Remote Memory References at Block Granularity

Hagit Attiya[1] and Gili Yavneh[2]

1 **Department of Computer Science, Technion, Haifa, Israel**
`hagit@cs.technion.ac.il`
2 **Department of Computer Science, Technion, Haifa, Israel**
`giliyav@cs.technion.ac.il`

Abstract

The cost of accessing shared objects that are stored in remote memory, while neglecting accesses to shared objects that are cached in the local memory, can be evaluated by the number of *remote memory references* (*RMRs*) in an execution. Two flavours of this measure – *cache-coherent* (CC) and *distributed shared memory* (DSM) – model two popular shared-memory architectures. The number of RMRs, however, does not take into account the *granularity* of memory accesses, namely, the fact that accesses to the shared memory are performed in *blocks*.

This paper proposes a new measure, called *block RMRs*, counting the number of remote memory references while taking into account the fact that shared objects can be grouped into blocks. On the one hand, this measure reflects the fact that the RMR incurred for bringing a shared object to the local memory might save another RMR for bringing another object placed at the same block. On the other hand, this measure accounts for *false sharing*: the fact that an RMR may be incurred when accessing an object due to a concurrent access to another object in the same block.

This paper proves that in both the CC and the DSM models, finding an optimal placement is NP-hard when objects have different sizes, even for two processes. In the CC model, finding an optimal *placement*, i.e., grouping of objects into blocks, is NP-hard when a block can store three objects or more; the result holds even if the sequence of accesses is known in advance. In the DSM model, the answer depends on whether there is an efficient mechanism to inform processes that the data in their local memory is no longer valid, i.e., cache coherence is supported. If coherence is supported with cheap invalidation, then finding an optimal solution is NP-hard. If coherence is not supported, an optimal placement can be achieved by placing each object in the memory of the process that accesses it most often, if the sequence of accesses is known in advance.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.4.1 Process Management

Keywords and phrases false sharing, cache coherence, distributed shared memory, NP-hardness

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.18

1 Introduction

In a typical multiprocessor, processes communicate by concurrently accessing objects in the shared memory. To reduce the high cost of access to the shared memory, a fast memory, local to each process, is used to cache recently-used objects. The cost of a *cache hit* – finding an object in the process's local memory – is negligible relative to the cost of a *cache miss*, which requires an access to the shared memory. We assume that a block is not evicted from the local memory unless it is required by some other process; that is, the local memory is large and fully associative.

A *remote memory reference* (*RMR*) [25] is incurred for every cache miss, according to one of two models: In the *cache coherent* (*CC*) model, the first time a process accesses an

object it puts the object in its local memory; subsequent reads and writes to the same object by the same process are free, as long as no other process modifies the object between them; otherwise, the object must be updated in the local memory again, incurring an RMR. In the *distributed shared memory* (*DSM*) model, each object is created in the local memory of some process, before the first access to it occurs; an access to the local memories of other processes incurs one RMR, while an access to the local memory is free. If cache coherence and cheap data invalidation is supported then, like in the CC model, the object will be copied by other processes for read operations, while write operations invalidate the copies at other processes. Making a copy incurs an RMR, while accesses to an existing copy do not. If invalidation is not supported, the object remains in the local memory of the process that created it, and any access to it by a different process incurs an RMR.

Both models, however, ignore the fact that access to the shared memory is performed in *blocks*, namely, several objects are placed together and moved together between local memories and shared memory. When an object is read into the memory, all the objects in the same block are moved, so that later accesses to them incur a cache hit. Similarly, if one object is moved to another process, accesses to the other objects in the same block causes a cache miss, even if no other process has accessed them in between. This phenomenon is called *false sharing* [23, 11].

Many works deal with cache-conscious organization of the memory, namely, the placement of objects in blocks so as to increase the number of cache hits, e.g., [8, 9, 20]. However, as we discuss in Section 1, these works consider only single-process scenarios and do not take into account the effects of concurrent access to blocks. Algorithms and lower bounds on the number of RMRs, which capture the effects of concurrency, do not take into account the granularity of memory accesses, which are done in blocks.

Our Contributions

We introduce the *block RMRs* complexity measure. In the CC model, when a process accesses a block, an RMR is incurred when the block is brought to its cache, and later accesses to the same block are free, as long as no other process writes an object in this block. For this model, we prove that finding an optimal placement of objects into blocks, i.e., a placement with a minimal number of block RMRs, is *NP-hard* when blocks can hold three or more objects; the result holds even when the sequence of accesses is known and all objects have the same size (Section 3.2). The problematic access sequence is a natural one, in which two processes perform a *traversal* on a graph. When blocks can hold two objects of the same size and the access sequence is known, we present an efficient algorithm for placing objects in blocks for the block-based CC model (Section 3.1).

In the *block-based DSM* model, each block is created in a specific process when the execution starts, and each access to a block not in a process's own cache incurs an RMR. If cache coherence is supported then an object may have several copies, in which case, consecutive accesses to the same block are free, as long as no other process writes an object in this block. If cache coherence is not maintained, then the object remains in the local memory of the process that created it, and accesses from other processes incur an RMR, while accesses of the creating process are free.

For the block-based DSM model, we prove that the number of block RMRs depends on the cost of invalidation and existence of cache coherence. If invalidation cost is negligible and cache coherence is supported then finding a placement of objects into blocks is *NP-hard*, even if the access sequence is known and all objects have the same size. If cache coherence is not supported, we show that the number of block RMRs is indifferent to the order of accesses

in the sequence, and use this result to design an algorithm that finds a placement with an optimal number of block RMRs, when the access sequence is known in advance. (The results for the DSM model appear in Section 4.)

For both models, we prove that handling objects with varying sizes makes the problem NP-hard (Appendix A). This result is achieved with an access sequence in which two processes traverse a tree one after the other. Because of this result, the rest of the paper concentrates on the case where all objects have the same size.

Related work

Remote memory references have been proposed as a way to predict the scalability of shared-memory programs [25]. They have been applied for evaluating the complexity of mutual exclusion algorithms and presenting lower bounds for it, in the CC and DSM models (see the survey [3]). Other works investigated problems like leader election [16] and renaming [2].

False sharing [23, 11, 6] occurs when different objects, placed in the same block, are accessed by different processes, causing cache misses that would not occur if the objects were in separate blocks. Bolosky and Scott [6] introduced models for false sharing and compared how well they align with the intuitive understanding of this phenomenon. Their *interval definition* says that the cost of false sharing is the difference in performance between a policy that makes optimal placement decisions, but enforces consistency on a whole-block basis, and one that enforces consistency only for real conflicts between accesses. Our definitions capture this formally by the difference between the number of block RMRs and the number of RMRs, and allow us to prove the NP-hardness result conjectured in [6].

