

THE CASE FOR MESSAGE PASSING ON MANY-CORE CHIPS

Rakesh Kumar (Illinois) Timothy G. Mattson (Intel) Gilles Pokam (Intel) Rob Van Der Wijngaart (Intel)

The debate over shared memory versus message passing programming models has raged for decades, with fairly cogent arguments on both sides. In this paper, we revisit this debate for multi-core chips and argue that message passing programming models are often more suitable than shared memory models for satisfying the unique goals presented by the many-core era.

The many-core era is different. The nature of programmers, the nature of applications, and the nature of the computing substrate are different for multi-core chips than the traditional parallel machines that drove the parallel programming debate in the past. Specifically, while traditional parallel computers were programmed by highly-educated scientists, multi-core chips will be programmed by mainstream programmers with little or no background in parallel algorithms, optimizing software for specific parallel hardware features, or the theoretical foundations of concurrency. Hence, multi-core programming models must place a premium on productivity and must make parallel programming accessible to the typical programmer. Similarly, while the history of parallel computing is dominated by highly specialized scientific applications, multi-core processors will need to run the full range of general purpose applications. This implies a drastically increased diversity in the nature of applications and an expanded range of optimization goals. This will heavily impact the choice of the programming model for multi-core chips. The programming models for multi-core architectures should also be capable of adapting to and exploiting asymmetry (by design and accident) in processing cores. We argue that the above goals are often better served by a message passing programming model than a shared memory-based programming model.

1. Metrics for Comparing Parallel Programming Models

To compare shared memory models and message passing models, we could take the familiar approach of defining a series of benchmarks, thereby turning this into a quantitative performance effort. However, it is difficult to

distinguish the relative impact of the programming models from the relative quality of the implementations of the underlying runtime systems in such a comparison. A true comparison should deal with qualitative “human factors” and how they impact the programming process. We believe that a fair comparison of programming models must consider the end-to-end cost of the full life cycle of a parallel program. The full life cycle can be summarized as:

- Write the parallel program.
- Debug the program and validate that it is correct
- Optimize the program
- Maintain the program by fixing bugs, porting to new platforms, adding features, etc.

A head-to-head comparison of the programming models for different stages of the program life cycle will allow us to make qualitative conclusions about the relative efficacy of the programming models. We modified the cognitive dimensions from [Green96] to define a set of concrete metrics for our comparisons:

Generality: The ability to express in the programming model any parallel algorithm, such that a comparable level of concurrency as embodied by the algorithm materializes on the execution platform.

Expressiveness: Does the programming model help programmers express the concurrency in their problem succinctly, safely, and clearly for the classes of parallel algorithms for which the model was designed? An expressive programming model provides concise abstractions that help a programmer identify concurrent tasks and specify how data is shared (or decomposed) between tasks. Expressiveness does not imply generality.

Viscosity: Does the programming model let a programmer make incremental changes to a working program? If not, the risk of adopting the programming model is high. Viscosity includes the following aspects:

- Is it possible to gradually introduce concurrency into an original serial version of a program? Usually this is not the case if the model implies a new language.
- How much effort is required to add or change functionality of an existing parallel code?

Composition: Does the programming model provide the isolation and modularization needed to support programming by composing parallel modules?

Validation (correctness): Is it easy to introduce cognitive slips when creating a program thereby introducing errors into the code? Can the program's correctness be reasonably validated? How difficult is it to find and remove bugs? Bugs that do not manifest themselves each time a code it run are difficult to find and remove. Such bugs can be due to non-determinism, or to the fact that there may be a big gap between formal specification and implementation of the programming model.

Portability: Does the programming model let a programmer write a single program that can be recompiled and mapped efficiently onto all systems that are relevant for the target user community? This includes the potential for support of heterogeneous systems.

Of these metrics, we highlight composition and validation, whose importance, while very high today, continues to increase. Composition, the ability to build complex applications by composing smaller modules, is the cornerstone of modern software development. It must be supported in parallel software if we hope to migrate our software onto multi-core systems.

