On the realizability of hardware microthreading. Revisiting the general-purpose processor interface: consequences and challenges

Poss, R.C.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 11

Placement and platform partitioning
—Highlight on specifics, opus 4

The way forward seems to be to provide reasonable operations, and make costs somewhat transparent.

Andrei Matei

Abstract

When adding new architectural features that do not have corresponding abstractions in the operating software stack, it is sometimes necessary to create “holes” in the lower level interfaces so that the new features can be exploited directly. However the constructs to drive these new features should still be designed to abstract away uninteresting characteristics of the implementation, and expose instead its general principles. We illustrate this with the example of work placement. As explained by [PHA10] and previous work, placement optimization cannot be fully automated in hardware and the efficient execution of fine-grained concurrent workloads over large many-core chips will require dedicated interfaces to control the mapping of computations to specific cores in operating software. However, no consensus yet exists as to what these interfaces should be. Upcoming many-core system stacks will thus need to explore placement issues by exploiting primitives that are not yet part of the usual abstraction tool box. In this chapter, we discuss how such primitives from the architecture proposed in part I can be exposed to the operating software providers.

Contents

11.1 Introduction .............................................................. 176
11.2 On-chip work placement: exposing control .......................... 177
11.3 On-chip work placement: how to exploit ............................ 182
Summary ................................................................. 186
11.1 Introduction

In chapter 5, we outlined the special relationship between platform providers and the providers of “first level” operating software, such as the C programming environment. We also highlighted the integration process for innovations at the level of the machine interface: first level operating software is first extended beyond the scope of established standards, then applications start to use non-standard features, and then features confirmed to be useful across a range of applications are retrospectively integrated in new revisions of the standards. To enable this process, there must exist “holes” in the abstraction layers to provide direct access to new features. An example of this is the \texttt{asm} construct in most C implementations, which allows C programs to use machine instructions not yet used by existing C language features.

Placement of computations and data on physical resources on a chip is another example of a platform feature still missing standard abstractions.

In contemporary systems, explicit placement is usually available through a system API with a large granularity: a program wishing to \textit{bind} a computation, usually a thread or a process, to a particular processor will request so through an interface to the operating system’s scheduler. Examples are Linux’ \texttt{sched_setaffinity} or FreeBSD’s \texttt{cpuset_setaffinity}. This usually requires special privileges and the overhead of a system call, and in practice is usually only performed once at program start-up to assign “worker” threads to specific processors, which then cooperatively negotiate the dispatch of tasks through data structures in shared memory. Even when considering the task dispatching process itself as a placement mechanism, overheads are also incurred by the cache coherency required to communicate the task data structures across cores in the memory network.

In contrast, the concurrency management primitives available with the architecture from part I allow programs to optionally and explicitly control the mapping of computations onto cores via a dedicated “delegation” NoC. The delegation messages are issued via special ISA instructions. This protocol enables the dispatch and synchronization of multi-core workloads within a few pipeline cycles; the direct use of the corresponding ISA instructions by programs is therefore orders of magnitude faster than the overhead of calling a system API, and also faster than sharing task data structures via a memory network.

However, direct use in programs also implies that the first level interface languages in the operating software must expose the hardware primitives as first-class language constructs that can be used at any point in algorithms. In the case of C, for example, this is a radically new requirement as C had never exposed knowledge about the physical location of sub-computations previously.

What should this language interface look like? One the one hand, we can predict that a first level interface can hide hardware-specific details such as the encoding of the special instructions. On the other hand, there is obviously not enough information at the lower level of the abstraction stack to devise high-level abstractions, for example “automatically perform the best work distribution for a given parallel computation.” This knowledge and the corresponding decision mechanisms are typically found at a higher level in the operating software stack, for example in the run-time system or the user of a “productivity” language like Chapel [CCZ07] or SAC [GS06].

This situation thus warrants creating an “abstraction hole,” to delegate to operating software providers at a higher abstraction level the responsibility to make placement decisions. In this chapter, we illustrate how to achieve this with the proposed architecture in our interface language SL, previously introduced in chapter 6.
11.2 On-chip work placement: exposing control

11.2.1 Architectural support

- The machine interface in chapter 4 was designed with two mechanisms for placement in mind. Historically, the first mechanism was a design-time partitioning of the many-core chip into \textit{statically named clusters}. Later on, a flexible \textit{dynamic partitioning protocol} was introduced as well, which we jointly designed and prototyped with the platform implementers. Both can be used in combination in a design, where larger, static cluster can be “internally” divided further at run-time into fine-grained sub-clusters. We detail the latter fine-grained placement protocol in Appendix E.

