POLITECNICO DI MILANO
DEPARTMENT OF ELECTRONICS, INFORMATION AND BIOENGINEERING
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

EXPLORING ARCHITECTURAL SUPPORT FOR APPLICATIONS WITH IRREGULAR MEMORY PATTERNS ON DISTRIBUTED MANYCORE SYSTEMS

Doctoral Dissertation of:
Marco Ceriani

Supervisor:
Prof. Gianluca Palermo

Tutor:
Prof. Andrea Bonarini

The Chair of the Doctoral Program:
Prof. Carlo Fiorini

2010 – XXVI

Abstract

With computing systems becoming ubiquitous, numerous data sets of extremely large size are becoming available for analysis. Often the data collected have complex, graph based structures, which makes them difficult to process with traditional tools. Moreover, the irregularities in the datasets, and in the analysis algorithms, hamper the scaling of performance in large distributed high-performance computing (HPC) systems, optimized for locality exploitation and regular data structures.

The aim of the PhD research has been proposing an approach to system design that enable efficient execution of applications with irregular memory patterns on a distribute, many-core architecture, based on off-the-shelf cores. A secondary goal for the architecture design has been simplifying the programming using a simple Shared Memory model.

The four key elements of the proposed architecture are: (1) a global address space that allows the use of simple programming models, (2) prevention of dynamic formation of hotspots, (3) latency tolerance through lightweight multitheading and finally (4) fine-grained synchronization.

The feasibility and effectiveness of the approach has been evaluated with a prototype on FPGA, that uses off-the-shelf cores and communication subsystem, with the addition of a set of custom-designed components. These components offer the fore-mentioned four capabilities with a minimal impact on the chip architecture. In spite of the small size of the prototype, the performance scaling of typical irregular kernels proved the effectiveness of the approach. In addition, the FPGA prototype allowed to evaluate the technical issues related to the implementation of the proposed architecture,

suggesting the technical details required for supporting other commodity processors.

The performance data obtained from the prototype have also be used to formulate an analytical model, which identify the system bottlenecks and can be used for dimensioning a large scale distributed system.

The research has moved on with the creation of a lightweight system simulator, in order to evaluate additional features using high level performance models instead of a full HDL implementation. The simulator neglects modelling the details of cache and memory hierarchy, which are irrelevant for applications which lack data locality, thus improving the simulation speed. On the other hand, it models the extended memory behaviour the fine-grained locking routines introduced by the custom hardware components. The simulator allowed to evaluate the impact of additional architectural features, such as the support for atomic operations on the global address space. The use of remote atomic operations, in place of lock routines, allowed to significantly reduce the synchronization overhead and exposed larger amounts of parallelism enhancing the effectiveness of the many-core system.

Contents

CHAPTER 1

Introduction

The study of large dynamic systems with complex interactions, such as human interactions and biological systems, produces the creation of very large collections of unstructured data. The example most visible to everyone is that of social networks. In ten years social networks gathered an enormous number of users, from the 271 million active users of Twitter [3, 47] to the 1.31 billions Facebook users [27]. The analysis of the users activities and interactions is hard, because it requires to elaborate very large and dynamically evolving connectivity graph-based structures. But the usefulness of analyzing large semi-structured data is not limited to social networks. It also allows to better comprehend many complex systems, such as biological systems [23, 43], economic systems [61], or the spreading of diseases [19].

The algorithms designed to analyze semi-structured data generally need to traverse very large pointer-based data structures, such as graphs or unbalanced trees, which can be composed of many millions of nodes and edges. Because of the large amount of information, often it is not feasible to store the whole data structures in the main memory of a single computer. Hence distributed machines are required to run the algorithms without the overhead of continuous disk accesses. However, the low regularity of the data structures makes it very difficult to partition the data effectively across the

nodes. In addition, the dynamic nature of the data structures limits the effectiveness in time of any attempt of optimizing the data layout. Also, the accesses to these structures present poor spatial and temporal locality with respect to computational intensive algorithms. This causes frequent fine-grained requests to both the memory hierarchy and the network, resulting in a very irregular mix of long and short latencies[37].

Traditional High Performance Computers (HPCs) mainly target computationally intensive applications, which process highly regular data structures, such as matrices. These applications usually feature high arithmetic intensity and data locality, which benefits from powerful processor pipelines and complex cache hierarchies. In addition, the regularity of the data structures facilitates the partitioning of the algorithms into loosely dependent tasks, and the optimization of data movements across multiple memories. Therefore, modern HPC systems are designed as large clusters composed of nodes with powerful multi-core processors, connected by communication networks optimized for maximum bandwidth. However, the deep cache hierarchies used for computational intensive applications are not beneficial to algorithms with irregular memory patterns, which frequently access the main memory[4]. Furthermore, because of the high and irregular connectivity existing in many real world graphs, the performance of large scale knowledge discovery applications are dominated by the communication time when run on distributed systems [66].

From the situation described above, two issues arise relative to irregular applications. The first one is that to improve the execution of parallel algorithms with poor locality in the data access, any attempt to reduce the memory access latency using complex cache hierarchies, either local or distributed, is poorly effective. A different approach has been attempted in the design of the Cray XMT supercomputer [28]. Instead of reducing the latency of each access, this supercomputer can hide that latency executing other threads, thanks to a high number (128) of hardware thread contexts concurrently active in a custom pipeline. This design choice has been successful in achieving the intended goal. However, the cost paid for such a high number of hardware threads is a low clock frequency. Paired with a complete lack of cache hierarchy, this penalty causes poor performance when executing computation intensive applications. This limits the effectiveness of the XMT to a reduced set of application domains.

The second issue is related to the manual optimization of the algorithms. In distributed systems it is imperative to minimize the cost of communication between nodes. Hence, the algorithms have to be designed focusing on data placement and accesses, instead of the control flow, identifying

all possible sources of data locality and optimizing the distribution of the structures on the system nodes. Also, it may be necessary to reorder or even modify the algorithm steps in order to minimize inter-node communication and to aggregate multiple messages to minimize the network overhead. The consequence is that an algorithm optimized for a distributed system can be very different, and much more complex, than the basic version designed for a simpler sequential machine with uniform memory. This is problematic for HPC teams, in which system and algorithm experts usually work jointly with domain experts, and often face novel problems, requiring an incremental approach to the software design.

Finally, paradigms for novel processor designs are shifting towards many-core architectures. The hundreds of low-power cores included in those processors are ideal for building HPC systems which run massively parallel applications. However, for performance reasons the processor architectures favor distributed programming models focused on locality exploitation, which, as already explained, is not beneficial to irregular, graph-based, algorithms.

Accordingly, the aim of this thesis is to understand how to evolve novel multi-core and many-core architectures to address the characteristics of parallel applications with irregular memory access patterns.

1.1 Proposed approach

The main goal of this thesis work is to design an abstract computer architecture capable of efficiently running algorithms with irregular memory access patterns, such as graph-based algorithms, implemented using a simple programming model based on the well known Shared-Memory Single Instruction Multiple Data (SIMD) machine.

The key elements of the architecture are:

- it provides a global address space, shared and distributed among the nodes of the system and transparent to the application;
- it reduces the dynamic formation of hot spots in the network and memory subsystems, by using a fine-grained translation function from the shared address space to the physical memory addresses;
- it provides automatic context switch on remote memory requests, effectively hiding the network latency;
- it provides fin-grained synchronization primitives for run-time conflict avoidance;

- to increase application performance, it implements most of the features inside custom hardware blocks, which do not modify the internal design of the processor cores, and have minimal requirements; hence it allows efficient execution of both application with regular or irregular access patterns;

In addition, the thesis presents a concrete architecture prototype, implemented using multiple Field Programmable Gate Array (FPGA) devices. Accordingly to the declared goal, each node of the system includes a traditional multi-core processor, composed of off-the-shelf components. The architecture is extended with custom hardware components, connected to the bus interfaces, proving that the full set of features can be implemented without modifying the core internal structure. The main limits of the prototype are the reduced operating frequency and available RAM memory. However, the processor frequency and network bandwidth are scaled by a similar factor with respect to analogous commercial components, allowing to measure realistic performance metrics.

Finally, the proposed work included the development of performance evaluation and estimation tools, consisting in an analytical model and a high-speed light-weight x86 system simulator. The simulated system includes all the custom components which are part of the proposed architecture, providing an accurate simulation of the application system behavior and performance. On the other hand, the simulator neglects architectural components that we assume disabled in the use case, such as the cache hierarchy, to speed up the simulation with respect to other parallel simulators. Thanks to using the Intel PIN[41] binary instrumentation framework, the simulator can run native x86 and x86-64 multi-threaded applications, implemented with a POSIX-like interface.

The thesis is organized as follows. Chapter 2 provides the relevant background on parallel applications with irregular memory patterns and their role in the High Performance Computing context. Following, chapter 3 describes the existing languages, libraries and architectures designed to improve the performance and productivity of HPC applications, with a focus on irregular applications. Chapter 4 introduces the abstract computer architecture and describes the custom components which are part of it, focusing on the generic or portable aspects. The specific low-level details regarding the implementation of the architecture on FPGA are provided in chapter 5, together with an analysis of possible issues or changes required to integrate the proposed components on architectures based on existing commercial processors. Chapter 5 also describe the simulation platform, focusing on the models used for simulating the timing and performance. The experi-

ments carried on for evaluating the architecture performance are described and analyzed in chapter 6, which in addition defines an analytical model which allows for a quick and wide design space exploration.

1.2 Dissemination of Results

The content of this thesis work has been published in various international conferences and journals. In details:

International Conferences

- **M. Ceriani**, S. Secchi, A. Tumeo, and O. Villa, *Prototyping Hardware Support for Irregular Applications* in Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '13), January 2013.
- **M. Ceriani**, S. Secchi, A. Tumeo, and O. Villa, *Exploring Manycore Multinode Systems for Irregular Applications with FPGA Prototyping* in Proceedings of the 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), April 2013.
- S. Secchi, **M. Ceriani**, A. Tumeo, O. Villa, G. Palermo and L. Raffo, *Exploring Hardware Support for Scaling Irregular Applications on Multi-node Multi-core Architectures* in 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), June 2013.

Journals

- **M. Ceriani**, S. Secchi, O. Villa, A. Tumeo, G. Palermo, *Exploring Efficient Hardware Support for Applications with Irregular Memory Patterns on Multinode Manycore Architectures* in IEEE Transactions on Parallel and Distributed Systems.

CHAPTER 2

Background

Since the dawn of computer history, one of the directions taken to increase the speed of computation has been the exploitation of the intrinsic parallelism existing in algorithms, in all its forms, such as Instruction Level Parallelism (ILP), Data Level Parallelism (DLP) or Task Level Parallelism (TLP). Nowadays, massive parallel execution is the main factor which drives the progress in computer performance, from small embedded systems to large scale High Performance Computing (HPC) systems, passing from personal computers. In the early 2000s multi-core processors started superseding single-core processors in all market sections, signing the end of the frequency scaling era. In the same years, the advent of general-purpose computing on GPUs allowed even small laboratories and companies to use hundreds of cores to run massively parallel algorithms. These changes had an impact also on HPC computing designs. Single processors and symmetric multi-processor (SMP) systems have given way to massive parallel processing (MPP) systems or clusters with a number of cores in the order of 10^4 to 10^6. The top supercomputer in June 2014, Tianhe-2, includes 3,120,000 cores to achieve a peak performance of 33.86 Pflop/s [58].

Traditionally, the approach used to parallelize an algorithm was to iden-

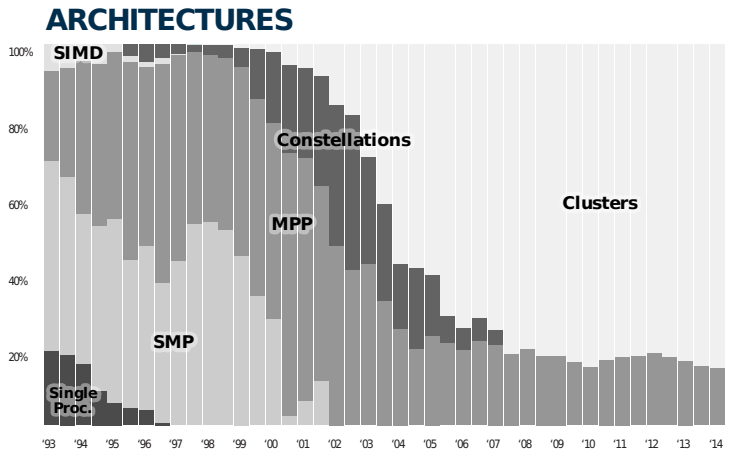


Figure 2.1: *Top 500 supercomputers by architecture. Source: top500 project[57]*

tify statically the dependencies among operations, which can be represented by a *dependency graph*. At low level, ILP can be exploited by identifying dependencies within small sequences of instructions, and executing them on different execution units. This activity can be performed automatically by a compiler or the processor itself, and is beneficial to processors with complex, super-scalar pipeline or vector processing units. A particular case, used by vector units, is the concurrent execution of independent iterations of the algorithm loops, which can be analyzed manually or even automatically by a parallelizing compiler. Loop parallelization is also the most common approach used to identify concurrent tasks to be performed by different threads in a multi-core system. An advantage of this method is that the amount of data shared between different tasks can be minimized with an appropriate partition of the data set, hence reducing the need for run-time synchronization. Also, data can be efficiently distributed between the tasks to maximize locality of accesses.

However, static loop analysis and dependency analysis work only in *"regular"* algorithms that use multidimensional dense arrays, e.g. FFTs, finite difference methods, algebraic computations and image filtering. Many *"irregular"* algorithms show a more general form of data parallelism, which require more complex run-time strategies and show different performance characteristics.

2.1 Data Parallel Algorithms

In computer science there are different definitions of regular and irregular algorithms or implementations. They are distinguished by what features of the algorithm are considered, but usually the concept of regularity is linked with the concept of performance optimization. The cause is that regularity, of any kind, has often been used as a source for code optimization, and its exploitation has been the goal of many architectural features, such as cache hierarchies or vector processors. Therefore, the common element in all definitions for *irregular algorithms* is the necessity to find new and different tools to solve the problem of their efficient execution.

In this research work, we use the following empirical definition:

Definition 2.1.1. *We consider as Irregular algorithms those algorithms that process large pointer-based data structures, such as sparse graphs of unbalanced trees, or show similarly irregular memory access patterns.*

Examples of problems solvable with irregular algorithms are: knowledge discovery from social networks and other human activities, e.g. economic transactions, optimization problems based on SAT solvers, or physic simulation of non homogeneous objects based on non uniform meshes or trees, e.g. N-body simulation.

The algorithms used to solve all these problems share a general form of data parallelism which is significantly different from the parallelism existing in matrix-based or vector-based applications. A detailed study of irregular data parallel algorithms is presented by Pingali et al. [51] in «The Tao of Parallelism in Algorithms». To classify the algorithms they use a data-centric formulation, called *operator formulation*, on the abstract data structure. Three key definitions are provided: *active elements*, *neighborhood* and *ordering*.

The active elements are the elements of the data structure on which computation might be performed in each point during the program execution. The processing of an element is performed by applying an operator to it. The neighborhood of an element is the set of other elements that are read or written while applying the operator to it. In the general case, the neighbors of an elements according to this definition do not coincide with the neighbors defined in graph theories. Figure 2.2 depicts 3 example structures, with active elements and neighborhoods highlighted. Finally, the ordering of the set of active elements is a distinctive characteristic of each algorithm. In some case, the algorithm is *unordered*, and the implementation can peek any element from the active set. In order cases, for example in event-driven

simulation, the algorithm imposes a partial or complete ordering to the active sets. Clearly, the ordering of an algorithm caused by potential dependencies between the elements reduces the available parallelism.

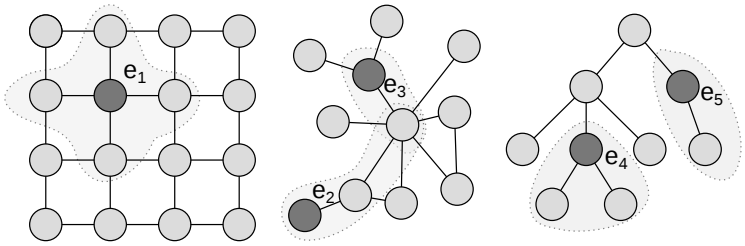


Figure 2.2: *Example of topologies, active elements and neighbors.*

Notably, a few algorithms rely on random selection of the unordered active elements. This happens, for example, when the execution of some of the elements does not contribute towards the overall progress of the algorithm. In this situation random selection ensures that eventually useful elements will be selected.

2.1.1 Algorithms Analysis

Figure 2.3 shows three analysis dimensions which can be used to classify data parallel algorithms. The *topology* dimension describes the structure of the data. The *active nodes* dimension describes the dependencies between active nodes and how the they are activated during the execution. Finally, the *operator* dimension describes the local effects of the operator applied to an active node.

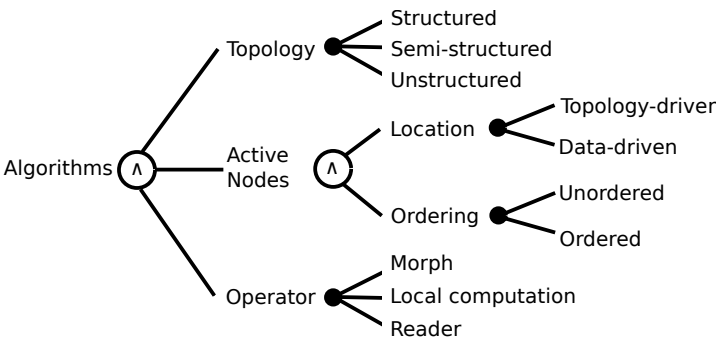


Figure 2.3: *TAO structural analysis. Source: Pingali et al. [51]*

The following paragraphs will briefly describe this analysis schema, and use it to highlight the main characteristics of *irregular applications*, which makes them different from regular ones.

Topology

The topology of the data is the first and most important characteristic of a parallel algorithm. The shape of highly regular structures can be described with a small number of parameters, for example the width and height of a rectangular matrix. This allows to define concise arithmetic expression for homogeneously partitioning the structure among a set of parallel tasks. Semi-structured topologies, e.g. balanced trees, have invariant properties that can help the partitioning. Conversely, the topology of general graphs is a property of the single instances, hence dependencies between elements have to be computed at run-time.

In addition, in regular topologies every element apart from the borders has the same number of neighbors as the others. Thus, in most cases the operator execution times tend to be homogeneous over the whole structure. Oh the other end, in general graphs and trees the size of the neighborhood varies from element to element, unbalancing the workload.

Finally, multidimensional arrays can be linearized, or blocked, to optimize the use of the cache hierarchy. Conversely, the complex connectivity of unstructured topologies reduces the locality of data accesses and generates unpredictable access patterns.

Active nodes

As already discussed, one of the characteristic of an algorithm is the existence of *ordering* constraints on the set of active elements. Another characteristic is the *location* of newly activated elements, that can be topology- or data-driven. In the former case, the activation depends only on the topology, for example performing a full visit of the structure. In the latter, the activity on one element may activate other elements, producing unpredictable and irregular workloads. Applications such as event-driven simulation are part of this group. A difficulty in parallelizing algorithms with data-driven activation is that the execution is not distributed evenly on the data structure, causing workload distribution and locality exploitation work against each other.

Operator

The way the operator modifies the data structure has a fundamental impact in the possibility to identify or prevent conflicts between concurrent activities. The simplest class is composed of operators that access the data structure only with read operations, and store the results in a different structure. Are part of this class all the algorithms that use a tree index to speedup the accesses to a second irregular structure, or to cluster it. A second class includes the algorithms that perform local computation in the active nodes, but do not modify the topology. Finally, morphing operators may modify the topology by adding or removing elements in the neighborhood of the active one.

The second and third classes of algorithm requires run-time mechanisms to grant exclusive access to the elements, and prevent incoherent states to be seen by concurrent tasks. The basic mechanism is the use of lock based critical sections that prevent concurrent access to the elements. A more sophisticated approach allows to define transactions that can be executed speculatively, and rolled back in case of conflict. Both approaches inevitably introduce an overhead in the execution, hence powerful but lightweight mechanisms for conflict avoidance are even more important for irregular applications than for regular ones.

Another characteristic of both classes of algorithms is that the problems solved by them may have multiple valid solutions, and each order in which the active elements are evaluated produces a different solution. This is an kind of intrinsic parallelism which requires to drop compatibility with serial execution to achieve optimal performance. For this reason, it is difficult to exploit it using a parallelizing tool, without explicit hints from the developer. It also increases the effort required to implement an application, because the non-determinism of the outputs prevents using some of the basic testing and validation strategies.

2.1.2 Execution Models and Run-time Support

The previous review of the characteristics of data parallel algorithms introduced some element required for their run-time execution, specially for irregular algorithms. This section is clarify them and presents the possible execution models.

In the most general case, the execution of a data parallel algorithm requires two collaborating activities: the identification of possible conflicts due to overlapping of the elements neighborhoods, and the scheduling of the operations according to the required ordering and the conflicts. Without

regularities in the structure of the data each of these two activities has to be performed at run-time, using the mechanisms provided by the system.

For the conflict identification and prevention we can identify various approaches:

Critical sections. The algorithm designer identify parts of the code that must not be executed concurrently, and marks them using functions provided by the language or libraries used. The critical sections are serialized at run-time each time two or more processors access the same piece of data. The serialization guarantee correctness of the algorithm, but introduce an overhead in the execution every time one thread of execution has to wait before entering a critical section. This overhead can be in the form of stalls of the processor, or additional context switches if the processor executes multiple threads. A special case of critical sections is represented by atomic instructions, which guarantee that an operation is completely executed and completed without conflicts from other processors or devices.

Transactions. Critical sections are generally implemented by locking a resource until completion. A different approach is to execute speculatively the sections inside a transaction that is completed only in absence of contention, and is re-executed otherwise. This approach is mostly beneficial whenever it is not known if a section will generate a conflict before it is executed. In this scenario, a lock-based critical section would cautiously stall parallel tasks even when it unnecessary.

To support this approach it is necessary to isolate all the data updates in a local copies and committing them in a single update. The mechanism can be implemented both in hardware or software. For more than a decade hardware support has been a hot research topic, and recently Intel has released the line of microprocessors based on the Haswell architecture, which expands the x86 instruction set with transactional synchronization extensions [34]. However, even when efficient hardware implementations exist, a software fallback path is required to handle the case when transaction completion is impossible. This introduces overhead in the execution and complexity in the algorithm design. In addition, managing memory transactions in the scenario of distributed HPC systems is challenging, and even more in the context of irregular applications with lack of data locality.

Dependency graphs. When possible, the construction of a dependency graph allows to compute independent sets of element, which can be

elaborated in parallel without requiring conflict detection. Regular data structures like rectangular matrices allow to identify all possible conflicts when designing the parallel algorithm. Hence, the parts of the data structures can be distributed on the system processors so that possibly conflicting elements are elaborated serially by the same processor, while different processors elaborates independent sets. Some topology-driven algorithm may permit the use of an inspector before the actual execution, to analyze the data structure and identify the neighborhoods of each element and compute a dependency graph. The inspector approach is useful when the data structure is immutable, and the inspection time is negligible with respect to the total execution. In all the other cases run-time coordination is required.

Regarding the scheduling of the operations, the main issues are two. The first one exists only on ordered algorithms, and is the decision on when to execute an operator. The cautious solution is to elaborate an element only after all the elements with higher priority, even when no conflict is detected. The reason for the wait is that each application of the algorithm operator may activate an element with priority higher than those already existing in the active set and conflicting with them. If a transaction mechanism is used, this cautious constraint can be relaxed, allowing the concurrent execution of independent activities as soon as possible but delaying the commit.

The second, and more general, issue is the distribution of the elements to the system processors or computation threads. Work distribution has the goal of optimizing the use of system resources, to reduce execution time. On one hand, this involves assigning a similar amount of work to each processor, to even load and improve the processors utilization, on the other hand it can exploit data locality to reduce the cost of the data accesses. The simplest approach is to divide statically the data structure in a number of equally sized parts equal to the number of processing elements. This approach is suitable for regular data structures, or topology-driven algorithms that allow the identification of conflicts at run-time using an inspection step. In the general case, instead, a run-time scheduling mechanism is required, which assigns elements to the processors according to the dynamic workload distribution. Irregular algorithms for graph analysis usually make use of a work list that stores the nodes to be processed. Therefore, the scheduling consists in arbitrating the parallel accesses to this list. The design of the distribution mechanism is a delicate element in the parallelization of the algorithms. Multiple solutions exist in literature such as centralized queues, distributed queues, and work stealing. However, none of the solution is universally optimal, and the effectiveness depends on how costly is

the overhead due to synchronized accesses on a given system architecture.

