Exascale Computing without Threads*

A White Paper Submitted to the
DOE High Performance Computing Operational Review (HPCOR)
on Scientific Software Architecture for Portability and Performance
August 2015

Matthew G. Knepley[1], Jed Brown[2], Barry Smith[2], Karl Rupp[3], and Mark Adams[4]

[1]Rice University, [2]Argonne National Laboratory, [3]TU Wien, [4]Lawrence Berkeley National Laboratory

knepley@rice.edu, [jedbrown,bsmith]@mcs.anl.gov, rupp@iue.tuwien.ac.at, mfadams@lbl.gov

Abstract

We provide technical details on why we feel the use of threads does not offer any fundamental performance advantage over using processes for high-performance computing and hence why we plan to extend PETSc to exascale (on emerging architectures) using node-aware MPI techniques, including neighborhood collectives and portable shared memory within a node, instead of threads.

Introduction. With the recent shift toward many cores on a single node with shared-memory domains, a hybrid approach has became in vogue: threads (almost always OpenMP) within each shared-memory domain (or within each processor socket in order to address NUMA issues) and MPI across nodes. We believe that portions of the HPC community have adopted the point of view that somehow threads are "necessary" in order to utilize such systems, (1) without fully understanding the alternatives, including MPI 3 functionality, (2) underestimating the difficulty of utilizing threads efficiently, and (3) without appreciating the similarities of threads and processes. This short paper, due to space constraints, focuses exclusively on issue (3) since we feel it has gotten virtually no attention.

A common misconception is that threads are *lightweight*, whereas processes are *heavyweight*, and that interthread communication is fast, whereas inter-process communication is slow (and might even involve a dreaded *system call*). Although this was true at certain times and on certain systems, we claim that in the world relevant to HPC today it is incorrect. To understand whether threads are necessary to HPC, one must understand the exact technical importance of the differences (and similarities) between threads and processes and whether these differences have any bearing on the utility of threads versus processes.

To prevent confusion, we will use the term *stream of execution* to mean the values in the registers, including the program counter, and a stack. Modern systems have multiple cores each of which may have multiple sets of (independent) registers. Multiple cores mean that several streams of execution can run simultaneously (one per core), and multiple sets of registers mean that a single core can switch between two streams of execution very rapidly, for example in one clock cycle, by simply switching which set of registers is used (so-called *hardware threads* or *hyperthreading*). This hyperthreading is intended to hide memory latency. If one stream of execution is blocked on a memory load, the core can switch to the other stream, which, if it is not also blocked on a load, can commence computations immediately. Each stream of execution runs in a *virtual address space*. All addresses used by the stream are translated to physical addresses by the TLB (translation look-aside buffer) automatically by the hardware before the value at that address is loaded from the caches or memory into a register. In the past, some L1 caches used virtual addresses, which meant that switching between virtual address spaces required the entire cache to be invalidated (this is one reason the myth of heavyweight processes developed); but for modern systems the L1 caches are hardware address aware and changing virtual address spaces does not require flushing the cache. Another reason switching processes in the past was expensive was that some systems would flush the TLB when switching virtual address spaces. Modern systems allow multiple virtual address spaces to share different parts of the TLB.

With Linux, the operating system of choice for most HPC systems, the only major difference between threads and processes is that threads (of the same process) share the same virtual address space while different processes have different virtual address spaces (though they can share physical memory). Scheduling of threads and processes is handled in the same way, hence the cost of switching between threads or between processes is the same. Thus, the only fundamental difference is that threads share all memory by default, whereas processes share no memory by default. The one true difference between threads and processes is that threads can utilize the entire TLB for virtual address translation, whereas processes each get a portion of the TLB. With modern large TLBs we argue that this

*Note that this discussion relates to threads on multicore systems and does not address threads on GPU systems.

is not reason enough to select a more complex programming model, since it has only, at most, second-order effects on performance.

Another issue that must be understood in order to compare thread versus process models is how memory is shared between streams of execution. Streams run with virtual addresses that are translated to physical addresses. If two streams access the same physical location, each of the stream's virtual addresses translate to the same physical address. If two streams share the same virtual address space they also share the same physical addresses. Unix shared-memory is a way to allocate a block of physical memory and then provide virtual addresses that point to the same physical memory to multiple streams. Once the memory has been allocated, access to that memory by any of the streams is at exactly the same cost as if that memory was private to that stream. Communication between threads and processes via mutexes is also identical; there is no performance penalty for processes sharing mutexes versus threads sharing mutexes.

We have demonstrated that using multiple threads versus multiple processes is similar at the operating system and hardware level both in what takes place and in performance. Thus, we believe that a discussion on threads versus processes boils down to "shared everything by default" versus "shared nothing by default". Using threads is not an issue of necessity, but merely one end of a spectrum of possible programming models.

Independent of the issue of threads versus processes is the question of how many streams of execution are most effective for multicore systems and how one should control the number of streams of execution during different portions of the computation. For example, in the classic loop-level parallelism of OpenMP one has one stream of control in the program, running on one core of the node, until a loop is reached. At that time multiple threads are launched, perhaps one per physical core or one per hyperthread, each handling different ranges of the loop values. Once the loop end is reached, all but one of the streams of execution is removed until the next loop is reached. With standard MPI programming there is always one stream of execution per core during the entire computation. Other programming models may instantiate several or many streams of execution per core either in parts of the computation or, with oversubscribed MPI, during the entire computation. The hope with these models is that by having "extra" streams of execution any latencies that block a particular stream are hidden by having one of the other streams running during that time. In our experience, the *launching* of additional streams of control from a given stream of control by requesting threads via OpenMP, is extremely time consuming, taking, for example thousands of clock cycles. Pulling them from a pool of pthreads is less time consuming, but still significant and cumbersome. Since the goal of HPC is to utilize all the computational units (for example, wide vector units and multistage pipelines), all the time, every cycle spent waiting for the launch of additional streams is a wasted cycle. Hence optimum use of the hardware for many HPC applications means keeping the same number (this number would be based on hardware features of the system) of steams of execution running at all times, and not launching and stopping streams of execution during the simulation. Based on this philosophy, some members of the HPC community are now using OpenMP to launch a collection of threads with OpenMP at the beginning of their programming and keeping that same running set of threads until the program exits. In our minds, since this model utilizes threads as if they were processes and processes have no fundamental disadvantage in performance from threads, one is better off just using MPI processes with appropriate use of MPI 3 based shared-memory data structures - if they are needed because of memory size constraints - within a node is more appropriate than using the MPI + OpenMP hybrid model.

MPI and PETSc. The PETSc team is open to proposals to replace the pure MPI programming model, but only with an alternative that is demonstrably *better*, not with something more complicated that has not been demonstrated to lead to faster, more maintainable code. We believe that the MPI 3 features of neighbor collectives, nonblocking reductions, and portable access to shared memory within a node provide all the functionality needed to achieve exascale performance for PETSc and many other HPC applications, without the complexity and potential drawbacks of utilizing threads. Ever since the the IBM SP2 in 1996 we have been told that "pure MPI won't scale to the next generation of machine"; this has yet to be true, and we have no reason to believe that it will be true even at the exascale.

Acknowledgments. The authors were supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357.