  In both cases, the “allocate” instruction that reserves bulk creation contexts accepts a “placement” operand. This is a resource address which triggers remote access, through the delegation NoC, to the named cluster if the address does not match the \textit{local address} of the thread issuing “allocate.” With static clusters, the address corresponds to the master core connected both to the delegation NoC and its neighbour cores in the cluster; with dynamic clusters, a \textit{virtual cluster address} is formed by the address of a first core, together with a cluster size and security capability, combined in a single integer value.

  Jointly to this addressing mechanism for new delegations, two machine instructions “getpid” and “getcid” are introduced. “getpid” produces the address of the cluster where the current thread was delegated to, also called \textit{default} or \textit{current} placement. “getcid” produces the address of the local core within its cluster. To avoid the overhead of extra instructions, “allocate” also recognizes the symbolic value 0 to implicitly reuse the default placement; and it also recognizes 1 to use the local core only, i.e. avoid multi-core execution altogether.

  With large static clusters, an operating software would likely not bind small computing activities to the static clusters at a high rate. In other words, the granularity of placement over large static clusters is coarse enough that this type of placement can be requested via system APIs. It is the existence of small clusters and the ability to dynamically partition clusters at run-time, within a few hardware cycles, which justifies the need for low-overhead, direct access via language primitives.

11.2.2 Language primitives

- We propose to transparently pass the run-time value of the “place” family configuration expression, which is the 2nd positional parameter to the \textit{“sl\_create”} construct (clauses I.5.8.1§3 and I.5.8.1§26), as the placement operand of the “allocate” operation. If the place expression is not provided (it is optional in the input syntax), the value 0 is used. With this translation in place, a program can specify, at the point of family creation:
  - either no place expression at all, which uses the default placement implicitly, or
  - one of the symbolic values 0 or 1, which designate the default placement or the local core explicitly, or
  - a fully specified target cluster address.

- Separately, we extend the library environment as follows:
  - we encapsulate the “getpid” and “getcid” machine instructions as C macros, defined in a SL library header, named “\textit{sl\_default\_placement()}” and “\textit{sl\_local\_processor\_address()},” respectively;
### Table 11.1: Core addressing operators in the SL library.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl_placement_size(C)</td>
<td>Number of cores in the cluster $C$.</td>
<td>2</td>
</tr>
<tr>
<td>sl_first_processor_address(C)</td>
<td>Address the first core in the cluster $C$.</td>
<td>2</td>
</tr>
<tr>
<td>at_core(P)</td>
<td>Address the relatively numbered core $P$ in the current cluster.</td>
<td>5/6</td>
</tr>
<tr>
<td>split_upper()</td>
<td>Split the current cluster and address the upper half.</td>
<td>5</td>
</tr>
<tr>
<td>split_lower()</td>
<td>Split the current cluster and address the lower half.</td>
<td>5</td>
</tr>
<tr>
<td>split_sibling()</td>
<td>Split the current cluster and address the sibling half from the perspective of the current core.</td>
<td>11</td>
</tr>
<tr>
<td>next_core()</td>
<td>Address the next core within the current cluster, with wraparound.</td>
<td>13</td>
</tr>
<tr>
<td>prev_core()</td>
<td>Address the previous core within the current cluster, with wraparound.</td>
<td>13</td>
</tr>
</tbody>
</table>

1 Number of processor cycles required with the Alpha or SPARC ISAs.

- we define a C typedef name “sl_place_t” wide enough to hold placement addresses.

With these primitives, a thread in the program can read and manipulate its own placement information, for example to compute thread distributions explicitly.

### 11.2.3 Integer arithmetic to compute placements

A characteristic feature of the proposed protocol in Appendix E is that it exposes the format of core addresses on the delegation NoC. In contrast to most contemporary systems where knowledge about the inter-processor fabric is entirely encapsulated in system services queried via privileged APIs, the proposed protocol allows programs to construct processor addresses “from scratch” without involving a system service.

To achieve this, the protocol advertises that a valid virtual cluster address is constructed by concatenating together a security capability\(^1\), the address of the first core in the cluster and the desired cluster size in a single machine word. This encoding creates the opportunity for programs to compute relative core addresses with the overhead of a few integer arithmetic instructions. For example, in our work we have captured the common patterns in table 11.1 as inlineable functions in a SL library header. Their implementation is given in Appendix E.3.