2.2 Algorithms on Graphs

Algorithms based on graphs, or sub-graphs such as trees, are the most irregular kind of algorithm. Typically, they express at the highest level both kind of irregularity and unpredictability, both in the memory access patterns and distribution of the workload across the data structure. Graph algorithms are also the natural expression of many complex problems, from the alignment of genomes [32], to cybersecurity [65] and biology research [23]. Because of the importance of data intensive applications, a big effort has recently been spent in defining benchmarks to guide the evolution of hardware architectures and software systems to support them. The basic kernel most used in the benchmark is an ordered graph traversal, the Breadth First Search (BFS) [31]. The performance of this kernel are very important because not only it is very representative of data intensive irregular applications, but it is also a corner stone in the implementation of more complex algorithms. Examples of algorithms based on BFS are the construction of minimal spanning trees, the computation of centrality measures to identify the importance of elements in the networks, the identification of connected components or heuristic search algorithms such as A*.

2.2.1 Breadth First Search

The breadth first search algorithm is a complete visit of a graph graph $G = (V, E)$, where V is the set of vertices and E is the set of all pairs (u, v) with $u, v \in V$ corresponding to the edges. The search starts from a given vertex $v_s \in V$, and constructs a spanning tree of the graph containing all the nodes that can be reached from v_s, and the edges which connects them to the root through a path with minimum distance. The result of the algorithm can be a map which stores the predecessor of each node, or the annotation of the predecessors inside the nodes themselves.

Algorithm 1 outlines a typical sequential implementation of the BFS. The algorithm explores the graph by expanding the set of visited nodes one level at a time. The nodes that have been reached but not expanded are stored in a work list, Q in the listing. This set is referred to as *frontier*. A second list, named QN stores the nodes reachable from the vertices in Q but not already visited. At the beginning (lines 1-4) the algorithm initializes Q with the starting vertex v_s, and sets all the other vertices as *not visited* and their predecessor as *undefined*. The algorithm proceeds in steps, expanding the frontier one level at a time, with the loop at line 6. The body of the

Algorithm 1 Sequential BFS algorithm

Require:

```
1:  function BFS(V, E, v_s)
2:      Q = {v_s}
3:      for all v ∈ V do
4:          Vist(v) = false
5:          Pred(v) = undefined
6:      while Q ≠ ∅ do // Expand one level at the time
7:          QN = ∅
8:          for all u ∈ Q do
9:              for all v|(u, v) ∈ E do
10:                 if Vis(v) = false then
11:                     QN = QN ∪ {v}
12:                     Vis(v) = true
13:                     Pred(v) = u
14:         Q = QN
        return Pred
15: end function
```

loop prepares QN to store the next frontier, emptying it at line 7. Then, it scans the frontier and explores the vertices that can be reached from it(lines 8-9). All vertices that have not been visited yet are added to QN, are sets as visited, and their predecessor is assigned (lines 11-13). At the end of each level, the list Q is updated with the content of QN, preparing the next level. The algorithm concludes when all the reachable nodes have been visited, and therefore the work list becomes empty.

The algorithm has a few notable characteristics. First of all, we should consider the complexity. Each vertex reachable from the starting one is added to the work list exactly once when first visited. Also, all the edges outgoing a vertex are visited once at line 9. If the graph is undirected and entirely connected, all the graph nodes and vertices are explored. Hence, the block at lines 11-13 is executed $|V|$ times, and the test at line 10 is executed $|E|$ times. In social networks and other interesting graphs the number of edges can be two orders of magnitude higher than the number of nodes [47]. Under this graph properties, the accesses to $Vis(v)$ at line 10 become the crucial element to be considered in the optimization optimization. Secondly, the patterns in the memory accesses executed at line 10 to test if a node has been visited reflect the irregularity existing in the graph structure. As just stated, these accesses are executed tens or hundreds of times more than the queue operations, hence the lack of locality and predictability harms the overall algorithm performance. Finally, the structure

of the algorithm, composed by multiple nested loops, is a perfect example of irregular data parallel algorithm. Each iteration can potentially be executed in parallel with the others, but conflict identification is required at run-time. The next sections will summarize the existing techniques for parallelizing the algorithm.

2.2.2 Parallel BFS

The algorithm listed in Algorithm Algorithm 2 is a parallel version of Algorithm Algorithm 1, obtained with a straightforward loop parallelization. The initialization can be performed in parallel, distributing the vertices evenly among the processors. In the main body of the algorithm, the loop at line 8, which visits the vertices in the current work list, is converted in a parallel loop that distributes the different iterations on the various processors and/or threads. On the other hand, the adjacency list of the vertices is iterated sequentially by a single processor, at line 9. The parallelization requires the introduction of 2 critical sections. The first one is constituted by the entire inner loop body, at lines 10-15, and prevents concurrent visits of a single node. The second critical section, at line 12, is required to insert a newly visited vertices in the work list. Finally, the parallel execution is synchronized at the end of each level, to guarantee that the graph is explored in strict order of distance from the source vertex.

One of the critical aspects of Algorithm 2 is the implementation of the largest critical section. The listing uses *lock(v)* to represent a lock at the granularity of a single vertex. The fine granularity is required to maximize the parallelism, but a more coarse granularity is possible, for example grouping multiple vertices. A possible implementation of the lock is to allocate a mutex variable for each graph vertex. This simple option has a very huge impact on total memory required, adding an overhead directly proportional to the graph size. A different implementation could store the addresses of the locked vertices in a table and use a single mutex to protect the table, effectively trading execution time for memory space. However, both approaches incur in high overhead when the lock is not taken. A few common optimizations used to reduce the impact of synchronization are added in Algorithm 3, which is proposed in [4] for a single socket architecture. First of all, it uses a bitmap to mark the visited vertices, reducing the memory size. Then, at lines 12-13 a simple check is performed on the bitmap to test if a vertex has already been visited. Since each vertex can have many input edges, most of the times the test will be positive and the critical section can be skipped altogether. If a vertex has not been visited

Algorithm 2 Parallel BFS algorithm: straightforward approach

Require:

```
1:  function BFS(V, E, v_s)
2:      for all v ∈ V do // in parallel
3:          Vist(v) = false
4:      for all v ∈ V do // in parallel
5:          Pred(v) = undefined
6:      Q = v_s
7:      while Q ≠ ∅ do
8:          QN = ∅
9:          for all u ∈ Q do // in parallel
10:             for all v|(u, v) ∈ E do
11:                 lock(v) // enter critical section
12:                 if Vis(v) = false then
13:                     QN = QN ∪ {v} // atomically
14:                     Vis(v) = true
15:                     Pred(v) = u
16:                 unlock(v) // exit critical section
            barrier
17:         Q = QN
        return Pred
18: end function
```

previously, and only in that case, the algorithm uses a potentially more expensive atomic operation *read_and_set* to atomically read the bit and set it to one. The atomic operation is required because multiple processors can execute lines 12-13 concurrently. The union of these techniques greatly reduces the probability of run-time contention and increases the scalability.

A second critical element is the parallel access to the work lists. The list QN produced at each iteration is often divided into multiple private lists, one per processor or processing thread. The private lists are joined at the end each iteration of the outer loop, corresponding to a whole level in the search. The partition has two positive effects: first of all the queuing can be performed without synchronization, secondly the a form of data locality is introduced in the algorithm because each processing element access only a part of the list. The current work list, Q can either be partitioned statically before each algorithm step, or dynamically distributed using small critical sections for the updates. Each option has its own advantages. The dynamic approach naturally provides a balanced workload distribution, while the static division can be used to expose data locality. This pattern of distribution of the global work lists into private ones is always used in distributed systems, where accessing remote memory is much more expensive than

Algorithm 3 Parallel BFS algorithm

Require:

```
1:  function BFS(V, E, v_s)
2:      Q = v_s
3:      for all v ∈ V do // in parallel
4:          Bitmap[v] = 0
5:      for all v ∈ V do // in parallel
6:          Pred(v) = undefined
7:      while Q ≠ ∅ do
8:          for all processor p do
9:              Q_{next,p} = ∅
10:         for all u ∈ Q do // in parallel on each processor p
11:             for all v|(u, v) ∈ E do
12:                 a = Bitmap[v]
13:                 if a = 0 then
14:                     prev = read_and_set(Bitmap[v], 1)
15:                     if prev = 0 then
16:                         Pred(v) = u
17:                         QN = QN ∪ {v}
            barrier
18:         Swap(Q, QN)
        return Pred
19: end function
```

local one.

A particular approach is described in [56]. The authors exploit two characteristics of the BFS algorithm. First of all, the algorithm accepts a limited degree of non-determinism: it does not matter which is the path taken to reach a vertex, it is enough that the path has the shortest length. In other words, each vertex may have multiple predecessors, and each of them is an acceptable piece of the final solution. In addition, the algorithm expands the frontier in sequential phases, and all the vertices in the frontier of a phase has the same distance from the source vertex. Therefore, the order used to iterate on the work list is not relevant for the overall correctness. In addition, a vertex can be inserted in the work list multiple times within the same phase, without compromising the algorithm correctness. The neighborhood of replicated vertices are visited more than once, but all the visits after the first one have no effect. The authors use this property to design a set of algorithm implementations that use special queue implementations and optimistic parallelization and avoid synchronizing the queues operations. The effectiveness of the approach depends on the structure of the graph and is limited to shared memory systems. However, the idea of allowing a small

number of useless visits to reduce the synchronization overhead is worth mentioning.

2.2.3 Distributed BFS

The parallel BFS algorithms described in the previous section achieve good performance on single processor systems, with uniform memory access times. However, the majority of high performance systems are massive parallel processing systems, or clusters, with hundreds or thousands of processors and distributed memories. In addition, each node of the system can host multiple processors on multiple sockets. In this case, accesses to data cached in different sockets of the same node have different latencies. Because of these reasons, the most studied versions of parallel BFS are based on the distributed memory model. In this model, the graph is partitioned across the whole system, and each node process only the part of the graph that is allocated on its memory. The nodes exchange messages with each other using explicit routines, which are also used as synchronization points. This approach has the advantage of exploiting data locality and focusing on data placement rather than control flow. The drawback is that all distributed BFS implementations are significantly more complex than the serial implementation listed in Algorithm 1.

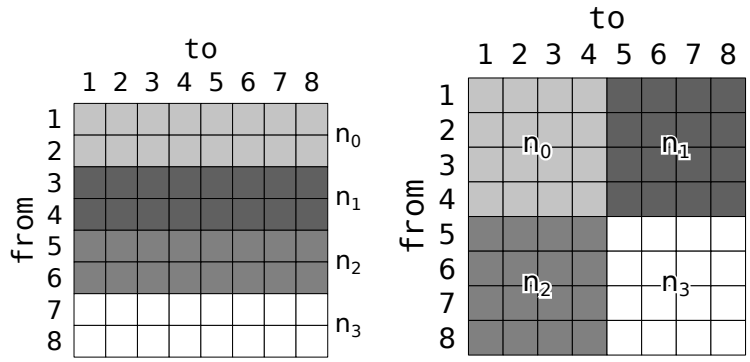
Agarwal et al. present a scalable BFS for multi-core processors, which target a dual socket Nehalem EP processor[4]. In addition to the techniques presented in Algorithm 3, the authors explicitly partition the data across the two sockets and introduce first in first out (FIFO) channels for the communication between sockets. Algorithm 4 gives a high level view of the implementation. The global work lists Q and QN are replaced with a set of lists, one per socket (node). The initialization phase performs the same operations as before, but adapted to the new number of lists. In addition, the all FIFO channels are initialized empty. The main algorithm loop is split in two steps. The first step iterates over the local work lists $Q[this]$ and visits the neighborhood of the vertices as in the sequential version. For each neighbor vertex the algorithm determines the socket (node) on which the vertex is mapped. The local vertices are handled as in Algorithm 3, using only local memory accesses. On the other hand, if the destination of an edge (u, v) is mapped to a different socket, the edge is enqueued in the corresponding channel (line 18). After this first phase, all processors synchronize with each other using a barrier, to ensure that every vertex from the work lists has been either visited or enqueued in a channel. Then, the visit is repeated with each processor fetching the edges from its channel.

The use of channels ensures each of the work lists, bitmaps and predecessor maps are accessed by a single processor. This allows to exploit data locality and optimize the use of the caches. The communication between sockets happens through the channels, and is split into two steps: a local enqueue operation at line 18 and a dequeue operation at line 20. During the dequeue, all cache lines that are still valid are invalidated and migrated to the destination socket.

Algorithm 4 Distributed BFS algorithm: high level

```
Require:
 1: function BFS(V, E, v_s)
 2:     for all v ∈ V do // in parallel
 3:         Bitmap[v] = 0
 4:         Pred[v] = undefined
 5:     for all n ∈ Nodes do // in parallel
 6:         Q[n] = ∅
 7:         QN[n] = ∅
 8:         Channel[n] = ∅
 9:     if systemNode(v_s) = this then
10:         Q[this] ← {v_s}
11:     while Q[this] ≠ ∅ do
12:         for all u ∈ Q[this] do // in parallel
13:             for all v|(u, v) ∈ E do
14:                 n = systemNode(v)
15:                 if n = this then
16:                     // Local execution
17:                 else
18:                     Channel[n] ← (u, v)
            barrier
19:         while Channel[this] ≠ ∅ do
20:             (u, v) ← Channel[this]
21:             // Local execution
            barrier
22:         Swap(Q[this], QN[this])
23:         QN[this] = ∅
        return Pred
24: end function
```

Even if this algorithm is designed for a shared memory system, its structure is the same used for distributed memory systems. For this reason, the listing in Algorithm 4 uses the term *node* instead of socket. In distributed systems the communication does not happen through cache line migration but through messages sent over a network between the two computation steps. By introducing a single communication step, the distributed version



(a) *1D* **(b)** *2D*

Figure 2.4: *Partitioning approaches for the adjacency matrix in the BFS distributed algorithms.*

permits also to aggregate multiple edges directed to the same destination in a single long message, improving the network efficiency. One drawback is that in this high level view of the algorithm communication and computation are temporally decoupled, causing a drop in the overall efficiency. For this reason, actual implementations have to carefully select and optimize the low level routines used for communication, in order to allow overlapping of computation and communication.

An large impact on the performance of distributed BFS implementations is given by the schema adopted for distributing the vertices on the system. Two approaches exist in literature, 1D and 2D decomposition. Figure 2.4 represents two adjacency matrices for a small graphs with eight vertices. In the picture, each square represents a possible edge, while the row and column labels are respectively the identifiers of the source and destination vertices. The matrices are colored according to the two partitioning approaches, 1D on the left and 2D on the right. In 1D decomposition the vertices are partitioned among the nodes of the systems, in groups with similar size. The whole lists of adjacent vertices are stored on the same node as the source vertex. In 2D decomposition, instead, the adjacency matrix is split in blocks hence each neighbors list is split among multiple nodes. The first approach is more straightforward, has a cleaner implementation with less overheads. On of the advantages of 2D decomposition are that long lists of neighbors are automatically partitioned and explored in parallel, improving load balancing. A second advantage is that the *all-to-all* communication step can be replaced with two steps involving the rows respectively and the

columns of the matrix. This decomposition allows for more optimizations of the network layer. Both approaches have appeared in scientific publications. The 1D approach is used, among other publication, in Agarwal et al. [4], in Villa et al. [59] and in Checconi and Petrini [18]. Instead, 2D decomposition is appears in Checconi et al. [17] and in Yoo et al. [66].

2.3 Conclusions

Applications with irregular workloads and memory patterns have an increasing importance in the High Performance Computing world. Their characteristics prevent them to exploit the features of modern HPC processors, designed to speed up the execution of regular algorithms with high arithmetic and computational intensity. The typical, and most crucial example of irregular algorithms are those that use generic, sparse graphs as data structures. The most frequent operation executed are ordered visits, hence a significant number of research projects has been focused on the optimization of search algorithms such as the breadth first search. The result of that research is a considerable increase of the performance and scalability of BFS implementations on distributed systems.

However, each of the optimized BFS implementations add a huge level of complexity on the simple sequential algorithm, and require a complete shift in the design approach, from the control flow to the data placement and move. For these reason, when designing new irregular algorithms for HPC and evaluating their effectiveness, it is very difficult to decide between the significance of the data set, the test execution time, and the effort required to reduce it. The HPC and scientific community could benefit from a system architecture that allows to execute irregular applications on distributed systems, with reasonably good performance and at the same time a simple programming model based on the common shared memory paradigm.

CHAPTER 3

State of the Art

Since decades the High Performance Computing has developed and used parallel and distributed systems to overcome the performance limitations of sequential processors. During this period various programming models and languages have been established as de facto standards for their effectiveness and performance, while others have been proposed more recently to increase productivity, reduce code complexity or solve portability issues. This chapters contains the most significant examples of the state of the art in parallel and distributed systems, parallel languages and support libraries.

3.1 Distributed systems

Distributed systems are computing systems composed of multiple software components distributed over the nodes of a computer network. The concept of distributed computation is general enough to include programs like online games or peer-to-peer applications, each developed using different technologies. In the context of HPC, distributed systems are used to scale parallel applications over thousands of processing cores and memories, generally using a Single Program Multiple Data (SPMD) control model. The simplest approach is to use the numerous processors to exe-

cute multiple instances of the same application on different data sets. Since the data sets are independent, very little communication is required between application instances, thus this approach can scale indefinitely with the number of processing nodes and is highly tolerant to network latency. This approach takes the name of grid computing[30], and allows to aggregate loosely coupled computers, which can be heterogeneous and even geographically distributed.

A more interesting case is the execution of tightly coupled application processes in a super-computing center to solve a single extremely large problem instance. In this scenario the design of algorithms requires a different approach with respect to shared memory systems, because the non-uniform and explicit nature of the inter process communication makes its role much more significant. The natural control model for distributed systems is Message Passing (MP). In the MP model an application is composed of multiple processes that communicate by explicitly interchanging messages among each other. With respect to conventional programming models, in the design of message passing algorithms the data placement and partitioning is a first-class element, on par with the control flow definition. A second difference is that remote sub-activities are not invoked by name, as the routines used in shared memory programs. Rather, the receiving process is responsible for executing the proper code after receiving the messages. The explicit decoupling of communication and computation allows for message passing applications to scale well with the number of processors used, which is the main advantage. Also, the message abstraction guarantee portability over a wide range of network architectures. Various realizations of the message passing model exist, such as the Aggregate Remote Memory Copy Interface (ARMCI)[49]. However, one standard have dominated all the others over time, the Message Passing Interface (MPI) and is become the de facto standard for HPC parallel applications.

3.1.1 Message Passing Interface

Message Passing Interface (MPI) is an interface specification for realizing libraries that support message passing operations[29]. It specifies names and calling conventions of communication subroutines for their use in C and Fortran applications, hence it allows to reuse existing languages, compilers and development environments. The result is a practical, portable and efficient interface for distributed computing. The initial target of MPI, in the 1980s and 1990s, were *distributed memory* architectures with one processor and one memory on each node. Thus, MPI favors the single

program multiple data (SPMD) program structure, and to a lesser extent Master/Worker. Later, the implementations have been extended to also exploit shared memory multi-processor architectures and hybrid systems, while maintaining the initial distributed memory model.

In MPI each process has a separate memory address space. The data structures have to be manually partitioned by the programmer, and the computation tasks mapped over the application processes. Data transfers are performed by copying a part of a process address space into the memory of another process. The most common API calls are for two-sided communication, in which data transfers are cooperative: they require a first process to execute a send operation and a second process to execute a receive operation. Examples are the $MPI\_send()$, $MPI\_recv()$ and $MPI\_Sendrecv()$ calls. Aggregate and collective operations are also available to facilitate communication within groups of processes, such as the $MPI\_Bcast()$, $MPI\_Reduce()$, and $MPI\_Alltoall()$. Because of the collaborative nature of the communication primitives, the algorithms must be break down into parallel steps with explicit synchronization points. However, one-sided communication primitives have been introduced starting from version 2 of the MPI standard, allowing remote memory access and decoupling data transfer from synchronization.

Since version 2, MPI also explicitly accepts a hybrid programming model, in which multiple threads are executed inside each process to exploit the multiple cores of SMP processors. The standard defines 4 levels of thread safety that the programmers have to select which pose increasing restrictions on the way the API can be called by the threads. The levels allow to reduce the need for the run-time checks and the corresponding overhead in applications that use a single thread for inter-node communication, while ensuring correctness if needed. There are no further details on how multithreading should be used, nor APIs for intra-process synchronization, however this allows to freely mix MPI with shared memory parallel paradigms such as standard POSIX Threads or OpenMP.

3.1.2 Active Messages

Active Messages (AM) are an extension to the message passing model that allow messages to perform computation on their own upon reception, asynchronously with the activities executed by the destination process [21]. The purpose is to reduce the overhead of message passing interfaces, with emphasis on reducing the latency of communication and removing the need for buffering. To achieve this goal, active messages are designed to exploit

the capabilities of the network hardware and provide low-level access to the software programmer. The basic idea is to introduce in the header of the messages the address of a user-level handler that will extract information from the message and integrate the data in the ongoing computation of the destination process. The handler must execute quickly and to completion, thus the heavy weight computation has to be performed by the main threads. The same requirements for quick and complete execution match very closely those of the interrupt handlers executed by most network architectures on message arrivals. Hence, active messages can use the mechanisms already existing, given adequate support from the operative system and the runtime libraries.

Active Messages are not buffered, except when required by for network transport, thus they avoid the overhead related to copying the data. In addition, by allowing the hardware interface to run user-level handlers it is possible to reduce the overhead up to an order of magnitude. Also, the asynchronicity greatly reduces the latency of the operations and allows to reduce the need for synchronization steps with respect to message passing model.

Active Messages is not a complete parallel programming paradigm, but a mechanism useful to implement these mechanisms efficiently. All participating processes must share the addresses of the message handlers, or corresponding table indexes, hence the mechanism is mainly suited to the SPMD programming model. Multiple implementations of AM exist, for various network protocols and communication libraries, such as UDP and MPI. The success is evidenced by the introduction of one-way communication primitives in MPI version 2.

3.1.3 Hybrid programming

Distributed memory and shared memory programming paradigms have different strengths and weaknesses, and they typically map on parallelism at different levels of granularity. Therefore the two models are often combined in the design and implementation of parallel applications, creating a hybrid programming model. The basic idea is to use message passing across the distributed nodes of the system, usually with MPI, and a shared memory model within each thread. The approach allows to increase performance with respect to the basic MPI implementation by exploiting a finer granularity of parallelism at the node level. Some applications clearly expose multiple levels of parallelism, hence greatly benefit from this approach. Introducing shared memory parallelism into an existing MPI ap-

plication has also various drawbacks, such as the overhead introduced by thread creation and synchronization, the interaction between two runtime libraries, and limitations in controlling work distribution and synchronization. The main weakness of the hybrid model is the increased complexity due to the combination and interaction of two different programming models, however in many cases it provides clear performance improvements [6, 10, 42].

3.2 Partitioned Global Address Space

Message passing, the de facto standard for HPC distributed systems, is generally considered harder to develop than shared memory programming models. However, distributed memory architectures achieve better scalability than shared memory ones, and the abstraction introduced by the communication interface allows for better portability. For these reasons, Distributed Shared Memory (DSM) models have been proposed since the early 1980s to combine both approaches by providing the illusion of a shared memory space on distributed hardware architectures [50]. Various implementations have been proposed which can be grouped in three main categories: hardware implementations that extend conventional cache techniques, operating systems that provide data sharing through virtual memory management, and compilers that automatically convert shared accesses in synchronization primitives.

Usually DSM systems provided a relaxed memory coherence model, because the latency of remote memory accesses are high and non uniform making strong coherence costly in terms of performance. To force coherence on individual accesses, the systems offer mechanisms that must to be explicitly specified by the programmer or high level software routines. An application can behave incorrectly when executed on systems with a weaker coherency model than the one used for the original implementation, hence the relaxed coherency of DSM models significantly reduces the portability of the applications. The problem is further enhanced by the complete lack of locality of access and awareness of data placement. Thus, DSM models do not completely realize the goal of simplifying the execution model, nor promote fast and portable algorithm implementations.

The Partitioned Global Address Space model is a more recent approach, which extends the DSM model by implementing a locality-aware paradigm [20]. In the PGAS model multiple SPMD processes share a part of their address space, like in the DSM one, however the global space is partitioned and a portion is local to each processor. In addition, PGAS implemen-

tation usually make distinction between local and remote memory references, permitting the allocation of local structures and explicit buffering. Hence, PGAS programs can exploit data locality, by having each process perform computation on the data located on its portion of the shared address space and using the local (non shared) space for temporary computation. Data structures can be allocated either locally or globally, and typically the distribution of global data structures is under control of the programmer. Programmers can access remote data using simple assignments or pointer dereference operations, the compiler and runtime support are then responsible for converting remote operations into lower level messages. Runtime libraries and language have been developed to support the PGAS memory model, most of the under the DARPA's High Productivity Computing Systems (HPCS) project[22] and are described in the following sections.

