(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
19 January 2017 (19.01.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/011223 A1

(51) International Patent Classification:
G06F 9/50 (2006.01)

(21) International Application Number:
PCT/US2016/040970

(22) International Filing Date:
5 July 2016 (05.07.2016)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
62/191,219 10 July 2015 (10.07.2015) US

(71) Applicant: **RAMBUS, INC.** [US/US]; 1050 Enterprise Way, Suite 700, Sunnyvale, CA 94089 (US).

(72) Inventor: **LOWERY, Keith**; 1050 Enterprise Way, Suite 700, Sunnyvale, CA 94089 (US).

(74) Agent: **PORTNOVA, Marina**; Lowenstein Sandler LLP, 65 Livingston Avenue, Roseland, New Jersey 07068 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report (Art. 21(3))*

WO 2017/011223 A1

(54) Title: THREAD ASSOCIATED MEMORY ALLOCATION AND MEMORY ARCHITECTURE AWARE ALLOCATION

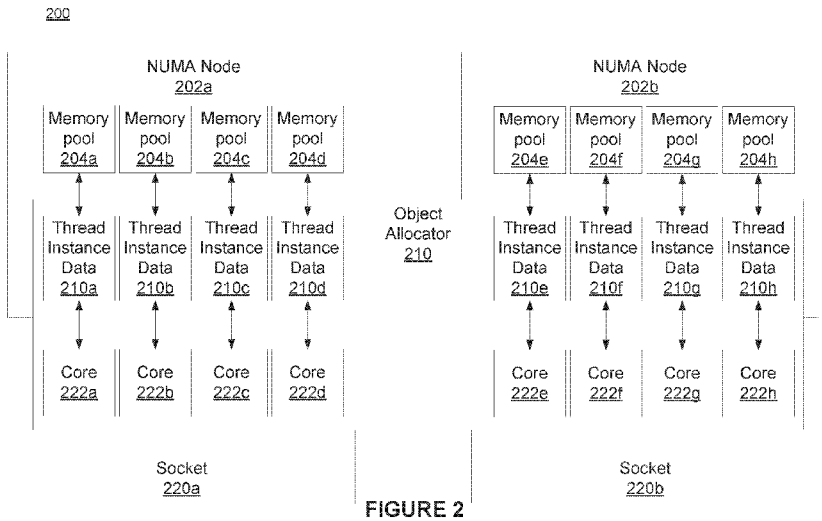


FIGURE 2

(57) Abstract: A method and system for thread aware, class aware, and topology aware memory allocations. Embodiments include a compiler configured to generate compiled code (e.g., for a runtime) that when executed allocates memory on a per class per thread basis that is system topology (e.g., for non-uniform memory architecture (NUMA)) aware. Embodiments can further include an executable configured to allocate a respective memory pool during runtime for each instance of a class for each thread. The memory pools are local to a respective processor, core, etc., where each thread executes.

THREAD ASSOCIATED MEMORY ALLOCATION AND MEMORY ARCHITECTURE AWARE ALLOCATION

BACKGROUND

[001] Increasingly, larger amounts of information are being stored and processed by computing systems. At a base level, these computing systems are configured with one or more processors, each controlling access to corresponding memory. However, efficient use of resources with such large amounts of information can be challenging.

BRIEF DESCRIPTION OF THE DRAWINGS

[002] Further aspects of the present disclosure will become apparent from the following description which is given by way of example only and with reference to the accompanying drawings in which:

[003] Figure 1 shows a block diagram of a system with a non-uniform memory architecture (NUMA), in accordance with various embodiments.

[004] Figure 2 shows a block diagram of system components and associated memory allocations, in accordance with various embodiments.

[005] Figure 3 shows a block diagram of components of a process and allocated memory portions, in accordance with various embodiments.

[006] Figure 4 shows a flowchart of an exemplary electronic component-controlled process for compiling code, in accordance with various embodiments.

[007] Figure 5 shows a flowchart of an exemplary electronic component-controlled process for executing code and allocating memory, in accordance with various embodiments.

[008] Figure 6 is a block diagram of an exemplary computing system including various embodiments.