### 11.2.4 Example usage: inner product

The abstraction above allows programs to express relative constraints on placement of computations, useful to distribute reductions over variable numbers of cores.

\(^1\)The capability is optional and can be used by a coarse-grained system service to provide isolation between computing activities running on separate large-scale core clusters on the chip.
11.2. ON-CHIP WORK PLACEMENT: EXPOSING CONTROL

```
double kernel3(size_t n, double *X, double *Z)
{
    sl_create(, , n, , , innerk3,  
              sl_shfarg(double, Qr, 0.0),  
              sl_glarg(double*, , Z), sl_glarg(double*, , X));
    sl_sync();
    return sl_geta(Qr);
}

sl_def(innerk3,, sl_shfparm(double, Q),  
       sl_glparm(double*, Z), sl_glparm(double*, X))
{
    sl_index(i);
    // Q += Z[i] * X[i]
    sl_setp(Q, sl_getp(Z)[i] * sl_getp(X)[i] + sl_getp(Q));
}
```

Listing 11.1: Concurrent SL code for the Livermore loop 3 (inner product).

Figure 11.1: Parallel reduction using explicit placement, for problem size \( n = 16 \) and a virtual core cluster containing 4 cores.

The boxes indicate the generated logical thread indexes.

As an example, consider the naive implementation of the vector-vector product in listing 11.1. This defines a C function `kernel3` which computes the inner product of the \( n \)-sized vectors \( X \) and \( Z \) using a single family of threads. Because there is a carried dependency between all threads, this implementation cannot benefit from multi-core execution: the first logical thread on every core but the first will wait for the last output from the last logical thread on the previous core (cf. also side note E.2).

Instead, we can modify this program source using the new placement primitives as indicated in listing 11.2. For clarity, we omit the extra logic necessary to handle array sizes that are not a multiple of the number of cores. The top-level function `kernel3` deploys two levels of reduction, where all threads at the first level run on the same core, and each
double kernel3 (size_t n, double *X, double *Z) {
    sl_place_t pl = sl_default_placement ();
    long ncores = sl_placement_size(pl);
    long span = n / ncores;

    pl = sl_first_processor_address(pl) | 1 /* force size 1*/;

    sl_create( , /* local placement: */ 1,
               0, ncores, 1, , , reductionk3 ,
               sl_shfarg(double, Qr, 0.0),
               sl_glarg(double*, , Z), sl_glarg(double*, , X),
               sl_glarg(long, , span), sl_glarg(sl_place_t, , pl));
    sl_sync();
    return sl_geta(Qr);
}

sl_def(reductionk3 , , sl_shfparm(double, Q),
      sl_glparm(double*, Z), sl_glparm(double*, X),
      sl_glparm(long, span), sl_glparm(sl_place_t, pl))
{
    sl_index(cpuidx);

    long lower = sl_getp(span) * cpuidx;
    long upper = lower + sl_getp(span);

    sl_create(, sl_getp(pl) + cpuidx * 2,
               lower, upper, 1, , , innerk3 ,
               sl_shfarg(double, Qr, 0.0),
               sl_glarg(double*, , sl_getp(Z)), sl_glarg(double*, , sl_getp(X)));
    sl_sync();
    sl_setp(Q, sl_geta(Qr) + sl_getp(Q));
}

sl_enddef

Listing 11.2: Concurrent SL code for the Livermore loop 3, optimized.
family created at the second level runs on a different core within the cluster. The resulting placement at run-time is illustrated in fig. 11.1.

The effect of this strategy on performance is given in fig. 11.2. To obtain these results, we ran the kernel on different virtual cluster sizes, with the input \( n = 64000 \). As described further in section 13.2 and table 13.1, the microthreaded cores run at 1GHz and the baseline is a sequential program running on 1 core of an Intel P8600 processor at 2.4GHz.

As can be observed, the parallel reduction is advantageous and offers a performance improvement over the baseline past 32 core. As described later in chapter 13, microthreaded cores are expected to be significantly smaller in area than cores in the baseline architecture. The performance per unit of area on chip therefore compares advantageously. Moreover, the proposed design does not use speculation, which suggests that the performance per watt may also be lower. At any rate, multi-core scalability is achieved without a priori knowledge about the number of cores while writing the program code.