3.2.1 GASNet

GASNet [9] is a language-independent specification for a low-level networking layer. It provides communication primitives specifically designed for implementing parallel global address space in SPMD languages and libraries. The specification is partitioned in two layers to maximize portability. The lower layer (core) is implemented on top of the features of each network architecture. It provides an interface strongly inspired by Active Messaging (see subsection 3.1.2) and allows each process to register a memory segment for shared access. The upper layer (extended) provides APIs for remote memory access and various collective operations. It provides barrier-based synchronization, but no mutual exclusion. GASNet is mainly intended as a compilation target for PGAS languages or as a tool for developing runtime libraries, not for direct use. In addition to the specification, reference implementations are available for several widely used networks and communication interfaces, such as InfiniBand, UDP and MPI.

3.2.2 X10

X10 [54] is a parallel language developed since 2004 by IBM, as part of a project funded by the DARPA's HPCS program[22]. The language is object-oriented and derived from the Java language, however an extension of C is also available, called Habanero C. The main goal of the language was to permit a tenfold improvement in productivity when programming large scale, high-performance super-computing systems. The programming model adopted is called the Asynchronous, Partitioned Global Address Space (APGAS) model. It extends the standard serial programming

model with two core concepts: *places* and *asynchrony*. The memory is partitioned in places, each with one or more lightweight thread of execution. The concept of places can be naturally mapped to the nodes in a cluster, or to the single tiles in a tiled many-core system. Each thread can access its own memory, switch place, or spawn asynchronous threads. Data locality is exploited explicitly by using the places, while communication between places is performed when an activity is moved to a different place along with its context. Mutual exclusion is offered in the form of conditional critical regions, and synchronization is realized by grouping asynchronous activities with an implicit barrier.

3.2.3 Chapel

Chapel [16] is a portable and open source parallel programming language, designed and developed under the DARPA's HPCS program[22]. Chapel targets general parallel programming, without focus on a specific type of parallelism or granularity. It supports a multi-threaded execution model in which each process is composed of multiple concurrent *tasks*. Each task can create other tasks, permitting nested parallelism. The memory model used is the PGAS model, and data structures can be described at different levels of abstraction, using either a global or local view. The global view allows to compute on distributed data structures, using global indices, while the local view allows to exploit data locality and even use message passing. Synchronization is available in the form of a *sync* keyword that causes the task to wait for all the sub-tasks created within its dynamic scope. Chapel provide a lightweight mechanism for mutual exclusion through *synchronization variables*, inspired by the Cray MTA and XMT features (see section 3.3). Synchronization variables are like normal variables, but also have a *full/empty* state which is used by blocking read and write operations.

3.2.4 Global Arrays

The Global Arrays (GA) [48] toolkit is a portable API for providing shared memory programming interface on distributed-memory computers. It was created in the U.S. Pacific Northwest National Laboratory (PNNL) and has been in public domain since 1994. Global Arrays provides a logical shared view of physically distributed dense arrays and multi-dimensional arrays, with asynchronous one-sided access to shared data. It exposes the non uniform memory access of distributed systems to the programmer, but gives high flexibility in partitioning the data and provides APIs to obtain locality information dynamically. The execution model is the multiple instructions

multiple data (MIMD) model, compatible with Message Passing Interface (MPI). The programmer is free to mix both the shared-memory and the message-passing paradigms in the same program and take advantage of the underlying message passing libraries. The various processes can be synchronized using mutexes, fences and barriers.

3.3 Cray XMT

The high latency of Distributed Shared Memory (DSM) systems is several order of magnitudes higher than the latency of arithmetic instructions. This causes the processors of DSM systems to frequently stall waiting for data. The traditional solution to reduce the processor stalls is to use a hierarchical memory structure, with multiple levels of cache. However only the first level is actually capable of keeping up with the processor speed, while in large DSM systems almost all the memory references hits remote memories. The consequence is that conventional algorithms designed for systems with comparable memory and processor speeds are not applicable.

Cray multithreaded architectures MTA-1, MTA-2 and XMT proved that multithreading is an efficient mechanism for hiding memory latencies, rather than reducing them[28, 36]. When a thread is stalled waiting for a memory result a different one can be executed, keeping the processor busy. On the Cray XMT threads are lightweight objects mapped onto hardware streams. Each processor has 128 streams, each composed of 32 general purpose registers, a target register and a status word. In addition, 64 KBytes of instruction cache are available. On each clock cycle the processor selects one stream from those eligible for execution, in a fair manner. The processor stalls only when no stream has instructions ready. On the other hand, each thread can have a single instruction active inside the pipeline. The pipeline includes three functional units: the A unit performs arithmetic operations as well as bit operations and fused multiply-add, the C unit can execute either a control or an add operation, finally the M operation can issue a read or write operation. The clock speed is 500 MHz, producing a peak performance of 1.5G floating point and 500M memory operations per second.

The memory system of the Cray XMT is composed of commodity DDR components, logically arranged in a global, shared address space. The memory unit (M) in the processor can distribute and/or scramble virtual addresses when mapping them to physical ones. Certain memory regions are mapped to local addresses unscrambled, such as the code section, while most of the address space is both distributed and scrambled. The distribution and scrambling guarantee that memory references of patterns larger

than a single cache line are evenly spread across all network interfaces, memory controllers and banks.

The memory system also provide a fine grained synchronization mechanism, thanks to an extended memory semantic. To each memory word are associated several additional bits: a full/empty bit, a pointer forwarding bit and two trap bits. Pointer forwarding marks memory words which contains memory pointers, allowing to automatically generate new memory references to follow a pointer chain. Trap bits are used by the runtime for synchronization or workload distribution. The full/empty bits instead is used by blocking memory requests that are written in the form $operationXY$, where X is the state required to proceed with the operation and Y is the state after the operation. For example, the $readfe$ operation waits for a memory word to be full and atomically sets the bit empty. The extended memory semantic allows for extremely lightweight and fine grained thread synchronization.

The MTA-1 and MTA-2 were developed using expensive custom ASICs, making them difficult to sell. On the other hand, the XMT is built on the Cray XT3, a distributed memory systems which supports the MPI programming model, by replacing the AMD Opteron processors with the Threadstorm processors, greatly reducing the cost. However, the Threadstorm design is highly focused on irregular DSM applications, and cannot exploit the locality existing in more conventional algorithms. Also, the architectural mechanisms introduced in the processor to support massive multithreading pose a limit on the processor frequency and decreases the throughput of single threads. The reduced performance of applications implemented with the message passing model reduce the effectiveness of the XMT as a general purpose HPC system, and relegates it to graph processing applications.

3.4 Conclusions

The objective High Performance Computing systems is to execute complex applications on extremely large data sets at the highest possible performance. This require the use of multi-core and distributed systems, and consequently of parallel programming models, which are generally designed for regular and partitionable workloads. Multiple programming models and languages have been proposed to address graph-based applications with irregular memory patterns, mainly centered around the concept of a distributed memory space. Software approaches introduce an abstraction layer over the underlying architecture which allows for greater portability, but introduce overheads over bare execution. On the other hand, the cost of

hardware architectures, such the Cray XMT, and above all their limited flexibility greatly harms their commercial viability. For this reason there is a need for more cost-effective hardware architectures capable of executing both regular and irregular algorithms.

CHAPTER 4

Abstract Architecture

This chapters introduces the architecture template that has been designed in this research project to address the problems of efficient execution of irregular applications. We start by providing a high level overview of the system. Then, we delve into the details of the required features, and how the system provides them. The details given in this chapter are as much independent as possible from the specific features of existing commercial processors and devices. This allows the abstract architecture to have multiple concrete implementations, adapting to the specific components selected. Platform specific details are described in Chapter 5.

An overview of the proposed HPC system architecture is shown in Figure 4.1. The left part shows a cluster level view, while a detail of each node architecture is shown on the top right. Finally a zoom on each core is provided in the bottom right corner. At the higher level, the system consists in a cluster composed of many nodes, which are interconnected through a high performance network. The network topology and protocol used for the communication are not relevant for the architecture operation. The network detailed characteristics only affects the system performance and the parameters required to optimize the system configuration. Every node of the cluster includes a many-core system composed of many processing

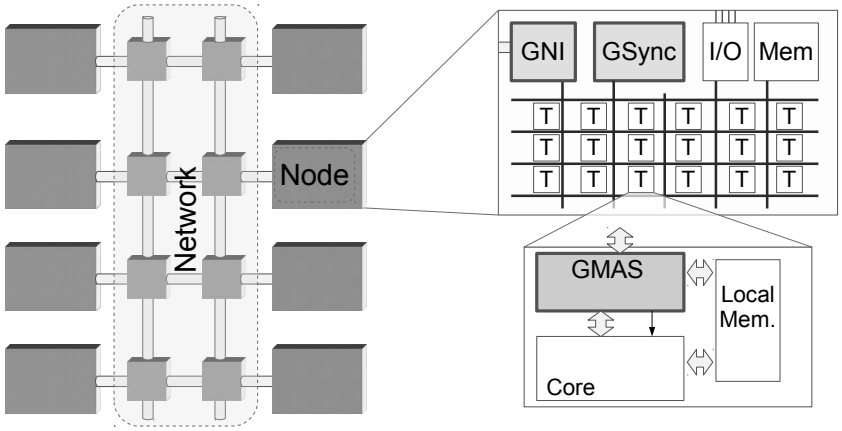


Figure 4.1: *Hierarchical overview of the architecture: whole system, single node and single tile detail*

tiles, a memory controller, peripheral devices and a network interface. All the tiles, labeled T Figure 4.1, are connected to each other and to the shared devices by a network, or equivalent communication subsystem. Each tile, in turn, includes a core, a local memory and a custom memory interface adapter. The private memory exists for performance reasons, and its purpose is storing in proximity of the core those portions of the application address space that shows locality of accesses, such as the code segment and the call stack. Hence, the abstract concept of local memory can be mapped either to a concrete private scratchpad, or to a private first level cache. The requirement of local memory components is not a real constraint, because they are already present in all existing multi-core and many-core architectures, in one of the two variants mentioned.

We have enhanced the base multi-core system with three on-chip custom components, highlighted in Fig. 4.1 with a bold border: the Global Memory Access Scheduler (GMAS), the Global Network Interface (GNI) and the Global Synchronization manager (GSync). These components jointly provides a set of features that enable efficient execution of irregular applications:

- a transparent global address space;
- reduction of dynamic hot-spots formation in the network and memories;
- hiding of network latency through multithreading;
- global, lightweight and fine-grained synchronization primitives.

The GMAS is a small component placed between the data port of each core and the on-chip network, in order to intercept all the data accesses and modify the to extend the memory address space and the semantic. The GMAS also connects to the interrupt port of the core, in order to trigger the execution of specific software routines. The GNI, instead, is a special network interface, shared among all the tiles in a node. It provides access to the remote sections of the global address space though memory mapping. Lastly, the GSync provides support for the synchronization primitives. It includes the structures required for fine-grained synchronization at the granularity of a single memory word.

All the blocks are designed with the constraint of requiring minimal changes to pre-existent components, such as processors, networks and memories. Thanks to this low impact on the system architecture, the components can be disabled at run-time when executing traditional HPC application with high computational intensity and data locality. This creates an architectural template capable of achieving good performance with irregular applications without degrading the performance of regular ones.

4.1 Global address space

The first feature offered by the proposed architecture is the hardware support for a global address space, shared across all nodes and application instances. A single, global address space allows to use simple programming paradigms based on shared memory, and simplifies the prototyping of new irregular applications.

To create the global address space, a part of the memory available in each node is reserved to be part of the global range. All these regions are then composed in a global address range, logically contiguous but physically distributed on the whole system. The hardware implementation of the global memory consists in exposing the whole global address space to each of the cores thanks to the GMAS, and in automatically forwarding remote memory requests to the corresponding nodes. Three different address spaces existing in the system are shown in Fig. 4.2. On the left, there is the address space visible to each core. This includes the eventual private memory space in the local memory, the memory mapped peripherals, the node-local memory range, and the system wide global memory range. The address space represented in the middle is composed of the physical addresses seen one the on-chip network. The difference with respect to the first space is that the virtual global memory space is replaced with the physical memory ranges of the components on which it is mapped onto, that is the DRAM

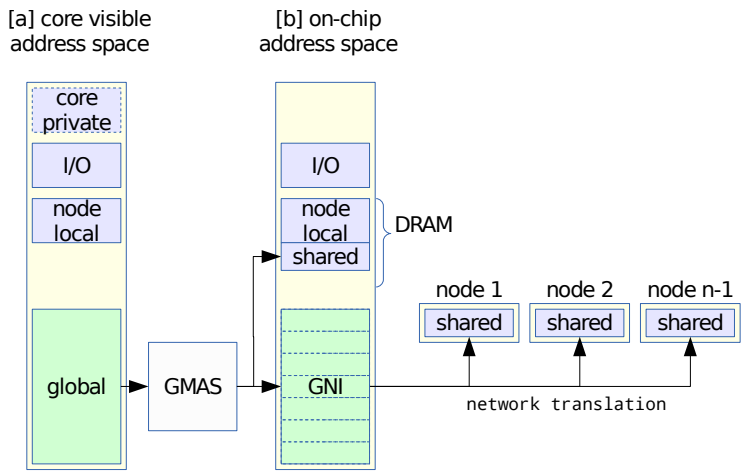


Figure 4.2: *Address Space Layout, as seen by the cores (a) and the on-chip network (b)*

modules and the GNI interface. A small part of the global address space is mapped into a reserved portion of the DRAM memory present in the node. The remaining addresses are remapped onto the memory mapped interface of the GNI. The mapping of the addresses is performed by the GMAS inserted between the data port of the cores and the on-chip communication network, by replacing the addresses in each data transaction. This allows to directly access to the portion of the global address space that is allocated locally in the node, with the smallest overhead possible. A second mapping function is included inside the GNI, which injects the remote memory portions in the on-chip address space and translates the memory addresses of the requests into network addresses.

The GNI acts as a transparent bridge between the node-local network and the system network, converting the transactions from one protocol to the other. Its simplified, high level, architecture is shown in Fig.4.4. The actions performed by the GNI are the decoding/encoding of memory transactions, their encapsulation inside network packets and the translation of addresses and network identifiers. The GNI handles outgoing and ingoing transactions concurrently, using two separate paths. Both categories of transaction are handled with a First Come First Served (FCFS) policy. Long latency requests are received by the GNI as memory mapped load/store transactions on its internal memory mapped interface. It extracts from the memory address both the identifier of the destination node and the physical memory address, used at the destination. Then, it creates a network packet that includes the requested operation and its arguments as payload. The

GNI also generates a globally unique transaction identifier, formed joining the address of its node to the internal transaction identifier used by the communication subsystem (also called transaction descriptor, or tag by different protocols). Hence, each packet includes the whole information required to route back the response and match it with the originating core. The GMAS connected to a core has complete information on the threads running on it, and can match each response with the corresponding thread. When a packet reach its destination, the GNI decodes incoming the request and converts it into a memory access towards the local DRAM, then sends the responses back to the originating core along path an analogous that followed by the request.

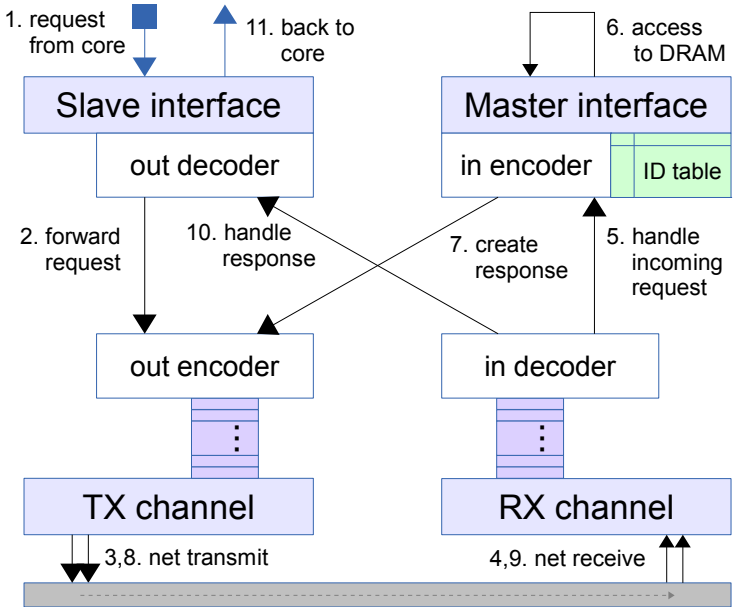


Figure 4.3: *High level structure of the GNI*

In the proposed architecture all the global load and store transactions are non-posted, i.e. both type of transaction require a response from the receiver. The consequence of requiring a response for store transactions is that they require a full round-trip over the network, as it happens for the load transactions, increasing the completion time. The reason for this decision is to guarantee sequential consistency in the memory model between the requests generated by each single application thread, in order to facilitate programming the system. The sequential consistency is specially critical between memory and synchronization operations, to ensure that at the end of the critical sections the correct memory state is visible by other

processors. Relaxing this coherency model in a distributed system could improve performance but would introduce complex and un-deterministic behaviors, nullifying one of the purposes of the proposed architecture that is reducing the development effort.

As previously said, the GMAS is placed between the cores and the on-chip network, hence it can remap a part of the global address space onto the local physical memory, allowing a direct access to it without passing through the GNI. In addition, the GMAS has the possibility of rewriting all the addresses of global memory operations, modify the control information associated to them and even modify the request types. This feature has been used to design the synchronization primitives presented later in this chapter. Also, it can be used to overcome the limitations of the pre-existing on-chip interconnection when designing an actual implementation of the abstract architecture.

4.1.1 Reducing hot-spot formation

One of the problems in the optimization of parallel and distributed systems is the run-time formation of hotspots, due to a temporary or permanent concentration of the requests or activities on few nodes of the system. This kind of concentration corresponds to a double drop in performance. First of all, the concentration is caused by uneven distribution of the data or unbalanced distribution of the workload, which means that part of the system is underused. Secondly, the hotsposts become the bottleneck and damper the scalability of the application. If a significant fraction of global memory accesses are directed to a single node, then the system performance is bound to the bandwidth of its network interface. The desired optimal behavior, instead, is to evenly distribute the accesses on all the nodes.

One way to achieve even distribution of the memory requests is to use the complex, non-linear mappings of memory addresses to nodes. The mapping must be designed to minimize the clashes caused by concurrent visits on the shared data structures. This approach has been used for optimizing parallel access to memory banks, e.g. in [63], and to distribute the accesses to the nodes of the Cray MTA and XMT supercomputers [7]. We followed the same approach, and added a hashing function inside the GMAS, which scrambles global memory addresses before mapping them on the system nodes. The function is configurable, and preserves very few last-significant bits, in order to distribute the addresses on the nodes with a very fine granularity, e.g. 64 bytes as the Cray XMT [28].

The hashing of memory addresses has the effect of breaking any struc-

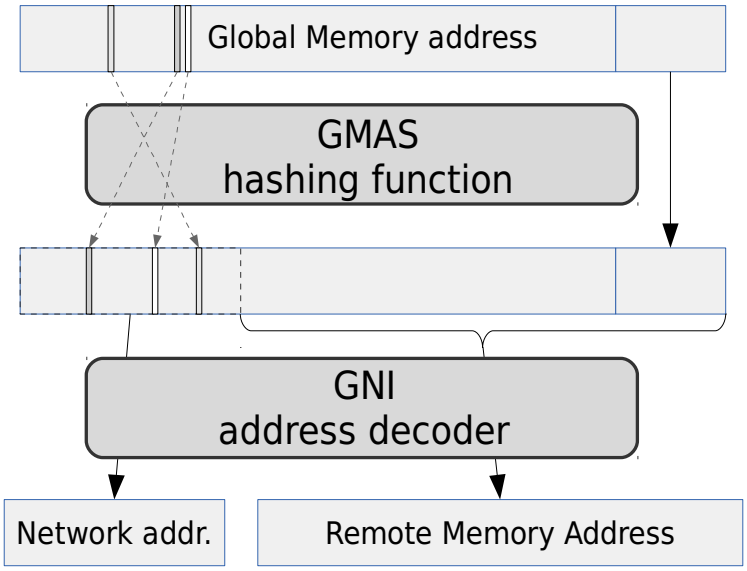


Figure 4.4: *Hashing and translation of global addresses.*

ture and variables which is larger than the hashing granularity, and distributing the accesses on the system. This is useful when during serial accesses to data structures such as the adjacency lists commonly used in graphs and trees. During those sequential scans, the probability of distributing subsequent requests on multiple nodes is increased by the scrambled address space, and as a result the the network traffic is more balanced. In addition, the hashing function allows to spread the data structures evenly on all the nodes in the system, independently on their size and without requiring an explicit configuration of the application [60].

4.2 Latency tolerance

In addition to providing the global address space, the GMAS includes various features to hide the long latency of network transactions through multithreading. The goal is to automatically exploit task-level parallelism to improve both the processor and network bandwidth utilization, by executing multiple threads while waiting for a response from the network. To achieve this, the GMAS performs the following actions when it identifies a remote request. First of all, it takes charge of waiting for the answer, internally keeping track of the pending status of the current thread. Then, it notifies the event to the core to start a context switch and, finally, it selects

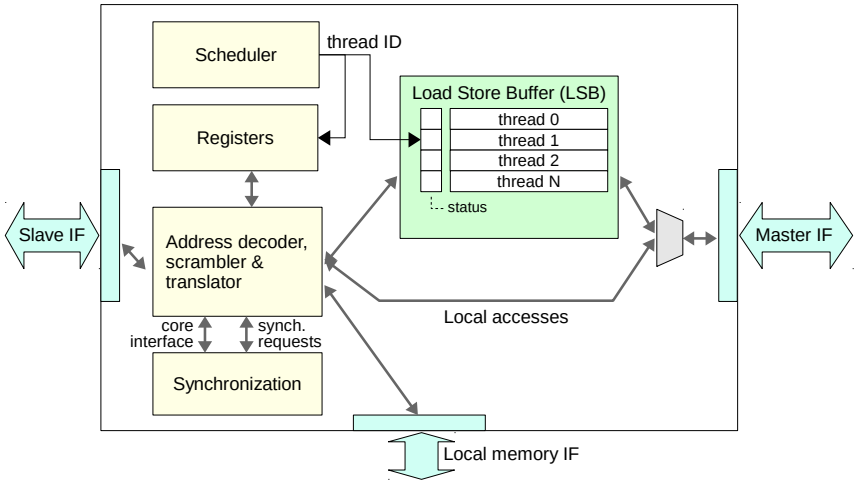


Figure 4.5: *GMAS internal structure*

the next thread to execute. These actions involve a set of hardware components inside the GMAS and of software routines for the context switch.

Figure 4.5 shows the internal structure of the GMAS. The largest block is the Load/Store Buffer (LSB), which stores information required to handle the remote requests. Each slot in the LSB corresponds to one of the threads that can run on the processor. The slots include a status bit and a field to store the result of remote requests when they are received. While a request for remote transaction is stored inside the LSB the GMAS also sends an interrupt to the core. In turn, the software interrupt service routine (ISR) associated to the GMAS asks the scheduler to perform a context switch. From that moment the core execution is decoupled from the network transaction, and the latency of the remote operations is hidden. The specific details on how the core pipeline handles an external interrupt and jump to the ISR vary from core to core. However, this mechanism allows to immediately identify long latency operations independently from the core characteristics, and without changing its internal implementation.

One reason for introducing the buffer inside the GMAS is to support the use of simple and low-power cores with in-order pipeline. The LSB provides the memory space to manage multiple pending memory requests even if the core does not include internal buffers because it is designed to execute a memory access at the time. Also, even in cores that support concurrent memory accesses, the buffer can be smaller than required to cover the very long latencies existing in distributed systems.

In many architectures the memory instructions that reach the memory access stage, or an equivalent stage, are not allowed to complete, preventing the call to the interrupt handler. This would prevent the core from activating the scheduler and performing the context switch to execute a different thread. We want our architecture design to support also cores with this limitation. Hence, in addition to the interrupt the GMAS also sends to the core a fake memory response. When restoring a thread context and exiting from the interrupt service routine, the program counter is set to the address of the last instruction executed, while normally execution resumes from the next address. This change allows the core to re-issue the memory request that triggered the switch. When the GMAS identify this new execution, the result is fetched from the LSB and returned to the core.

Another key component of the GMAS shown in Figure 4.5 is the hardware scheduler. The scheduler selects the next thread for execution after identifying a remote request, and exposes its ID to the software through one of the memory mapped registers. The GMAS always triggers the scheduler together with the interrupt signal sent to the core. Thus the scheduler latency is masked by the time required to save the current thread context. In addition, the hardware scheduler inside the GMAS does not require to access the main memory, because the statuses of the threads running on its core is stored in the LSB.