As for validation, these costs often exceed system acquisition and software creation costs, a situation that will only worsen as more and more software being produced exploits parallelism. Anecdotal evidence of the difficulty of validating parallel software abounds, we merely cite a single source [Lee06]:

"We wrote regression tests that achieved 100 percent code coverage. The nightly build and regression tests ran on a two processor SMP machine, ... No problems were observed until the code deadlocked on April 26, 2004, four years later."

Section 3 discusses how message passing programming models fare against shared memory models for the above metrics.

2. Comparison Framework

To evaluate different parallel programming models, we need to define a framework that captures a programming model's impact on the design and implementation of the most common parallel algorithms. Our comparison framework consists of different categories of parallel algorithms strategies and different algorithm patterns within each category. Following [Carriero89] we define the following three distinct strategies for parallel algorithm design:

- **Agenda parallelism:** Parallelism is expressed directly in terms of a set of tasks
- **Result parallelism:** Parallelism is expressed in terms of the elements of the data structures generated in the course of the computation.
- **Specialist parallelism:** parallelism is expressed in terms of a collection of tasks each of which are specialized to a distinct function. In other words, data flows between a set of specialized tasks that execute concurrently.

This provides the top level structure of our framework. Each parallel algorithm strategy consists of the following common algorithm patterns used in practice [Sottile09],:

Table 1 Brief taxonomy of parallel algorithms

Parallel algorithm Strategies	Algorithm Design Patterns	
	Agenda parallelism	Task parallelism
Result parallelism	geometric decomposition	data parallelism
Specialist parallelism	Producer/consumer (pipeline)	Event-based coordination

These patterns are well known by experienced parallel programmers (details are available in [Mattson04] and [Keutzer09]). The framework is not complete, but we submit that it covers the broad cross section of the most important algorithms.

Using these patterns combined with our earlier metrics, we can turn our intuition about a programming model into specific (and testable) hypothesis about why different programming models dominate.

3 Comparing Message Passing and Shared Memory

We start with two generalizations concerning message passing vs. shared memory programming models. These concern validation and composition. To compose software modules, you must assure isolation of the modules. Interaction can only be allowed to occur through well defined interfaces. To validate a program, you must assure that every legal way the operations in all active threads can interleave produce a correct answer. Both of these metrics are compromised by a shared address space. Message passing by design provides a mechanism of isolation since the threads of processes in a computation by definition execute in their own address spaces. As for validation, the message passing programmer only needs to check the allowed orderings of distinct message passing events. In a shared address space programming model where all threads access a single address space, proving a program to be race free has been shown to be an NP complete problem [Klein03]. Hence, regardless of the type of parallelism involved, we assert that message passing has a strong advantage in terms of the ease with which a program can be validated and the ability to support software composition.

In the remainder of this section we will work through the algorithm design patterns described in Section 2 using the metrics we defined in Section 1 to compare shared memory and message passing models based on software features and actions required of programmers. The results are summarized in Table 2.

3.1 Agenda Parallelism

Design patterns associated with the “agenda parallelism strategy” are expressed directly in terms of tasks. The two cases differ in how the tasks are created; either directly as a countable set (task parallelism) or through a recursive scheme (Divide and conquer).

For the task parallelism pattern, both the message passing and shared address space programming models are highly expressive and are general enough to cover most algorithms associated with this pattern. The message passing programming model is particularly well suited since data decomposition is typically a straightforward extension of the decomposition of the problem into a set of tasks. This means that the ease of validation common to distributed memory environments is easy to exploit with message passing, task parallelism problems.