### 11.2.5 Relationship with memory consistency

Regardless of how heterogeneous a chip is, and thus how much information on-line resource managers must maintain (cf. section 2.4), a general-purpose, multi-core chip must expose the accessibility of memory from processors as a basic property of the system. Indeed, the topology of the memory network impacts the implementation of operating software at all layers of the abstraction stack, as we outlined in section 3.4.1 and chapter 7.

This is mandatory even for so-called “shared memory” chips, because as we explained previously in chapter 7, upcoming and future chip design may feature weak consistency models as an opportunity to simplify and optimize cache coherency protocols. As multiple consistency granularities start to appear, the distributed nature of many-core chips becomes unavoidable to the operating software providers [BPS*09].

To characterize these aspects in a language abstract machine, we introduced the notion of Implicit Communication Domain (ICD) and Consistency Domain (CD) in section 7.3. These describe subsets of a platform where implicit communication between memory store and load operations is possible between asynchronous threads. Communication is fully
implicit within a CD, whereas it requires some semi-explicit coherency actions in a ICD. Across ICDs, the memory address space is not shared. For example, a Symmetric Multi-Processor (SMP) system with full cache coherency would form a single CD and ICD; different nodes in a distributed cluster would form distinct ICDs; and nodes in a distributed cache system\textsuperscript{2} may form a single ICD but distinct CDs.

The placement operators we introduced in the previous sections are tightly related to the memory topology. Indeed, a program code which assumes a single CD cannot safely spread work across a virtual core cluster spanning multiple distinct CDs. For example, on the reference implementation of the proposed architecture, coherency between L1 caches sharing a single L2 cache uses a fully coherent snoopy bus, whereas coherency between L2 caches is lazy: writes may not be propagated to other L2 caches until a thread terminates. This implies that a group of cores sharing the same L2 cache form one CD, but separate L2 caches form separate CDs. A program wishing to use implicit memory-based communication between separate threads, without resorting to explicit memory barriers, must thus ensure that the threads are assigned to cores sharing a single L2 cache. How should this knowledge be negotiated between the hardware implementation and the operating software?

As a first step in this direction, we propose the following two primitives:

- \texttt{local\_consistent\_cluster(C)}: for a given cluster address \(C\), produce the address of the largest cluster that encompasses \(C\) but provides a single CD. For example, on the proposed architecture this would evaluate to an address for the local L2 core group, whereas on a conventional SMP this would evaluate to an address for the entire local system.
- \texttt{local\_implicit\_cluster(C)}: for a given cluster address \(C\), produce the address of the largest cluster that encompasses \(C\) but provides a single ICD. For example, on the proposed architecture this would evaluate to an address for the entire chip, whereas on a distributed chip with separate local memories for each core this would evaluate to an address for the local core only.

We propose that these primitives be used in the run-time system of a higher-level programming language, to select at run-time which implementation to run (whether suitable for a single CD or multiple) depending on the actual resources available locally to a computation.

### 11.3 On-chip work placement: how to exploit

The primitives provided so far allow a program to control thread placement within a virtual cluster at run-time, including partitioning an existing virtual cluster into physically separate sub-clusters. With the proposed encoding of addressing information, the corresponding operations can be carried out within a few pipeline cycles. This suggests a high level of dynamism, where placement can be decided at a fine grain in algorithms at run-time.

#### 11.3.1 Related work

We have found similar considerations about on-chip placement in the three high-productivity languages for large parallel systems produced during the DARPA project \textit{High Productivity}

11.3. ON-CHIP WORK PLACEMENT: HOW TO EXPLOIT

Computing Systems [DGH+08]: Sun’s Fortress [ACH+08], Cray’s Chapel [CCZ07], and IBM’s X10 [CGS+05]. We describe them below.

Also, we supervised an application of our proposal for cross-domain consistency to distributed clusters, which we describe as related work in section 11.3.1.3.

