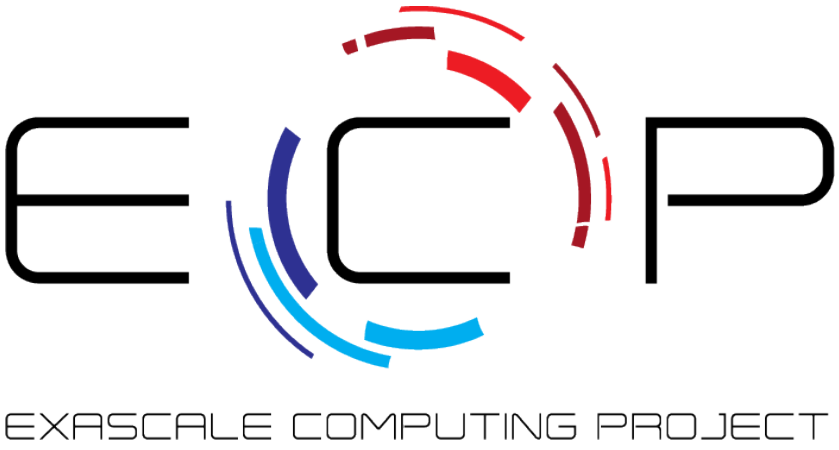
Hardware Evaluation Outreach: Application development Challenges Now and for the Exascale Era

Ray Bair[1], Jeanine Cook[5], David Donofrio[2], Jeff Kuehn[3], Shirley Moore[4 1]

[1]Argonne National Laboratory, Chicago, Illinois, USA
[2]Lawrence Berkeley National Laboratory, Berkeley, California, USA
[3]Los Alamos National Laboratory, Los Alamos, New Mexico, USA
[4]Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
[5]Sandia National Laboratories, Albuquerque, New Mexico, USA

April 12, 2019



SAND2019–4136 R

1 Introduction

The intent of this document is to assist the programmer in understanding details of contemporary and Exascale hardware system design and how these designs provide opportunities and place constraints on next-generation simulation software design. We attempt to clarify hardware organization and component details for our most current and Exascale systems to help program developers understand how software needs to change in order to take best advantage of the performance available.

Exascale success is specifically defined for ECP as a 50x improvement over baseline in the aggregate "capability volume" on several KPP axes, of which raw floating point performance is only one, but also includes characteristics such as problem size, system memory size, node memory size, power, and efficiency. This multi-axis approach is particularly important to understand in the context of delivered improvements in real applications, since, for instance, the floating point computation may comprise less than 10% of the actual computational work required.

Given the Exascale requirements and the constraints these requirements put on the performance expectations of fundamental system components, the programmer will be forced to re-think several application implementation details in order to achieve exaflop performance on these systems. The remainder of this document aims to present more detail on Exascale era system hardware and the specific areas that the programmer should address to extract performance from these systems. We attempt to give the programmer guidance at both a high- and low-level, providing some abstract suggestions on how to refactor codes given the expected system architectures and some low-level recommendations on how to implement these modifications. We also include a section on training resources that are helpful for both programmers that are just beginning to understand code modifications for contemporary and Exascale systems and for those that have done some refactoring and are now trying to extract maximal application performance from these systems.

2 Programmer Challenges for the Exascale Era

The previous decades have been witness to an incredible increase in computing performance - from the relentless rise of single threaded performance to the multi- and many-core scaling that is still happening today. However, due to the convergence of multiple technical and physical limitations we see that future HPC systems cannot continue to scale using general purpose processors alone.

Current technology trends do not point to an exascale system being feasible by 2024 without the use of accelerators. As an example, the two IBM Power9 CPUs in OLCF Summit's node deliver a peak performance of 1.1TF compared with the 15TF delivered by just two of the six GPUs in a node. Hypothetically, building a relatively large system composed of 65,000 dual socket Power9 nodes would only deliver 71.5PF peak performance and the CPUs alone would consume almost 13MW - as much as the entire 200PF Summit system. It is clear that even a 10X increase in CPU performance will still fall well short of the goal of an exaflop capable system by 2021.

