1 Introduction

Uniprocessor computer architecture has reached a steady state in which new developments are evolutionary, not revolutionary. In contrast, parallel computing has long been in a state of flux, marked by projects proposing radical departures. Differences between the various approaches are so fundamental in nature that they are difficult to quantify and compare objectively. However, the steady development of sequential processor technology is having its influence on parallel computing, and is constraining the design space, forcing the various architectures to converge to a point where a multiprocessor consists of ordinary workstations-class computers interconnected by a high-speed network. The primary difference remaining among the various designs is the way in which communication is integrated into the overall architecture.

This dissertation focuses on the introduction of communication into the parallel computer design. It introduces the concept of a *communication architecture* to describe the aspects of the architecture that are related to communication, i.e., the extension of a sequential architecture into a parallel one. Active Messages is proposed as an efficient communication architecture and is shown to subsume traditional communication architectures.

Active Messages provides a small set of simple communication primitives which are *efficient*, *versatile*, and *incremental*. The primitives are *efficient* in that they map high-level parallel languages well onto communication primitives and that these primitives map efficiently onto the processor and network hardware structures. The Active Message primitives are *versatile* in that they can be composed to implement a variety of existing and emerging parallel programming languages. The Active Messages

communication architecture is *incremental* to the processor architecture, in that it leverages the investment in sequential processor design. Implementations on two state-of-the art parallel machines demonstrate these properties of Active Message and bring the cost of the communication primitives down by an order of magnitude, enabling efficient mapping of message passing, shared memory, and message driven programming models onto a single machine without requiring language-specific hardware support.

1.1 The Architecture Design Process

The driving force behind the development of Active Messages is the desire to formulate a communication abstraction that is well suited as a compilation target for emerging high-level parallel languages and can be implemented efficiently with conventional hardware structures. In this respect, the goals of Active Messages are very similar to the ones motivating reduced instruction set computer architectures (RISC) in the early 80s.

The development of RISC was a revolutionary phase of instruction set architecture (ISA) design in that established boundaries were questioned and responsibilities redistributed. The insistence on two *architecture design principles* were among the most important aspects of the process: (i) to take the entire system into consideration—from program characteristics down to VLSI implementations—and (ii) to evaluate alternatives using strictly quantitative criteria. The instruction set architecture focused the revolution process and the surrounding debate on the definition of this interface in which the contract between the designer and the user is explicit, the design goals can be stated clearly, and performance measurements can be gathered and evaluated objectively.

The development of Active Messages in this dissertation follows that of RISC instruction set architectures: the discussion is focused around the communication architecture which serves as interface between the designer and the user. Furthermore the boundaries established in existing multiprocessor systems are questioned. Reviewing the arguments surrounding the RISC vs. CISC debate clarifies the role of Active Messages in parallel computer architecture design.

1.1.1 The RISC argument

The instruction set design principles most commonly advocated in the 70s were regularity, orthogonality, and composability [Wul81]. By following these principles, the case analysis performed by the compiler during code generation was supposedly simplified. This concern in simplifying the compilation process was pushed further in high-level language (HLL) architectures that provided complex instructions to allow a one-to-one mapping from language constructs to the instruction set[1]. Even more general architectures, such as the VAX, increased the complexity of the instruction set in an attempt to capture language constructs in as few instructions as possible. The underlying goal was to "close the semantic gap" [Gag73] between the high-level languages and the instruction set with the benefit of simplifying compilations and improving code density. Small programs were equated with fast programs.

All these efforts in raising the level of the instruction set were predicated on the existence of a level of interpretation built into the hardware, typically a micro-code engine. The RISC proponents recognized that the compiler represents a far more powerful level of translation than the micro-coded interpreter[2]. In addition, closing the semantic gap was shown to increase the danger of "semantic mismatches" [Wul81]: the situation where the semantics of an instruction almost, but not quite, match that of a language construct, preventing the use of that instruction. The full recognition of the compiler's role in mapping language constructs onto hardware structures changed the design criteria for instruction sets: to "provide primitives, not solutions", that can be composed efficiently by the compiler to implement the constructs found in high-level languages.