11.3.1.1 Relationship with Chapel and Fortress

Virtual clusters in our setting correspond closely to the concept of locales in Chapel and regions in Fortress:

In Chapel, we use the term locale to refer to the unit of a parallel architecture that is capable of performing computation and has uniform access to the machine’s memory. For example, on a cluster architecture, each node and its associated local memory would be considered a locale. Chapel supports a locale type and provides every program with a built-in array of locales to represent the portion of the machine on which the program is executing. [...] Programmers may reshape or partition this array of locales in order to logically represent the locale set as their algorithm prefers. Locales are used for specifying the mapping of Chapel data and computation to the physical machine [...]. [CCZ07, p. 13]

A locale is a portion of the target parallel architecture that has processing and storage capabilities. Chapel implementations should typically define locales for a target architecture such that tasks running within a locale have roughly uniform access to values stored in the locale’s local memory and longer latencies for accessing the memories of other locales. As an example, a cluster of multi-core nodes or SMPs would typically define each node to be a locale. In contrast a pure shared memory machine would be defined as a single locale. [Cra11, Sect. 26.1]

Every thread (either explicit or implicit) and every object in Fortress, and every element of a Fortress array (the physical storage for that array element), has an associated region. The Fortress libraries provide a function region which returns the region in which a given object resides. Regions abstractly describe the structure of the machine on which a Fortress program is running. They are organized hierarchically to form a tree, the region hierarchy, reflecting in an abstract way the degree of locality which those regions share. The distinguished region Global represents the root of the region hierarchy. The different levels of this tree reflect underlying machine structure, such as execution engines within a CPU, memory shared by a group of cores, or resources distributed across the entire machine. [...] Objects which reside in regions near the leaves of the tree are local entities; those which reside at higher levels of the region tree are logically spread out. [ACH+08, Sect. 21.1]

Like with locale and region variables in Chapel and Fortress, our approach allows programs to manipulate clusters of cores using an abstract handle (a virtual cluster address in our setting). We also suggest in our reference implementation a form of data locality between cores within a locale (cf. fig. 3.9). Furthermore, both Chapel and Fortress provide explicit placement of computations on named locales/regions (with the keyword on in Chapel, at in Fortress) and relative addressing via the special name “here,” which corresponds to sl_default_placement() in our setting.
CHAPTER 11. PLACEMENT AND PLATFORM PARTITIONING

Neither Chapel nor Fortress formalize a consistency model for concurrent accesses to memory, like we do. Moreover, our questions about the acquisition of additional resources and how to determine appropriate cluster sizes, mentioned above, do not have an answer in Chapel nor Fortress either.

Despite these similarities, both Chapel and Fortress are considerably more mature with regards to data locality. In particular, both languages define primitives that allow programs to assign variables, especially arrays, to named virtual core clusters. When these constructs are used, any computation using these variables is implicitly distributed across virtual clusters so that each unit of computation becomes local to its operands.

In our setting, this feature is not available natively: variables are either defined in the global scope, in which case they are shared by all clusters visible to the program, or in the local scope, in which case they are only conceptually local to the thread where they are defined and all other threads running on the same core.

In practice, memory locality is obtained automatically by any thread on its local core in our reference implementation, because as seen above this uses a distributed cache protocol which migrates cache lines automatically to the point of last use. However the work on Chapel and Fortress suggests that explicit control of memory placement may be desirable in the language semantics. Moreover, explicit data placement could provide a reliable handle on allocation if our architecture were extended to support multiple distributed memories on chip. We consider this further in section 11.3.1.3.

11.3.1.2 Relationship with X10

Places in IBM’s X10 are similar to Chapel’s locales and Fortress’ regions:

A place is a collection of resident (non-migrating) mutable data objects and the activities that operate on the data. Every X10 activity runs in a place; the activity may obtain a reference to this place by evaluating the constant here. The set of places are ordered and the methods next() and prev() may be used to cycle through them.

X10 0.41 takes the conservative decision that the number of places is fixed at the time an X10 program is launched. Thus there is no construct to create a new place. This is consistent with current programming models, such as MPI, UPC, and OpenMP, that require the number of processes to be specified when an application is launched. We may revisit this design decision in future versions of the language as we gain more experience with adaptive computations which may naturally require a hierarchical, dynamically varying notion of places.

Places are virtual — the mapping of places to physical locations in a NUCC system is performed by a deployment step [...] that is separate from the X10 program. Though objects and activities do not migrate across places in an X10 program, an X10 deployment is free to migrate places across physical locations based on affinity and load balance considerations. While an activity executes at the same place throughout its lifetime, it may dynamically spawn activities in remote places [...] [CGS+05, Sect. 3.1]

Like Chapel and Fortress, X10 allows both explicit assignment of computations and data to processing resources, and the local sub-partitioning of resources. With regards to
consistency, X10 defines places as sequentially consistent, which in our model would translate as a constraint that core clusters do not span multiple consistency domains.

The new concept with X10 is the run-time mapping of virtual places (visible by programs) and physical resources (invisible by programs). This reduces the amount of knowledge about the environment provided to programmers but creates the opportunity to optimize load balance at run-time outside of the program specification.