The divide and conquer design pattern can be mapped onto message passing and shared address space models. These algorithms, however, are difficult to express with a message passing model. The problem is that as a task is recursively divided into a number of smaller tasks, the data associated with the individual tasks must be analogously decomposed. Programming models that require explicit data decomposition are difficult to apply when tasks are created so dynamically. Shared address space programming models, however, avoid this problem altogether since all threads have access to the shared data space. Furthermore, a key feature of implementations of divide and conquer algorithms is the need to dynamically balance the load among units of execution. For example, if tasks are managed in queues, it is possible that one unit of execution will run out of work. If it only needs to steal work descriptors from a neighboring queue without the need to move data, these work-stealing algorithms are natural to express. This is clearly the case for shared address space models, but not for message passing models.

Overall, both models are well suited for the Agenda parallelism strategy. The task parallelism design pattern works well for both types of programming model, but it slightly prefers the message passing model. For the divide and conquer pattern, however, the shared address space model is substantially better suited.

3.2 Result Parallelism

Design patterns associated with the result parallelism strategy center on how data is decomposed among the processing elements of a system. In most cases the decomposition is well suited to a static decomposition or if dynamic,

the dynamic structure is well defined algebraically and well suited to explicit data management schemes. Hence these algorithms work well with message passing and shared address space programming models.

The classic “result parallelism” pattern is geometric decomposition. Message passing models have been used extensively with this pattern. The sharing of data is explicit through messages, making geometric decomposition programs that utilize a message passing programming model both robust and easy to validate. Shared address space programming models work well also, but since race conditions are possible due to the fact that data is “shared by default”, these programs can be difficult to validate.

Message passing program with geometric decomposition patterns are also highly portable. Since it is natural in these problems to define how data is shared between processes, the programming models are highly portable, allowing easy movement between shared memory and distributed memory systems.

Data parallel algorithms follow a similar analysis. They work well with message passing and shared address space model. Shared address space models, however, have a slight edge over message passing, however, because they don’t require complicated data movement operations when collective operations are encountered. This is only a slight advantage, however, since the most common collectives are included in message passing libraries,

Overall, both models work well for these algorithm strategies. The message passing model, however, has a slight edge due to the greater ease of validating a program once written.

3.3 Specialist Parallelism

These algorithms can be challenging for message passing and shared address space programming models. The essential characteristic of the design patterns associated with the specialist parallelism strategy is that data needs to flow between specialized tasks.

For the pipeline algorithms, both models work, but the message passing provides more disciplined movement of data between stages. Messages are a natural way to represent the flow between stages in the pipeline making message passing programming models both expressive and robust. Shared address space programming models work, but they require error prone synchronization to safely move data between stages. For an API that lacks point to point synchronization (such as OpenMP) this can lead to the need to build complicated synchronization protocols that depend on the details of how a flush works. Even expert OpenMP programmers find flush challenging to deal with in all but the most trivial cases [Hoeflinger05].

These problems are even worse for the event-based coordination algorithms. Message passing models work but robustness is compromised since the event models require anonymous and unpredictable flow of messages between processes. This compromises the robustness and validation properties and creates one of the few situations where race conditions can be introduced in a message passing program. The key is to use a higher level model to apply discipline to how messages are used in these algorithms. For example, an actors model maps well onto event-based coordination algorithms. Actors is by its nature a message passing model. It can be implemented in a shared address space, but it requires complex synchronization protocols and can lead to programs that are difficult to validate. The table below summarizes how the two models fare against each other for different metrics.

4. Architectural Implications

Programming models place requirements on the hardware that supports them. A programming model that requires a shared address space, in order to run efficiently, requires hardware support. In practice, this comes down to the question of hardware supported cache coherence.

As the number of cores and the complexity of the on-chip networks grow, overhead in service of the hardware cache coherency protocol limits scalability. For example, each directory entry will be 128 bytes long for a 1024 core processor supporting fully-mapped directory-based cache coherence. This may often be larger than the size of the cacheline that a directory entry is expected