[009] Figure 7 is a block diagram of an exemplary operating environment, in accordance with various embodiments.

DETAILED DESCRIPTION

[0010] Reference will now be made in detail to the various embodiments of the present disclosure, examples of which are illustrated in the accompanying drawings. Furthermore, in the following detailed description of the present disclosure, numerous specific details are set forth in order to provide a thorough understanding of the present disclosure. However, it will be understood that the present disclosure may be practiced without these specific details. In other instances, well-known methods, procedures, components, and circuits have not been described in detail so as not to unnecessarily obscure aspects of the present disclosure.

[0011] The limits of frequency scaling as used for improving single-thread performance in central processing units (CPUs) have resulted in multiple cores being put into processors to allow more parallelism. The implications of this from a software perspective are significant because parallel programming is often challenging. This is due to the fact that parallel programming necessitates software developers reasoning about problems in ways that can be non-intuitive to human beings.

[0012] The complexity of symmetric multiprocessor (SMP) systems can require software developers reason, not only about parallel algorithms, but also about system-level bottlenecks that impact overall performance. As such, instead of being able to focus on programming domain specific problems, developers may have to consider system architectures in order to achieve maximum performance.

[0013] System-level memory architectures can have a significant impact on performance. Many CPUs include memory controllers on-die and there is a global address space for the CPUs. Motherboard topologies are such that memory performance varies depending on the proximity of a specific location to a specific core that is running code that is accessing the memory. Non-Uniform Memory Architectures (NUMAs) can have runtime characteristics that negatively impact application performance. For example, in one exemplary quad socket system with a random distribution of memory locations for an application's data, 75% of the memory references would be "far" references involving data traversals across the motherboard. In some cases, these quad socket systems are avoided because memory access patterns of dual socket systems perform better by having 50% far memory references as compared to the 75% far memory references. Because the reduction of far memory references can improve performance, developers can consider system topology and memory management when designing the thread architecture of an application in order to achieve improved performance out of SMP systems.

[0014] In contrast, the evolution of software development languages has, for decades, moved away from system topology awareness and toward abstractions thereby making reasoning about non-system problems easier. While object orientated languages and associated runtimes allow developers to reason about a complex problem, object oriented languages hide system and architectural complexities.

[0015] Embodiments discussed below can allow NUMA-aware memory allocations in object-oriented languages in a way that 1) preserves the simplicity for software developers while 2) transparently providing much improved NUMA memory allocation behavior of running applications. In some embodiments, a combination of C++ templates, C++ inheritance, thread-local storage, and NUMA-specific memory pools can be used to override the default "new" and "delete" operators (or similar operators for other programing languages). Developers can thereby be free to program in the way they are accustomed to while the "new" and "delete" operators access thread-specific lock-free memory pools for allocating instance data associated with new objects. Object allocation is fast due to the lock-free design and object allocation is NUMA-local to the node where the allocating thread executes. The lock-free design improves object allocation speed by removing the negative performance impacts of a lock based design. The object allocation is NUMA-local to the node where the thread requesting the object allocation executes thereby improving memory access performance (e.g., by reducing access latency).

[0016] With the memory allocation functionality in class templates, the compiler can generate distinct NUMA memory allocators for each unique class hierarchy that is based upon the underlying template.

[0017] Using the __thread storage class for certain memory variables can further ensure that each class-hierarchy allocator will advantageously allocate from a thread-specific pool of memory, thereby, in some circumstances, substantially eliminating the need for locking or concurrency control at runtime. The __thread storage class allows variable allocation such that there is an instance of the variable per thread. Table 1 shows exemplary C++ code for implementation of thread-specific lock-free memory pools for allocating instance data associated with new objects according to one embodiment.

```
template <typename T>
class fastobject {
```

```
public:
   fastobject() {}
   fastobject() {}
   void * operator new(size_t size)
   {
      …
   }
void operator delete(void *pObject)
   {
      …
   }
   friend void fastobject_init<T>(void);
   friend void fastobject_make_key<T>(void);
protected:
   static pthread_key_t _key;
   static __thread pthread_once_t _once;
   static __thread struct mempool **_pools;
private:
};
```