4.3 Global synchronization

Regular data structures, such as dense matrices, allow to identify most of the data dependencies at design or compile time. Conversely, algorithms based on arbitrary graphs and trees require run-time conflict avoidance, because the interference depends on the data instance. Therefore, the synchronization primitives offered by a system have a huge impact on both the applications complexity and performance. The natural granularity for managing concurrent accesses to graph structures is the single node. Thus, common APIs based on mutex variables, such as the POSIX Threads[33], incur in excessive overhead, both in term of memory size and time. But allocating a mutex for each node causes a huge increment in the size of the data structure. It is possible to reduce the number of mutexes while supporting the same granularity using software mechanisms, for example storing the addresses concurrently accessed in a shared look-up table. But software approaches increase the time for acquiring mutex ownership. Consequently, there is a need for hardware support for fine-grained synchronization, which permits precise control of the threads with low overheads.

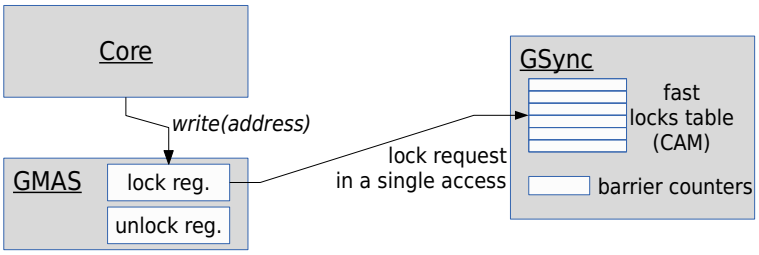


Figure 4.6: *Lock-based synchronization offered by GMAS and GSync components.*

One of the custom hardware components added to the system is the GSync, which includes the structures required to provide support for fine-grained synchronization. In order to adapt the abstract architecture to any existing instruction set architecture (ISA) and any on-chip communication subsystem, we decided to implement the synchronization interface using a set of memory mapped registers inside the GMAS. The application programming interface (API) exposed to the software applications defines three operations, $lock(address)$, $try\_lock(address)$ and $unlock(address)$, which respectively take ownership of a memory location, attempts to take ownership or releases it. The software can execute any of these operations by writing in a dedicated register. For a minimal implementation of the GMAS, with small area requirements, the $lock$ operation can be provided by the software run-time, using the hardware try_lock. The use of memory mapped registers makes the interface independent from the ISA, and allows using cores that do not provide instructions for mutual exclusion. The decision of placing the interface inside the GMAS, instead, allows to use global addresses in the synchronization operations, and to reuse the infrastructure for the remote operations introduced to provide the global address space. On the other hand, the GSync physical interface uses physical addresses, and its interface has to be adapted in the implementation.

Inside the GSync is located a fast, associative table that keeps track of the addresses locked by the application. The hardware implementation has two main benefits over software tables. First of all, using a hardware associative memory allows to greatly reduce the time required to check if an address is included in the locks table. In addition, since the lock table and the corresponding logic is enclosed in the GSync component, synchronization requests can be implemented using a single request across the on-chip interconnection and the system network. By requiring a single round trip, the GSync reduces the bandwidth used for synchronization purposes, and solves the problem of concurrent accesses by the multiple cores inside a

system node.

In addition to exclusive access to the shared data structures, parallel applications require also a mechanism for ordering the steps of the algorithms. One of the solution is the use of *barriers*, which force the threads/processes to stop until all other threads have reached the same barrier. The use of barriers in parallel algorithms is specially natural in data-parallel applications, when the parallelism is obtained by executing concurrently different iterations of a loop. An example is given in Algorithm 2 and 3. Dynamic barrier constructs can be implemented in software using the operations for mutual exclusion provided by the system. However performing multiple memory accesses across the network introduce a large latency, which increases with the diameter of the system network, reducing the effectiveness. For this reason the GSync component can be extended to add the barrier counters and the hardware logic to provide scalable barrier synchronization. Studies on efficient implementation of barrier synchronization on distributed systems has been studied extensively in the past, e.g. [44]. Hence this research work does not go in details about the low-level details required to maximize scalability and reduce overhead.

4.4 Programming and Execution Model

This section describes the programming model and execution flow of the designed architecture. Thanks to the custom hardware modules specifically designed for this architecture, the programming model can replicate the shared-memory semantic on top of a distributed memory system, with POSIX-like multithreading and lock-based synchronization. The applications are designed following the Single-Program-Multiple-Data (SPMD) paradigm, where every core of the system runs the same application with different inputs. At system startup an unique identifier is assigned to each core of the system, which is stored inside the GMAS fore quick retrieval. In addition, each node is assigned a sequential identifier in addition to its network address. These identifiers are exposed to the software run-time and to the application, to distinguish the different instances and distribute the workload across the system. When launching an application, a predefined master core initializes the global data structures used by the system run-time, while the remaining application instances are blocked waiting on a barrier.

As discussed in Section 4.2, software multithreading is used to tolerate the latency of remote memory accesses. Every time a remote memory reference is issued, the thread is automatically preempted and a different

thread is scheduled on the core. Alternatively, the threads can explicitly yield the core on which are run, if the algorithm includes a low-priority segment. Since irregular applications access the global memory very frequently, context switches caused by remote accesses are highly frequent. Therefore, the execution model does not include the concept of preemption based on periodic time slices, as it happens instead in traditional multi-threaded systems. It is still possible to add interval based preemption if needed by a specific implementation, by scheduling a periodic interrupt, but generally the mechanism is not required.

Communication among threads should take place only through the global address space, whether they run on different cores or on the same one. The portion of the memory in each node which is not used for the global address space can be used by the run-time or for optimizing the applications, e.g. storing local work-lists. Conflicts between threads on the data structure is prevented using the explicit *lock*, *try_lock* and *unlock* primitives, described in Section 4.3. The *barrier* routine is available to synchronize the algorithm execution across all the cores in the system or inside a single node. Since the synchronization happens in the global address space, it can generate remote requests and automatically trigger context switches. Moreover, every time a thread cannot acquire a lock it explicitly yields the core, in order to prevent deadlocks.

4.5 Conclusions

The abstract architecture described in this chapter provides the set of features ideal for the efficient execution of irregular applications, implemented with a simple programming model. The three custom components proposed can be introduced in existing architectures with minimal modifications, reducing the cost of the system and permitting the execution of traditional workloads with regular data structures. The remainder of this manuscript goes into the details required for implementing the architecture using existing commercial processors and presents the tools developed to assess the effectiveness of the approach.

CHAPTER 5

Architecture Implementation

The abstract system architecture designed during the research project was described in chapter 4. The abstract architecture includes the high level design of the components, and the generic structures and logic required to provide the features required for efficient execution of irregular applications. Part of the design is necessarily generic, because one of its goal is to be applicable to a wide range of existing distributed systems, composed of different processors and devices. The details in the actual implementation of the new custom components depends on the characteristics of the commodity components, so they have to be detailed in a later design step.

This chapter describes the details of a concrete prototype implementation on FPGA, then discusses possible implementation issues that can be encountered when using commercial multi-core processors and possible solutions. Finally, it presents a lightweight simulator that has been developed to assess the performance of hypothetical implementations without incurring in the cost of a full implementation.

5.1 FPGA Prototype

To evaluate a concrete design of the architecture we implemented a prototyping platform using multiple Field Programmable Gate Array (FPGA) devices. The decision of implementing a prototypes is due to the fact that FPGA systems can integrate a very large number of cores and I/O interfaces, hence they allow to emulate multi-core and many-core systems with lower execution times than achievable using simulators, and at the same time emulate all the low level details [62].

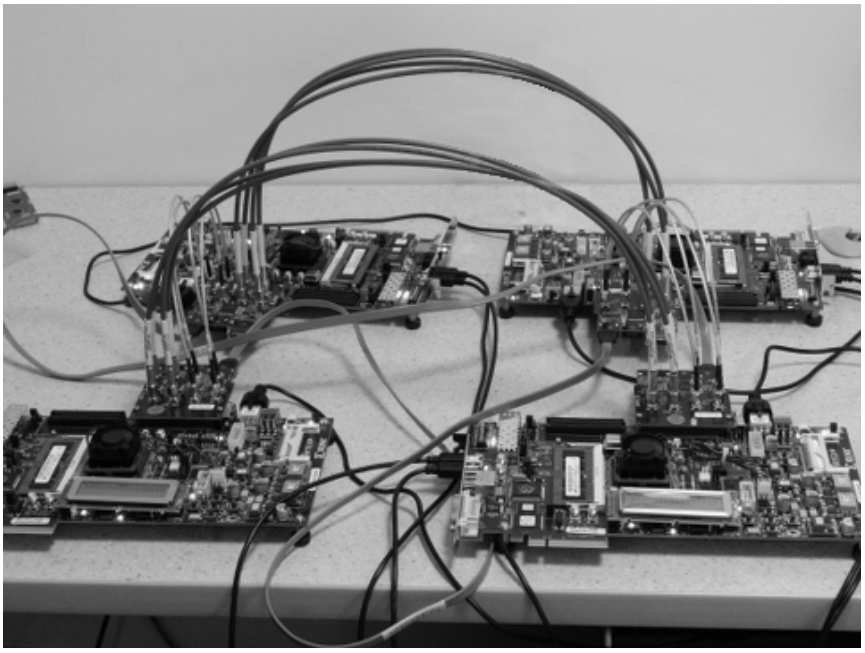


Figure 5.1: *System prototype, implemented with 4 Xilnx ML605 boards.*

The FPGA design has been implemented using 4 Xilinx ML605 boards, each mounting a LX240T Virtex-6 FPGA and representing a node of the distributed system. Since the ML605 board has a limited number of general purpose connectors, each board mounts an FMC XM104 daughter board[25], which provides 8 SMA connectors and 2 Serial ATA (SATA) connectors. The four boards communicate through a fully connected network, composed of six links. The eight SMA connectors in each daughter board are used to implement two full duplex links, with differential signaling. The third link in each board is realized using a SATA cable connected to the daughter board. Both SMA and SATA connectors are connected to identical RocketIO GTX transceivers inside the FPGA, which share a single

clock source. Hence, the only difference between the links is the physical medium. Communication between the boards is done using the Aurora protocol, a lightweight link-layer protocol developed by Xilinx for high-speed serial links [64].

The architecture of each node is composed of off the shelf building blocks provided by Xilinx, with the addition of custom implementations of the components presented in 4. The standard components used in the prototype are the ones distributed along with the Xilinx ISE Embedded Design Suite, version 13.4. The many-core node architecture is composed of multiple instances of the Xilinx MicroBlaze core[45]. The MicroBlaze is a 32-bit reduced instruction set computer (RISC) embedded processor, with in-order, single issue pipeline. Its architecture is simple but highly configurable. For the prototype we used a very compact configuration, in order to increase the number of cores in the FPGA, and consequently the number of concurrent hardware threads. Since the target benchmark are not computation intensive, we omitted both the floating point unit and the hardware divider. In addition, neither the optional memory management unit (MMU) nor the data cache are useful for irregular benchmarks, so are not included in the cores. The instruction caches, instead, have been included to eliminate the accesses to the main memory during the execution of the benchmark kernels. Finally, each core is connected to a private Block RAM (BRAM) memory, which is used by the applications as scratchpad, During the configuration of the FPGA, the BRAM is initialized with a minimal firmware, used for the system boot. With this configuration we were able to instantiate 32 cores on each node, using 30.9% of the available look-up tables (LUT)s.

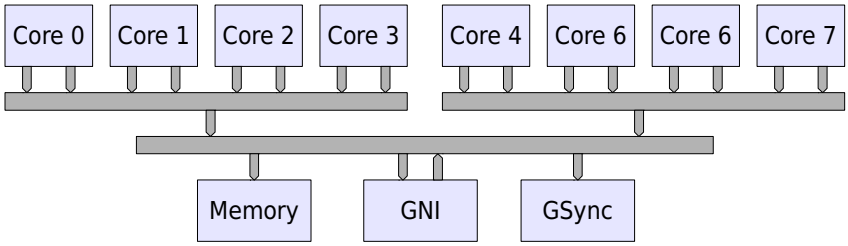


Figure 5.2: *Schema of the on chip communication subsystem, implemented using a two level hierarchy of AMBA AXI buses.*

The on-chip interconnect sub-system provided by Xilinx is the Advanced Microcontroller Bus Architecture (AMBA) AXI4 bus, a common interconnect solution for the design of systems-on-chip and embedded systems. The

Xilinx implementation of the bus supports up to 16 masters, along with split transactions, pipelined requests and burst messages [2, 40]. The platform also includes a DDR3 RAM controller which provide access to the node memory, a 512 MB SODIMM module. Finally, each node includes an UART controller and a hardware debug module (MDM) that can connect to up to 8 processors. Internally, the cores communicate with the memory controller, and the other devices through a two level hierarchy of AXI4 buses, as shown in Figure 5.2. The reason for adopting a bus hierarchy is that the Xilinx implementation of AXI supports only 16 masters on a single instance, not enough for a many-core architecture. Each top level bus is shared among 4 cores and has 8 master ports, corresponding to the instruction cache and data ports, and a single slave. Hence it is a degenerate 8-to-1 bus with simplified decoding logic. The bottom level connects all the top level buses to the shared devices: the DRAM memory, the Global Network Interface (GNI) and the GSync. The GNI acts both as master and slave on this lower level bus. The slave port is used by the cores to send remote requests, while the master port is used by the GNI to access the memory and GSync to execute the requests.

The clock frequency of the cores and other components in the FPGA is 100 MHz, and the GTX transceiver provides a bandwidth of 625 Mbit/s. Since our prototype is based on FPGA, its performance is limited by the low operating frequency for the processing cores, custom logic and the on- and off-chip interconnects. However, we downscaled the network bandwidth to match the reduced clock frequency of the FPGA, maintaining the relative performance of the various components coherent with those of commercial systems. Therefore, the downscaling of processor operating frequency and channel bandwidth does not invalidate the information obtained from the prototype on performance scaling.

The following sections describe implementation specific details of the custom components, and characteristics specific of the Xilinx building blocks which impact the architecture design.

5.1.1 GNI

As described in section 4.1, every packet sent across the network includes the address of the source node and an locally unique identifier that distinguish the thread which generated the request. In the prototype platform we use the AXI transaction IDs as local identifiers. During the synthesis of the architecture, the bus masters are enumerated and each receives a fixed numeric identifier, unique at node level. The architecture includes 33 bus

masters, divided into 32 cores and one GNI, hence at least 6 bits are required to represent the master IDs. In addition, the AXI protocol allows each master to extend its identifier with additional bits. The reason for the additional fields is to optionally permit out-of-order completion of the memory requests. This characteristic is useful for our architecture, because in a distributed system the latencies can vary widely. The out-of-order support allows to process the responses to remote accesses in the order they are received by the GMAS, and resume the respective threads. Therefore, the transaction ID generated by each GMAS is composed by its own fixed identifier, extended with the number of the current thread. This extended transaction ID, coupled with the source node address, can uniquely identify network transactions and are used to route the responses back.

Because of the way they are constructed, the total size of the address and identifier fields has to be large enough to number all the threads executed on the entire system. However, the AXI is a parallel bus, and the transaction IDs are implemented as additional physical links which contribute to the total system area. For this reason, the GNI has to store the network routing data inside an internal table. The table stores network information together with a local transaction identifier, used for accessing the AXI bus and uses the data to create the response packets. The size of the table is dimensioned to handle the peak request rate, according to the rate of the network interface, and.

Not all architecture implementations necessarily need a table in the GNI to match global and local identifiers. For example, a system which uses a custom on-chip network could directly include the global identifier inside the local packets. This other option would reduce the amount of memory required inside the GNI, but at the cost of increasing the overhead in the on-chip network. However, the approach used in the prototype implementation is generally preferable, because decouple the on-chip and system networks.

The most restrictive limit in the prototype GNI implementation is caused by the use of the AMBA AXI bus. The Xilinx implementation of the bus allows each slave component to define the maximum number of read and write operations that can process concurrently. When the limit is reached subsequent requests are blocked by the bus arbiter until the slave replies to the pending ones, moving the availability check from the slave to the bus itself. Unfortunately, the maximum limit configurable is 32 for each of the read and write channels. Since the GNI is forwarding local transactions across the network, only 32 remote operations of the same type can be performed concurrently, limiting the maximum bandwidth achievable. The effects of this limit are shown in the results of the experiments presented in

chapter 6.

5.1.2 GSync

The main component of the GSync prototype implementation is a lock table exposed on the on-chip address space. The table stores one bit for each entry of the table, which assumes the value 1 if the corresponding memory address is locked or 0 when the lock is not take. When the GSync receives a request for a lock it sets to 1 the corresponding entry and returns the previous value, implementing a *try_lock* semantic. If the returned value is 0 it means that the lock is successfully taken, otherwise it means that the lock was taken by a different thread. When instead the it receives an unlock request it sets the lock bit to 0.

Since the GSync is a device shared at node level, each access from the cores, or the corresponding GMAS, has to be atomic with respect to other bus masters. The version 4 of the AXI bus has removed the support for locked transactions previously existing, instead it supports paired exclusive read and write operations with the semantic of load-link and store-conditional. This kind of exclusive access is well suited to implement read-modify-write atomic operations such as test-and-set often used by the processors to atomically modify the memory. However, the requests to the GSync require to execute the accesses in the inverted order, in fact the core has to send (store) the address to the device and then read (load) the result of the operation. Therefore, the exclusive access mechanism provided by the AMBA AXI4 is not suitable for implementing atomic access to the GSync.

Instead of requiring atomic access support from the bus we mapped the two operations to simple memory accesses, which are by definition atomic when executed with the granularity of a memory word. The GMAS component transforms *try_lock* requests into read accesses to the GSync, which allow the GMAS to receive the previous status of the lock as read data. Instead, *unlock* requests are transformed into write accesses, because the do not require a response from the component. The GSync interprets the accesses to the memory mapped lock table as synchronization operations and performs the logic explained above. Remote requests are sent across the network as normal read or write transactions, using the memory address of the remote GSync components instead of the DRAM modules. Therefore, all *try_lock* and *unlock* requests are implemented using single transaction operations.

This implementation has two drawbacks. The first one is that only two

operations are possible, therefore the *lock* synchronization primitive is implemented as a software spin-lock using the hardware *try_lock*. After each failed attempt, the software routine has to explicitly yield the core to allow other threads to continue their execution, hence preventing deadlocks. But this means that each failed lock imply a whole context switch, in addition to the latency of the GSync access. The second drawback is that the address which is the argument of the synchronization operations has to be sent as part of a load request, which has no data payload. To achieve this, we exposed the lock table directly in the GSync address range, and introduced a direct mapping of memory addresses to table entries. The address mapping is performed by the GMAS, by inserting a part of the locked address in the address of the GSync request. Since the address range of the GSync is smaller than the address space of the local memory, multiple addresses share the same entry causing potential collisions. The false collisions do not invalidate the fundamental semantic of the single synchronization operations, rather they introduce unnecessary spins of the spin lock decreasing performance. However, deadlock are possible in presence of nested locks, when the arguments of the two locks collide on the same table entry. This limitation could be a hard constraint for an actual implementation of the architecture, however during the experimental evaluation we found that it was possible to implement our benchmark algorithms without dangerous nested locks. On the other hand, a more complete implementation would require to extend the AXI4 protocol specification or misuse its features, but the effort required is too high for a low-priority and platform specific activity.

5.1.3 Run Time

The prototype platform does not run a traditional operative system, but a thin run-time layer that provides basic functionality. The run-time layer has been specifically designed and implemented for the prototype platform, so its design depends on the features provide by the hardware components of the prototype.

One of the activities required for the correct functioning of the system is the assignment of unique identifiers to the cores and network addresses to the nodes, in order to identify the various components and coordinate the application execution. The core identifiers are statically determined during the synthesis of the FPGA design, and stored inside the corresponding GMAS components. Each core can read its own identifier from a memory mapped register inside the GMAS. In addition, each GMAS include a

register in which the run-time layer copies the network address of the node during the system boot, in order to provide quick access to it. The combination of network address and core IDs is used to compute a globally unique core identifier, used by the applications to distribute the workload. In order to simplify the implementation, also the network addresses are assigned statically. Since all the FPGAs are configured using the exact same configuration bit-stream, to assign different network addresses we used different configurations of the user DIP switch included on the ML605 boards. At start up the GNI reads the value of the DIP switch and copies it in a memory mapped register, which is available to the cores. The same register is used by the GNI router to route the network packets towards the correct outgoing link.

During the boot of the system, the GNI automatically initializes the network links, without requiring software intervention. In the meantime, each of the cores executes a routine stored inside its local BRAM memory, which clears the core status registers and the scheduler inside the GMAS, prepares the memory space for the thread stacks in the BRAM, and finally copies the node address in the corresponding GMAS register. After the network initialization is completed, a predefined core initializes the system data structures required by the run-time and allocated in the global address space. The remaining cores are forced to wait on a special barrier variable, which is allocated at a fixed address inside the global address space. The specific addresses of the initial barrier are not mapped onto one of the nodes memories, like all the other global addresses, but on memory mapped registers included in the GSync of the first node. These special registers are automatically initialized during the FPGA configuration, thus during the boot the barrier is already available for use by the various cores.

Because of the single program multiple data (SIMD) nature of the prototype platform, each core starts with one active thread and all of them executes the same application. Core identifiers and barrier based synchronization are used by the application to perform the serial sections, such as allocating and initializing the data structures. On the other hand, during the parallel sections the threads on each core participate in the execution. Also, each thread can spawn additional threads, which are allocated on the same core. The scheduling of active threads is performed by the hardware scheduler existing inside the GMAS components. In addition to the automatic preemption on remote requests the threads can yield the processor by writing the command in a GMAS register. The write operation makes the GMAS raise the interrupt signal, in the same way as if a remote request had been issued. The scheduling policy we decided to adopt is the round

robin. In detail, the scheduler is implemented using a sequential scan over the pending bits in the load store buffer of the GMAS, one cycle at a time until a ready thread is found or all the buffer slots have been considered. The maximum duration of the scheduling procedure depends on the number of LSB slots, which is 16 in the prototype. This simple implementation reduces to the minimum the complexity of the logic and the corresponding area. Still, it requires less time than needed to copy the 32 registers on the thread stack to save the context, hence the scheduler latency is completely hidden by the context save routine.

5.1.4 Area metrics

Table 5.1: *FPGA resources occupation and maximum frequency of the custom components.*

	# of slice registers	# of slice LUTs	$f_{max}[MHz]$
GMAS	1063 (0.4%)	1566 (1.0%)	230.3
GNI	450 (0.1%)	1915 (1.3%)	253.3
GSync	4711 (1.6%)	8054 (5.3%)	241.1
MicroBlaze	1510 (0.5%)	1457 (1.0%)	155.7
AXI4 (sum)	29694 (9.9%)	28868 (19.0%)	153.4

Table 5.1 shows the FPGA resource occupation and maximum operating frequencies of the three custom modules presented in chapter 4, and the two main building blocks from the Xilinx library, the MicroBlaze core, and the AXI4 bus. The data are obtained from a full synthesis of the prototype, with the percentage numbers relative to the Virtex-6 LX240T FPGA. Among the custom components, the GMAS occupation is the most critical, because the system hosts a GMAS module connected to each processing core, while each node has only one GNI and GSync modules. For the GSync, almost all the resources are used to implement the lock table, configured to have 4096 entries. Finally, the interconnection is also quite significant, consuming one fifth of the available LUTs. In terms of operating frequency, none of the custom components is critical.

5.2 Applicability to Commercial Processors

The prototype architecture we implemented is based on the building blocks provided by Xilinx, which are not designed for high performance systems. Namely they are the MicroBlaze core, which is a 32-bit scalar processor, and the AMBA AXI bus, which is used mainly in the design of embedded

systems. However, one of the claim of this thesis is that the approach used to design the abstract architecture can be applied to existing commercial processors and systems. For this reason, this section will describe in detail how the multithreading support can be implemented in architectures based on widely used processor families, such as Intel and ARM. The automatic preemption of the threads on remote accesses is one of the main features of the proposed architecture, but at the same time it is the only operation that strictly interacts with the processor internal structure. Hence, the applicability of the architecture design to the cited families of processors is of paramount importance, and shows the generality of the approach.

In the description of the multithreading support, in section 4.2, we stated that the GMAS component is connected to the interrupt port of the corresponding core. In fact, processors usually provide two similar mechanisms for interrupting the execution of a program or task: interrupts and exceptions. Exceptions are caused by the execution of an instruction or by one of its effects, such as unaligned memory accesses or memory faults. Interrupts, instead, are caused by external and asynchronous events, generally related to one of the system devices. The requirement for supporting automatic context switches on remote memory accesses is simply the existence of a mechanism for interrupting the application, whether it is in the form of interrupts or exceptions.