It is conceptually possible to introduce this feature in our setting by indirecting every bulk creation through a system service in charge of placement, which would hold the mapping between place identifiers (manipulated by the program) and physical cluster addresses. The trade-off of this approach is a mandatory overhead at each bulk creation, which would defeat the benefit of architectural support for the effective parallelisation of small workloads. This observation highlights the fact that the primary benefit of hardware support for concurrency, that is low overheads for concurrency creation and synchronization, can only be reaped in applications if the program has direct access to the machine interface.

11.3.1.3 The Hydra run-time

Under our supervision, the author of [Mat10] has thoroughly investigated the use of the proposed consistency semantics across clusters of SMPs. In this work, more SL language primitives have been designed which integrate consistency semantics, placement and object (data) declarations.

This work extends the vision that data should be migrated automatically to the point where it is used, to provide locality implicitly. In contrast to the three “high-productivity” languages mentioned above, the proposed language semantics provide a placement-neutral declaration for variables, and provides instead explicit declarative constructs to inform the run-time system where data may be needed. The run-time system then exploits this information to determine the minimum amount of necessary communication sufficient to satisfy the consistency semantics assumed in the program. For example, when multiple logical threads explicitly declare inter-domain communication for different subsets of a data item, the run-time system automatically merges these declarations during execution so that only one communication event occurs per node for all logical threads sharing a view on the data item.

11.3.2 Open questions to operating software providers

If the proposed fine-grained interface to placement is to be adopted, operating software providers will need to answer the following questions:

- how does a program obtain access to a virtual cluster initially? Implicitly, we have assumed so far that the top-level thread(s) of a program are placed at some virtual cluster by an “operating system” upon program start-up. With the proposed language constructs, a program can then address cores within this initial cluster. However, system services must be designed to gain access to additional clusters if automatic scalability to varying resource availability is desired.
- in an attempt to answer the previous question in our own work, we have prototyped a cluster allocation service able to partition the entire chip into sub-clusters on demand. However, we could not find a satisfying protocol to query this service: what should be its request parameters?
Our naive initial choice was to define a “desired number of cores” and “whether a single consistency domain is desired” as request parameters. However, this was misdirected: programmers and operating software abstractions do not reason in terms of number of cores and consistency domains; instead, they have real time constraints (e.g. guaranteed minimum throughput on an algorithm) or optimization goals (e.g. maximize the throughput on an algorithm). We are not aware of any existing methodology to translate these requirements to placement requests at a fine grain.

- in more general terms, *can application-level extra-functional requirements on throughput and latency be expressed only as constraints on the number and location of cores involved?* The proposed architecture tries hard to achieve this simplification with its distributed cache, which conceptually provides a uniform view over a shared off-chip backing store and isolates traffic between separate core clusters using a hierarchical topology (cf. section 3.4.1).

However, we did not find any results that suggest that these properties hold in an actual implementation for larger number of cores. Quite the contrary, using the implementations described in section 4.7 we observed memory bandwidth and latency interference between independent core clusters. We thus cannot exclude that future many-core chips will feature Non-Uniform Memory Access (NUMA) topologies with load-dependent bandwidth and latency properties. Were this to happen, the *communication activity* between program components must be characterized and taken into account by the operating software infrastructure. This would in turn imply that operating software providers will require to know about the semantics of memory consistency protocols on chip, and language designers will need to derive communication requirements between units of work expressed to run concurrently in program source code. We are not aware of any coordinated approach in that direction at the time of this writing.

Nevertheless, the integration of placement at the machine interface creates a *common vocabulary* to reason and talk about these issues between operating software providers. Such a common ground is sorely missing in current architectures, where issues of work placement are resolved using different abstractions in each operating software environment, and are thus difficult to compare analytically. We should thus expect that a common set of primitives will *facilitate* further research activities.

**Summary**

- In this chapter, we have introduced preliminary support for explicit placement of computations to clusters of cores defined dynamically. This support is sufficient to control thread placement in operating software and accelerate multi-stage parallel computations. It also avoids requiring programmers or code generators to make assumptions about the overall topology of the system.

- We have also isolated further research questions that must be answered before a full integration of these primitives into higher-level abstractions can be achieved. By studying related work we observe that these questions have not yet been fully answered by other modern, contemporary parallel programming languages; we suggest that common abstractions in the machine interface will facilitate future research in this area.