Table 1 – Exemplary code for thread specific and class specific memory allocation

[0018] The exemplary code of Table 1 combines templates, __thread specific memory pools, and overloading of the object allocation operators (e.g., new and delete). It is appreciated that any class used as a parameter to this template can cause the compiler to automatically generate variables in memory that are either global in scope for the generated class (e.g., _key) or global in scope per thread for the resulting compiler-generated class. The __thread-scoped variables receive per-thread initialization. The memory pools that are initialized to fulfill allocation requests can be explicitly initialized by the template implementation to be from the local NUMA node. This initialization generally occurs the first time a developer calls the "new" operator on a class derived from the class used to initialize the template. Each call to the "new" operator from a different thread can allocate objects from a thread-specific memory pool on the current NUMA node. Thus, a developer can advantageously use object-oriented abstractions of the language without regard to the system topology while achieving location-aware object allocation by default.

[0019] By using a combination of templates with __thread-scoped variables, a majority of the work for doing thread-specific allocations can be done by the compiler and runtime, while the actual allocation code can be simplified down to initializing the NUMA-specific memory

pool and providing allocations therefrom. The __thread-scoped variable and object hierarchies derived from the templates can ensure that the "new" and "delete" operators occur in a thread-specific context.

[0020] The Figures illustrate example components used by various embodiments. Although specific components are disclosed in the Figures, it should be appreciated that such components are exemplary. That is, embodiments are well suited to having various other components or variations of the components recited in the Figures. It is appreciated that the components in the Figures may operate with other components than those presented, and that not all of the components of the Figures are required to achieve the goals of embodiments.

[0021] Figure 1 shows a block diagram of a system with a non-uniform memory architecture (NUMA), in accordance with various embodiments. Figure 1 depicts an exemplary NUMA system 100 having four multi core processors 110a-d with associated local memories 130a-d. The NUMA system 100 includes NUMA nodes 102a-d, local random access memories (RAMs) 104a-d, multi-core processors 110a-d, a Peripheral Component Interconnect (PCI) or PCI express (PCIe) bus 180, and a Network Interface Card (NIC) 150. The PCI bus 180 couples the NUMA nodes 102a-d, multi-core processors 110a-d, and the NIC 150.

[0022] The NUMA nodes 102a-d include RAMs 104a-d, multi-core processors 110a-d, which include memory controllers 115a-d. The local RAMs 104a-d are coupled to one or more processing elements (e.g., CPUs, cores, field-programmable gate arrays (FPGAs), etc.) in the multi-core processors 110a-d. The NUMA nodes 102a-b are coupled via a cross-node link 160 which allows communications between the NUMA nodes 102a-b. The NUMA nodes 102c-d are coupled via a cross-node link 170 which allows communications between the NUMA nodes 102c-d. The NUMA nodes 102a-d can further include processor or core local caches, a generic cache (e.g., shared cache), dynamic random access memory (DRAM), memory coupled to a system data bus, and other memory stores, etc.

[0023] The NIC 150 is configured for coupling the system 100 to a network thereby facilitating communications of system 100 with one or more other systems. The NIC 150 can include receive queues configured for receiving communications (e.g., data packets) from a network coupled to the NIC 150 and communications (e.g., data packets) from one or more processing elements in multi-core sockets 110a-d to communicate to the network.

[0024] The multi-core processor 110a includes cores 111a-d, level 1 (L1) caches 112a-d, level 2 (L2) caches 113a-d, a level 3 (L3) cache 114, and a memory controller 115a. The multi-core processor 110b includes cores 121a-d, L1 caches 122a-d, L2 caches 123a-d, a L3 cache 124, and a memory controller 115b. The multi-core processor 110c includes cores 131a-d, L1 caches 132a-d, L2 caches 133a-d, a L3 cache 134, and a memory controller 115c. The multi-core processor 110d includes cores 141a-d, L1 caches 142a-d, L2 caches 143a-d, a L3 cache 144, and a memory controller 115d. Each core of the multi-core processors 110a-d can have respective associated L1, L2, and L3 caches and local RAM. For example, the core 111a is associated with L1 cache 112a, L2 cache 113a, L3 cache 114, and the local RAM 104a. The multi-core processors 110a-d can each be coupled to local RAMs 130a-d and other components of system 100 via respective multi-core sockets (not shown).