A large body of research studies memory locality for sequential computing. *Cache-conscious* algorithms take into account the structure of the cache, i.e., cache size, block size, placement of objects in blocks and other parameters, in order to minimize cache misses. Petrank and Rawitz [20] showed that the problem of partitioning data into the blocks of a single cache *of limited size* is NP-hard, and in fact, it is hard to approximate. Our NP-hardness proof for the CC model employs multiple processes, instead of bounding the size of the cache, and is achieved using traversals, rather than with an arbitrary access sequence as in their result. However, we do not show that approximating the optimal solution is hard; in fact, it is not, since the number of RMRs for the sequence divided by the size of the block is a lower bound on the number of block RMRs.

Lavaee [18] showed that the problem of *data packing*, using a fully associative, limited-size cache and for a single process, is NP-hard. By using multiple processes, our result is achieved with a more natural access sequence that does not require dummy objects in order to fill the cache and cause data to be evicted.

Afek et al. [1] introduce a memory allocation scheme that is cache index-aware, i.e., takes into account that objects are placed in a cache "row" that coincides with their cache *index*, which is a consecutive subset of the bits in their memory address.

Cache-oblivious algorithms [14, 22] optimize the object layout in the memory and their design is indifferent to cache parameters, such as size and block size. For example, a partitioning of tree nodes into blocks that reduces the number of blocks accessed is given by van Emde Boas trees [24]. Cache-oblivious algorithms were suggested for matrix transposition and FFT [14] for priority queues [4] and for sorting [14, 7, 12, 13]. All these algorithms are designed for a single process and do not take multi-threading into account.

More recent work attempts to provide cache-oblivious algorithms in multiprocessing environments. One such variant is the *parallel cache-oblivious* model [5]. In general, the number of cache misses for algorithms in this model is higher than in the regular cache-oblivious model, due to restrictions needed to minimize false sharing and memory imbalances between sub-tasks. However, for some tasks, the asymptotic bounds are not affected.

Multi-core oblivious cache-oblivious algorithms [10] are unaware of both the number of cores and the cache parameters. In these algorithms, different processes cooperate in order to complete a single task and data placement is designed to avoid cache misses as much as possible. Our work focuses on multi-process environment as well, but takes into account possible contention between noncooperative tasks run by different processes which may access the same object, for example, concurrent accesses to the same data structure. Our NP-hardness results hold even when the block size B is known and use only two processes. This makes them stronger and they also immediately apply to Multi-core oblivious cache-oblivious algorithms.

2 RMRs and Block RMRs in the Cache Coherence Model

We consider an asynchronous system in which a set of n processes, $P = \{p_1, \ldots, p_n\}$, execute concurrently and communicate by accessing a set O of shared objects. Each process has a local cache memory associated with it. Objects placed in the local memory can be easily retrieved, and the cost of doing so is negligible compared with the cost of fetching objects from another process's local cache, or from the main shared memory. A process may read an object's current value, or write a new value to an object.

Objects may be part of the same data structure, for example, vertexes of a graph. A common access pattern to such data structures is a *traversal*: a sequence of accesses by a single process to the vertexes of the graph, where each pair of consecutive accesses in the sequence π, either access the same vertex or adjacent vertexes.

The local memory is partitioned into *blocks* of *size* B, each of which can contain objects whose combined sizes is at most B. The local memory can hold an unbounded number of blocks and therefore, blocks are not evicted due to lack of space. A *B-block placement* of O is a partition of the objects in O into disjoint sets (*blocks*), $\tilde{O} = \{O_1, \ldots, O_\ell\}$, each containing objects with a combined size of at most B. We assume each object can be placed in a single block, and is not spread across blocks.[1]

If all objects have the same size, we will consider their size to be 1. In this case, a *B-block placement* of O is $\tilde{O} = \{O_1, \ldots, O_\ell\}$ such that for each block $|O_i| \leq B$.

An *access sequence* π is a sequence $(p_{i_1}, a_1, o_{j_1}), \ldots, (p_{i_m}, a_m, o_{j_m})$ such that $p_{i_h} \in P$, $a_h \in \{\text{read,write}\}$ and $o_{j_h} \in O$, for every h, $1 \leq h \leq m$.

There are two models for counting the number of *remote memory references* (*RMR*s) in an access sequence π, the *cache coherent* (*CC*) model and the *distributed shared memory* (*DSM*) model (see Figure 1). In this section, we concentrate on the CC model; the DSM model is discussed in Section 4.

In the CC model, a remote memory reference (RMR) is incurred either on the first access (whether it is a read or a write) to an object by a process or on the first access after a write to the same object by another process. Formally, for an access sequence $\pi = (p_{i_1}, a_1, o_{j_1}), \ldots, (p_{i_m}, a_m, o_{j_m})$

$$\begin{aligned} \#rmr^{\text{CC}}(\pi) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | (\forall k, 1 \leq k < h, (p_{i_h} = p_{i_k}) \implies (o_{j_h} \neq o_{j_k})) \text{ or} \\ (\exists k, 1 \leq k < h, (o_{j_k} = o_{j_h}, a_k = \text{write, and } p_{i_k} \neq p_{i_h}) \\ \text{and } (\forall \ell, k < \ell < h, (p_{i_h} = p_{i_\ell}) \implies (o_{j_h} \neq o_{j_\ell})))\}| \ . \end{aligned}$$

[1] Objects with size larger than B need more than one block. By placing an object in the minimal number of blocks that can contain it (i.e., all parts of the object except perhaps one are placed in a block alone), our results hold in this model, by ignoring the parts of the object that fill full blocks and taking into account only the part of the object that remains and does not fill a full block.

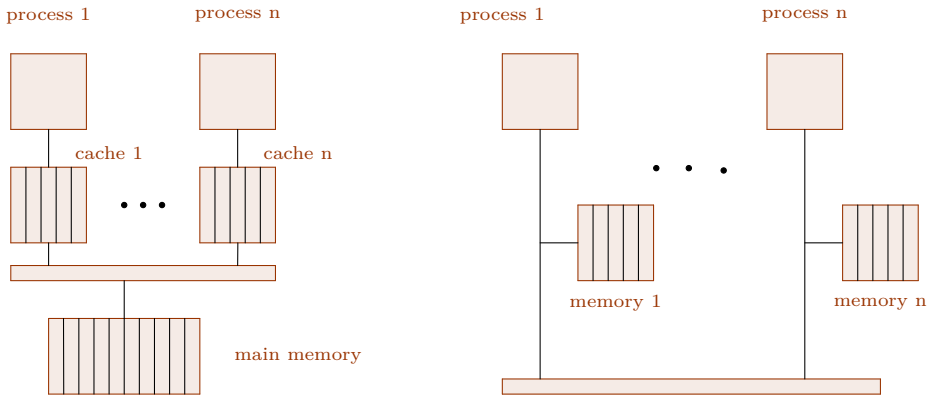

Figure 1 Models: Cache Coherence (left) and DSM (right).