The primitives embodied in RISC instruction sets are a rather thin abstraction over the micro-architectures found at the heart of many traditional CISC architectures. This meant that RISC instructions could be decoded with simple logic and directly executed in hardware using similar function units [Pat85, Rad82]. The level of interpretation represented by the micro-code was eliminated.

---

1. Examples are Lisp, Prolog, and FORTRAN machines in which the instruction set directly supported dynamic typing, range checks and/or array indexing.
2. Attempts to eliminate the compiler, such as the SYMBOL project, failed dramatically. Thus a compiler would always have to compose instructions to implement language constructs.

Pipelining the execution of RISC instructions proved to be as simple as pipelining micro-code execution, but more effective because the awkward boundary between micro-sequences disappeared. In addition, the pipeline could be exposed to the compiler and integrated into code generation to avoid pipeline interlocks to a certain degree. Eliminating the micro-code engine not only removed the overhead of the micro-code interpreter, but enabled compiler optimizations across more levels. In particular it allowed the compiler to specialize the translation of a high-level construct to use only the primitives necessary in each particular instance. The hardware designer is then free to optimize the frequent simple case and rely on the software to implement the general form when necessary.

Specialization significantly reduces the number of instructions required from what a straightforward expansion would suggest. By providing only simple primitives, the cost model for the compilation remains simple and reliable which simplifies optimizations and improves their accuracy. Ultimately this results in the execution of fewer operations at the micro-architecture level, translating into fewer clock cycles.

RISC architectures create further opportunities for optimization by exposing all storages resources that cannot be trivially managed in hardware [Hen84]. This principle applies not only to registers which the compiler can allocate more efficiently, but also to structures in memory such as stacks, queues, and heaps.

Note that RISC architectures did not really invent any new architectural mechanisms[3]: the micro-architectures of many machines had the traits of a RISC, in particular, Cray's high-performance architectures contained a RISC subset. The contribution of RISC was to identify the critical aspects, to remove the extra features, and to establish the relationship to VLSI design and to high-level language compilation.

In summary, two *design criteria* proved critical to the success of RISC architectures: efficiency and versatility. RISC architectures strive to be:

- *efficient* in that the mapping from high-level languages to instructions is efficient (measured in instructions per program) and the implementation of the instruction set is efficient (measured as cost/performance, and in clocks per instruction multiplied by the clock cycle time), and
- *versatile* in that they support a wide variety of programming languages sufficiently well, i.e., well enough to offset the small performance advantages of more specialized architectures with the steeper performance growth curve of mainstream parts[4].

3. Except for register windows, arguably.

4. While not necessarily a primary concern of the original RISC proponents, the versatility was arguably instrumental in assuring wide acceptance. Projects such as SOAR [UBF+84] demonstrated that RISC architectures could be improved for certain programming languages, but that such hardware support is not a prerequisite for running programs in these languages and that the versatility of RISC machines is more valuable than a slight performance improvement in special cases.

1.2 Communication Architecture Design

The communication architecture plays the same role for communication than the instruction set architecture plays for computation. It therefore seems natural to require its design to follow the same principles and to be measured using the same criteria. A communication architecture must be:

- *versatile* in that it supports a variety of parallel programming models and emerging parallel languages, and
- *efficient* in that the mapping from language constructs to communication primitives is efficient and that these primitives map efficiently onto the processor and network hardware structures.

In addition, because all modern parallel machines are collections of workstation-class computers the communication architecture must seamlessly extend the existing sequential instruction set architecture, i.e., it should also be

- *incremental* in that it complements the existing architecture without disturbing its versatility or efficiency.

This incrementality should allow parallel computing to leverage the investment in mainstream technology and ride the same performance curve.