For application developers this focus on accelerators presents both a challenge and an opportunity. While it will become ever more critical to ensure larger sections of an application are able to take advantage of accelerators, both programming environments and hardware features are working to ease the transition. Below is a partial list of challenges exposed to application developers by hardware architectural shifts:

- **Increasing Parallelism**: In order to capture peak performance on an exascale class machine an application will need to exploit incredible levels of parallelism in terms of flops, memory accesses and interconnect bandwidth. Considering a reasonable CPU clock speed of O(2GHz) an exascale class system will have O(10^8) floating point units. Given the latency of floating point units, an application dominated by flops would need to expose O(10^9)-way parallelism. While memory bandwidth has increased dramatically with the availability of HBM to O(2TB/s) the latency to access memory has remained relatively flat at O(200ns). With this ratio, it is necessary to have 400KB worth of loads in flight at all times - this translates to 50,000 DP Floats. Network injection bandwidth continues to increase, although not as quickly as node performance. In order to fully saturate an HPC interconnect multiple processes will

need to be injecting data simultaneously. This applies to both CPUs and accelerators - both will require multiple threads dedicated to off-node communication.

- **Data Locality**: With the majority of performance being accessible through accelerators it is necessary for the user to strongly consider the locality of their data. Two options will be available - manual partitioning and explicit management of locality or reliance on a runtime to automatically migrate pages from host to accelerator. In all cases, minimizing data movement between host and the accelerator will be most advantageous, however, relying on runtimes to perform automatic migration may result in strong NUMA penalties.

- **Machine Balance**: Even with increases in memory bandwidth, most nodes will continue to see compute (flops) scale faster than the available bandwidth. This drives application developers to push their codes to simultaneously minimize unnecessary data movement while also attempting to drive the memory interface as close to the limit as possible. Note that this balancing of compute and bandwidth can be applied to all levels in the memory hierarchy - including caches, HBM, DDR, etc. While memory bandwidth has increased, memory capacity is not keeping pace. This leads to a new ratio for developers to balance - bytes of capacity to bytes of bandwidth. In current server class CPUs it takes O(1 sec) to touch every address in memory - i.e. 128GB of DDR could be accessed in 1 second at 128GB/s. However, with memory bandwidths approaching O(2 TB/s) and capacities in the O (10s) GB it becomes possible to touch every address within HBM in milliseconds. Again, this drives application developers to maintain good control over data movement and locality. With respect to problem size, current system designs operating today have memory sizes on the order of 1PB, O(10^{15} bytes), and it is very easy to imagine that an exascale system could extend this to 10PB, O(10^{16}) bytes. However, the cost in dollars and power of both memory and cores continues to increase, pushing per node memory capacity to at best remain constant and to more likely decrease.

3 Programming Recommendations

This section aims to provide both high- and low-level recommendations to programmers for reasoning about/developing/refactoring applications to execute more efficiently on most contemporary, very near-term, and Exascale timeframe systems. We start with high-level suggestions and strategies that application developers must consider based on hardware trends, and follow this with some additional lower-level application refactoring suggestions to better take advantage of these hardware features.

The following is an outline of hardware trends that are beginning to appear in some form on the most current systems, but will continue to move along these paths into the Exascale era. We describe the trend and how applications should respond to this trend to extract adequate performance.

1. Core counts per socket will increase modestly, with a likely maximum of around 64 cores/socket. For processor systems, the primary sources of computational performance will come from the acceleration units and/or the exceptionally wide SIMD units per CPU core (8 to 16 Double Precision words). Applications will need to exploit wide SIMD through extracting as much vectorization as possible. However, maximal performance will only be attained through implementing codes on accelerators.

2. Performance will come primarily from accelerators, and in the Exascale timeframe, these will be GPU technologies that may be more optimized for compute than in prior generations. Developers should use existing systems such as Summit to port and optimize their codes on GPUs now. A common trend is a unified/coherent memory between CPUs and GPUs, creating the concept of "supernodes." Expect nodes to comprise more GPUs than CPUs and that GPUs can directly communicate. Accelerators will diversify beyond Exascale.