Metrics	Agenda Parallelism		Result Parallelism		Specialist Parallelism	
	task parallel	divide and conquer	geometric decomp	data parallel	pipeline	events
Generality	=	Shar +	=	Shar+	Msg +	Msg +
Expressiveness	=	Shar +	=	Shar +	Msg +	Msg +
Viscosity	Shar +	Shar +	Shar +	Shar +	Msg +	=
Composition	Msg +	Msg +	Msg +	Msg+	Msg +	Msg +
Validation	Msg +	Shar +	Msg +	Msg +	Msg +	=
Portability	Msg +	Msg +	Msg +	Msg +	Msg +	Msg +

Table 2: Comparing Message Passing (Msg) and Shared Memory (Shar) programming models for design patterns from Table 1. A “+” indicates when a model dominates for a given case. An “=” indicates that the two models are roughly equivalent for that particular case.

to track. As another example, writes in a sequentially consistent shared memory processor may not proceed until all the shared lines have been invalidated, even the ones residing in cores that may be 10s of hops away.

Hence, as the number of cores increases, the overhead associated with the cache coherency protocol grows. In particular, the additional cost due to the cache coherency protocol as each core is added to a many core chip grows. This increasing cost per core means that as the core counts grow, a “cache coherency wall” eventually limits the ability of a program to extract increased performance from the system.

Compare this to the situation for a many core chip that does not support cache coherency. Such a chip would be fine for software based on a message passing model. In this case, the cost as each node is added to a chip is fixed based on the topology of the network. Hence, there is not coherency wall and these message passing chips can scale to much larger numbers of cores.

5. Discussion and Conclusion

Table 2 summarizes our comparisons of shared memory and message passing programming models. We consider a range of design patterns for each of our metrics.

As we indicated earlier, message passing programming models have distinct advantages due to the relative ease of validation and the fact they support the isolation required for composition. . Furthermore, as we pointed out in the previous section, a message passing programming model is more portable as well due to the fact the model places fewer constraints on

programming models have an advantage. In some cases, these advantages can be quite stark. This often leads to disqualification of message passing upfront, since the most salient first impression that programmers have of a programming model is its expressiveness. Higher expressiveness is often associated with higher programmer productivity. However, validation and composition constitute a very large portion of the downstream cost of an application’s lifecycle. The shared memory programmer trying to validate a program and understand its composition with other software modules must understand the underlying memory model of the system; a task that even challenges experts in the field. This makes those costs much greater for shared memory models than a message passing model.

When you look at the full software life cycle and the full range of metrics (not just expressiveness), we submit that message passing models are more suitable than shared memory models for a large class of applications. Hence, message passing models are an important, if not the only alternative for programming multi-core and many-core chips. The benefits only increase as the number of cores and the complexity of the network on a processor chip increase.

6. References

[Carriero89] Nicholas Carriero and David Gelernter, “How to Write Parallel Programs: A Guide to the Perplexed”. ACM Computing Surveys, 21(3) pp 323-357, September 1989.

[Green96] Thomas R.G. Green and M. Petre, “Usability Analysis of visual Programming

Environments: a “Cognitive Dimensions” framework”, *Journal of Visual Languages and computing*, vol 7, pp. 131-174, 1996.

[Hoeflinger05] Hoeflinger, J.P., de Supinski, B.R.: The OpenMP memory model. In: *Proceedings of the First International Workshop on OpenMP - IWOMP 2005*. (2005)

[Keutzer09] K. Keutzer and T.G. Mattson, “A design pattern language for engineering (Parallel) software“, *Intel Technology Journal*, in Press, 2009.

[Klein03] Philip N. Klein, Hsueh-I Lu, and Robert H. B. Netzer, *Detecting Race Conditions in Parallel Programs that Use Semaphores*, *Algorithmica*, vol. 35 pp. 321–345, Springer-Verlag, 2003

[Lee06] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 29, no. 5, pp. 33-42, May 2006

[Mattson04] T.G. Mattson, B. A. Sanders, B. L. Massingill, *Patterns for Parallel Programming*, Addison Wesley software patterns series, 2004.

[Sottile09] M.J. Sottile, T.G. Mattson, and C. E Rasmussen, *Introduction to Concurrency in Programming Languages*, CRC Press, 2009.