1.3 Previous Approaches to Communication Architecture

The formulation of these explicit design criteria allows a clear comparison of the multitude of existing parallel machines by reviewing the communication architectures embodied in their designs. The subsections below briefly evaluate in what sense the major parallel architectures—message passing, shared memory, and message driven—satisfy the three requirements for a successful communication architecture. (Chapter 5 discusses each of the three models in detail.)

1.3.1 Message driven architectures[5]

Message driven architectures have been developed specifically to support languages with fine-grain dynamic parallelism such as CST [HCD89], Multilisp [Hal85] and Id90 [Nik91]. The basic argument for such execution models is that long, unpredictable communication latencies are unavoidable in large-scale multiprocessors [ACM88, AI87, Jor83, SBCvE90, WG89] and that multithreading at the instruction level provides an attractive avenue towards tolerating them. To support the resulting frequent asynchronous transfers of control (context switches) efficiently message driven architectures integrate communication deeply into the architecture.

From the perspective of the RISC vs. CISC debate, these architectures are essentially HLL communication architectures in that they implement the communication model with a one-to-one mapping from high-level language constructs to instructions in mind. For example, the programming models on these machines include the concept that load balancing occurs by sending small tasks to remote processors where they allocate storage and start executing. This concept translates directly into sending messages (or tokens) to another processor where, on message arrival, the hardware allocates storage and enqueues the task into the scheduling queue.

The message reception mechanism with its storage management and task scheduling is implicit in the execution model and is implemented in hardware in its full generality. Message driven architectures include a powerful set of task synchronization and scheduling mechanisms in the architecture. These high-level constructs prevent the compiler from optimizing each use and generating specialized expansions. Experi-

ments with sophisticated compilation strategies show that in many cases composing simpler mechanisms on these machines yield higher performance [Tra91, SGS+93].

While dataflow and message driven architectures suffer from the typical inefficiencies of HLL architectures and are not incremental to efficient sequential processors, Dally has demonstrated [DW89] that the communication mechanism embodied in the J-machine is very versatile. It supports a variety of parallel programming models including message passing, dataflow, shared memory, and actors. Dataflow architectures are more specialized and target specific high-level parallel languages.

1.3.2 Message passing architectures

Message passing architectures were developed to a large degree at Caltech [Sei88, FK91] and JPL following a pragmatic strictly incremental approach. Simple single-board computers (SBCs) were mounted in racks and interconnected through back-to-back parallel ports. Today, most commercial parallel computer vendors offer descendents of these early machines, using mostly commodity technology: all current massively parallel processors (MPPs) are essentially workstations interconnected by a custom network.

Communication is integrated into the architecture by exposing the communication micro-architecture to the kernel which implements the send&receive primitives available to the user. The *send* primitive takes a memory buffer and sends it to a destination node. *Receive*, conversely, accepts a message from a

[5] In this dissertation, dataflow is considered to be a variant of message driven architectures. Although the two architectures originated in different research communities, they integrate communication into the overall architecture in a similar manner.