[0025] The memory controllers 115a-d can control access to respective local RAMs 104a-d. The local RAMs 104a-d can function as a shared memory space for allocations of memory for one or more of multi-core processors 110a-d. As a result, memory allocations could occur that are not local to the processing elements. For example, a portion of local RAM 104a could be allocated to a process or thread executing on core 141a which will have increased latency because of being remote from core 141a.

[0026] Embodiments are configured to perform NUMA-aware or topology based allocations of memory thereby advantageously allocating memory that is local to where a process or thread executes. For example, memory allocations for processes or threads executing on cores 111a-d are allocated within local RAM 104a. As another example, memory allocations for processes or threads executing on cores 141a-d are allocated within local RAM 104d. The NUMA-aware or topology based allocations can further be within caches (e.g., L1, L2, and/or L3 caches). For example, memory allocations for processes or threads executing on core 111a of multi-core processor 110a can be from L1 cache 112a, L2 cache 113a, and/or L3 cache 114). The NUMA-aware allocations within the L1, L2, and/or L3 caches can be performed to the extent that a processor allows for cache manipulation and/or allocations at each cache level (e.g., based on the cache level granularity of cache manipulation the processor exposes to software).

[0027] Figure 2 shows a block diagram of system components and associated memory allocations, in accordance with various embodiments. Figure 2 depicts how a multi-threaded application with an object allocator executing on a multi-socket system leverages thread-specific memory pools to satisfy allocation requests. The diagram 200 includes NUMA nodes 202a-b (e.g., NUMA nodes 102a-d), an object allocator 210, and sockets 220a-b.

[0028] The sockets 220a-b can support multiple processing elements, e.g., multi-core processors and/or multiple CPUs. For example, the multi-core processor 110a can be coupled with socket 220a and the multi-core processor 110b can be coupled in socket 220b. As shown, the socket 220a includes cores 222a-d and the socket 222b includes cores 222e-h. The cores 222a-h are configured to execute one or more processes which may have one or more respective threads.

[0029] The object allocator 210 is configured to allocate memory from within NUMA nodes 202a on a per-thread basis. The object allocator 210 allocates memory pools 204a-d corresponding to respective thread instance data 210a-d. The thread instance data 210a-d is associated with respective threads executing on cores 222a-d. The object allocator 210 allocates memory pools 204e-h corresponding to respective thread instance data 210e-h. The thread instance data 210e-h is associated with respective threads executing on cores 222e-h. The object allocator 210 thus allocates the memory pools on a per-instance (e.g., class instance), per-thread basis, and allocates to respective NUMA nodes based on the NUMA topology. Embodiments are thus able to perform class-based allocation of memory that are topology and micro architecture aware.

[0030] For example, the memory pools 204a-d can be allocated from a memory pool A that includes the local RAMs 104a-b and the L1-L3 caches of multi-core processors 110a-b. The memory pools 204e-h can be allocated from a memory pool B that can include the local RAMs 104c-d and the L1-L3 caches of multi-core processors 110c-d. The memory pool A (e.g., a memory pool class A 320a) can be used for memory allocations associated with a class A and the memory pool B (e.g., a memory pool class B 320b) can be used for memory allocations associated with a class B.

[0031] Figure 3 shows a block diagram of components of a process and allocated memory portions, in accordance with various embodiments. Figure 3 depicts exemplary threads of a process 300 with instances of classes and associated memory pools. For each thread there is a respective memory pool allocated for each respective class instance of each thread. The process 300 includes thread 302 and thread 304.

[0032] The thread 302 includes class A instance 310a which is associated with respective memory pool class A 320a. The thread 302 includes class B instance 310b which is associated with respective memory pool class B 320b. The thread 302 further includes class C instance 310c which is associated with respective memory pool class C 320c.