3. Nodes will make use of HBM (high bandwidth memory) as the primary memory tier and in the 2021 exascale timeframe, HBM will have very limited capacity in comparison to DDR memories, offering

a Bytes-to-FLOP ratio < 0.01 (a factor of 10x to 100x lower capacity per peak execution rate in comparison to earlier projections). HBM memory capacity may be supplemented with a near capacity tier composed of additional channels of DDR4 or DDR5 memories. This will exacerbate NUMA effects that will have to be managed by the programmer. Developers need to really understand their data, its size, and how it is used throughout the computation to understand how to place data to optimize locality. This may require refactoring data structures and the computation that operates on it. This HBM capacity limitation will likely remain until packaging issues are resolved.

4. Non-volatile storage will look as if it were in the memory address space rather than going through a block I/O device (known as storage class memory). Not only is this more convenient (now you can memcopy() from a volatile data array to a non-volatile data array), it is also lower latency and more efficient because you no longer need to go through a deep software stack to effect storage.

5. Rack disaggregation strategies driven by the larger datacenter market may be implemented in some systems, which has driven the explosion of memory fabric technology concepts such as HPE's Gen-Z, IBM's Open-CAPI, and NVIDIA's NVLink. These high performance fabrics enable flexible configuration of hardware resources within a rack by supporting formation of customized supernodes. For ECP, this feature could enable applications to configure "fat nodes" or "thin nodes" from rack resources, depending on their requirements. However, disaggregation will exacerbate NUMA effects for large node configurations. The memory fabrics being designed for datacenters include integrated hardware support for a global address space model for data movement. Global Address Space (GAS) makes non-local memory globally visible to threads within an application or hardware domain, although it might not be cache coherent. Advantages include lower overhead for messaging and enabling lightweight GPU threads to participate as peers in interprocessor communication. The disadvantage of GAS is that it does not offer specific acceleration for MPI. Vendors have not finalized the GAS communication and synchronization primitives, and there is a risk that they may not be usable by MPI3 RMA or will be different enough that they may not be performant.

The next sections dive deeper into programming considerations given the observed and expected changes in system hardware. These hardware/system details cover our most contemporary systems through the expected Exascale systems.

3.1 Extracting Parallelism

As mentioned in Section 2, Exascale era systems will require extraction of massive application parallelism in order to achieve performance. This parallelism requirement is FP-, memory-, and network driven. Applications will need to be decomposed at multiple levels into threads of execution that can drive the O(10^9)-way parallelism provided by FPUs (floating-point units) and the 400KB of loads that memory will allow these systems to sustain. Multiple processes will also be required to drive the network injection bandwidths that will be available. Problem decomposition will have to be done at multiple levels and at finer grain than in the past in order to take advantage of the available parallelism and to achieve the performance gains expected by an exascale system.

Good, high-level advice in this area and many others is for programmers to modify their existing codes to run well on existing contemporary machines. Systems such as ORNL's Summit support high levels of parallelism through high bandwidth memory and GPU accelerators. Leveraging existing frameworks such as OpenMP, Kokkos, and Raja work for extracting CPU and accelerator parallelism and will likely work well on future systems. These frameworks limit the programmers exposure to portability issues by abstracting the complexity of the underlying hardware into the programming interface. OpenMP may be a reasonable choice of framework. While it may not deliver performance, it will likely be supported on future platforms.

Lower-level advice on extracting parallelism involves both defining finer-grained threads of execution and modifying data structures to enable these threads to efficiently store and operate on the data. An example of a multi-level problem decomposition is the following:

1. The top-level is a MPI rank, which is how we've decomposed parallelism in CPU systems for decades.
2. Intermediate-level parallelism uses cores or accelerators to execute threads within each MPI rank. These core/accelerator threads can be considered a *tile* of execution assigned to a core or an accelerator. In support of tiles, data structures may be represented as an array-of-structs-of-arrays, where each structure represents a tile.
3. The lowest-level parallelism is extracted within each thread through dense SIMD loop computation that takes advantage of the vast number of wide vector units expected to be available in next-gen systems.

To fully realize parallelism, there must be the ability to place multiple tiles on a single MPI rank, the ability for each MPI rank to have a tile computed by a thread, and the ability for each thread to compute dense SIMD. The size and number of tiles must be flexibly varied to accommodate different problem sizes.

Load balancing concerns may drive more than one intermediate level of parallelism above or below the tile level, or it may dictate a larger number of tiles based on application drivers. In some applications, where the amount of computation varies among tiles, an active load balancer may be required to distribute tiles to ranks and threads.