Fix a B-block placement $\tilde{O} = \{O_1, \ldots, O_\ell\}$ for an access sequence π, as above. We count the number of *block* accesses in π that incur a cache miss, i.e., the first access (either a read or a write) to an object in a block, or an access after a write by another process to *some object in the same block*:

$$\begin{aligned} \#brmr^{\mathrm{CC}}(\pi, \tilde{O}) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | (o_{j_h} \in O_l) \text{ and} \\ ((\forall k, 1 \leq k < h, (p_{i_h} = p_{i_k}) \implies (o_{j_k} \notin O_l)) \text{ or} \\ (\exists k, 1 \leq k < h, (o_{j_k} \in O_l, a_k = \text{write, and } p_{i_k} \neq p_{i_h}) \\ \text{and } (\forall \ell, k < \ell < h, (p_{i_h} = p_{i_\ell}) \implies (o_{j_\ell} \notin O_l)))))\}| \,. \end{aligned}$$

3 Block RMRs in the CC model

In the cache coherence model, all objects are in the main memory before the access sequence is executed. Therefore, once the access sequence π is fixed, no optimization can reduce the number of RMRs and it can be computed by a sequential pass over the access sequence π, while tracking the last process that modified each object. Specifically, for every access, if there was no previous access in the sequence to the same object by the process or if the previous access was a modification by a different process, we increase the number of RMRs by one and update the latest process to access the object.

Note that when each object is placed in a separate block, that is, in the assignment $\tilde{O} = \{O_1, \ldots, O_n\}$, with $|O_i| = 1$, for every i, $1 \leq i \leq n$, we have:

$$\#brmr^{\mathrm{CC}}(\pi, \tilde{O}) = \#rmr^{\mathrm{CC}}(\pi).$$

Therefore, $\#rmr^{\mathrm{CC}}(\pi)$ is an upper bound on the minimal number of block RMRs for the sequence π.

Given a B-block placement of O, $\tilde{O} = \{O_1, \ldots, O_n\}$, let $\#rmr^{\mathrm{CC}}(o_i, \pi)$ be the number of accesses to the object o_i that incur an RMR, i.e., an access to o_i that caused the block containing o_i to be brought to the process's local memory. For every access to o_i that incurs an RMR, there is a block RMR that brings o_i to the accessing process, incurred by either the access itself or a previous access to another object in the block. Therefore, the number

of block RMRs for objects in O_j is at least

$$\max_{o_i \in O_j} (\#rmr^{\mathrm{CC}}(o_i, \pi)).$$

Since each block contains at most B objects, the number of block RMRs is at least

$$\frac{\Sigma_{o_i \in O_j} (\#rmr^{\mathrm{CC}}(o_i, \pi))}{B}.$$

The overall number of block RMRs is the sum of block RMRs for the objects in each block O_j and therefore, it is at least

$$\Sigma_{O_j \in \tilde{O}} \frac{\Sigma_{o_i \in O_j} (\#rmr^{\mathrm{CC}}(o_i, \pi))}{B} = \frac{\#rmr^{\mathrm{CC}}(\pi)}{B}.$$

Therefore, for every access sequence π,

$$\frac{\#rmr^{\mathrm{CC}}(\pi)}{B} \leq \min_{\tilde{O}} \#brmr^{\mathrm{CC}}(\pi, \tilde{O}) \leq \#rmr^{\mathrm{CC}}(\pi).$$

We consider the question of optimally placing objects into blocks, in order to minimize the number of block RMRs. We prove that for $B > 2$, the problem is NP-hard, even if the sequence of accesses is known in advance, by showing a polynomial-time reduction from the *graph partitioning problem* [17]. But first, we prove that there is a polynomial-time algorithm for this problem when $B = 2$.

3.1 A Polynomial Algorithm for 2-Block Placement

A 2-block placement can be found with an algorithm for finding a maximum weighted matching for a graph, which can be done in $O(|V|^2|E|)$ using linear programming [21].

▶ **Definition 1** (Maximum Weighted Matching)**.**
Input: Undirected graph $G = (V, E)$, weights $w(e) \in Q$ for each edge $e \in E$.
Question: Which matching, i.e., a set of pairwise non-adjacent edges, has maximum weight (the sum of the weights of the edges in the matching)?

Given an input to the 2-block placement problem, we take the complete graph $G = (V, E)$ with a vertex $v_i \in V$ for every object $o_i \in O$ and an edge between every pair of vertexes.

The edge weights are calculated according to the access sequence π. For each access by process p_i to an object o_1 and every object o_2, we determine whether the access would incur an RMR if o_1 and o_2 are placed in the same block. To do this, we check if o_2 was previously accessed by p_i. If there is a previous access to o_2 by p_i, we check whether an access by another process in between could invalidate the block in the local memory and decrease the weight of the edge between o_1 and o_2. Otherwise, we check which previous access could save an RMR if o_1 and o_2 are placed in the same block and increase the weight of the edge between o_1 and o_2. For example, Figure 2(left) shows the final weights for the next access sequence on four objects A, B, C, D:

$$(p_1, \text{write}, A)(p_1, \text{write}, B)(p_2, \text{write}, C)(p_3, \text{write}, D)(p_2, \text{write}, C)$$

The weight of (A, B) is 1 because placing them in the same block would save a block RMR on the access to B. The weight of (C, D) is -1 because placing them in the same block would incur an extra block RMR on the second access to C (which does not happen if C is placed in a singleton block).

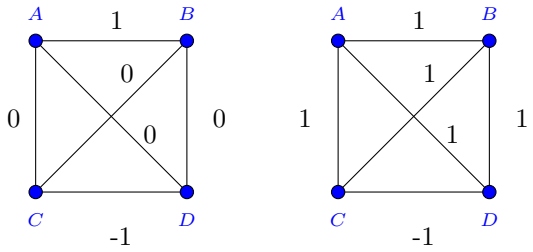

Figure 2 Graph with original weights (left) and after incorrect weight adjustment (right).

After the weights are calculated, we find a maximum matching in the weighted graph. A matching and a 2-block placement naturally correspond to each other: the endpoints of edges in the matching represent disjoint pairs of objects, and each pair can be placed in a single block in the memory. The remaining objects are placed in singleton blocks.