[0033] The thread 304 includes class A instance 312a which is associated with respective memory pool class A 330a. The thread 304 includes class B instance 312b which is associated with respective memory pool class B 330b. The thread 304 further includes class C instance 312c which is associated with respective memory pool class C 330c. The memory pools 320a-c and 330a-c can be allocated on a topology-aware basis, as described herein.

[0034] With any C++ class, a variable can be declared that is scoped to a thread. Each thread that references that class will receive a unique instance of that variable. That variable can further be initialized in a local NUMA memory pool. For the class hierarchy that derives from that class, each thread that references the class will have a respective memory pool. In some embodiments, the compiler generates code for the runtime to manage allocations and data structures. The compiler generates code that initializes a memory pool for each thread. It is appreciated that each allocation of an object that is performed out of that class uses a different memory pool based on the thread that is executing. The memory pools can be a portion of memory that is closest to the processor or core where the thread is running.

[0035] The use of templates in combination with the thread scope variables to get class specific variables allows a respective memory pool for each thread. A respective memory pool is allocated for each class instance for each thread. In other words, each class instance that is unique, for each thread, has a respective memory pool.

[0036] A compiler can be configured to use thread scoped variables combined with templates to have allocation of a thread specific memory pool initialized with the thread scoped variable in the memory pool object or a chunk of memory. The memory pool is further allocated in a location of memory that is in the nearest NUMA node to where the thread is executing. When the compiled program, executes the language, runtime performs the allocation of thread and class specific memory pools allocated based on the system topology. The compiler thus automatically generates code for allocation of thread and class specific memory pools that allocated based on system topology.

[0037] The memory pools are initialized locally based on system topology, e.g., the nearest NUMA node from a system topology standpoint. Thus, for a thread running on a processor requesting an allocation of memory for an object, it will have the memory allocated be local to the processor where the process is running.

[0038] In some embodiments, the thread specific memory pools can be used with non multi-socket systems where there are NUMA issues and where allocations out of a memory pool that are lock free are desired for, among other reasons, performance reasons. The lock free allocations avoid the negative performance impacts of locking, synchronizing, and coordination among threads.

[0039] Embodiments include an executable configured to allocate memory based on object oriented structures (e.g., one or more classes) and an executable configured to allocated memory based on topology. The allocations of memory based on system topology can increase performance, save time, and reduce energy consumption.

[0040] Figure 4 shows a flowchart of an exemplary electronic component-controlled process for compiling code, in accordance with various embodiments. Figure 4 depicts a process 400 that is executed by a compiler in generating compiled code configured for per class and per thread topology (e.g., NUMA) aware allocations.

[0041] The generated code can be configured to allocate memory pools based on topology alone, class alone, or based on a combination of class and topology. The generated code can be configured to allocate first based on class to select a memory pool and then based on topology for a particular portion of the selected memory pool. For example, allocations for an instance of a class A of a process executing on the multi-core processor 110a may be made from a memory pool A which includes local RAMs 104a-b while allocations for an instance of a class B of a process executing on the multi-core processor 110c may be made from a memory pool B which includes local RAMs 104c-d. The specific locations of memory used for allocations within the memory pools, by the generated code, can be based on topology of the NUMA system. For example, a thread executing multi-core processor 110a would have memory allocations within the local RAM 104a selected instead of memory allocations within the local RAM 104b because the local RAM 104a is closer to the multi-core processor 110a. As another example, a thread executing on the core 111a would have memory locations within the L1 cache 112a selected instead of memory locations within the L1 cache 112b because of the proximity of the L1 cache 112a to the core 111a.

[0042] At block 402, code is accessed. The compiler can access the code from a data store or receive the code from a data source.