3.1.1 Loop Structure Recommendations

Future architectures will support SIMD-like features with varying degrees of start-up cost for SIMD-loops. Because of the variable start-up expense for these loops, large trip-counts and large operation counts on the inner-most loops are desirable. Loop-carried dependencies should be avoided to help the compiler extract locality as discussed below. This dictates:

1. High trip counts: minimum inner-loop iterations $\sim 32 - 2048$
2. High number of arithmetic operations per iteration count: $\sim 100 - 1000$ operations/iteration

Start-up costs are expected to improve in future generations, but operation density within inner-loops will still need to remain high.

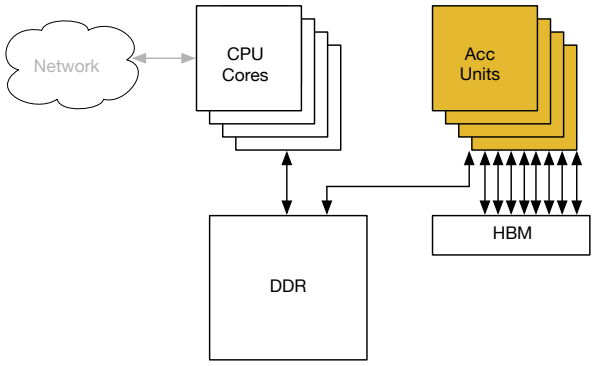
3.1.2 Data Structure Recommendations

As mentioned above, in support of the tile abstraction for parallelism decomposition, data structures may be represented as an array-of-structs-of-arrays, where each structure represents a tile. We also stated that the size and number of tiles must be flexibly varied to accommodate different problem sizes. A reasonable expectation is that $\#tiles * tilesize$ will be roughly consistent with 2X increase at each system generation, but with a trend toward increasing on future systems to the limit of on-package memory (HBM). Data copying should be avoided – attempt to place the data near the computational element.

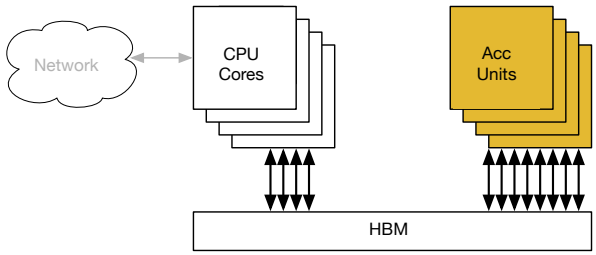
All low-level data structures supporting SIMD-loops should be array-like and designed to avoid loop-carry dependencies within the loops that might break SIMD optimization opportunities. As long as loop-carry dependencies are avoided, we expect compilers to be capable of loop-fission if required to extract locality, however, loop-fusion is, in practice, more difficult for the compiler to perform automatically.

3.2 Data Movement and Locality Considerations

Data movement is, and will continue, to be a key part of performance optimization and as systems trend towards greater heterogeneity the ability to flexibly partition an application across various compute resources will allow greater performance scaling on the exascale machine regardless of node configuration. With this in mind, programmers will continue to strongly consider locality as they develop applications. One critical component in this partitioning will be the disparate bandwidth/capacity capabilities presented by various memory technologies. The trade-off between memory capacity and bandwidth as seen in contemporary systems will persist as High-bandwidth memories (HBMs) will continue to provide increased bandwidth albeit



(a) Potential node architecture with asymmetric memory organization. CPU connected to capacity (DDR) memory only, HBM access exclusively through accelerators. Accelerator can access DDR memory as well. Strong incentive to run majority of code on accelerator.



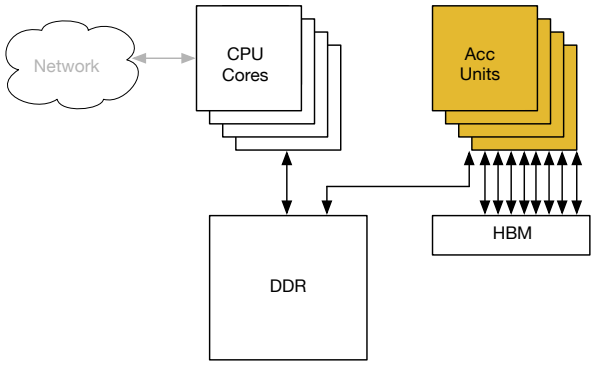
(b) Potential node architecture with symmetric memory organization. Vast majority of performance still provided by accelerator, however, CPU now has access to HBM. Overall node memory capacity severely impacted due to limitations imposed by HBM capacity.