The weights are calculated according to the access sequence π (Algorithm 1). We initialize the weight for each edge to zero. Next we go over the access sequence, and for each access to an object, check which previous accesses to other objects could either increase or decrease the number of RMRs if they are put in the same block as the current object. For each access $(p_{i_h}, a_h, o_{j_h}) \in \pi$, let $(p_{i_k}, a_k, o_{j_h}) \in \pi$ be the previous access to the same object in π. If no such access exists, for example, if (p_{i_h}, a_h, o_{j_h}) is the first access to the object, we look for previous accesses by the same process. For every object o, whose most recent access (prior to the h-th access) is by p_{i_h}, we increase the weight of (o, o_{j_h}) by 1, since placing them in the same block will avoid a block RMR on the access (p_{i_h}, a_h, o_{j_h}).

Now we assume (p_j, a_j, o_j) exists. If $p_j = p_i$, then for each object o', if there is a process $p' \neq p_i$ such that there is an index k such that $j < k < i$ and $p_k = p'$ and $o_k = o'$, and this is the most recent write to o' prior to the i-th access, we decrease the weight of the edge (o', o_i) by 1. For example, for three processes, objects $O = \{A, B, C, D\}$, and the sequence:

$$(p_1, \text{write}, A), (p_2, \text{write}, B), (p_1, \text{write}, C), (p_3, \text{write}, B), (p_1, \text{write}, A),$$

the weight of the edge (A, B) will be decreased by 1, since there is a write to B by a process other than p_1 between the two accesses to A. Intuitively, if A and B are placed in the same block then we will incur an extra RMR for this part of the sequence, which would not be incurred had they been placed in different blocks.

If, on the other hand, $p_j \neq p_i$, then for each object o' such that there is an index k, $j < k < i$, $p_k = p_i$ and $o_k = o'$, and this is the most recent write to o' prior to the i-th access, we increase the weight of the edge (o', o_i) by 1. For example, for the sequence:

$$(p_4, \text{write}, D), (p_2, \text{write}, B), (p_1, \text{write}, C), (p_2, \text{write}, B), (p_2, \text{write}, D)$$

the weight of the edge (B, D) will be increased by 1, since there is a write to B by process p_2 between the two writes to D by p_2. Intuitively, if D and B are placed in the same block then we will incur one less RMR for this part of the sequence than would have been incurred had they been placed in different blocks.

Figure 3 shows the final weights of the graph for the sequence that is the concatenation of the two previous sequences (all actions are writes):

$$(p_1, A), (p_2, B), (p_1, C), (p_3, B), (p_1, A), (p_4, D), (p_2, B), (p_1, C), (p_2, B), (p_2, D)$$

Algorithm 1 Sequential pseudocode for calculating the weights.

```
    calc_weights(G = (V, E), w, acc)
 1: init w to zeros
    //latest and latest_write are arrays of size |V| representing,
    //respectively, the most recent access and write to an object
 2: init latest to nulls
 3: init latest_write
 4: for i = 1...|acc|:
 5:       (curr_proc, curr_act, curr_obj) = acc[i]
 6:       prev_acc = latest[curr_obj]
 7:       if (prev_acc is null):
              // Find which objects were previously accessed by the
              // same process, and increase the corresponding weight
 8:           for obj = 1...|V|:
 9:               if (obj ≠ curr_obj and latest[obj] ≠ null):
10:                   (proc, temp1, temp2) = acc[latest[obj]]
11:                   if (proc == curr_proc):
12:                       w((obj, curr_obj))+ = 1
13:      else:
14:          prev_proc = acc[prev_acc][0]
15:          for obj = 1...|V|:
                 // Check if the obj was accessed between the previous access
                 // to curr_obj and the current access
16:              if(obj ≠ curr_obj and latest[obj] ≠ null
                 and latest[obj] > prev_acc):
17:                  (proc, temp1, temp2) = acc[latest[obj]]
                     // Putting obj and current_obj together can save a block RMR
18:                  if (prev_proc ≠ curr_proc
                       and proc = curr_proc):
19:                      w((obj, curr_obj))+ = 1
                     // Putting obj and current_obj together can incur an extra block RMR
20:                  else if (latest_write[obj] ≠ null
                       and latest_write[obj] > prev_acc
                       and prev_proc = curr_proc
                       and acc[latest_write[obj]][0] ≠ curr_proc):
21:                      w((obj, curr_obj))− = 1
      // Update latest and latest_write according to the access
22:   latest[curr_obj] = i
23:   if curr_act is a write:
24:       latest_write[curr_obj] = i
```

▶ **Lemma 2.** *If the matching $M_{\tilde{O}} \subseteq E$ corresponds to a 2-block placement $\tilde{O} = \{O_1, \ldots, O_\ell\}$, then $\#rmr^{\mathrm{CC}}(\pi) - \Sigma_{e \in M_{\tilde{O}}} w(e) = \#brmr^{\mathrm{CC}}(\pi, \tilde{O})$.*

Proof. Consider some block $O_j \in \tilde{O}$. If $O_j = \{a\}$ is a singleton block, then the number of RMRs incurred by access to a is $\#rmr^{\mathrm{CC}}(a)$, since accesses to other objects do not cause it to move to a different cache. On the other hand, if O_j is mapped to some edge $e_{O_j} \in M_{\tilde{O}}$ it contains two objects, a and b. We count the block RMRs incurred by accesses to a and b. For every access to a (and similarly b) there are three cases:

1. The access is counted as an RMR in $\#rmr^{\mathrm{CC}}(\pi)$ and a block RMR in $\#brmr^{\mathrm{CC}}(\pi, \tilde{O})$.
2. The access is counted as an RMR in $\#rmr^{\mathrm{CC}}(\pi)$, but not in $\#brmr^{\mathrm{CC}}(\pi, \tilde{O})$. This happens only if the access to a was preceded by an access to b, that brought O_j to the accessing process' cache. This can happen in either the first access to a or in subsequent ones. We note that for each such occurrence we increased $w(e_{O_j})$ by 1.

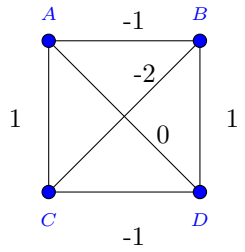

Figure 3 Graph after the weights are calculated.

3. The access is counted as an RMR $\#brmr^{\mathrm{CC}}(\pi, \tilde{O})$, but not in $\#rmr^{\mathrm{CC}}(\pi)$. This happens only if the access to a was preceded by an access to b, which moved the block O_j from the process' cache. We note that for each such occurrence we decreased $w(e_{O_j})$ by 1.