[0043] At block 404, attributes are determined. The attributes are configured to communicate to the runtime thread specific attributes for memory allocations. The attributes can include whether to allocate memory pools based on topology alone, class alone, or based on class and topology. The attributes can be configured to allocate first based on class to select a memory pool and then based on topology for a particular portion of the selected memory pool. For example, allocations for an instance of a class A of a process executing on the multi-core processor 110a may be made from a memory pool A which includes local RAMs 104a-b while allocations for an instance of a class B of a process executing on the multi-core processor 110c may be made from a memory pool B which includes local RAMs 104c-d. The specific locations of memory used for allocations within the memory pools, by the generated code, can be based on topology of the NUMA system. For example, a thread executing multi-core processor 110a would have memory locations within local RAM 104a selected instead of memory locations within local RAM 104b because local RAM 104a is closer to multi-core processor 110a. As another example, a thread executing on core 111a would have memory locations within L1 cache 112a selected instead of memory locations within L1 cache 112b because of the proximity of L1 cache 112a to the core 111a.

[0044] At block 406, memory allocation code is generated based on the attributes. The code is generated in a form that the runtime executable will use to create data structures in a thread-specific way. The memory allocation code is generated based on the attributes and the runtime executable can make thread specific memory allocations and class specific memory pool allocations based on the attributes. The application runtime is configured by the compiler to allocate memory from specific thread relative memory pools that are in nearby memory locations (e.g., based on system topology). The generated code thus allocates memory pools that are topology-local, for each instance of a class for each thread.

[0045] In some embodiments, the compiler accesses one or more portions of a template. The template is used to generate a class at compile time based on the template and the code that was accessed. The use of templates allows the generation of code with a class-hierarchy-specific allocator. The compiler can generate code for performing the NUMA aware memory allocations on per-thread and per-class basis before a call of the "new" operator in the code will be executed.

[0046] The generated code can be configured to allocate memory pools based on topology alone, class alone, or based on class and topology, according to the determined attributes. The generated code can thus be configured to allocate first based on class to a selected memory pool and then based on topology to allocate a particular portion of the selected memory pool.

[0047] At block 408, compiled code is generated (e.g., by a compiler). The compiled code includes code generated based on the attributes and based on the source code accessed (e.g., function calls, etc.).

[0048] At block 410, the compiled code is output. The compiled code can be output to a data store and be an executable file.

[0049] Table 2 shows example C++ code for an exemplary class with thread specific allocations, as described herein.

```
class customer {
        static string TableName="Cust_rec";
        int cust_ID;
        __thread static int mem_pool = near_mempool();
};
```

Table 2 – Exemplary portion of class for memory allocation

[0050] The code of Table 2 includes a class called customer with a static string variable, a non-attributed int variable, and a __thread static variable. When an instance of the class customer is created with the "new" operator, a portion of memory will be allocated for each variable that does not have a static or thread attribute. For example, a portion of memory will be allocated for an integer called the cust_ID. Memory can also be allocated for the tablename string variable in a single global location for all occurrences of the customer class.

[0051] The __thread static int mem_pool variable allocates a respective portion of memory for each thread. In some embodiments, a thread identifier is assigned to each memory pool so that each thread for a respective class instance will have a respective memory pool. The compiler generated code can include an automatic initialization portion to initialize memory pools based on system topology so that the memory pools are allocated in a NUMA-aware manner. For example, the automatic initialization of a language runtime could be configured to initialize 4 or 8 threads on execution and allocate memory pools for each instance of each class for each thread. The automatic initializer can be called the first time an object is instantiated.

[0052] The near_mempool() function allocates a memory pool to the mem_pool variable based on system topology (e.g., for NUMA systems). The near_mempool() function may be called once per thread automatically by the compiler generated code. The runtime will detect

whether the thread specific mem_pool variable has been allocated and initialized. In some embodiments, the near_mempool() function examines the system topology during execution and determines a portion of memory to allocate for the memory pool based on the system topology. A thread specific variable can be allocated for each instance of the class and each different derivation of the class when the template is parameterized with the derived class and not a parent class. The near_mempool() function can return an address or pointer of an allocated memory pool.

[0053] Figure 5 shows a flowchart of an exemplary electronic component-controlled process for executing code and allocating memory, in accordance with various embodiments. Figure 5 depicts a process 500 of executing compiled code that is configured for topology and class aware allocations on a per-class and per-thread basis. Process 5000 can perform allocation of memory based on the proximity of a core executing a thread and memory proximate to the core executing the thread. For example, portions of the local RAM 104d can be allocated for a thread executing on the core 141d.