The choice between interrupts and exceptions depends on how a specific core handles the two classes of interruptions, and the possibility to apply a little modification to the internal architecture of the core. The MicroBlaze used in the prototype supports external interrupts, triggered by an external interrupt signal, and optionally supports exceptions caused by the application execution. The exceptions can be caused by the decoding logic (e.g. wrong operation code), the execution unit or by the optional memory management unit (MMU). But all the components responsible for exceptions are hidden inside the black box of the core, which is provided in binary form. Thus it is not possible to generate exceptions different from the existing ones, and interrupts are the only viable implementation option. The drawback of using interrupts is that the interrupt handler is called after the current instruction completes and updates the internal state of the core. As a result, the routine has to explicitly move back the instruction pointer, in order to re-execute the memory instruction after resuming the thread. In addition, the destination register of the memory instruction is replaced by a fake value, sent by the GMAS. Hence, when the instruction is executed again after the thread is resumed, the GMAS may not recognize this new memory access as the re-execution of the remote one, in case the register

used to compute the address has been overwritten by the fake value[1]. On the other hand, exceptions caused by the MMU, such as a memory fault, do not modify the register file, and automatically restore the program counter to the address of the instruction that caused the exception. Hence, memory exceptions would be the right mechanism to be used for providing automatic preemption on remote memory accesses. In order to use it, the only requirement would be the possibility to trigger memory exceptions from the external core interface.

In current commercial processors we find the same differences between external interrupts and exceptions described above for the MicroBlaze. In particular, the Intel software developer's manual[35] describes 3 kinds of exceptions: *faults*, *traps* and *aborts*. Faults are exceptions that can be corrected and allow the program execution to resume with no loss of continuity. The processor restores the core state to the state before the beginning of the faulting instruction, and the address passed to the exception handler is that of the faulting instruction instead of the next. Traps are similar to faults, but report the address of the instruction following the faulting one. Finally, aborts are caused by severe errors such as hardware errors, and do not allow for program continuation. With respect to interrupt, the only guarantee is that they are taken on an instruction boundary, but are not associated to a specific instruction. As for the MicroBlaze, the optimal implementation of the GMAS for an Intel processor would require to trigger a fault instead of an interrupt. Hence, logically the GMAS should be positioned between the internal memory unit and the first level cache. Actually, because of the similarities in the functionality provided, the GMAS could be implemented as an extension to the existing MMUs.

The operation of ARM processors with respect of application interruptions is the same as the Intel processors. According to the ARM architecture reference manual[1], in general faults are synchronous to the associated executing instructions, apart from few imprecise exceptions caused by system errors that are reported asynchronously to the instruction. Interrupts, instead, are always treated as events asynchronous to the application flow. Data memory access errors are provided among the causes for exception. Naturally, the same considerations done for Intel apply to ARM cores.

[1]This problem has been avoided in the prototype evaluation, by inspecting the benchmarks code to ensure that problematic instructions were not present.

5.3 Platform Simulation

Architecture prototyping has a lot of advantages over simulation, among which the higher fidelity of the results, the faster execution of the benchmarks, and last but not least the attention to details that can be overlooked in an abstract model. However, the process of designing, implementing and specially validating hardware components is more time consuming than writing software. In addition, many of the low level details of the prototype implementation may not bring useful insight on the overall flexibility and scalability of the architecture. Architecture simulation can solve both problems, by allowing the use of more productive software languages and development environments and by permitting to emulate and simplify uninteresting or low-impact details. In addition, platform simulators allow to emulate slow system activities such as accesses to the file system, greatly reducing the time with respect to full simulation. Still, simulation provides more accurate information than analytic models, because it can consider the actual dynamic behavior, including workload distribution and the conflicts on shared resources. A platform simulator has been added to the set of architecture realizations, for two reasons. The first is to provide an initial assessment of an extension to the synchronization mechanism, without dealing with all the low-level implementation details. The second reason is the ability to easily introduce multiple performance counters in the simulator and obtain detailed information regarding the system performance, even for the components that can not be modified in the prototype, like the cores.

The particular characteristics of the proposed distributed architecture pose a few requirements on the model used for simulation, and at the same time allow ignore some of the most detailed components modeled in common simulators. First of all, the target applications do not use data caches in the main application bodies. Eventual data caches can be used to speed up the run-time initialization, but the global address space bypasses the cache hierarchy. Hence, a big portion of the traditional performance models used in simulators for parallel architectures is superfluous, and should be disabled or removed in order to obtain correct simulation results. A second characteristic is that the very long latencies caused by global memory accesses are orders of magnitude higher than the latencies measured inside the cores pipelines. Therefore, the simulation of the control units or branch predictors has low priority, because the increased accuracy of the simulator is hardly worth the increase in the simulation times. Finally, the simulator should support not only shared memory multi-core and many-core ar-

chitectures, but also permit the definition of a tiered architecture in which different multi-core processors are connected using an off-chip network.

Many simulators for multi-core architectures have been proposed and released for public use. However, most of them are focused on accurate modeling of the existing traditional shared memory multi-core architectures, to provide precise timing, power or temperature metrics. Usually this kind of simulator is used to evaluate scheduling policies, voltage scaling or small scale architectural variations. To name a few, McPAT [39] provides an integrated timing and power model for accurate design space exploration of manycore processors. It includes detailed models for all fundamental components at circuit and technology level, but the level is too detailed for our needs, specially the cache hierarchy and does not support distributed systems. Graphite [46] is a parallel and distributed cycle accurate simulator, which aims to simulate manycore processors with thousands of cores using multiple machines. Sniper [11], based on Graphite, is focused providing fast and scalable execution by using high level abstract models and interval-based simulation. Hornet [52] is a parallel cycle-accurate simulator designed for studying networks-on-chip, which can run synthetic traffic or emulate a MIPS-based multicore. McSimA+ [5] is a cycle-level detailed timing simulator aimed at studying heterogeneous and asymmetric architectures, which offers full control on thread placement and management. Citing Ahn et al., all of these simulators have their own merits and serve their different purposes well. However, none of them was ready to be used in our research out of the box. The main problems were the necessity to remove the data cache simulation and to integrate the behavior of the custom hardware components. After considering the effort required to adapt an existing simulator we decided to develop a new, lightweight simulator tailored on the specific needs of our research. The following is a summary of the main characteristics:

- models the performance of a multi-core and distributed architecture, with globally shared memory, and automatic preemption on remote transactions.
- simulation is performed at application-level: only the application is executed in the simulator environment without requiring a whole operative system simulation. Any system activity required for managing I/O events or to spawn new threads is emulated and executed by the hosting operative system. The thread scheduling policy is the only exception, since multithreading support is one of the focal point of this study.

- decoupled functional and performance simulation: while a model of the distributed architectural is used to simulate the performance of the application execution, the functional simulation of core pipelines and and shared memory is replaced with direct execution on the host. Synchronization operations are a special case, because in their execution timing and functionality are intertwined.

- targets fast simulation over accuracy of secondary components, for early design evaluation and exploration.

- supports x86 application binaries.

Among the above cited simulators, Sniper was the one which matched the requirements more closely, for its high-level but accurate approach, and because it is not excessively focused on a specific technology or architectural aspect. Therefore its general structure and techniques have been used as a reference for our implementation. The high level structure and the interfaces of the simulation framework is shown in Figure 5.3. The two main blocks in the picture are the application and the simulator, which roughly correspond to the two aspects of the simulation process: functional behavior and timing. To simplify the development, the exact functional behavior is obtained by directly executed on the host machine a version of the application instrumented using the Pin tool [41]. This solution avoids the need for parsing binary executables and decoding complex instruction sets, like the x86 one. Also, native execution provides faster execution because it eliminates the need to interpret each single instruction. Also it permits not to replicate all the machine state information inside the simulator, such as the content of registers and memory. The drawback is that it prevents the cross-simulation of different instruction sets, but this limitation is not relevant for our research since x86 processors are widely used in HPC systems.

By instrumenting the application it is possible to send events to the performance model, obtain timing information and even modify both the timing and functional behavior of the application. Pin is an instrumentation system created and released by Intel, which uses dynamic just-in-time compilation to instrument x86 executables as they are running, inserting analysis routines provided by instrumentation tools called *pintools*. The resulting code is optimized using several techniques to reduce the instrumentation overhead, like inlining, register allocation and instruction scheduling. Using Pin it is possible to insert calls to the main simulator interface before each instruction of the application, obtaining a complete and detailed trace at instruction granularity. In addition it is also possible to replace entire

routines with different ones, to change or enrich their behavior. This functionality is used to emulate calls to the operative system, specially those regarding memory allocation and multithreading. It also permits to the application to query the size of the simulated architecture and obtain the virtual identifiers of threads, cores and nodes. In order to use the particular APIs offered by the architecture a thin library must be included in the application. When the application is run and instrumented by Pin, all the procedures of the library are replaced with calls to the simulator interface, which emulates their execution as explained in the abstract architecture description (chapter 4). If instead the application is executed without instrumentation, the library emulates the multithreading and synchronization APIs using internally the POSIX threads and mutexes, allowing quick test of the algorithms without the overhead of the performance simulation.

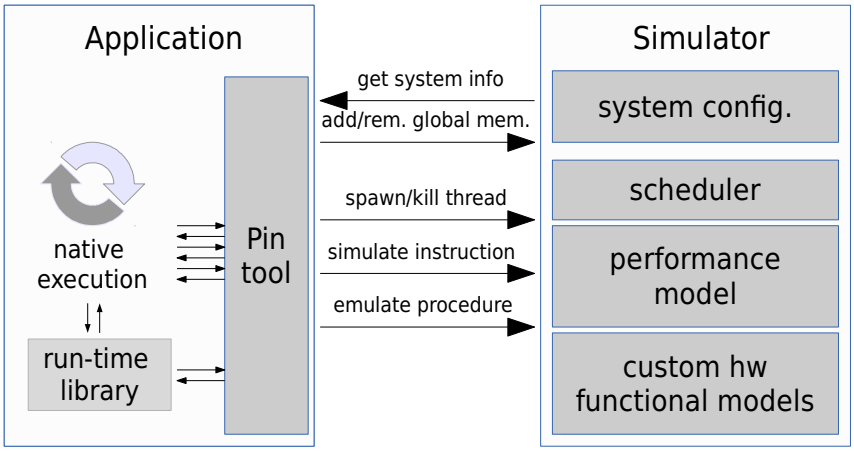


Figure 5.3: *High-level interface of the platform simulator.*

The main messages exchanged by the functional and performance parts are depicted in Figure 5.3. Allocation of data in the virtual global address space is performed using the function g_malloc, which behaves identically to the standard $malloc$. The only difference is that the values returned by the g_malloc are intercepted, and used to identify future accesses to the global structures. This approach allows to use the virtual memory space of the host system, without requiring an explicit customization of the application address space at compile time. In addition it allows the use of the standard $malloc$ routine to emulate allocation of data in the local memory. Global memory ranges are removed when the corresponding memory is deallocated by the application.

Thread management is performed using the POSIX Thread interface.

The thread creation and destruction events are intercepted, as well as the $pthread\_join$ procedure, to control the placement of the threads on the simulated cores and schedule their execution. In addition, the execution of the most common pthread API functions is ignored by the timing model, which replace them with fixed, configurable values. This allows to emulate lightweight tasks which creation and destruction is faster than the pthread, and also makes the timing model independent from the configuration of the host machine.

5.3.1 Performance Model

The execution of an application on a system with distributed memory involves latencies that range from a single clock cycle for simple arithmetic operations to thousands of cycles for remote memory accesses. This disparity makes a perfect accuracy of the short operations performed by the cores not necessary. For this reason the performance model is not cycle accurate, but uses various approximations to increase simulation speed. First of all, the simulation of control instructions and arithmetic instructions performed on registers are assumed to take a single clock cycle. To this cost it is added the latency of accesses to the resources shared by multiple cores: the memory, the GNI and the GSync components. Hence an instruction that increments the value in a local memory includes the cost of two memory accesses plus one cycle for the arithmetic unit. To each shared component is associated a fixed duration that is used to represent the latency required for an access in absence of contention. The durations are configurable at runtime to allow the exploration of the design space.

One of the main components of the timing model is the scheduler model. This component is responsible of assigning spawned threads to the cores, using a round robin policy. In addition, it blocks the threads when they are spawned by the host and allow only one thread per core to continue execution. When the GMAS identify a remote memory operation the scheduler blocks the current thread, selects the next one and resumes its execution, hence forcing a scheduling policy on top of the host scheduler. The same effect is obtained when the application explicitly calls the $yield$ procedure. At the same time, the scheduler module emulates the latency required for saving and restoring the thread contexts, forwarding the core time and updating the utilization statistics. As required by the abstract architecture, when a remote transaction is concluded the scheduler module is notified and puts the corresponding thread in the ready list. To match closely the prototype implementation, the scheduler uses a round robin policy over the

threads, according to the simulated time.

Some of the components integrate a contention model to emulate the increased time due to dynamic competition between the cores. The components are: the memory controller which provides access to the node DDR memory, and the GNI network interface. These are assumed the two components which are more likely to experiment high utilization in the system, a part from the cores. The dynamic latency of accesses to the memory controller is computed by measuring then dynamic contention and feeding the value to a queue model, which returns the expected wait time. The contention is measured counting the number of accesses performed inside a moving window, which slides forward in time as new accesses are modeled. The model adopted is taken from the queuing theory. More precisely the $M/G/1$ queue, written in Kendall's notation, which models a system with *M*arkovian request arrivals, service time with *G*eneral distribution and a single server. This queue model has the advantage of being solvable in closed form using the Pollaczek - Khinchin formula, and allows to use actual measured statistics for the service time instead of assuming an a-priori probability distribution. The model assumes that the requests behave like a Markovian process, with Poisson distribution of the arrivals, and does not guarantee correctness under different distributions. However one of the properties of the the Markovian process is that the interval between successive requests is independent from the previous ones. This property is very plausible for a multi-core processor in which different cores issue memory request indifferently from each other. In addition, queue models with Markovian arrivals are the only ones with simple closed-form solutions, usable for a quick estimation of the delays. The Pollaczek - Khinchin is given below, where W is the average waiting time, ρ is the utilization, λ is the rate of arrivals and x is a random variable which describes the time required to service them.

$$W = \frac{\lambda \cdot E(x^2)}{2(1-\rho)}$$

The contention on the GNI interface is actually simulated and not estimated. The main limit identified in the prototype implementation is not due to the rate at which it can serve the remote requests, but by a limit on the number of concurrent transactions, which is 32 per read and write operations each. At the best of our knowledge there are no closed form solutions for queue models with multiple servers, and in particular simple enough to replace the simulation. For this reason each GNI interface keeps a list composed of one time interval for each allowed transaction. Each

slot contains the start and finish times of one transaction performed. Each transaction is assumed to take a time equal to one network round trip time (RTT), after that time the slot in the list can be reused for a different one. When the GNI receives the request for a network transaction it identify the first available slot and stores the request time inside it. If the first slot available is subsequent to the arrival time, then the request is forwarded to the network adding a corresponding delay, and the slot is overwritten with the data of the new transaction. Transmission times are written in a timestamp inside network packets, which are used to model their progress across the network.

One of the issues in parallel simulation is how to maintain synchronized a global time between the simulated components. Event based simulators use queues to sort all the events and simulate them in order. In order to speed up simulation time, we added a timestamp inside each simulated core and allow them to proceed concurrently as long as the simulated operations have local effects. Since the host scheduler is free to schedule the threads, the core timestamps inevitably start to de-synchronize. Even phenomenons of starvation are possible, in which a simulated core receives a small portion of the host cpu time. For this reason, after a configurable period has elapsed all the cores are blocked on a synchronization barrier, which poses an upper bound to the de-synchronization of the cores local times. The configuration allow to select a trade-off between accuracy and simulation speed.

Global, system wide events such as global memory access are simulated out of order. For example, two threads mapped to two different cores can be executed sequentially by the host and perform global memory requests, and the simulated clock of the core simulated last can be previous to the clock of the other core. This reordering is accepted for various reasons. First of all, the timing simulation is decoupled from the functional one. In addition the exact ordering of global operations is not guarantee by the architecture model outside of synchronized sections, hence a correct implementation of the algorithms should not be invalidated by out of order timings of the memory accesses. Secondly, a reordering in the operations does not modify the total contention on the resources, hence introduce only small perturbations on the overall performance. In order to minimize the number of out-of-order global operations, the network operation is simulated using a dedicated thread which forwards the packets one at a time in order of timestamps. The same thread also simulates the activities performed by the GNI on reception, such as accessing the memory and creating response packets, and keeps these activities ordered with respect to the other network packets. However, since the cores execution is not syn-

chronized, global requests can be injected in the network subsystem out of order, after subsequent ones have already been processed by the simulator. By combining the best-effort delivery of network packets and the checkpoint synchronization between cores it is possible to arbitrary increase the simulation accuracy at cost of reduced simulation speed.

5.3.2 Architecture Extension

Thanks to the simulation interface it was possible to extend the architectural design and explore an alternative solution to the problem of race conditions during the modification of the shared data structures. The proposed solution based on variable locking is both complete and flexible. Every time a shared structure is modified it is possible to lock an address which represent it and obtain a critical section with mutual exclusion, independently from the number and kind of modifications performed. However, the extension of the lock protocol to a distribute address space increments the latency of the operations, and in turn increases the duration of critical sections, potentially causing a high number of conflicts. Figure 5.4 provides a schematic representation of a critical section involving one or more remote addresses. Each of the two synchronization operations require a whole round trip time (RTT). In addition, the critical section can include multiple memory accesses, each requiring a remote access. For example, to increment a variable two memory operations are required, which summed to the synchronization ones require four RTTs. This duration can correspond to thousands of clock cycles, and the problem is exacerbated by the fact that the average RTT grows with the network size. During this time other processors or threads can attempt to modify the same part of the data structure, and be forced to wait for the release of the lock. Hence it is interesting to evaluate alternative approaches, eventually limited to special but frequent use cases, in order to reduce the probability of contention.

Two approaches are adopted by modern multicore architectures, both involving an extension of the ISA. The first one is the definition of a set of atomic operations, which use low level hardware locking to provide exclusive access to a memory location during the execution of a single operation. The second one is the use of a paired set of memory operations, usually called *load-link/store-conditional*, which allow to optimistically execute an instruction without locks, and write the result into memory only if the destination address has not been modified in the meantime. To cite some example, the x86 ISA provides a $LOCK$ prefix that can be applied to a small set of instructions to turn them into atomic ones. The ARM, instead,

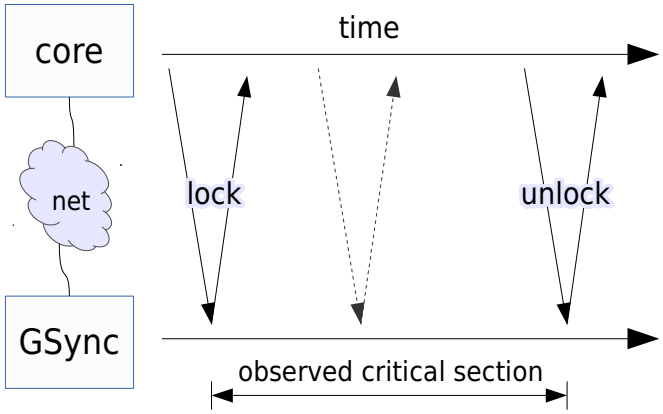


Figure 5.4: *Duration of a critical section involving remote memory addresses.*

has abandoned the used of bus locking in favor of the pair of conditional instructions $LDREX$ and $STREX$[1].

In the algorithms used for experimental evaluation the most frequent critical sections correspond to setting a single flag or increase a counter. Hence a small number of atomic operations could solve most of the need for mutual exclusion. In addition, the optimistic approach is less attractive because each failed attempt correspond to a significant amount of time wasted, due to the long latencies involved. For this reason the support for atomic operations has been integrated in the simulated architecture, as an optional feature. In particular, a single atomic operation has been provided which increments a single memory word, and is accessible by the application by calling a function offered by the run-time library. Functionally, the atomic increment is simulated by taking a global simulation lock before performing the increment, to prevent concurrent execution of atomic operations. The timing model, instead, simulates a single transaction across the network, with both the memory address and the increment value stored in the request packet. On reception of an atomic request the GNI models two accesses to the memory with once cycle in the middle for the increment, using the same latencies used to model the core functioning. This timing model assumes that the on-chip architecture already supports atomic operations performed by the processor, and permits the GNI to use the same mechanism. However, a similar approach can be applied also to ARM based architectures, by sending an atomic request over the network and using the GNI to perform the increment optimistically at the destination.

5.4 Conclusions

This chapter has two purposes. The first one is to address the possible issues encountered in concrete implementations of the abstract architecture, due to the specific characteristics of the processor used. A full featured prototype has been implemented on FPGA which demonstrates the feasibility and scalability of the approach. The prototype uses off-the-shelf components provided by Xilinx for both the processor and on-chip interconnection without modifications. The custom components do not limit the processor clock frequency, and add an area overhead comparable with the area of the MicroBlaze core in its smallest configuration. Furthermore, the chapter compares the two mechanisms existing for preempting threads, i.e. interrupts and faults, and analyzes the pros and cons of their use with commercial processors based on the two popular instruction sets x86 and ARM.

In addition, this chapter introduce a lightweight simulator which models the abstract architecture. Both the prototype and the simulated platform are used in the experiments described in the next chapter 6.

CHAPTER 6

Performance Evaluation

The experiments performed to evaluate the effectiveness of the proposed architecture are described in this chapter, and the results analyzed. Experimental evaluation has been performed using both the FPGA prototyping platform and the simulator, using a set of benchmarks commonly used to represent applications with irregular access patterns. One of the aims of the experiments is assessing the scalability of both the distributed global address space and the synchronization operations. In addition, the experiments analyze in detail the performance of the various components, and how they impact on the overall performance of the benchmarks. The experiments are preceded by the definition of an analytic model, which helps in identifying the correlation between the various performance metrics, and better understand the experimental results.

6.1 Performance model

To evaluate the effectiveness of the abstract architecture presented in this research project, we present a mathematical model that correlates the main architectural features with system level performance metrics. The use of the model provides interesting insights on the importance and impact of each

Table 6.1: *Definitions of terms used in the model*

	Hw parameters
N	Number of nodes in the cluster
C	Number cores on a node
f	Clock frequency [Hz]
D_{net}	Latency of a network request (average) [seconds]
D_m	Latency of a memory request (average) [cycles]
S	Size of network request (average) [bytes]
	Sw parameters
K_r	Ratio between remote requests and global requests
K_l	Ratio of global requests managed locally
T	Number of threads per core
I_{sw}	Number of processor cycles between global requests
D_{csw}	Context switch delay [cycles]

feature, and allows to estimate the utilization of the components to identify potential bottlenecks. The model uses asymptotic analysis to quickly provide bounds to the performance and evaluate different scenarios, neglecting low level details. Also, asymptotic analysis is accurate enough in situations with saturated resources, which is a desirable situation for HPC systems.

The execution time of data-irregular applications is essentially dictated by the frequent occurrence of long latency remote memory requests. Therefore, the model is focused on the memory and network subsystems, and on the number of accesses to the global address space. Internal details of the cores, such as branch prediction or instruction caches, are not explicitly modeled because their effects are minor. The model explanation starts with computing the latency of remote operations and the single-threaded throughput. We then proceed with modeling multi-core utilization and other system level performance metrics. All the definitions used in the model are listed in Table 6.1.

The reference system is a networked cluster composed of N nodes, each hosting a multi-core processor with C cores that run at frequency f. Part of the memory of each node is shared with the others, as part of a global address space. Every memory access to the global address space may be directed towards the local memory controller or forwarded to a remote note. We call the latencies (delays) of these accesses respectively D_m for memory, and D_{net} for network transactions. The values are considered as the average delays measured on the system, to simplify the model formulation. However, maximum values can be used to compute a pessimistic scenario. We introduce two parameters to represent the factions of remote and local accesses: K_r is the ratio of remote requests over all global memory ac-

cesses, and $K_l = 1 - K_r$ is the ratio of local ones. In the ideal situation of an evenly balanced system, the ratios depends on the number of nodes in the systems, and assume the values $K_r = \frac{N-1}{N}$ and $K_l = \frac{1}{N}$. The parameter K_r tends to 1 with increasing numbers of nodes, following a common hyperbolic law. As a reference, Table 6.2 shows the values of the two parameters for a range of system sizes. In large systems with 1024 nodes the percentage of local accesses is negligible, less than 0.1%, and even in medium sized systems they constitute a small minority of the total memory accesses.