Therefore, the number of block RMRs incurred by a and b is exactly

$$\#rmr^{\mathrm{CC}}(a) + \#rmr^{\mathrm{CC}}(b) - w(e_{O_j}).$$

Summing over all blocks, we get:

$$\#brmr^{\mathrm{CC}}(\pi, \tilde{O}) = \Sigma_{O_j \in \tilde{O}, O_j = \{a\}}(\#rmr^{\mathrm{CC}}(a)) + \Sigma_{O_j \in \tilde{O}, O_j = \{a,b\}}(\#rmr^{\mathrm{CC}}(a) + \#rmr^{\mathrm{CC}}(b) - w(e_{O_j})).$$

And thus:

$$\#rmr^{\mathrm{CC}}(\pi) - \Sigma_{e \in M_{\tilde{O}}} w(e) = \#brmr^{\mathrm{CC}}(\pi, \tilde{O}).$$ ◀

Given this lemma, it is easy to see that if $\Sigma_{e \in M_{\tilde{O}}} w(e)$ is maximized, then our target function $\#brmr^{\mathrm{CC}}(\pi, \tilde{O})$ is minimized. Finding the maximum for $\Sigma_{e \in M_{\tilde{O}}} w(e)$ is finding a maximum weight matching in a weighted graph, which can be solved in $O(|V|^2 \cdot |E|)$ steps [21]. Since $|V| = |O|$ and the graph is complete, this is in $O(|O|^3)$. The total complexity of the algorithm depends on the calculation of edge weights. Using a straightforward approach, for every access we must go over all previous accesses and update an edge to every object, resulting in an $O(|\pi|^2 + |\pi| \cdot |O| + |O|^3)$ time complexity. This can be improved by using a hash table to remember the last read and write for each object while going over the access sequence. This alleviates the need to go over all previous accesses; instead, we go over the objects and find which were accessed in the relevant part of the sequence. This results in $O(|\pi| \cdot |O| + |O|^3)$ time complexity.

The algorithm does not guarantee that between two solutions with the same minimal number of block RMRs, it chooses the one with the smallest number of blocks. This can be done by maximizing the number of blocks containing two objects, or equivalently, maximizing the number of edges in the matching. Therefore, we look for a maximum-weight matching with as many edges as possible.

We note that any such solution cannot contain negative-weighted edges, since removing them would increase the total weight. Note also that removing 0-weighted edges still leaves a maximum-weight matching. Therefore, the maximum-weight matching with the maximal number of edges is comprised of a maximum-weight matching with as many 0-weighted edges added to it as possible. By adding a small positive weight to the 0-weighted edges before running the maximum-weight matching algorithm, we can ensure that as many such edges

as possible will be added to the solution, since adding them will only increase the total weight. This weight must be the same for all edges, since we do not prefer one 0-weighted edge over the other. In addition, the weight must be small enough to ensure that the new solution contains some original maximum-weight matching as a subset. For example, every maximum-weight matching for the graph of the sequence:

$$(1, A)(1, B)(2, C)(3, D)(2, C)$$

must contain the edge (A, B), since it is the only edge with positive weight.

If the weight added to 0-weighted edges is 1, as shown in Figure 2(right), then a maximum-weight matching on this graph is either the edges $(A, D), (B, C)$ or $(A, C), (B, D)$, neither of which contain the edge (A, B). However, if we give all the 0-weighted edges a weight $< \frac{1}{|E|}$ (for example $\frac{1}{|V|^2}$), the total weight of all the 0-weighted edges is smaller than 1. Since all the positive-weighted edges weigh at least 1, any combination of originally 0-weighted edges weighs less than the weight of those edges. Therefore, the solution found by the algorithm must contain an original maximum-weight matching as a subset, otherwise, such a matching will produce a higher total weight, contradicting the algorithm's optimality in finding a maximum-weight matching. This implies the next theorem:

▶ **Theorem 3.** *In the CC model, there is a polynomial algorithm for finding a 2-block placement, with the minimal number of block RMRs for a given access sequence, while minimizing the total number of blocks used.*

3.2 Hardness Proof for B-Block placement, $B > 2$

We now prove a hardness result for B-block placement with $B > 2$, by showing a polynomial-time reduction from the graph partitioning problem, known to be *NP-complete*, even for a fixed $K \geq 3$ and even if all vertex and edge weights are 1 [17].

▶ **Definition 4** (Graph Partitioning)**.**

Input: Undirected graph $G = (V, E)$, weights $w(v) \in Z^+$ for each vertex $v \in V$ and $l(e) \in Z^+$ for each edge $e \in E$, positive integers K and J.

Question: Is there a partition of V into disjoint sets $V_1, \ldots, V_m$, such that $\Sigma_{v \in V_i} w(v) \leq K$ for $1 \leq i \leq m$ and such that if $E' \subseteq E$ is the set of edges that have their two endpoints in two different sets V_i then $\Sigma_{e \in E'} l(e) \leq J$?

We redefine B-block placement as a decision problem.

▶ **Definition 5** (B-Block Placement Decision)**.**

Input: Two positive integers B and R, a set of n processes $P = \{p_1, \ldots, p_n\}$, a set of memory objects O, a sequence of accesses $\pi = (p_1, o_1), \ldots, (p_m, o_m)$ such that for every i, $1 \leq i \leq m$, $p_i \in P$ and $o_i \in O$.

Question: Is there a partition of the objects in O into disjoint sets $\tilde{O} = \{O_1, \ldots, O_k\}$ such that $|O_i| \leq B$ and $\#brmr^{\mathrm{CC}}(\pi, \tilde{O}) < R$?

Given that it is easy to calculate the number of RMRs, and it is an upper bound on the number of block RMRs, a polynomial time algorithm for the decision problem can be used in conjunction with a binary search to solve the optimization problem.[2] Therefore, if we prove that the decision problem is NP-hard, the minimization problem is also NP-hard.

[2] That is, what is the minimal number of block RMRs for the sequence π. An optimal placement of objects to blocks is not found in this way.

We will use the input graph for which partitioning must be found as the underlying data structure that two processes access simultaneously. Each process performs a traversal and accesses the edges in DFS order. Processes access the edges in round-robin order, each accessing the endpoints of an edge one after the other, before control is passed to the next process. This access pattern ensures that putting the two endpoints of an edge in the same block results in fewer block RMRs than if they are in different blocks. Therefore, given a placement with fewer block RMRs, the partition into blocks induces a partition of the graph into disjoint sets of vertexes, which gives a good solution to the graph partitioning problem. Conversely, we prove that if there is no such placement, then there is no valid solution to the graph partitioning problem.

In more detail, the input is an undirected graph $G = (V, E)$, in which all edge and vertex weights are 1, and positive integers $K \geq 3$ and J.

