

# **I Own What I Own When I Own It**

(Memory Ownership Protocols in a Manycore World)

**Paul Petersen, Intel**

The first step in fixing a problem is finding the root cause of the problem.

The typical problem in parallel computing is that parallel applications are not reliable. Applications may crash, hang or produce wrong results, possibly in random ways. These problems are not new. Even on sequential hardware, when dealing with concurrency (i.e. threads in a shared memory space), these problems arise. Over the years research teams have tackled this problem, with commercial spin-offs to aid software developers (The Intel® Thread Checker is one such tool, <http://www.intel.com/software/products/tcwin> ). Tools that can detect the existence of potential data-race and deadlock defects in a parallel application can dramatically shorten the software development cycle. What took days of random, trial-and-error search with print statements and debuggers now is shortened to a few hours of CPU time that in the end generates a report of many faults in the parallel application.

These tools, while a massive productivity gain for the industry, do not change the fundamental manner in which parallel software is created. They can speed up the development cycle, but, in the end, because these are essentially testing tools; the developer is responsible by design for the total correctness of the resulting application.

The software industry has been here before. The topic then was memory allocation. C and C++ applications relied on the explicit use of the malloc () and free () APIs to allocate and deallocate memory. Many developers used these APIs incorrectly, resulting in illegal memory accesses and memory leaks. Purify\* was an early product to detect these problems ( <http://www.ibm.com/software/awdtools/purify/> ). Tools like Purify and the Intel Thread Checker provide the ability to detect the “needle-in-the-haystack” of the elusive bug that escapes the developer’s ability to locate.

But it is well known, that you cannot “test” correctness into an application. It needs to be designed in from the start. Analysis tools like this are best used as aids to a professional software developer, making a good developer even better.

If analysis tools are not the answer for creating correct software, then what is? We may find some help by examining the history of memory management. As mentioned above, historically memory was manually managed with the software developer responsible for correct allocation and deallocation. Also many of the languages with these low level interfaces placed correctly manipulating pointers into this memory in the programmer’s hands.

But many in the industry have suggested that these approaches are error prone and should be replaced. The popularity of managed languages like Java and C# is testimony to developers seeing the advantage of living in an environment where certain classes of errors cannot happen due to restrictions placed on the allocation, deallocation, and access to memory in these environments.

Does this imply that a similar evolution will occur in parallel programming? Unconstrained access to a shared memory heap may be declared too dangerous to be used as the basis for constructing robust manycore software.

In the field of High Performance Computing, problems like data-races typically do not exist because much of this field has moved to distributed computing where disjoint address spaces are the norm. But, the experience from this field appears to support the statement that message passing does not fix the problems with threading; it just replaces one set of problems with another set of problems. While you remove the danger of asynchronous memory accesses causing random behavior, you introduce the problem of the data never being located where you need to use it, and thus you introduce cumbersome memory copies to move the data to its new location.

Although it appears that abandoning shared memory efficiency for message passing may not be successful, we may be able to adopt the beneficial aspects without suffering with the additional performance costs.

One concept that message passing provides is the notion of data-ownership. Threads (processes) own their own data address space, and they “share” data by acquiring a copy of the data, and the access rights to have exclusive ownership of this instance while it is being modified. Then ownership may be transferred again to communicate the modified data back to the original owner.

Message passing does this ownership transfer with explicit data-movement operations. However, it seems that if one were to leverage the benefits of a shared memory environment on a manycore processor, transferring ownership should not require one to actually transfer the data. This also opens up the possibility of defining shared-read-only data access ownership where the benefits of a shared memory address space can really shine.

The question is how to enforce such a scheme. As mentioned before in the domain of memory management, garbage collection technology, dynamic array subscript checks, and severe restrictions on pointer manipulations produced a solution to the problem.

Can similar approaches work for parallelism?

Functional languages or functional patterns in imperative languages provide one answer. If the developer does not need to share the data then it should be provably safe from a parallel correctness standpoint. Defining and enforcing that a thread's stack space is

private would be another technique. This has been used in Java and does decrease the types of threading bugs that can exist.

Another approach that is being explored today in the research community is with the use of transactional memory. This approach temporarily enforces ownership for the duration of transaction. However, it appears to approach the solution backwards. Instead of saying my application is guaranteed to be safe and predictable except where I want to do something dangerous, it takes the opposite stance. The transactional memory approach states, programs have no guarantees about correctness except in the few places that are marked as transactional. This is not really exactly true, as there is the assumption that the programmer has statically “proved” that nothing bad can happen outside the transactions. This is like saying the programmer had “proven” that the malloc () and free () APIs were used correctly. Putting this burden on the software developer means that mistakes will escape to hurt our customers.

This position statement is not proposing a fully baked, complete solution. Remember how this statement started, “The first step in fixing a problem is finding the root cause of the problem”. The problem in this case is not just a defect in a specific application, but instead could be a defect in the ecosystem in which we write parallel applications. Correct parallel programs require developers to reason about memory ownership, and allowed sharing modes. Moving in the direction which makes this more explicit and enforced seems like a good direction to pursue.