Table 6.2: *Remote memory percentage wrt. system size*

Size	K_r	K_l
2	0,5000	0,5000
4	0,7500	0,2500
8	0,8750	0,1250
16	0,9375	0,0625
64	0,9844	0,0156
128	0,9922	0,0078
512	0,9980	0,0020
1024	0,9990	0,0010

The software side of the system is considered as composed of two main components: the run-time (system) support layer and the parallel application. With regard to the run-time system, the measure of interest is the demand on the CPU required to perform a context switch, D_{csw}, because it imposes a hard limit on the effectiveness of multithreading in hiding memory latencies. The application behavior, instead, is summarized by the number of threads scheduled on each core and the average number of clock cycles between consecutive global memory requests, respectively T and I_{sw}.

The overall system performance is dependent on the rate of global memory requests generated and injected on the network. The two asymptotic values for the memory rate generated by a single core correspond to the cases of low processor utilization and processor saturation. Two hypothetical schedules corresponding to those scenarios are depicted in Figure 6.1. When a thread (T1) issues a remote request, it remains blocked for a whole round trip over the net (D_{net}). In the meantime, the core switches to a different thread, which uses the CPU until it also issues a remote request, after I_{sw} cycles. In the first example the processor utilization is low because there are only two threads, hence the core stalls until the first response finally arrives. After receiving the response, the core restores the context of T1, requiring $\frac{1}{2}D_{csw}$ clock cycles. We assume that context switch can be

approximately divided in two equal portions, corresponding to the context save and restore phases. Additional overhead is either negligible or evenly divisible. Only half of the context switch time is computed because the context save is performed while waiting for the response. When a local memory access is performed, the core simply stalls for a time D_m, without switching the context. Accordingly to these observations, the set of threads $\{T1, T2\}$ is executed every $I_{sw} + K_r(D_{net} + \frac{D_{csw}}{2}) + K_l D_m$ cycles. The parameter I_{sw} is defined as the number of cycles between global accesses, hence each thread performs a single global access in the period just computed. Therefore, the number of accesses performed by a core is equal to the number of threads T. Hence, the asymptotic formula for computing the global access rate of a single core, in low utilization regiment, results in Equation 6.1.

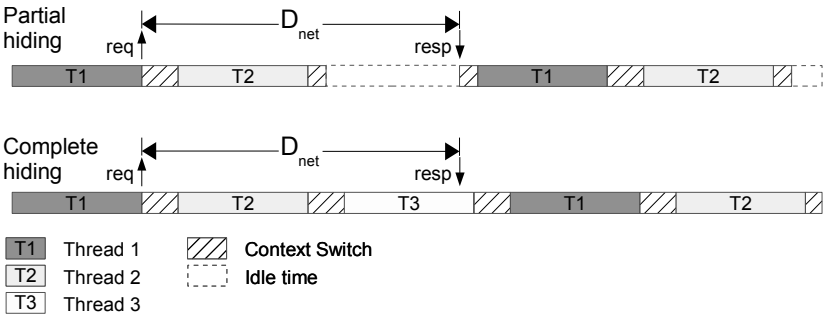


Figure 6.1: *Latency hiding through multithreading*

$$Rate_{low}^{glob} = \frac{T}{I_{sw} + K_r(D_{net} + \frac{D_{csw}}{2}) + K_l D_m} \tag{6.1}$$

In the scenario of a saturated processor, instead, by definition the core never stalls waiting for remote responses. Each time a core issues a global memory access, there is a probability K_l of it being stalled for a local memory access, and a probability K_r of executing a context switch in response to a remote request. However, it is never forced to wait for network transactions. Hence, in average each of the threads requires a total of $I_{sw} + K_r D_{csw} + K_l D_m$ clock cycles per each global memory request issued. Under fair scheduling, each thread gets an equal share of the core, hence each thread executes a global memory request every $N(I_{sw} + K_r D_{csw} + K_l D_m)$ cycles. This is also the total demand on the core for executing $N = N(K_r + K_l)$ global accesses. The resulting global access rate is given

by Equation 6.2, and is the maximum rate achievable by a single core. The value is dimensionless and expressed in references per clock cycles.

$$Rate_{sat}^{glob} = \frac{1}{I_{sw} + K_r \cdot D_{csw} + K_l D_m} \tag{6.2}$$

The intersection of Equation 6.1 and Equation 6.2 gives the number of threads required to completely hide the network latencies, maximize the network throughput and saturate the processor use. The parametric formula, is given in Equation 6.3, which is obtained by equating the response time of a single global request to its demand on the core, so that the core utilization is 100%.

$$T_{th} = \frac{I_{sw} + K_r(D_{net} + \frac{D_{csw}}{2}) + K_l D_m}{I_{sw} + K_r \cdot D_{csw} + K_l D_m} \tag{6.3}$$

In the equations proposed above, the network latency is a fixed value, not depending on the congestion of the network. This is acceptable for under-loaded networks, but is no more true when approaching the saturation point. Anyway, a second formula could be introduced to relate the delay of the traffic injected on the network with the latency. We do not provide an example because such formula depend on the details of specific network adopted. The resulting system composed of two equations can be solved to estimate the global access rate as in the case of fixed latency. Depending on the network, the equations can be solved symbolically, or require the use of iterative numeric solvers. However, as it can be seen from Equations 6.2 and 6.3, the maximum rate of global requests is independent from the network delay D_{net}, which only increases the number of threads required to achieve it. On the other hand, if network saturation has a significant impact on the effective delay, it means that the network itself is the bottleneck of the system and that remote latencies cannot be completely hidden with multithreading unless the network is improved.

Several system performance metrics depend on the single-core global memory access rate. First of all, the network bandwidth required by each node of the system on its interface. We assume that contention on the on-chip resources is negligible with respect to contention on the network interface, something possible by using high performance on-chip networks. Therefore, the throughput of a node is directly proportional to the number of cores C, up to network saturation. A fraction K_r of the global requests by each core is turned into a network message, each requiring in average S bytes. Even in this case we use the average value in the formulas. The

reason is that remote memory requests produce similarly small packet payloads, hence each packet can be approximated with the average one without the need to distinguish between request types. Equation 6.4 gives the asymptotic network bandwidth per-node, which is valid up to saturation of the network interface. The equation also can be reversed to compute the maximum rate as function of the system size, network bandwidth and message size.

$$Band_{node} = C \cdot f \cdot S \cdot K_r \cdot \begin{cases} Rate_{low}^{glob}, T < T_{th} \\ Rate_{sat}^{glob}, T \geq T_{th} \end{cases} \tag{6.4}$$

Similarly, it is possible to model the memory bandwidth utilization. The total memory bandwidth of the system is given by the number of nodes N multiplied by the bandwidth of a single controller. In the best-case scenario, the requests are evenly distributed among all the nodes. In this case, the demand on each memory controller is exactly equal to the request rate generated by the cores in a single node.

$$MemoryRate_{balanced} = C \cdot f \cdot \begin{cases} Rate_{low}^{glob}, T < T_{th} \\ Rate_{sat}^{glob}, T \geq T_{th} \end{cases} \tag{6.5}$$

Conversely, on an unbalanced system, a single memory controller can receive more requests than the others, potentially becoming the bottleneck. Anyway, in a large distributed system the majority of global accesses issued by the processors basses through the network, as previously shown in Table 6.2. For the same reason, the majority of the accesses to a memory controller arrives from the network interface instead of the local cores. Therefore, it is simple to determine whether the memory controllers are capable of sustaining the maximum traffic that can pass over the network, or whether memory is the possible bottleneck of the system.

A final interesting metric linked to the global access rate is the utilization of the cores. For each global memory access, the demand by the application on a core is given by the sum of the three components: the average computation time between global requests, the context switch caused by remote requests, and the stalls due to accesses to the local portion of the global address space. The components can also be labeled as user demand, system demand and idle time. The sum of the components is the same that at the denominator of Equation 6.2, which provides the maximum global

rate in case of processor saturation.

$$U_{cpu} = U_{cpu,user} + U_{cpu,sys} + U_{cpu,stall} \tag{6.6a}$$
$$= Rate_{glob} \cdot (I_{sw} + K_r D_{csw} + K_l D_m) \tag{6.6b}$$

6.2 Prototype Evaluation

This section presents the experimental evaluation of the architecture prototype described in section 5.1. We start the evaluation by measuring the latencies of the main system operations, including the use of the custom hardware components. Then, experimental data obtained from the prototype is used to validate the analytic model. Using both the experimental data and information provided by the model, we evaluate the prototype performance when running memory intensive and synchronization intensive benchmarks. The model is used to estimate metrics that were not directly measured on the prototype, such as the processor utilization.

6.2.1 System characterization

. The first step in evaluating the prototype performance is to measure the cost of its individual components. Table 6.3 reports the latencies of accesses to the main system components, as well as the duration of context switches. The data reported were obtained by measuring the duration of a tight software loop that performed 1 million accesses to the components, then removing the software overhead from the total. We measured all the latencies with a single core active, thus without contention on the on-chip bus hierarchy. Traversing the GMAS requires a single cycle for local memory accesses. On the other hand, remote accesses are forwarded to the GNI with a delay of 3 cycles. The processor receives a fake response within 1 cycle, so this overhead is not visible inside the processor. The 3 cycles delay is instead part of to the network round trip time, whole total is orders of magnitude higher. Be these considerations the overhead introduced by GMAS is very minor with respect to the other system times.

In addition to the experimental measurements performed on the prototype, we also measured the latency of a single access to the GNI interface using the ISE Simulator (ISIM) provided by Xilinx. Figure 6.2 display part of the control signals involved in the transaction. From top to bottom they are: the instruction opcode, the MicroBlaze data port, both the slave (S) and master (M) ports of the GMAS, used to forward the requests, the master port of the first AXI bus, placed between the two hierarchy levels, and

Table 6.3: *Latencies of the main operations.*

Action	Latency [clock ticks]
crossing GMAS	1 - 3
access to GNI	20
access to GSync	18
local memory access	28
remote memory access	505
context switch latency	213

finally the slave port (S) of the GNI, on which remote requests are received. For each port encountered by the transaction two signals are shown, respectively the address request (AR) and read response (R). As evident from the picture, the GMAS transparently forward the request to the bus and back to the core, while the bus hierarchy is responsible in large part for the transaction duration. The delay incurred at each stage is due to the presence of buffers at each port of the bus hierarchy, necessary to sustain 100MHz clock frequency. Indeed, FPGAs are somehow limited when implementing highly interconnected designs such as a many-core with a shared bus, which require a large number of long and fast connections. In spite of the relative long latency of local requests, the latency of remote transactions is 20 times higher, enough to validate the effectiveness of the architecture. Also, even in commercial systems accesses to the main memory are one or two order of magnitude slower than arithmetic instructions [38].

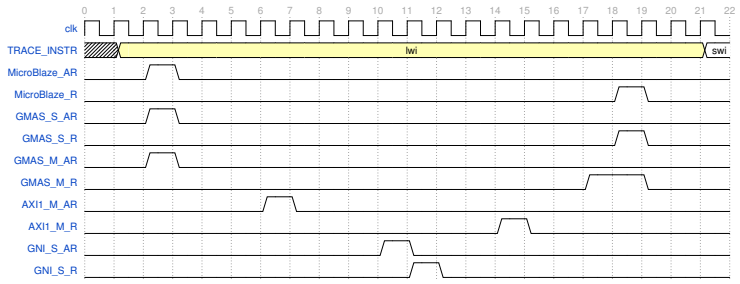


Figure 6.2: *Latency of an access to the GNI interface.*

The data obtained from profiling the prototype, used inside equations presented in section 6.1, provide the upper bound for the global of memory accesses generated by the system. The absolute maximum rate is obtained when every execute instruction is a memory access, without computation nor control, that is $I_{sw} = 0$ in Equation 6.2. In addition, applying the same assumption to Equation 6.3 gives the number of threads required to hide

network latency and saturate the cores utilization. Using the prototype latencies we obtain that the maximum rate per core is 600 thousand accesses per second, and 3 threads are enough to hide the latency. Therefore, the whole system composed of 4 nodes with 32 cores per node has a theoretical absolute maximum rate of 76.8 million references/second.

6.2.2 Pointer chasing

Some of the prototype metrics are hard to measure directly because related to black box components which do not provide the relevant counters, for example the utilization of the cores. Fortunately, these metrics can be computed from other measurable ones, using the relations presented in section 6.1. In order to validate the accuracy of the analytic model, its prediction has been compared with the performance measured on the prototype. The experiment has been performed using a pointer chasing benchmark, a basic irregular kernel that follows random chains of pointers stressing the global memory support. The kernel of the benchmark is presented in Algorithm 5.

Algorithm 5 Pointer chasing kernel

```
1:  function VISITLIST(list)
2:      if list.size = 0 then
3:          Return
4:      ptr = list.head
5:      while ptr ≠ end do
6:          ptr = ptr.next
7:  end function
8:  function MAIN(N)
9:      list ← createList()
10:     shuffleList(list)
11:     threads ← []
12:     for i ← 1, N do
13:         threads[i] ← threadSpawn(vistiList, list)
14:     for i ← 1, N do
15:         threadJoin(threads[i])
16: end function
```

The benchmark initially creates an array in the global address space, composed of 2^{20} elements, each containing a pointer. The pointers are linked to each other forming a single pointer chain that touches all the array elements in random order. The array is randomized using an algorithm inspired by the *riffle* technique commonly used to shuffle cards. At the beginning each pointer of the array is set to point to the next element, with the

last element pointing to the *NULL* value. The a number of shuffle steps is performed, equal to the logarithm of the array size. In each steps two empty lists are created, labeled left and right. Then list produced by the previous step is iterated over, and the elements are appended to one of the two lists, and the decision is taken randomly with equal probability. After iterating over the whole list, the *right* list is appended to the *left* one. Hence, each shuffle step maintains the property of connecting all the array items in the list, without introducing cycles or skipping elements. After a single step the array contains many short forward jumps, and a single backward jump, as shown in Figure 6.3. But each successive step introduce more backward jumps and increases the randomness. The result is similar to the execution of the *riffle* shuffle, because at each step the elements belonging to the second half of the list maintains their relative positions, but are randomly intermixed to the elements of the first half.

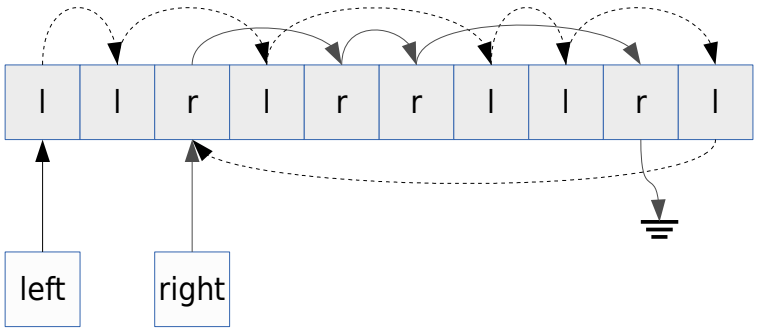


Figure 6.3: *The pointers array after one pass of the shuffling algorithm.*

The core of the kernel spawns a configurable number of threads that iterates over the whole list. This iteration results in a highly irregular access pattern to the memory, characterized by continuous jumps with random direction and distance. The implementation of the kernel performs a small number of accesses to the stack and control operations between consecutive global requests, corresponding to $I_{sw} = 67$ clock cycles. This benchmark requires no synchronization, apart from the final barrier used to ensure the termination on all the cores. In addition, thanks to the scrambled memory space and the single access to each array element, the memory accesses are perfectly distributed on the memories. Also, since every thread performs exactly the same sequence of operations the processor utilization is homogeneous. Therefore, pointer chasing execution is the closest possible to the perfectly balanced situation hypothesized in the analytic model.

Figure 6.4 shows the rate of remote requests generated by each single

node. The benchmark has been executed varying the number of cores used and the number of threads spawned on each core. The darker dots in the chart correspond to the prototype performance, while the lines and lighter dots are the estimation obtained with the model. As is visible, the two sets overlap in most of the configurations, proving that the model is highly accurate. The maximum difference between the modeled and experimental performance is 6.1%, with an average of 2.3%. Thanks to this accuracy it is possible to use the model to get reasonable insights on the unmeasured internal operation of prototype.

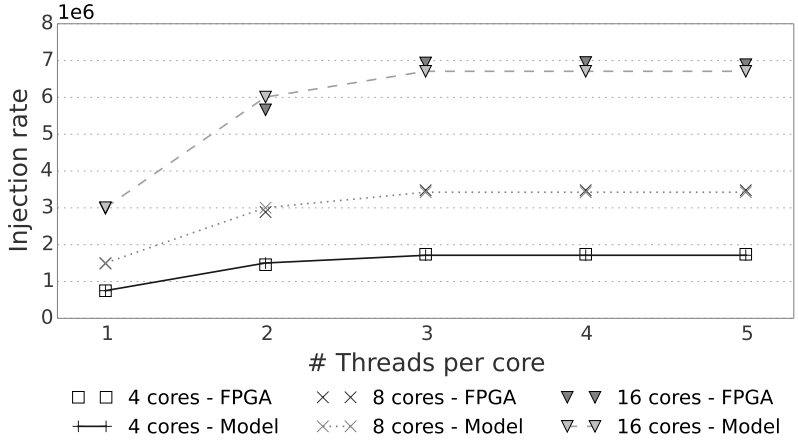


Figure 6.4: *Injection rate per node vs. number of threads per core, for different number of cores per node*

The pointer chasing benchmark is not only useful to validate the model. It also allows to evaluate the maximum performance scaling provided by the prototype, since it does not include mutual exclusion or synchronization points. The performance scaling exposed by the benchmark is a weak form, because each thread visits the entire linked list hence the problem size increases with the number of executors.

As shown in Figure 6.4, the throughput increases linearly from one to two threads, then reaches the maximum at three threads per core as predicted by the model. Adding further threads after the saturating the cores does not increase the performance, but does not even decrease it. This behavior is due to the thread stacks placed inside the private scratchpad memory attached to each core, instead of using a shared cache hierarchy. The threads mapped to a processor only contend on the internal pipeline, not on the memory, hence spawning more threads than needed has no negative

effect. On the other hand, to keep all the thread stacks in the scratchpad memories it is necessary to put a hard constraint on the number of threads. For the prototype evaluation this constraint is not a limiting factor, because the low diameter of the network allows to hide the latency and maximize core utilization by using just 3 threads. The speed up measured is linear also with respect to the number of cores. In fact, the 16 cores configuration has a speedup of 3.96 over the 4 core one, only 1% less then the linear scaling. This confirms the assumption that the on-chip bus is not among the bottlenecks in the prototype.

Using the analytic model it is possible to obtain an insight on the utilization of the individual components of the system. By analyzing the pointer chasing benchmark we identify the bottlenecks encountered when running a perfectly parallel and memory bounded algorithm, which lack synchronization and has very little computation. Figure 6.5 shows the utilization of four components: the core, the GNI internal interface, the network links and the DRAM memory module. The actual limit on the GNI interface is not given by its maximum sustainable throughput but by the limit of 32 pending operations enforced by the Xilinx implementation of the AXI4 bus specification. Since the network round trip time is 505 clock cycles, in average one remote access is routed every 15 cycles, while the bus interface accepts a much higher rate.

The processor utilization, shown as the first bin in Figure 6.5, is divided in three parts: the time used by the application, the time required by the system to switch contexts, and the stalls due to accesses to the local memory. This allows to quantify the overhead of the runtime layer. The configurations represented use from 1 to 3 threads and 8 or 16 cores. The remaining configurations have been omitted because less interesting. In fact, four or five threads have the same behavior as three, and the behavior of the four cores configuration is simply scaled with respect to the eight cores. With 8 cores per node the performance is limited by the saturation of the cores, when three threads are spawned, while memory utilization caps at only 16%. In the same configuration the average number of network transactions is 16.3, approximately half the maximum permitted by the bus. In the 16 cores configuration, instead, the number of pending operations in the GNI already approaches the limit with just 2 threads. The third thread allows to saturate both the cores and the GNI interface, resulting in the best overall system utilization. With regard to cores utilization, the majority of the time is spent on system code. In fact, when the core is saturated only 28.7% of the time is spent on the benchmark execution, while 68% goes in context switches. Anyway, even if the overhead of multithreading is high,

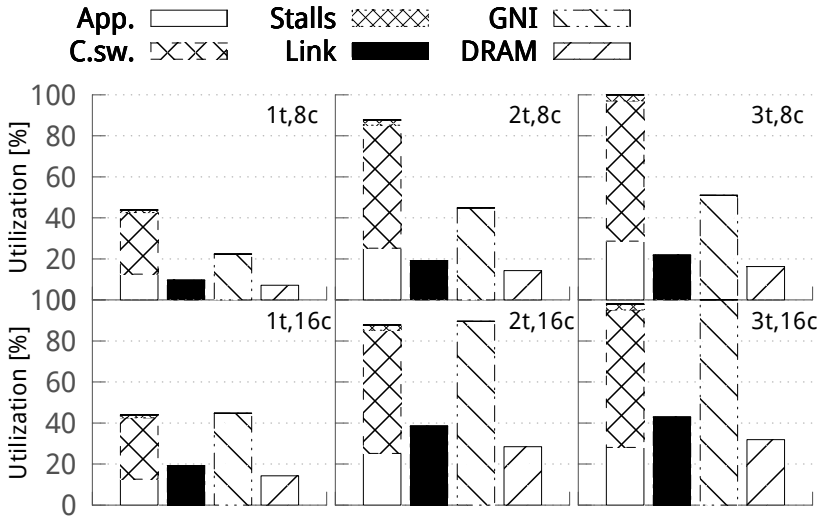


Figure 6.5: *Utilization of system components by the pointer chasing, on different configurations of threads and cores. CPU utilization is split into application time, context switch and stalls.*

it allows to exploit part of the time required for remote requests, increasing the user utilization from 12.6% to 28.7%.

6.2.3 Breadth First Search.

As introduced in section 2.2, the Breadth First Search (BFS) is a typical irregular algorithm used to evaluate HPC systems [31]. This computation kernel is widely used, because many problems in graph theory can be solved by an ordered visit of a graph or by algorithms based on it. Examples are finding the shortest path between two vertices, testing a graph for bipartiteness, or computing centrality measures that indicate the importance of a node in a network. We ported to the prototype APIs a version of the BFS algorithm originally designed for the Cray XMT [8], adapting it to the different synchronization and threading interfaces. The pseudo-code implementation is shown in Algorithm 6.

In the experimental evaluation the graph explored has 100.000 nodes. The graph generation algorithm selects the number of neighbors of each node randomly, using a uniform distribution between 1 and 80. The destination of each edge is also selected randomly with uniform distribution. All the repetitions use the same graph, which contains 3.998.706 edges and has diameter 6. Because of the uniform distribution of the outward edges, the

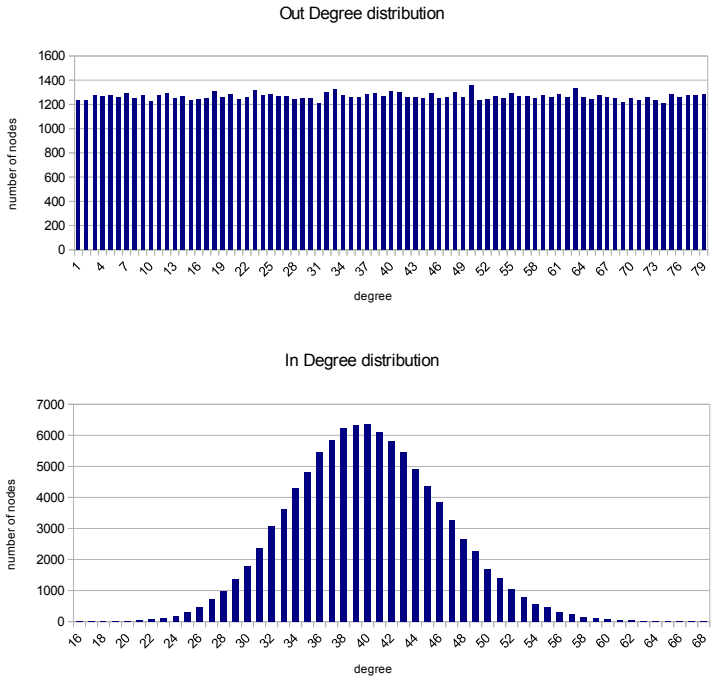


Figure 6.6: *In- and Out-degree distributions of the graph used in the BFS experiments.*

distribution of the indegree follows a bell-shaped distribution, with mode 40. The two distributions are shown in Figure 6.6.

The breadth first search is not only a memory intensive algorithm, but also synchronization intensive. In fact, it has two critical sections in a small loop body. The first one prevents different threads to concurrently test and set the visited flag of the same vertex. The second is required to insert the newly visited vertices in the shared work-list. Therefore, this benchmark significantly stresses the components providing synchronization support.