We take two processes p_1 and p_2 and an object o for each vertex $v \in V$. For each edge $e \in E, e = (v_i, v_j)$ and process p, we define a subsequence

$$\pi_{(p,e)} = (p, \text{read}, o_i)(p, \text{write}, o_i), (p, \text{read}, o_j)(p, \text{write}, o_j).$$

Let $\pi_e = \pi_{(p_1,e)}, \pi_{(p_2,e)}$. Consider the following traversal in DFS order of the edges of the graph: The traversal starts at an arbitrary vertex. When the traversal reaches a node, it either immediately retreats through the same edge, if the node was already visited, or it continues to visit the node's neighbors, and then retreats through the same edge. Therefore, each edge is visited twice during the traversal, and neighboring vertexes are accessed one after the other. Let $e_{i_1}, \ldots, e_{i_{2|E|}}$ be the sequence of edges in this traversal. We define $\pi = \odot_{1 \leq j \leq 2|E|} \pi_{e_{i_j}}$. For each process p, the sequence of accesses is $\odot_{1 \leq j \leq 2|E|} \pi_{(p,e_{i_j})}$, which is a traversal due to the choice of the order of the edges. The length of the access sequence π is in $O(|E)|$, since it is a concatenation of $2|E|$ constant size sequences, and therefore, the reduction is polynomial in the size of the input. Let $B = K$ and $R = 4(|E| + J)$.

▶ **Lemma 6.** *There is a B-block placement of O for sequence π with R or fewer block RMRs if and only if there is a partitioning of the graph G under the K and J weight sum constraints.*

Proof. Given a graph and a partition $V_1, \ldots, V_m$ of the vertexes such that $\Sigma_{v \in V_i} w(v) \leq K$, we define a B-block placement for $B = K$ where O_i contains all the objects that correspond to the vertexes in V_i. Since all weights are 1, the number of vertexes in V_i is at most K, and so is the number of objects in O_i. Similarly, a B-block placement of the objects induces a partition of the corresponding vertexes that satisfies the $K = B$ constraint on the graph.

Let E' be the set of edges with endpoints in different sets V_i. We argue that the number of block RMRs for the sequence π is $4(|E| + |E'|)$. After each subsequence π_e, each block is either still in the main memory or in the cache of process p_2, since p_2 modifies each block after p_1 modifies it. Therefore, if $e = (v_i, v_j)$ and o_i and o_j, the corresponding objects, are in the same block, then the number of block RMRs for π_e is exactly 2, since each process must bring the block into its cache (including the first process) in order to access o_i, and then proceed to access o_j. Otherwise, if o_i and o_j are in different blocks, the number of block RMRs for π_e is exactly 4, since each process must access both blocks in turn. Therefore, since every edge is traversed twice, the total number of block RMRs is $4|E \backslash E'| + 2 \cdot 4|E'| = 4(|E| + |E'|)$.

It remains to show that a solution to the B-block placement problem implies a solution to the graph partitioning problem. If there is no B-block placement for $B = K$ and the sequence π with $R = 4(|E| + J)$ or fewer block RMRs, then for every placement and corresponding graph partitioning, the number of block RMRs is $4(|E| + |E'|) > R = 4(|E| + J)$ and therefore, $|E'| > J$. Hence, there is no graph partitioning such that the size of every group is K or less and the number of edges between different groups is J or less.

Table 1 Remote memory references in variants of the DSM model.

	Reads	Writes	Invalidation
DSM - With Coherence	Free as long as data in local memory is valid. Incurs an RMR, otherwise.	Free if data in local memory is valid, and incurs an RMR, otherwise. Causes invalidation of object copies in other local memories.	Negligible
DSM - No Coherence	Reads by object creator are free. Other reads incur an RMR.	Writes to an object by a non creator process incur an RMR. Writes by the owner are free.	Not supported

Conversely, if there is a B-block placement for $B = K$ and the sequence π, with fewer than $R = 4(|E| + J)$ RMRs, then $4(|E| + |E'|) \leq R = 4(|E| + J)$, and $|E'| \leq J$. Therefore, the corresponding partitioning of G satisfies the constraints on K and J. ◀

This proves the next theorem:

▶ **Theorem 7.** *In the CC model, the B-block placement decision problem, for $B \geq 3$, is NP-hard.*

4 Block RMRs in the DSM Model

In the DSM model, an access to another process' local memory incurs an RMR, depending on the specific characteristics of the implementation. Some DSM systems incorporate some form of *cache coherence*, i.e., a mechanism that ensures processes only read or write valid data, and not data that was already overwritten by another process. We consider two variants of the DSM model, depending on whether invalidating an object in a remote memory is possible, and what is the cost of doing so (see Table 1).[3]

Each object o is created at a *creator* process, denoted *creator*(o), which is the owner of the local memory in which o originally resides. The *creator*(o) process is chosen before the first access to o. An object is valid when created, and therefore, accesses to the object are free as long as they are just by the creator process. In all variants of the DSM model, we may choose *creator*(o) according to whatever information we have on the access sequence.

If coherence is supported and invalidation cost is negligible (for example, see [19]), then each read or write to an object not in the local memory causes the object to be copied to the local memory, incurring an RMR. If the action is a write then other copies of the object are invalidated, and the cost of doing so is negligible. Given an access sequence $\pi = (p_{i_1}, a_1, o_{j_1}), \ldots, (p_{i_m}, a_m, o_{j_m})$, we formally define:

$$\begin{aligned}\#rmr^{\mathrm{DSM-CC}}(\pi) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | ((p_{i_h} \neq \mathit{creator}(o_{j_h}))& \\ \text{and } (\forall k, 1 \leq k < h, (p_{i_h} = p_{i_k}) \implies (o_{j_h} \neq o_{j_k})))& \\ \text{or } (\exists k, 1 \leq k < h, (o_{j_k} = o_{j_h}, a_k = \text{write, and } p_{i_k} \neq p_{i_h})& \\ \text{and } (\forall \ell, k < \ell < i, (p_{i_h} = p_{i_\ell}) \implies (o_{j_h} \neq o_{j_\ell})))\}| \,.&\end{aligned}$$

If coherence and invalidation are not supported, then every access, whether a write or a read, to an object by a process other than its creator, incurs an RMR. Accesses to an object

[3] In the full version of the paper we also consider a third variant, in which invalidation is supported, but its cost is similar to an RMR.

Table 2 *Block* remote memory references in variants of the DSM model.

	Reads	Writes	Invalidation
DSM - With Coherence	Free as long as data in local memory is valid. Incurs an RMR, otherwise.	Free if data in local memory is valid, and incurs an RMR, otherwise. Causes invalidation of other copies in other local memories.	Negligible
DSM - No Coherence	Reads by the object's creator process are free. Other reads incur an RMR.	Writes by owner process are free. Other writes incur an RMR.	Not supported

by its creator are free. That is:

$$\#rmr^{\mathrm{DSM-No-CC}}(\pi) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | p_{i_h} \neq \mathit{creator}(o_{j_h})\}| \ .$$

To define block RMRs in these variants of the DSM model, consider a B-block placement $\tilde{O} = \{O_1, \ldots, O_\ell\}$. Each block O_i is associated with a creator process $\mathit{creator}(O_i)$, which creates all objects in O_i. The RMRs incurred for reads and writes is shown in Table 2, and defined next.