Figure 1: A **physical view** of two node architectures illustrating how the CPU and accelerator are connected to capacity vs. high bandwidth memory

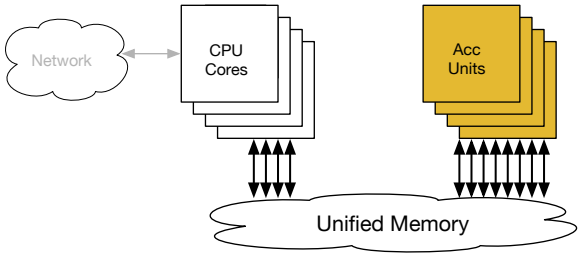
with lower capacity when compared to DDR. Physical constraints, such as packaging, ensure HBM capacity will likely be limited to O(10s)GB in the near term, eventually approaching O(100)GB. These capacity limitations are driving the implementation of heterogeneous memory systems that combine HBM for performance while simultaneously providing DRAM or NVM for capacity off-package. While movement/coherence between these memory technologies may be automatically managed by a combination of hardware features and system software the programmer should pay close attention to where data is stored / accessed to obtain the best performance.

Figures 1 and 2 illustrate potential node organization from both a hardware (physical) perspective as well as a software perspective with respect to how memory is connected / partitioned between CPU and accelerators. While the organization of HBM and DRAM with respect to the CPU and accelerators is not fully known for exascale era systems the two primary proposals, from a software perspective, are shown in Figures 2a and 2b. In Figure 2a, the majority of memory capacity is attached to the CPU and while the accelerators can read DDR the CPUs cannot access the HBM creating non-symmetric memory access. Here, the programmer must explicitly and manually manage locality and should try to minimize the amount of data moved between host and device. Figure 2b shows an organization where HBM can be accessed by both CPU and accelerators, making memory access symmetrical. Capacity memory may be provided by NVM or other technology that has significantly lower bandwidth than HBM. In addition, a node with this configuration will suffer from limited overall memory capacity due to physical constraints imposed by HBM technology discussed earlier. In this case, locality is likely managed by the runtime, with strong NUMA effects and penalties. In both organizations, networking is done through the host CPU.

While connectivity between accelerators and CPUs continues to improve moving both computation and data between them will remain expensive. Taking current GPU-based machines as a proxy for future exascale systems it can be observed that it is advantageous to move as much code as possible to the accelerator and view the GPU as the primary computational element rather than as an offload accelerator. For example, the time required to launch a kernel on a GPU is O(10us); in this time a modern CPU can easily execute O(10K) to O(100K) instructions. This implies that the kernel invoked on the GPU must execute for a significant amount of time to amortize the considerable latency/overhead in spawning. GPUs continue to trend towards being optimized for greater computational performance in lieu of 3D graphics performance while simultaneously improving their connectivity to the CPU and other GPUs. Overall, the best preparation for future exascale systems is to begin porting codes to multi-GPU nodes such as those found on Summit / Sierra.



(a) Asymmetric organization requires user to explicitly manage locality and pay careful attention to partitioning while minimizing data movement between CPU and accelerator memories.



(b) Symmetric memory organization allows runtime to manage locality for programmers. While convenient, there may be very strong NUMA effects and penalties if close attention is not paid to producer / consumer relationships.

Figure 2: A **software** point of view of two node architectures with differing memory organizations

3.3 Multi-GPU Considerations

Accelerators offer the best path to an exascale system that fits within the target power budget. It is likely that as in current systems, the exascale system will contain multiple accelerators per node. The accelerators may or may not be homogeneous and looking beyond exascale this heterogeneity is likely to increase. As a proxy for this increase in accelerator count we look at current multi-GPU node systems, such as OLCF's Summit, as a way for application developers to begin reasoning about programming models and tradeoffs.