First of all, we try to estimate the throughput using a model similar to the one described in section 6.1. We modified the model to computes the rate in terms of edge visits, instead of memory requests generated. In the new model the parameter I_{sw} identifies the average number of instructions executed per edge, and the number of memory requests and context switches are scaled accordingly. This model takes into account only part of the synchronization effects: it includes the lock and unlock operations in the total number of operations performed on the global address space. However, it ignores the run-time contention existing on the locks. Figure 6.7.a shows the model prediction, which has a behavior very similar to pointer chasing,

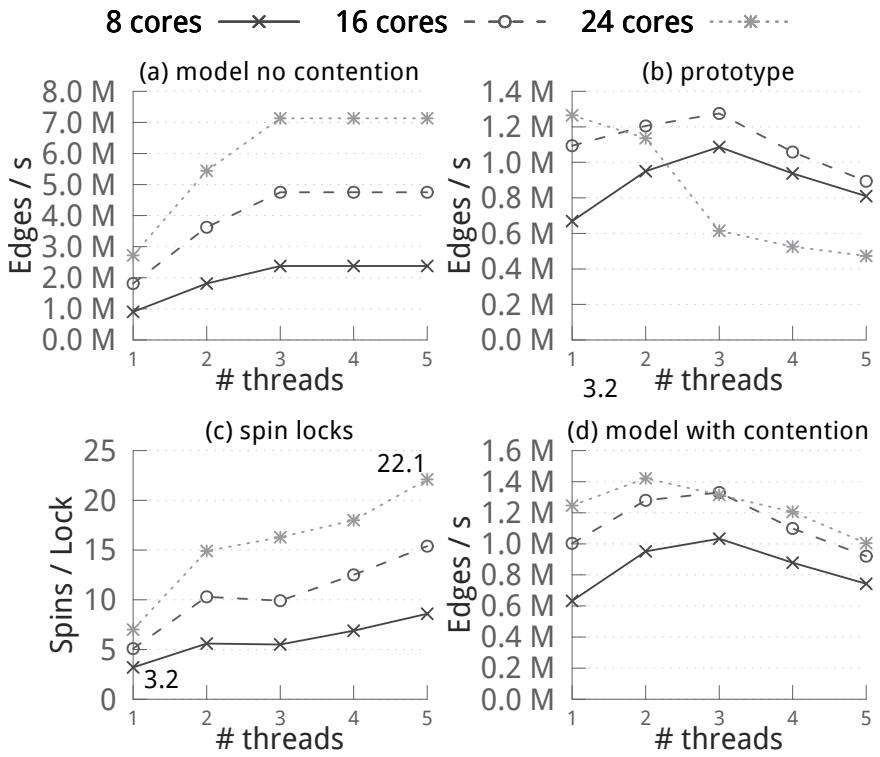


Figure 6.7: *Performance of the BFS benchmark from a synchronization-less model, on the prototype, and synchronization aware model*

characterized by core saturation at 3 threads per core, and a peak of 7M edges visited per second with 24 cores.

Fig. 6.7.b, instead, shows the performance obtained running the benchmark on the prototype. First of all, the execution speed on the prototype is reduced by almost an order of magnitude with respect to the model, in particular on configurations with more parallelism. The system with 8 cores and 1 thread per core is the most similar to the model, with a performance only 1.4 times slower than estimated. On the other hand, the configuration with 5 threads on each of 32 cores is 15 times slower. Secondly, adding threads to an already saturated system has a huge negative effect on the prototype performance, while the modeled performance remains constant. These two differences are due to the overhead caused by the contention on the critical sections.

The details of the overhead caused by the synchronization are shown in Figure 6.7.c, which presents the average number of spins required for ob-

taining a lock, measured in the prototype. Even in the small configurations, each lock needs to spin between 3 and 7 times before taking ownership, and the number increases with the number of threads used. The main cause is the insertions to the shared work-list that stores the nodes to be visited by the algorithm in the next phase. Each insertion in the list is a critical section that takes approximately 2000 clock cycles ($20\mu s$) because of multiple remote requests for both data and synchronization. The probability of contention during this long interval is significant, because many threads are trying to enter it with high frequency, specially in the central phases of the execution when most of the nodes are visited. Also, this overhead grows with the number of threads because of two trends. When the system is underused, it behaves accordingly to a modified Amdahl's law [24] that accounts for serialized critical sections. Because of the serialization, the waiting time grows linearly with the number of threads, limiting the speedup. In the prototype, this is demonstrated by the increasing number of spins on the locks from 1 to 3 threads. The saturation of the cores adds a second effect. When all the core time is used, the ratio available to each thread shrinks with the addition of others. Consequently, the time required to complete a critical section increases. This second effect is evident in Fig. 6.7.b and Fig. 6.7.c, when more than 3 threads per core are used.

We extended the model to take into account the number of lock attempts due to contention, obtaining a better approximation shown in Fig. 6.7.d. In this version, the number of global transactions per edge is increased with the number of cores and threads used. Hence, the model does not distinguish single critical sections, but captures the resulting average overhead. Even so, the accuracy is greatly improved, and captures the same trends observed on the prototype. This hints that the main limiting factor in the proposed architecture is the overhead due to the polled implementation of the lock primitive and the corresponding context switches caused by the synchronization conflicts. This is also demonstrated by the utilization of the components, shown in Table 6.4.

When running the BFS, the core utilization is much higher than the other components with respect to the pointer chasing benchmark. However, only 4% of the processor time is due to the application execution. The remaining time is consumed by the context switches that are forced by every *try_lock*, even when the request fails. In column $SyncBW$ we show the percentage of network utilization due to synchronization requests. In the best case, it equals the bandwidth used by memory accesses, but increases with the number of threads. This suggests that the most effective improvement to the architecture would be a hardware implementation of the spin lock.

Table 6.4: *BFS system utilization.*

Cores, Threads	utilization [%] CPU Total	CPU App.	Network Link	Network GNI	Network SyncBW	Memory DDR
8, 1	43.5	2.6	1.0	2.4	51.1	7.2
8, 2	93.2	3.9	1.5	3.5	62.0	13.6
8, 3	100.0	4.2	1.7	3.9	61.8	14.6
8, 4	100.0	3.6	1.4	3.3	66.3	13.9
16, 1	46.1	2.0	1.6	3.7	60.3	13.7
16, 2	99.3	2.6	2.0	4.8	73.7	25.6
16, 3	100.0	2.7	2.1	5.0	73.1	26.0
16, 4	100.0	2.2	1.8	4.1	77.0	24.9

6.2.4 SSCA#2.

Another benchmark representative of graph theory computational kernels is SSCA#2 [53]. It generates scale-free graphs, using a generator algorithm based on R-MAT [15]. Scale-free graphs, or networks, are characterized by a degree distribution that follows a power law, that is the fraction of nodes with k connections is $P(k) \sim k^{-\gamma}$. This distribution is interesting because is found in many networks such as the world wide web links, biological networks and social networks.

The SSCA#2 benchmark is composed of 4 kernels. The first one is the graph construction. It creates a sparse graph starting from a list of tuples, each representing an edge of the graph with start vertex, end vertex and edge weight. The reference implementation stores the graph data using 4 arrays: the first one is the list of vertices, which stores an integer label for each of them. The second array represents the edges, and stores the destination vertex for each one. The edges are listed grouped and sorted by the source vertex label. The third array stores the position in the edge list of the first neighbor of each vertex, and finally the last array contains the weights of the edges. The 2nd kernel visits all the edges of the graph to identify the set composed by the edges with maximum weight. The 3rd kernel extracts one sub-graph for each edge identified by the second kernel. The sub-graphs include the two extremes of the edge, and all the vertices that are reachable by the edge end vertex within a fixed number of steps. One possible implementation is the execution of one Breadth First Search per each sub-graph. Finally, the 4th kernel is the identification of the vertices with highest betweenness centrality score [26]. The betweenness centrality (BC) of a node represents the number of shortest paths from all vertices to all vertices that pass through that node. That is, the betweenness

centrality of a node v is

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

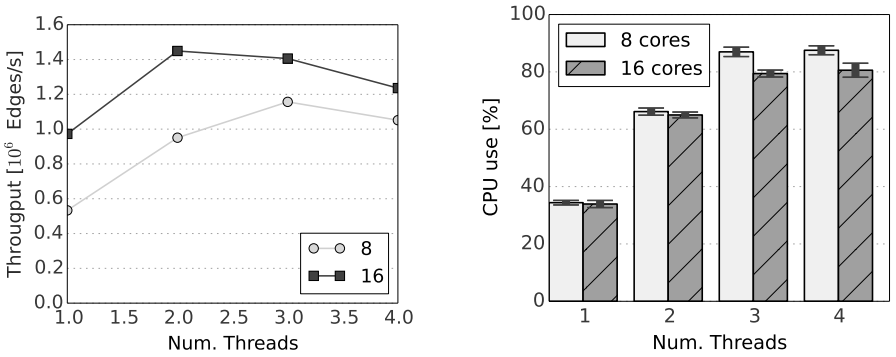
where σ_{st} is the total number of shortest paths from vertex s to vertex t and $\sigma_{st}(v)$ is the number of those paths that pass through v. The execution times of the four kernels, reported in Table 6.5, is highly unbalanced. The 4th one, betweenness centrality, accounts for more than 99% of the total execution time. Because of this extreme unbalance, only this kernel has been analyzed in the experimental evaluation.

Table 6.5: *Execution times of the SSCA#2 kernels.*

Kernel	duration	normalized
Graph construction	92.825.349	0,00361
Classify large sets	16.816.951	0,00065
Graph extraction	51.437.789	0,00200
Betweenness centrality	25.530.790.602	0,99373

The BC can be computed by performing multiple breadth first visits of the graph, one starting from each node of the graph, and assigns a centrality weight to the nodes according to the number of shortest paths it belongs to. For our experiments, we ported to the prototype APIs the reference implementation, which is written in the C language with OpenMP pragmas for parallelization. The benchmark permits to approximate the result by executing a limited number of BFS visits from randomly chosen nodes, in order to reduce the execution time, specially on very large graphs. In the experiments we configured this number to 256 BFS. When translating the code from OpenMP, we used a fixed round robin schedule for parallelizing the loops, except for the main loop at line 9 of Algorithm 2. For this loop we used a dynamic scheduling of loop iterations to threads, similar to the OpenMP *guided* schedule. To implement the dynamic scheduling we use both global index and local indexes. The threads increment the global index in large blocks using critical sections, than iterate over the block using local indexes. The block size decreases exponentially as the global index approaches the end of the queue. This allows to reduce the overhead at the beginning of the loop, when the blocks are larger, and allows fine grained parallelism at the end to improve load balance.

Figure 6.8 shows kernel throughput on the left and the utilization of the cores on the right. The core utilization is averaged over the entire execution. The performance scale very well in the unsaturated systems (1-2



(a) *Throughput* **(b)** *CPU utilization*

Figure 6.8: *Betweenness centrality performance*

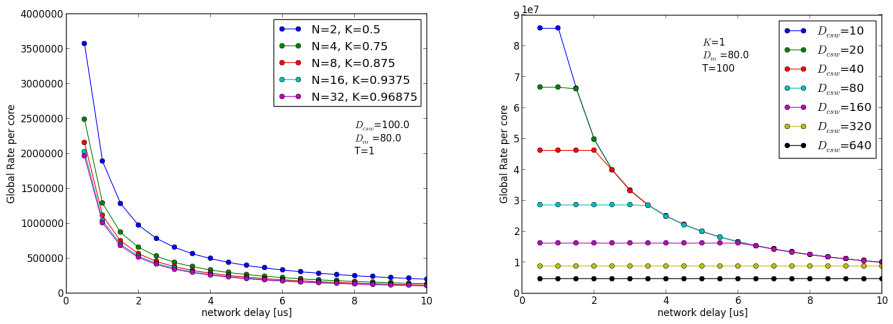
threads), and shows the same degradation observed in the BFS when the cores are fully used. The saturation occurs at 88% average CPU utilization, instead of 100%. There are two reasons: first of all the algorithm has a small serial section that is executed during each of the 256 BFS visits. More important, the algorithm uses multiple barrier synchronization points for each visit, hence with a higher frequency with respect to the previous BFS benchmark. Each barrier causes some of the cores to stall waiting for the remaining ones, hence to reach 100% of core utilization the workload must be perfectly balanced in each of the parallel sections, a task almost impossible to achieve without introducing overheads. Even if the multiple barriers prevents full utilization of the processor, the dynamic scheduling of the main loop distributes the workload evenly across the entire set of cores, as shown by the min-max bars in the plot. Even if multiple short stalls occur, none of the processors gets significantly underused. On the memory side, the number of remote memory requests generated by the various cores presents a relative standard deviation lower than 7%.

Algorithm 6 BFS benchmark pseudo-code

```
1:  function BFS_PARALLEL(V, Edges, v_s)
2:      for all v ∈ V do
3:          Vist(v) ← 0
4:      Q ← v_s
5:      threads ← []
6:      for ti ← 1, N do
7:          threads[ti] ← threadSpawn(BFS_body, ti)
8:      for ti ← 1, N do
9:          threadJoin(threads[ti])
10: end function
11: function BFS_BODY(ti)
12:     while Q ≠ ∅ do
13:         QN = ∅
14:         for i ← ti to N every num_threads do
15:             vertex ← Q[i]
16:             for all e ∈ Edges(vertex) do
17:                 w ← edgeDestination(e)
18:                 lock(w)
19:                 if Visit(w) = 0 then
20:                     Visit(w) ← 1
21:                     unlock(w)
22:                     lock(QN)
23:                     pos ← Q.size + 1
24:                     Q.size ← pos
25:                     unlock(QN)
26:                     Q[pos] ← v
27:                 else
28:                     unlock(w)
            barrier
29:         if ti = 0 then
30:             swap(Q, QN)
31:             level ← level + 1
            barrier
32: end function
```

6.3 Design exploration

Using the proposed model it is possible to perform an exploration of a wider range of system parameters to assess their effect. In Figure 6.9 is represented the inverse dependency between the rate of global memory accesses and average network delay, when varying two system parameters. In both cases a single thread executes loop that access the global memory every 25 instructions. The latency of the accesses to the local memory instead is 80 clock cycles. The delays are reported in clock cycles, with a clock frequency of 100MHz, apart from the network delays which are expresses in micro seconds. On the left plot (a) the number of nodes in the network assumes different values, from 2 up to 32, which correspond to a percentages of remote requests decreasing from 50% to 97%. With more than 8 nodes the probability of performing local accesses to the global address space is significantly low and the performance remain essentially unchanged. In addition, even in systems with a small number of nodes the performance varies only if the latency of a network transaction is under $2\mu s$. This indicates that even in middle sized HPC systems all the global memory accesses can be assumed as remote, and further reducing the local accesses does not impact negatively the performance.



(a) *Memory Rate vs. number of nodes*

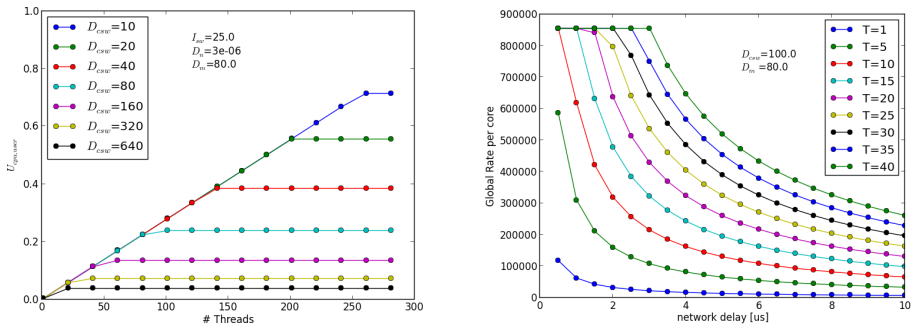
(b) *Memory Rate vs. context switch delay*

Figure 6.9: *Inverse dependency between global memory rate and average network latency*

The right plot (b) shows how the global memory rate changes with the time required for switching the threads contexts, while performing only remote requests ($K = 1$). The horizontal portions of the asymptotic lines represent the region in which the processor is saturated by the execution of the application and the context switch routines, and the performance is independent on the network latency. Instead, the hyperbolic lines correspond

to network bound cases, in which the processor has idle times that can be covered by using more threads. The plot shows that the scheduler overhead has to be very low for making the multithreaded approach feasible, motivating the need for the automatic preemption and the concurrent hardware scheduling performed by the GMAS.

Figure 6.10.a shows the correlation between the user core utilization and the network latency, on different values of the context switch time. The plot excludes the scheduling time and shows only the utilization due to the execution of the application code. The plot motivate two interesting considerations The first is that a good processor utilization requires a very fast scheduler routine. The second is that given a quick scheduler, the number of threads required to saturate the processor is very high, in the order of hundreds of threads. This number coincides with the 128 hardware contexts used by the Cray XMT supercomputer, which pays a cost in terms of clock frequency and single thread performance in order to provide this feature. This observation motivates the idea of maintaining the thread contexts in memory and using software routines for the switch. The concept is confirmed by Figure 6.10.b, which represents the memory rate in function of the network delay, when running various number of threads with a reasonably fast scheduler. The plot shows that only in very low latency networks a small number of threads is enough to cover the latency and maximize the throughput.



(a) *Core utilization vs. number of threads, with different scheduler overheads*

(b) *Memory Rate vs. network delay, with various numbers of threads*

Figure 6.10: *Effects on performance of the number of threads executed*

By filling the model with the timings measured on the prototype, it is also possible to estimate the effectiveness of the multithreading support in that specific implementation. The benchmarks execution on the prototype showed that most of the cores time is spent inside the scheduler routines,

which limits the number of threads that can be used and the user-level core utilization. Consequently, we evaluated how the performance would change with two extremely different multithreading approaches.

First of all, we considered how much the performance would degrade when using a software version of the same scheduler included in the GMAS. The scheduling policy has been implemented in software using the Xilinx MicroBlaze instruction set, to obtain the necessary timing information. However, its execution on the prototype is not possible, because the GNI network interface has been designed for interacting with the GMAS components, neglecting the features required for interacting with an operative system. The software scheduler stores the ready/waiting flags of the threads in the local scratchpad using a single word, with the most significant bit corresponding to the thread with identifier zero. The MicroBlaze ABI includes an instruction (*clz*) that counts the number of leading zeros in a 32-bit word. This instruction can be used to retrieve the identifier of the first available thread, using a fixed priority. For example, if the first byte has value 00101101 it means that the first thread executable is the 3rd one, which has identifier 2, equal to the number of leading zeroes. We added a second bit-mask, used to cancel out the flags of the most recently scheduled threads. This mask allows to rotate the highest priority position in a round-robin fashion. The whole scheduling sequence reads the pending status word from the local scratchpad, searches for a valid thread, updates the status, and eventually rotates the mask. This code requires between 45 and 50 clock cycles for completion, depending on the thread statuses. Adding this latency to the time required to save and load the contexts of the threads, the total increases to approximately 260 cycles.

However, the role of the GMAS is not limited to the selection of the next thread to execute. It also also receives the responses from the network, and manages the status flags of the threads without intervention from the core. A pure software support for multithreading requires that the scheduler interrogates the network interface to identify the completed requests and to map them to the respective threads. Therefore, we considered an hypothetical software scheduler (SW sched) that requires 100 clock cycles to interact with the GNI and select the next thread.

Fig. 6.11 shows the estimated performance, and compares it to the prototype performance when running pointer chasing on 16 cores. The maximum request rate of the cores drops by 24%, from 27.4 to 20.7 Mref/s. At this rate, the number of concurrent network transactions is less than the maximum, hence the network utilization and system efficiency is reduced with respect to the prototype. The plot in Fig. 6.11 also shows the esti-

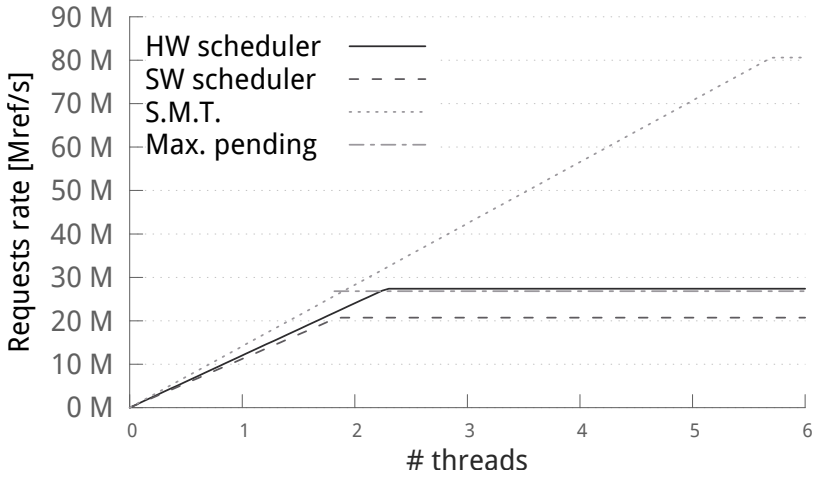


Figure 6.11: *Global request rate generated by the system with different context switch implementations: hw scheduler, sw routine, simultaneous multithreading*

mated performance of a system that executes the context switch in zero cycles. This system would require to modify the cores to add multiple register files, as in the Cray XMT. In this scenario, pointer chasing would reach 80.6 Mref/s, three times more than the rate obtained with the prototype. However, to achieve this high rate it is necessary to maintain 6 complete thread contexts in hardware registers, while the proposed architecture only stores the statuses of the threads. Also, 6 threads are only enough to cover the latency of a small, completely connected network, and the number must be increased with the network diameter. Hence, the considerable slow down of our prototype with respect to hardware multithreading comes with large savings in term of hardware complexity and cost, and more flexibility in increasing the number of threads, making it a valid trade-off.

6.4 Synchronization Performance

The support for mutual exclusion based on variables locking is flexible and, if used correctly, it permits to protect any data structure even during complex modifications. However, because the memory space is partitioned and distributed most of the lock requests are remote. Figure 5.4 presented in subsection 5.3.2 shows that a remote critical section involves at least 4 round trip times (RTT), from the point of view of the issuing core, and 3 RTT from the point of view of the destination GSync. Using the timings of the FPGA prototype, a typical critical section requires $3RTTs = 1515$

clock cycles. These numbers correspond to a pattern of memory accesses read-modify-write, which involves reading a value and storing the result of an operation. As the length of critical sections increase, also the probability of contention increases. In particular, threads residing on the same node of a locked variable have to try many times before finally acquiring it, causing a context switch each time. In presence of highly contended parts of the data structures the GSync can easily become a bottleneck.

For this reason we confronted the performance of the BFS kernel implemented using locks with a version of the BFS which uses atomic increments. The comparison was possible thanks to the simulation platform, which allowed to emulate the atomic operations in a reduced amount of time. Before confronting the performance it is useful to measure the distribution of the lock attempts on the 4 nodes which compose the system. The lock distribution is shown in Figure 6.12, using the rate of lock attempts per second, averaged over four execution of the kernel. Each quadrant of the Figure represent one of the nodes, and the corresponding GSync. To facilitate the analysis, one line is added to the bars corresponding to the average rate over all the configurations and nodes.

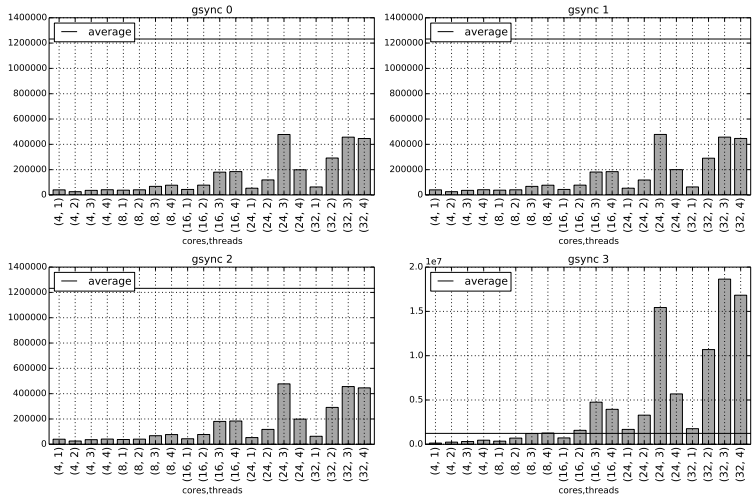


Figure 6.12: *Distribution of lock attempts on the system nodes.*

By comparing the bars and the average line, it is evident that in the configurations with higher parallelism exploited one of the nodes receives many more requests than the others. This node in question is the one that stores the head (and size) of the work-list used by the algorithm to keep track of the discovered vertices, that requires mutual exclusive access every

time a vertex is visited for the first time. This motivates the proposals of introducing atomic operations to handle possibly frequent but small critical sections. Notice that the problem can be alleviated also by working at the algorithm level, introducing local queues and distribution steps. However, atomic increments can be used also in other similar hot-spots, and also are a much easier tool for optimizing applications than a invasive algorithm modifications.