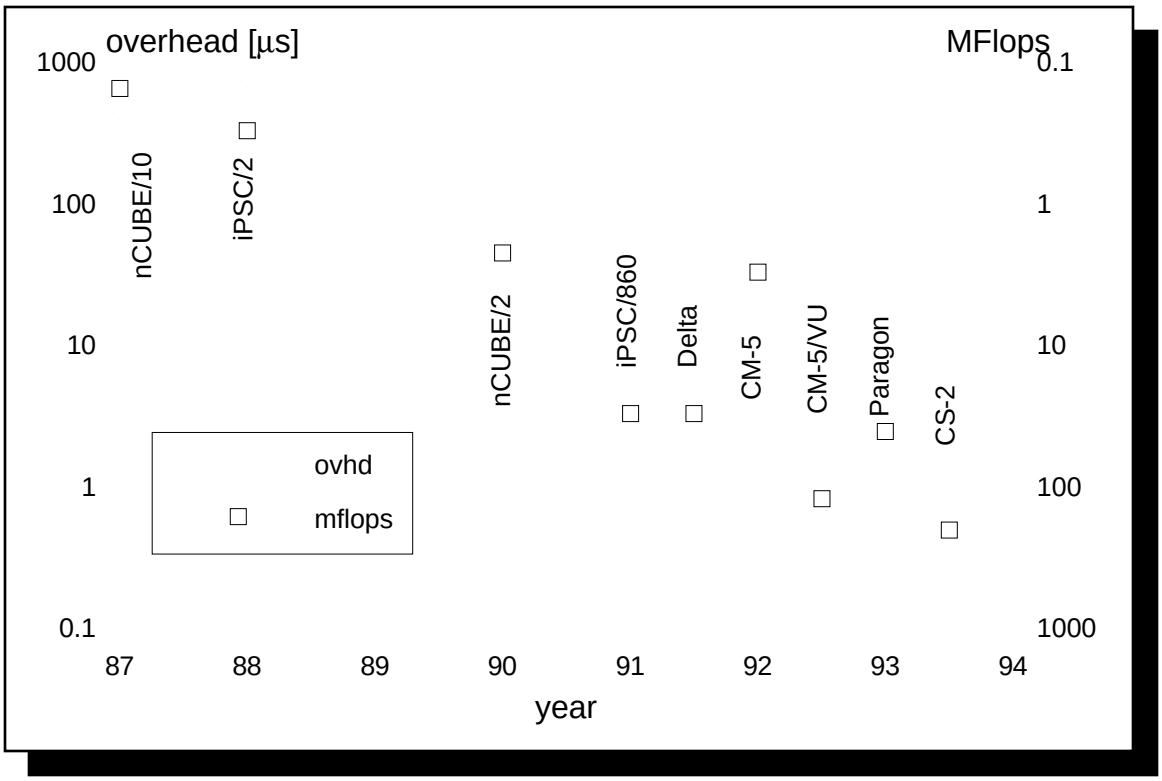

Figure 1-1: Evolution of communication vs. computation performance.

source node and stores it in a specified memory buffer. Many variants of these primitives exist and differ principally in their synchronization characteristics, e.g., whether a *send* blocks until the corresponding *receive* has been executed.

The problem with the send&receive primitives is that they are not suited for direct execution and require a level of interpretation, i.e., the kernel serves the same role as CISC micro-code interpreter. For example, blocking send&receive requires multiple message round-trips to synchronize the sender and receiver before the actual data is transferred. The semantics of non-blocking send&receive further require the interpreter to manage unbounded resources in the form of buffers necessary to hold messages that have arrived, but not received by the user process as well as messages which temporarily cannot be injected into the network.

The send&receive primitives are also not well suited as compilation target for HLLs. One problem is that with send&receive the address spaces of the two communicating processors are distinct and the sender cannot make use of any knowledge about the receiver's memory allocation. This creates inefficiencies, for example, in regular array computations where the compiler can pre-allocate boundary zones into which data from neighboring processors is copied at the beginning of every time step. Even though all storage is preallocated and all addresses are known by the sender, the message layer has to buffer and copy received messages because the sender cannot specify the target address at the destination. In an attempt to reduce the number of such semantic clashes most send&receive implementations provide a multitude of variants differing in buffer management, data placement (scatter-gather), tag matching, and synchronization (blocking). The addition of scatter-gather versions, in particular, parallels the quest for ever more addressing modes in CISC instruction sets.

The level of interpretation in the kernel and the resulting semantic mismatches prevent efficient and versatile use of message passing machines. Figure 1-1 shows that since the first generation of message

passing machines the communication performance (expressed as the processor overhead to send and receive a message) has barely

tracked the improvements in base technology (measured through the cycle time). In the same machines, the computational performance (measured in MFLOPS) has improved at a much faster pace due to advances in the micro-architectures and in the compilation process.