If coherence is supported with negligible invalidation cost, then each access to an object not in the local memory incurs an RMR; it causes a copy of the accessed block to appear in the local memory. If the access is a write then all other copies of the block are invalidated. Accesses to objects whose blocks are in the local memory are free. Formally:

$$\begin{aligned}\#brmr^{\mathrm{DSM-CC}}(\pi, \tilde{O}) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | (o_{j_h} \in O_l) \text{ and } \\ (((p_{i_h} \neq \mathit{creator}(O_l)) \text{ and } \\ (\forall k, 1 \leq k < h, (p_{i_h} = p_{i_k}) \implies (o_{j_k} \notin O_l))) \text{ or } \\ (\exists k, 1 \leq k < h, (o_{j_k} \in O_l, a_k = \text{write, and } p_{i_k} \neq p_{i_h}) \\ \text{and } (\forall \ell, k < \ell < h, (p_{i_h} = p_{i_\ell}) \implies (o_{j_\ell} \notin O_l))))\}| \ .\end{aligned}$$

When coherence is not supported, each access to a block by a process other than its creator incurs an RMR; accesses by the creator are free. Formally:

$$\#brmr^{\mathrm{DSM-no-CC}}(\pi) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | o_{j_h} \in O_\ell \text{ and } p_{i_h} \neq \mathit{creator}(O_\ell)\}| \ .$$

The DSM model with coherence is similar to the CC model, except for the first access to every block: If the first access to an object o is done by process $p \neq \mathit{creator}(o)$, then the step incurs an RMR; otherwise, no RMR is incurred. Therefore the number of RMRs depends on the choice of $\mathit{creator}(o)$ for these accesses, while other accesses are not affected.

We explain how the results for the CC model can be extended to this model. The polynomial algorithm for blocks with $B = 2$ is adapted by changing the edge weights to account for the objects being created in the process' local memory instead of the main memory. The NP-hardness result with $B > 2$ can also be adapted by a slight change in the access sequence and the way edge weights are calculated (see Section B).

In the DSM model without coherence, the number of RMRs is minimized when each object is created at the process that accesses it the most. Furthermore, it can be shown that the minimal number of RMRs does not depend on the partitioning into blocks, just on the choice of creator processes. This means that a simple greedy algorithm can be used to pick the creator process for each object.

5 Conclusions and Future Research Directions

This paper introduces a framework for studying the cost of accessing remote memory (whether shared memory or data stored at another process), which takes into account the fact that shared objects can be placed and moved together in larger blocks. We introduce a formal complexity measure, called *block RMR*s, for both the CC and the DSM models. Our main result shows that it is NP-hard to place objects into blocks in a way that minimizes the number of block RMRs, even for a fixed access sequence. The result holds for both the CC model and the DSM model with coherence and negligible invalidation cost, when a block can contain three objects or more.

In the common situation, however, the access sequence is not known in advance. Instead, we know it is from a *family* of sequences, typically, those generated by a particular concurrent algorithm, for example, interleavings of (partial) traversals or searches by a subset of the processes. It would be interesting to find block placement methodologies for such families, in a way that exploits the benefits of moving several objects together. Taking this a step further, it is interesting to look at probabilistic models where the access sequences are chosen from a family of sequences with a known distribution. The goal is to choose a B-block placement such that the *expected* number of block RMRs is minimized. In the DSM model without coherence, these problems may have simple solutions in the form of choosing the object's creator to be the process that is expected to access it the most.

It would also be interesting to study the effects of bounding the local memory size, so it can hold a bounded number of blocks, and limiting the cache associativity. It is likely that in general, the problem is *NP-hard* due to our results as well as [20, 18].

Acknowledgements. We would like to thank Youla Fatourou, Danny Hendler, Erez Petrank and the referees for helpful comments and useful discussions.

References

1 Yehuda Afek, Dave Dice, and Adam Morrison. Cache index-aware memory allocation. *SIGPLAN Not.*, 46(11):55–64, 2011. doi:10.1145/2076022.1993486.

2 Dan Alistarh, James Aspnes, Seth Gilbert, and Rachid Guerraoui. The complexity of renaming. In *IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 718–727, 2011.

3 James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distrib. Comput.*, 16(2-3):75–110, 2003. doi:10.1007/s00446-003-0088-6.

4 Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC, pages 268–276, 2002. doi:10.1145/509907.509950.

5 Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 355–366, 2011. doi:10.1145/1989493.1989553.

6 William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, 1993. URL: http://dl.acm.org/citation.cfm?id=1295480.1295483.

7 Gerth Stølting Brodal, Rolf Fagerberg, and Gabriel Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Proceedings of the 32nd International Conference on Automata, Languages and Programming*, ICALP, pages 576–588, 2005. doi:10.1007/11523468_47.

8 Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. *SIGPLAN Not.*, 33(11):139–149, 1998. doi:10.1145/291006.291036.

9 Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, PLDI, pages 1–12, 1999.

10 Rezaul Alam Chowdhury, Vijaya Ramachandran, Francesco Silvestri, and Brandon Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 73(7):911–925, 2013.

11 Susan J. Eggers and Tor E. Jeremiassen. Eliminating false sharing. In *Proceedings of the International Conference on Parallel Processing, ICPP. Volume I: Architecture/Hardware*, pages 377–381, 1991.

12 Rolf Fagerberg, Anna Pagh, and Rasmus Pagh. External string sorting: Faster and cache-oblivious. In *Proceedings of the 23rd Annual Conference on Theoretical Aspects of Computer Science*, STACS, pages 68–79, 2006. doi:10.1007/11672142_4.

13 Arash Farzan, Paolo Ferragina, Gianni Franceschini, and J. Ian Munro. Cache-oblivious comparison-based algorithms on multisets. In *Proceedings of the 13th Annual European Conference on Algorithms*, ESA, pages 305–316, 2005. doi:10.1007/11561071_29.

14 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, 1999.

15 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

16 Wojciech Golab, Danny Hendler, and Philipp Woelfel. An $O(1)$ RMRs leader election algorithm. *SIAM J. Comput.*, 39(7):2726–2760, 2010.

17 Laurent Hyafil and Ronald L. Rivest. Graph partitioning and constructing optimal decision trees are polynomial complete problems. Technical Report Rapport de Recherche no. 33, IRIA – Laboratoire de Recherche en Informatique et Automatique, October 1973.

18 Rahman Lavaee. The hardness of data packing. In *Proceedings of the 43rd Annual ACM Symposium on Principles of Programming Languages*, POPL, pages 232–242, 2016. doi:10.1145/2837614.2837669.

