \sum_{i=1}^{k} \tau(p_i) = \sum_{i=1}^{k} \mu(\Pi_i) \leq R = L$

That is

(15) $\sum_{i=1}^{k} s(u_i) \leq L$

Recall that, by (2), for all $u, u' \in U$, if $u \neq u'$ then $s(u) \neq s(u')$. By construction of Π, we then have that

(16) $\tau(p) = \tau(p')$ iff p and p' are siblings

Hence, p_i, the slowest process in Π_i and its siblings are the only processes in Π_i that do not suffer delays. Therefore, there are exactly $|P[u_i]| = v(u_i)$ processes that do not suffer delays in Π_i. Now, by assumption, Π has at least N processes that do not suffer delays. Hence

(17) $\sum_{i=1}^{k} v(u_i) = \sum_{i=1}^{k} |P[u_i]| \geq N = K$

Therefore

(18) $\sum_{i=1}^{k} v(u_i) \geq K$

Thus, the knapsack instance (U, s, v, L, K) has a solution.