The hardware used in message passing machines offers an interesting opportunity though: given that the micro-architecture is exposed to the kernel, is should be possible to implement a better communication architecture without hardware changes.

1.3.3 Shared memory architectures

Shared memory multiprocessors were first developed as simple extensions to uniprocessors by adding a few additional processors onto the memory bus and having all processors share the same memory. In larger scale systems such a simple design is no longer possible and memory is divided into numerous modules and all processor and memory nodes are interconnected by a network similar to that is used in message passing machines. Today's large scale machines continue to support the illusion that all memory is equally shared by all processors, which is formalized in the PRAM model in which access to any memory location takes unit time.

Shared memory architectures hide communication within the memory system and only let communication affect the instruction set architecture through peculiar memory access semantics. While these architectures are incremental in the sense that the processor is not directly affected, the changes to the memory system are substantial and require a significant design effort and real-estate investment. Arguably, the memory system in shared memory multiprocessors outreaches the processor in complexity (DASH numbers, Alewife?, KSR?).

The perception that shared memory read and write are simple abstractions over the capabilities of the micro-architecture, i.e., that these primitives can be mapped efficiently onto the hardware, is an illusion. Remote memory accesses can take an arbitrary amount of time (typically over 100 cycles) during which the processor is usually stalled. Not only does this lead to instructions with very long execution times, it also requires the memory or cache controller to constitute an autonomous agent which can service remote requests while the processor is stalled. Thus, in some sense, each node in a shared memory multiprocessor consists of two processors, only one of which can be used for computation.

Mapping HLLs to shared memory is difficult because of semantic clashes. Foremost, shared memory communication architectures only support data movement and do not provide any support for transmitting events. While this is sufficient as long as data is moved to the computation when necessary it does not support programming models in which the computation can be moved to the remote data. Shared memory also transfers data in fixed-size messages and only performs round-trip communication (writes require acknowledgments). Data parallel compilers and message passing programmers can often organize communication such that data is pushed to the consuming processor by the producer, eliminating communication latencies and only requiring one-way communication. Expressing this type of communication pattern is impossible with shared memory.

Shared memory is really a high-level programming model which is implemented directly in hardware in shared memory multiprocessors. It is not a communication architecture which can be implemented efficiently and to which many different high-level parallel languages can be mapped.

1.3.4 Conclusions

Existing approaches to parallel computer design are reminiscent of HLL architectures and provide solutions instead of primitives. Consequently, none of the approaches have the versatility to serve as basis for implementing a wide variety of high-level programing languages efficiently and, in particular, sematic clashes prevent any of the approaches to subsume the others. In addition, in all approaches the

hardware must implement general storage management and is forced to take high-level scheduling or data movement decisions without the type of information available during the compilation process.

The reluctance to adopt RISC principles for communication architecture design results in a situation where none of the existing architectures is able to provide an efficient versatile parallel computing platform. Instead, the community remains fragmented, the choice of programming languages available on each machine remains dictated by the hardware, and efficient portable parallel programs remains an oxymoron.

1.4 The Active Messages Approach

Active Messages applies the RISC approach to communication architecture and takes the entire system into consideration, from parallel programming languages down to the communication micro-architectures, and applies quantitative criteria to evaluate the alternatives. First the key issues that a communication architecture must address are defined and then a set of primitives that address these key issues efficiently are developed.

A new approach to communication, such as Active Messages, is enabled by two factors: the success of RISC processor architectures and the emerging consensus on network design which force a convergence of multiprocessor architectures, and the gradual shift towards high-level parallel programming languages which imply a translation step between the programming model and the underlying architecture. In addition, the first Active Messages implementations leverage the fact that message passing multiprocessors expose the communication micro-architecture to the software allowing the new communication architecture to be prototyped without hardware development.