19 Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, aug 1991. doi:10.1109/2.84877.

20 Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. *SIGPLAN Not.*, 37(1):101–112, 2002. doi:10.1145/565816.503283.

21 M.D. Plummer and L. Lovász. *Matching Theory*. North-Holland Mathematics Studies. Elsevier Science, 1986. URL: https://books.google.co.il/books?id=mycZP-J344wC.

22 Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massacusetts Institute if Technology, Cambridge, MA, 7 1999.

23 Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

24 P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Symposium on Foundations of Computer Science*, pages 75–84, 1975.

25 Jae-Heon Yang and James H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995. doi:10.1007/BF01784242.

A B-Block Placement for Objects with Varying Sizes

We prove that if objects have varying sizes, then the B-block placement problem is *NP-hard* even for a simple traversal on a tree. Assume that each object o_i has a size $a_i \in Z^+$, $a_i \leq B$, and each object is placed in a single block (and not across more than one block). A B-block placement is a partition of the objects in O into disjoint sets $\tilde{O} = \{O_1, \ldots, O_n\}$ such that, for every $1 \leq j < n$, $\Sigma_{o_i \in O_j} a_i \leq B$.

For both the CC and the DSM models, there is a reduction from the *bin packing* problem, a well-known NP-complete problem [15].

▶ **Definition 8** (Bin Packing).
Input: An integer R (the bin size), and ℓ items with sizes $a_1, \ldots, a_\ell$.
Question: What is the minimum number M such that there is a partition of the ℓ items into disjoint sets $S_1, \ldots, S_M$ such that $\Sigma_{i \in S_j} a_i \leq R$ for every j, $1 \leq j \leq M$?

Fix R to be the block size B, the number of objects to be ℓ and the size of an object o_i be a_i. We define a tree $T = (V, E)$ whose vertexes are v_i for each object o_i, and whose edges are $E = \{(v_i, v_{i+1}) | 1 \leq i < \ell\}$. For processes $p_1, p_2 \in P$, consider the access sequence $\pi = (p_1, \text{write}, o_1), \ldots, (p_1, \text{write}, o_\ell), (p_2, \text{write}, o_1), \ldots, (p_2, \text{write}, o_\ell)$, in which p_1 and p_2 write to each object once.

An optimal algorithm for B-block placement packs the objects into the smallest number of blocks possible: Since the size of the local memory is unlimited, any block read into the local memory remains there until it is required by a different process in the CC and DSM with coherence models.

Therefore, the number of block RMRs is equal to twice the number of blocks in which the objects reside in the CC model and DSM with coherence models.

Thus, an algorithm that finds the optimal solution to the B-block placement problem also finds a partitioning of objects of sizes $a_1, \ldots, a_\ell$ into blocks of size $B = R$, which minimizes the number of blocks used. This yields an optimal solution to the bin packing problem. Since bin packing is NP-complete [15], so is B-blocking for objects of varying sizes.

B NP-Hardness for DSM with Coherence and $B > 2$

We explain how to adapt the NP-hardness result for $B \geq 3$. Given the parameters to the graph partitioning problem, the parameters to the B-Block mapping are very similar: The input is an undirected graph $G = (V, E)$, weights $w(v) \in Z^+$ for each vertex $v \in V$ and $l(e) \in Z^+$ for each edge $e \in E$, and positive integers K and J. We assume that all weights are 1, and $K \geq 3$.

The access sequence is somewhat different: We take two processes p_1 and p_2, and an object o for each vertex $v \in V$. For each edge $e \in E, e = (v_i, v_j)$ and process p, we define a subsequence

$$\pi_{(p,e)} = (p, \text{read}, o_i)(p, \text{write}, o_i), (p, \text{read}, o_j)(p, \text{write}, o_j)).$$

Let $\pi_e = (\pi_{(p_1,e)}, \pi_{(p_2,e)})^{|E|}$. We define $\pi = \odot_{e \in E} \pi_e$. The length of π is $O(|E|^2)$ and therefore, the reduction is in polynomial time. Let $B = K$ and $R = 4|E|(|E| + J) - |V|$.

Given the part of the access sequence corresponding to an edge $e = (v_i, v_j)$,

$$\pi_e = ((p_1, \text{write}, o_i), (p_1, \text{write}, o_j)), (p_2, \text{write}, o_i), (p_2, \text{write}, o_j)^{|E|},$$

the number of block RMRs is as follows: If both endpoints of e are in the same block, then every access by p_1, except perhaps the first one, incurs a block RMR. This is because at

the end of $\pi_{(p_2,e)} = (p_2, \text{write}, o_i), (p_2, \text{write}, o_j)$, the block is in p_2's local memory. The total number of block RMRs is $2|E|$, from which we reduce 1 if this is the first access to the block containing o_i and o_j.

On the other hand, if the endpoints of e are in different blocks, then every access by either p_1 or p_2 must move a block into its local memory, except perhaps the first two accesses of p_1. The total number of block RMRs is $4|E|$, minus 1 or 2, depending on whether or not this sequence accessed blocks for the first time.

Since at the end of each subsequence π_e all blocks that were already accessed are in the local memory of process p_2, if $|E'|$ is the number of edges with endpoints in different blocks, the total number of RMRs is at most

$$2|E| \cdot 2|E \backslash E'| + 4|E| \cdot 2|E'| = 4|E|(|E| + |E'|)$$

and at least

$$2|E| \cdot 2|E \backslash E'| + 4|E| \cdot 2|E'| - |V| = 4|E|(|E| + |E'|) - |V|,$$

where $|V|$ is the maximum possible number of blocks.

If there is no B-block ent for $B = K$ and the sequence π with $R = 4|E|(|E| + J) - |V|$ or fewer block RMRs exists, then for every placement and corresponding graph partitioning the number of block RMRs is at least

$$4|E|(|E| + |E'|) - |V| > R = 4|E|(|E| + J) - |V|$$

and therefore $|E'| > J$. Hence, there is no graph partitioning such that the size of every group is K or less and the number of edges between different groups is J or less.

Conversely, if there is a B-block placement for $B = K$ with S blocks, and the sequence π, induces fewer than $R = 4|E|(|E| + J)$ block RMRs, then

$$4|E|(|E| + |E'|) - S \leq R = 4|E|(|E| + J) - |V|,$$

and therefore

$$4|E| \cdot |E'| \leq 4|E| \cdot J - (|V| - S).$$

Since $|V| - S > 0$, we have $|E'| \leq J$. Therefore, the corresponding partitioning of G holds under the constraints on K and J.

This proves the following theorem:

▶ **Theorem 9.** *In the DSM model with cache coherence and negligible invalidation, the B-block placement decision problem is NP-hard, for $B \geq 3$.*