A second version of the BFS has been implemented, using atomic increments both to insert new vertices in the shared work-list and to mark the vertices as visited. By using atomic increments all the exclusive insertions in the work-list are reduced to an atomic read-modify-write performed by a GNI on the memory that resides on the same node, much shorter than the lock-based version. This greatly reduces the contention and allows to better balance the memory and network pressure over the system. Since this version uses no locks, there is not an equivalent of Figure 6.12. On the contrary, the comparison is possible by confronting the distribution of memory and network transactions. Figure 6.13 and Figure 6.14 show the utilization of the GNI network interface. As in the case of the experiments performed on the prototype, the utilization is not related to the actual network bandwidth but to the maximum number of concurrent transactions, which is a similar but scaled metric.

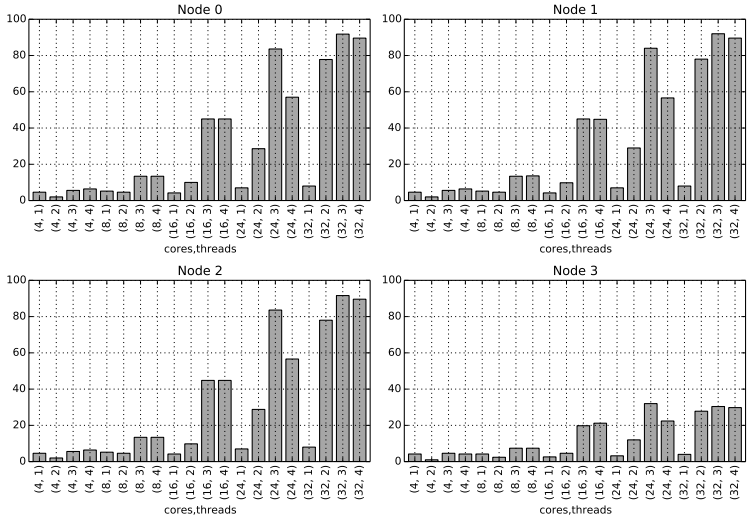


Figure 6.13: *Utilization of the GNI interface, by the lock-based BFS.*

The most evident difference is that when using the atomic increment the

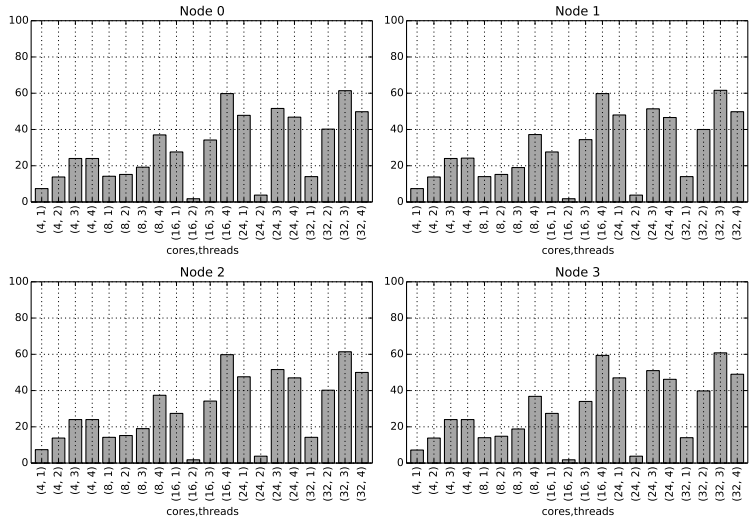


Figure 6.14: *Utilization of the GNI interface, by the BFS with atomic increment.*

outgoing traffic is equal for each node, while when using the locks the network traffic outgoing from the nodes 0,1 and 2 is much higher than that of node 3. The cause for the network unbalance is that much of the network traffic is due to lock requests that for node 3 are executed locally. A second difference, is that in the four- and eight-cores configurations, where contention is less frequent, the use of atomic operations improves the network utilization on all the nodes. Similar considerations are possible when considering the rate of global memory requests generated by each node, shown in Figure 6.15 and Figure 6.16. It is clear that the reduced contention on the critical sections allows the algorithm to execute faster and improves the rate of memory requests by more than three times.

Finally, it is interesting to analyze the impact on the scheduler of the polling implementation of the locks with respect to the poll-less atomic increment instruction. Figure 6.17 reports the number of scheduling events occurring in the system when executing the lock-based BFS. The events are divided in *blocks* of the threads because of remote transactions, and *yields* executed after failing a lock acquisition. The numbers are summed over all the nodes and cores of the system. As the parallelism exploited increases, also the scheduler overhead increase. Specially the real, hardware concurrency provided by multiple cores has a negative effect on the number of *yield* requested by the threads.

The situation is radically different when using atomic increments instead

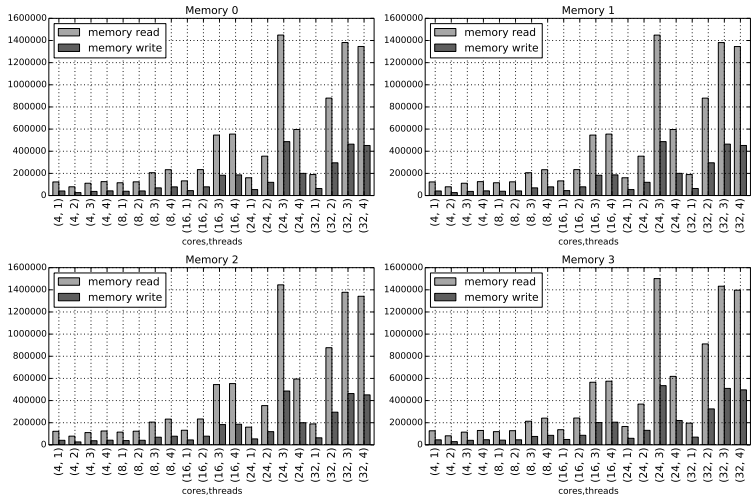


Figure 6.15: *Rate of global memory accesses performed by each node, running the lock-based BFS.*

of the locks, as shown in Figure 6.18. First of all, there is no need for the threads to *yield* the processor, because the atomic operations executed remotely can not fail hence can not cause deadlocks, unlike the polled *lock* routine. In addition the total number of thread block events caused by remote transactions is stable, independently from the number of cores and threads used. This is because each of the code sections requiring mutual exclusion, which are were small, is reduced to a single transaction that does not require multiple attempts. Therefore, the number of memory accesses and atomic operations is essentially given by the size and topology of the graph, hence is constant independently from the amount of parallelism exploited.

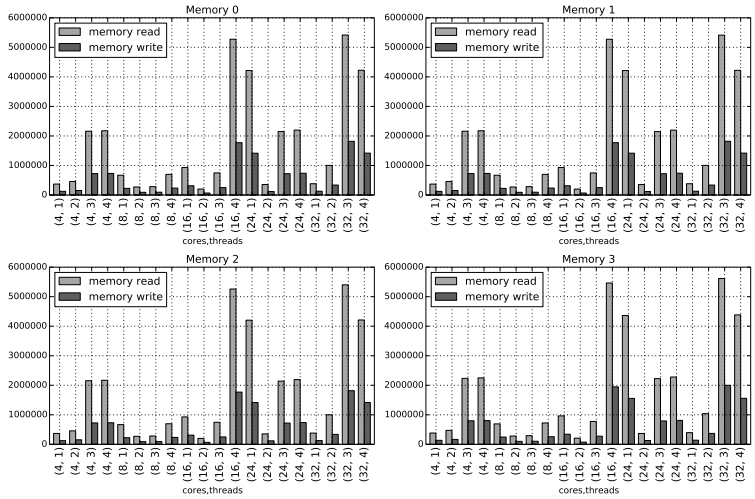


Figure 6.16: *Rate of global memory accesses performed by each node, running BFS with atomic increments.*

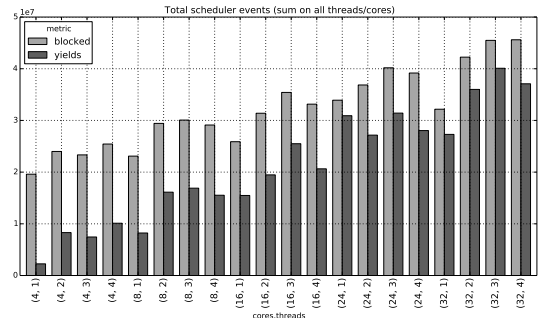


Figure 6.17: *Rate of global memory accesses performed by each node, running the lock-based BFS.*

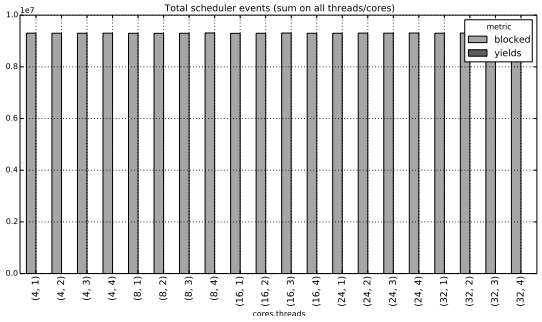


Figure 6.18: *Distribution of lock attempts on the system nodes.*

6.5 Conclusions

This chapter has described the experiments performed to evaluate the proposed abstract architecture and the prototype implementation. The prototype shows that the multithreading support is effective in hiding the latency of remote memory accesses increasing the rate of memory requests and in turn the application throughput. The performance of memory bounded kernels, such as the pointer chasing, scale very well with the number of cores and threads used. The performance of the prototype configurations with many cores is constrained by the limit on the number of concurrent transactions due to the commodity on-chip bus, not by the architecture design.

When the algorithms include sections with high contention the system has an optimal configuration beyond which increasing the parallelism has negative effects. The main issue is the polling implementation of the *lock* primitive performed from the source node, which requires multiple round trip time. Using the simulator we showed that the bottlenecks can be greatly reduced by moving the logic of the critical section to the destination node, using atomic operations. This benefit is possible without introducing complex optimizations in the algorithm implementation.

By using an analytic model, we showed that the architecture approach can scale well to larger systems, provided that lightweight tasks with fast context switches are used, and enough tasks are instantiated to hide the network bandwidth.

CHAPTER 7

Wrap-up and Conclusions

The reason and main goal of the research presented have been stated in the introduction and further motivated in the background and state of the art. Traditional High Performance Computing architectures have been designed to optimize computational intensive algorithms based on regular data sets, while algorithms on irregular data have had less interest. Because of the emerging interest on knowledge discovery from graph-based data structures, support for efficient execution of these applications is now required from future HPC architectures, as well as execution models that facilitate the development. In addition, the supporting architecture should be flexible enough to execute efficiently both regular and irregular algorithms, and reduce the costs in order to be marketable.

An abstract architecture has been proposed [12], that integrates three custom components inside a regular distributed and many-core system architecture. The design requires very limited modifications on the main system components. The architecture provides a partially distributed address space, which allows to share the main data structures without explicit partitioning. The scrambling of the address space allows to mitigate the dynamic onset of hot-spots, by breaking and distributing the data across the whole system. The distributed memory approach is made possible by executing

multiple threads on each core, and automatically preempting the threads on remote accesses to cover the network latency. By scheduling the threads in hardware, the switch delay is further reduced. In addition, the architecture support a generic mechanism for fine-grained mutual exclusion by providing lock primitives which work directly on memory addresses instead of requiring the allocation of mutex variables.

To prove the feasibility of the architecture, an actual implementation has been described which consists in an FPGA prototype [13, 14, 55]. In addition, commercial families of processors have been considered, and details have been provided regarding their use in the proposed architecture. The prototype demonstrates that the additional hardware components have an area overhead comparable to the size of a compact micro-controller, optimized for space.

In the last chapter the prototype is supported by an analytic model and a simulator in demonstrating the performance and scalability of the architecture. The prototype shows that the multithreading support is effective in hiding the latency of remote memory accesses, making distributed shared memories viable despite the long latencies. The utilization of the prototype cores attributable to the application is low, because of the relatively high time required to perform software context switches. However multithreading significantly improves the performance with respect to single threaded applications. This improvement is expected to grow on larger HPC systems with greater network diameters.

The lock-based synchronization procedures are effective for low-contention fine-grained critical sections, however become a bottleneck when hot-spots exists in the algorithm. The reason is the source-based polling performed by the threads in case of failed lock acquisition. The simulated system proves that by inserting atomic operations that perform atomic read-modify-write operations at destinations can eliminate the bottleneck in the most common use cases.

Finally, the analytic model shows that the architecture has the potential to scale to much larger distributed systems, provided that enough parallel tasks are available and that the software overhead due to the task switch is low. This promotes the need for further research on lightweight tasks for permitting the exploitation of the parallelism massively multi-threaded architecture.

Bibliography

[1] *ARMv6-M Architecture Reference Manual*. Tech. rep. ARM DDI 0419C. ARM, 2010.

[2] *AXI Reference Guide*. UG761. Version 13.4. Xilinx. 2012. URL: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf.

[3] *Abount Twitter, Inc.* 2014. URL: https://about.twitter.com/company.

[4] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. «Scalable Graph Exploration on Multicore Processors». In: *Proc. of the 2010 ACM/IEEE International Conf. for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11.

[5] Jung Ho Ahn, Sheng Li, O. Seongil, and N.P. Jouppi. «McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling». In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2013, pp. 74–85.

[6] Rocco Aversa, Beniamino Di Martino, Massimiliano Rak, Salvatore Venticinque, and Umberto Villano. «Performance prediction through simulation of a hybrid MPI/OpenMP application». In: *Parallel Computing* 31.10–12 (Oct. 2005), pp. 1013–1033.

[7] D.A Bader, Guojing Cong, and J. Feo. «On the architectural requirements for efficient execution of graph algorithms». In: *International Conference on Parallel Processing, 2005. ICPP 2005*. June 2005, pp. 547–556.

[8] David A. Bader and Kamesh Madduri. «Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2». In: *Int. Conf. on Parallel Processing*. 2006, pp. 523–530.

[9] Dan Bonachea. *GASNet Specification, v1.1*. Tech. rep. UCB/CSD-02-1207. CS Division (EECS), University of California, Berkeley, 2002.

Bibliography

[10] Steve Bova, Clay Breshears, Rudolf Eigenmann, Henry Gabb, Greg Gaertner, Bob Kuhn, Bill Magro, Stefano Salvini, and Veer Vatsa. «Combining message-passing and directives in parallel applications». In: *SIAM News* 32.9 (1999), pp. 10–14.

[11] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. «Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation». In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, 52:1–52:12.

[12] M. Ceriani, S. Secchi, O. Villa, A. Tumeo, and G. Palermo. «Exploring Efficient Hardware Support for Applications with Irregular Memory Patterns on Multinode Manycore Architectures». In: *IEEE Transactions on Parallel and Distributed Systems* PP.99 (2014), pp. 1–1.

[13] Marco Ceriani, Gianluca Palermo, Simone Secchi, Antonino Tumeo, and Oreste Villa. «Exploring Manycore Multinode Systems for Irregular Applications with FPGA Prototyping». In: *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society. 2013, p. 238.

[14] Marco Ceriani, Simone Secchi, Antonino Tumeo, and Oreste Villa. «Prototyping hardware support for irregular applications». In: *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM. 2013, p. 4.

[15] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. «R-MAT: A Recursive Model for Graph Mining». In: *SDM*. 2004, pp. 442–446.

[16] B.L. Chamberlain, D. Callahan, and H.P. Zima. «Parallel Programmability and the Chapel Language». In: *Int. J. High Perform. Comput. Appl.* 21.3 (2007), pp. 291–312.

[17] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A.R. Choudhury, and Y. Sabharwal. «Breaking the speed and scalability Barriers for Graph exploration on distributed-memory machines». In: *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. Nov. 2012, pp. 1–12.

[18] Fabio Checconi and Fabrizio Petrini. «Traversing Trillions of Edges in Real Time: Graph Exploration on Large-Scale Parallel Machines». In: *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 425–434.

[19] R. M. Christley, G. L. Pinchbeck, R. G. Bowers, D. Clancy, N. P. French, R. Bennett, and J. Turner. «Infection in Social Networks: Using Network Analysis to Identify High-Risk Individuals». In: *Am. J. Epidemiol.* 162.10 (Nov. 2005), pp. 1024–1031.

[20] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. «An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C». In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2005, pp. 36–47.

[21] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. «Active Messages: A Mechanism for Integrated Communication and Computation». In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ACM, 1992, pp. 256–266.

[22] E. Elnozahy. «Five Years with the High Productivity Computing Systems Program A Perspective». In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. 2007, pp. 1–1.

[23] Ernesto Estrada. «Characterization of topological keystone species: Local, global and "meso-scale" centralities in food webs». In: *Ecological Complexity* 4.1–2 (Mar. 2007), pp. 48–57.

[24] Stijn Eyerman and Lieven Eeckhout. «Modeling critical sections in Amdahl's law and its implications for multicore design». In: 38.3 (2010), 362–370.

[25] *FMC XM104 Connectivity Card User Guide*. UG536. Version 1.1. Xilinx. 2010. URL: http://www.xilinx.com/support/documentation/boards_and_kits/ug536.pdf.

[26] LINTON C. FREEMAN. «A Set of Measures of Centrality Based on Betweenness». In: *Sociometry* 40.1 (1977), pp. 35–41.

[27] *Facebook Reports Second Quarter 2014 Results*. 2014. URL: http://investor.fb.com/releasedetail.cfm?ReleaseID=861599.

[28] John Feo, David Harper, Simon Kahan, and Petr Konecny. «ELDORADO». In: *International Conf. on Computing Frontiers*. 2005, pp. 28–34.

[29] Message Passing Interface Forum. *MPI: AMessage-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS), 2012.

[30] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[31] *Graph 500*. 2014. URL: http://www.graph500.org/specifications#sec-6.

[32] Michael Höhl, Stefan Kurtz, and Enno Ohlebusch. «Efficient multiple genome alignment». In: *Bioinformatics* 18.suppl 1 (2002), S312–S320.

[33] ISO. *Information technology - Portable Operating System Interface (POSIX) Base Specifications, Issue 7 TECHNICAL CORRIGENDUM 1 - First Edition*. Tech. rep. ISO ISO/IEC/IEEE 9945 CORR 1. International Organization for Standardization, 2013.

[34] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2014. URL: https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[35] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Tech. rep. 325462-047US. Intel, 2013.

[36] Andrew Kopser and Dennis Vollrath. «Overview of the next generation Cray XMT». In: *Cray User Group 2011 Proceedings* (2011), pp. 1–10.

Bibliography

[37] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. «Lonestar: A suite of parallel irregular programs». In: *IEEE International Symposium on Performance Analysis of Systems and Software, 2009. ISPASS 2009*. 2009, pp. 65–76.

[38] David Levinthal. *Performance Analysis Guide for Intel® CoreTM i7 Processor and Intel® XeonTM 5500 processors*. Version 1.0. 2013.

[39] Sheng Li, Jung-Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. «McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures». In: *42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009. MICRO-42*. Dec. 2009, pp. 469–480.

[40] *LogiCORE IP AXI Interconnect*. DS768. Version 1.05.a. Xilinx. 2012. URL: `http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v1_05_a/ds768_axi_interconnect.pdf`.

[41] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. «Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation». In: *SIGPLAN Not.* 40.6 (June 2005), pp. 190–200.

[42] Phu Luong, C.P. Breshears, and Le N.Ly. «Coastal Ocean Modeling of the U.S. West Coast with Multiblock Grid and Dual-Level Parallelism». In: *Supercomputing, ACM/IEEE 2001 Conference*. Nov. 2001, pp. 25–25.

[43] Ana M. Martín González, Bo Dalsgaard, and Jens M. Olesen. «Centrality measures and the importance of generalist species in pollination networks». In: *Ecological Complexity* 7.1 (Mar. 2010), pp. 36–43.

[44] John M. Mellor-Crummey and Michael L. Scott. «Algorithms for scalable synchronization on shared-memory multiprocessors». In: *ACM Trans. Comput. Syst.* 9.1 (Feb. 1991), pp. 21–65.

[45] *MicroBlaze Processor Reference Guide*. UG081. Version 13.4. Xilinx. 2012. URL: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/mb_ref_guide.pdf`.

[46] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. «Graphite: A distributed parallel simulator for multicores». In: *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*. Jan. 2010, pp. 1–12.

[47] Seth A. Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. «Information Network or Social Network?: The Structure of the Twitter Follow Graph». In: *Proc. of the Companion Publication of the 23rd International World Wide Web Conf.* 2014, 493–498.

[48] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. «Global Arrays: a portable ldquo;shared-memory rdquo; programming model for distributed memory computers». In: *Supercomputing '94., Proceedings*. 1994.

[49] Jarek Nieplocha and Bryan Carpenter. «ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems». en. In: *Parallel and Distributed Processing*. Ed. by José Rolim, Frank Mueller, Albert Y. Zomaya, Fikret Ercal, Stephan Olariu, Binoy Ravindran, Jan Gustafsson, Hiroaki Takada, Ron Olsson, Laxmikant V. Kale, Pete Beckman, Matthew Haines, Hossam ElGindy, Denis Caromel, Serge Chaumette, Geoffrey Fox, Yi Pan, Keqin Li, Tao Yang, G. Chiola, G. Conte, L. V. Mancini, Domenique Méry, Beverly Sanders, Devesh Bhatt, and Viktor Prasanna. 1586. Springer Berlin Heidelberg, Jan. 1999, pp. 533–546.

[50] B. Nitzberg and V. Lo. «Distributed shared memory: a survey of issues and algorithms». In: *Computer* 24.8 (Aug. 1991), pp. 52–60.

[51] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. «The Tao of Parallelism in Algorithms». In: (2011), pp. 12–25.

[52] Pengju Ren, M. Lis, Myong Hyon Cho, Keun Sup Shim, C.W. Fletcher, O. Khan, Nanning Zheng, and S. Devadas. «HORNET: A Cycle-Level Multicore Simulator». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.6 (June 2012), pp. 890–903.

[53] *SSCA#2 v2.0 Specification*. 2014. URL: `http://http://www.graphanalysis.org/benchmark/`.

[54] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. *X10 Language Specification Version 2.2*. 2012.

[55] Simone Secchi, Marco Ceriani, Antonino Tumeo, Oreste Villa, Gianluca Palermo, and Luigi Raffo. «Exploring hardware support for scaling irregular applications on multi-node multi-core architectures». In: *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. IEEE. 2013, pp. 309–313.

[56] J.J. Tithi, D. Matani, G. Menghani, and R.A Chowdhury. «Avoiding Locks and Atomic Instructions in Shared-Memory Parallel BFS Using Optimistic Parallelization». In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. May 2013, pp. 1628–1637.

[57] *Top500, June 2014 poster*. 2014. URL: `http://s.top500.org/static/lists/2014/06/TOP500_201406_Poster.pdf`.

[58] *Top500 List - June 2014*. 2014. URL: `http://www.top500.org/list/2014/06/`.

[59] O. Villa, D.P. Scarpazza, F. Petrini, and J.F. Peinador. «Challenges in Mapping Graph Exploration Algorithms on Advanced Multi-core Processors». In: *Parallel and Distributed Processing Symp., 2007. IPDPS 2007. IEEE International*. Mar. 2007, pp. 1–10.

[60] Oreste Villa, Antonino Tumeo, Simone Secchi, and J Manzano. «Fast and accurate simulation of the cray xmt multithreaded supercomputer». In: *Parallel and Distributed Systems, IEEE Transactions on* (2012).

Bibliography

[61] Jiaoe Wang, Huihui Mo, Fahui Wang, and Fengjun Jin. «Exploring the network structure and nodal centrality of China's air transport network: A complex network approach». In: *Journal of Transport Geography* 19.4 (July 2011), pp. 712–721.

[62] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. «A Practical FPGA-based Framework for Novel CMP Research». In: *Proc. of the 2007 ACM/SIGDA 15th International Symp. on Field Programmable Gate Arrays*, 116–125.

[63] S. Weiss. «An aperiodic storage scheme to reduce memory conflicts in vector processors». In: *Proc. of the 16th annual international symposium on Computer architecture*. 1989, 380–386.

[64] *Xilinx Aurora. Available at www.xilinx.com/products/intellectual-property/aurora8b10b.htm*.

[65] C.C. Yang and T.D. Ng. «Terrorism and Crime Related Weblog Social Network: Link, Content Analysis and Information Visualization». In: *2007 IEEE Intelligence and Security Informatics*. May 2007, pp. 55–58.

[66] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. «A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L». In: *Proc. of the 2005 ACM/IEEE Conf. on Supercomputing*. 2005.