1.4.1 Active Messages

The basic idea in Active Messages is to provide primitives which allow the compiler to apply information gathered in the compilation process to the effective control of communication. The most natural way for the compiler to convey such information is to generate appropriate code, hence, Active Messages strives to allow the compiler to generate code for the critical aspects of communication. This is reflected in the central Active Messages mechanism which associates a small amount of user-level code in the form of a handler with the reception of every message. The handler is named by the message, typically by a pointer in the first word of the message, and is executed in the user process context immediately on message arrival. The role of the handler is to get the message out of the network and into the ongoing computation on the node. The Active Message handlers, in essence, allow messages to be executed instead of requiring interpretation. The key to efficient implementation of Active Messages is to limit the resources available to handlers which must execute quickly and to completion.

The Active Messages primitives correspond closely to the hardware capabilities in message passing machines where a privileged interrupt handler is executed on message arrival, they are an extension of the state machines handling read and write requests in

shared memory cache controllers, and they represent a useful restriction on message driven processors.

1.4.2 Contributions

The primary contributions of this dissertation are:

- the discussion of computer architecture's role in multiprocessor communication from which the notion of *communication architecture* is derived and which leads to the definition of the four key issues that a communication architecture must address,
- a critique of traditional communication architectures which shows that existing approaches to communication in multiprocessors are not well suited as efficient general-purpose communication architectures,
- the development of Active Messages, a new class of communication architectures which address the key issues and provide a set of versatile, efficient, and incremental communication primitives, and
- the demonstration that Active Messages subsumes existing communication architectures (e.g., that the programming models supported by traditional communication architectures can be implemented efficiently using Active Messages) and enables new high-level parallel programming languages.

1.4.3 Dissertation overview

The dissertation begins in Chapter 2 with a discussion of the role of computer architecture in multiprocessor communication. The introduction of the notion of a communication architecture focuses the discussion on the central element of multiprocessor architecture and allows the definition of the aspects of communication that a communication architecture must define.

In preparation for the detailed discussion of communication architectures, Chapter 3 provides the necessary background in the form of descriptions of communication micro-architectures. In particular the nCUBE/2 and CM-5 micro-architectures are used heavily in this dissertation and are presented in detail as the information available in the literature is incomplete. Beyond providing this background, the discussion of the micro-architectures crystallizes the key issues that any communication architecture must address if it is to allow efficient implementations.

Chapter 4 is devoted to the definition and implementation of Active Messages. After a discussion of the concepts underlying Active Messages, the development of the nCUBE/2 and CM-5 implementations demonstrate the incrementality of Active Messages. The efficiency of the mapping onto hardware structures is analyzed using micro and macro-benchmarks which show performance close to the hardware limits while maintaining a simple cost model. A careful examination of the implementation details reveals significant inefficiencies in both micro-architectures and suggests avenues for hardware improvements.

Active Messages is contrasted to the traditional communication architectures in Chapter 5. The discussion of message passing, message driven, and shared memory architectures shows that the communication primitives embodied in these architectures are tailored for a specific programming model and not suited as target for the compilation of other parallel programming languages. Chapter 5 further develops mappings of the message passing and shared memory communication primitives to Active Messages. Active Messages is well suited to support the communication needs of both, although hardware support for a local/remote check is desirable if shared memory is to be emulated directly.

The real efficiency and versatility of Active Messages comes into play in Chapter 6 which presents implementations of two novel parallel languages. Split-C supports message passing, data parallel and shared memory programming styles and contains several features made possible by the flexibility of Active Messages. The implementation of Id90 shows that dynamic parallel languages benefit from extensive specializations of the expansion of high-level constructs and from resource management in the compiler. The performance achieved with Id90 on the CM-5 is competitive with message driven architectures which were specially designed to support this type of programming language.

Chapter 7 concludes the dissertation and uses the experience gained with Active Messages to suggest improvements to existing communication micro-architectures, and to show that Active Messages offers an incremental path to higher communication performance.