When targeting a multi-GPU node it is important to consider your parallelization strategy, below are four options with their associated tradeoffs:

- **Single Thread, Multiple GPUs:** In this conceptual model a single thread will send data to the kernel that needs it, regardless of which GPU the kernel is executing on. This approach can require the developer to add additional loops to the code to manage all the devices and has the undesirable result of the CPU becoming a bottleneck as it is likely unable to stream data to multiple GPUs as fast as it can be consumed. This leads to underutilization of the node and is, in general, an undesirable approach.

- **Multiple Threads, Multiple GPUs:** Similar to the Single Thread model, this is relatively simple to conceptualize and relies on using OpenMP, Pthreads, or a similar model on the CPU where each thread manages its own GPU. This allows for a simple mapping of thread to device but does have the potential to conflict with existing threading in the application - be sure to pay close attention to affinity. This approach can realize improved utilization when compared to the Single Thread model.

- **Multiple Ranks, Single GPU:** If your application already makes use of MPI a straightforward model is to map each rank to a single GPU. Assigning multiple ranks to a node may allow use of multiple (all) GPUs. This may allow a developer to re-use existing domain decomposition but attention must be paid to MPI placement in order to maintain performance.

- **Multiple Ranks, Multiple GPUs:** This is the most complex of the four models and allows each rank on a single node to manage multiple GPUs within a node. This method allows all GPUs to share common data structures and enable direct communication between GPUs. While this model is quite challenging to get correct and is recommended to be used only when absolutely needed it will likely deliver the highest performance.

Section 4 provides links to significantly more detailed training material provided by both ECP as well as multiple DOE HPC Facilities.

4 Training Resources for Application Development: Getting Started and Further Optimization

A wealth of information is available on best practices for developing code and obtaining optimal performance on pre-exascale computers, with an eye toward the exascale systems. Here we provide a starting point for exploration of DOE classes organized by the Exascale Computing Project and the large DOE computing facilities. Most of the upcoming classes are open to all, though some have limited seats. In many cases slides and video are available online from past classes. You can also expect to see additional classes sponsored by the Hardware Evaluation team focused on other topics discussed above.

The ECP IDEAS Project is a good source of webinars, with an ongoing series called Best Practices for Software Developers (https://ideas-productivity.org/events/hpc-best-practices-webinars/). Webinars covering topics relevant to the discussion above include:

- Parallel I/O with HDF5: Overview, Tuning, and New Features
- Quantitatively Assessing Performance Portability with Roofline
- Basic Performance Analysis and Optimization – An Ant Farm Approach
- An Introduction to High-Performance Parallel I/O

Comprehensive training in performance portability using Kokkos is available from time to time at DOE Labs, ECP meetings, and other events. RAJA, which is also used for performance portability, might be useful to some developers. Resources on these are:

- Syllabus for a recent multi-day course at NERSC:
  https://www.nersc.gov/users/training/events/performance-portability-with-kokkos-march-26-29-2019/
- Shorter overview of Kokkos can be found on Youtube:
  https://www.youtube.com/watch?v=MrMgECniQhQ
- Tutorial on the RAJA performance portability abstraction layer:
  https://extremecomputingtraining.anl.gov/speakers/rich-hornung-llnl/

Other really useful resources include:

- The Argonne Training Program on Extreme-Scale Computing provides "intensive, two-week training on the key skills, approaches, and tools to design, implement, and execute computational science and engineering applications on current high-end computing systems and the leadership-class computing systems of the future." Web site for applications dates and the syllabus:
  https://extremecomputingtraining.anl.gov/.
  The web site also provides links to the videos from past training sessions.
- The list of training and tutorial programs at the Oak Ridge Leadership Computing Facility (OLCF):
  https://www.olcf.ornl.gov/for-users/training/,
  including archives of class videos relevant to recent large-scale machine deployments.
- The National Energy Research Scientific Computing Center (NERSC) provides targeted training:
  https://www.nersc.gov/users/training/events/, with recordings of past sessions.

Acknowledgement

This work was supported by the Exascale Computing Project (ECP), Project Number 17-SC-20-SC, a collaborative effort of two DOE organizations, the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem including software,

applications, hardware, advanced system engineering, and early testbed platforms, to support the nation's exascale computing imperative.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.