[0054] At block 502, compiled code is accessed. The compiled code can be accessed from a data store (e.g., local storage) or received for execution (e.g., over from a network).

[0055] At block 504, an initializer is executed. The initializer can be part of a language runtime as described herein. The initializer can allocate thread and class specific memory pools as part of the initialization process for one or more threads, as described herein. The initializer can be configured to select a memory pool based on class and/or topology and then select an area with the selected memory pool for one or more memory allocations based on topology. The initializer may access data related to the classes and other objects of a process that is to be executed. For example, an initializer for a process with a class A to be executed on core 111c of multi-core processor 110a can select a memory pool A including local RAMs 104a-b for memory allocations to be associated with the class A of the process. The initializer can further select portions of local RAM 104a based on the proximity of the local RAM 104a to multi-core processor 110a.

[0056] At block 506, one or more threads are executed. The one or more threads can be executed after the allocations by the initializer, as described herein. The compiled code including functions calls and other computations can be executed as one or more threads.

[0057] At block 508, additional memory pools are allocated. The additional memory pools can be allocated as more threads of a process are generated or as threads request allocations of memory for new instances of classes. For example, an initializer for a process with a class A to be executed on core 111c of multi-core processor 110a can select a memory pool A including local RAMs 104a-b for memory allocations to be associated with the class A of the process. The initializer can further select portions of local RAM 104a based on the proximity of the local RAM 104a to the core 111c.

[0058] Figure 6 is a block diagram of an exemplary computing system 600 including various embodiments. Figure 6 depicts a computing system 600 for performing the functions, processes, etc., described herein. For example, the computing system 600 may compile code and/or execute code for allocating memory, as described herein. Computing system 600 broadly represents any single or multi-processor computing device or system capable of executing computer-readable instructions. Examples of computing system 600 include, without limitation, workstations, laptops, client-side terminals, servers, supercomputers, distributed computing systems, handheld devices, or any other computing system or device. In its most basic configuration, computing system 600 may include at least one processor 614 and a system memory 616.

[0059] Processor 614 generally represents any type or form of processing unit capable of processing data or interpreting and executing instructions. In certain embodiments, processor 614 may receive instructions from a software application or module. These instructions may cause processor 614 to perform the functions of one or more of the example embodiments described and/or illustrated herein. For example, processor 614 may perform and/or be a means for performing, either alone or in combination with other elements, one or more of the receiving, accessing, identifying, determining, using, associating, allocating, executing, initializing, compiling implementing, and providing described herein. Processor 614 may also perform and/or be a means for performing any other steps, methods, or processes described and/or illustrated herein.

[0060] System memory 616 generally represents any type or form of volatile or non-volatile storage device or medium capable of storing data and/or other computer-readable instructions. Examples of system memory 616 include, without limitation, RAM, ROM, FLASH memory, or any other suitable memory device. Although not required, in certain embodiments computing system 600 may include both a volatile memory unit (such as, for example, system memory 616) and a non-volatile storage device (such as, for example, primary storage device 632. For example, the processor 614 can be the multi-core processor 110a and the system memory 616

can be the local RAM 104a. The local RAM 104a can store code compiled, as described with respect to Figure 4. The local RAM 104a can further be used for used for class and topology based memory allocations, as described with respect to Figure 5, for portions of compiled code executing on one or more of the cores 111a-d of the multi-core processor 110a.

[0061] Computing system 600 may also include one or more components or elements in addition to processor 614 and system memory 616. For example, in the embodiment of Figure 6, computing system 600 includes a memory controller 618, an I/O controller 620, and a communication interface 622, each of which may be interconnected via a communication infrastructure 612.

[0062] Communication infrastructure 612 generally represents any type or form of infrastructure capable of facilitating communication between one or more components of a computing device. Examples of communication infrastructure 612 include, without limitation, a communication bus (such as an ISA, PCI, PCIe, or similar bus) and a network. In one embodiment, system memory 616 communicates via a dedicated memory bus.

[0063] Memory controller 618 generally represents any type or form of device capable of handling memory or data or controlling communication between one or more components of computing system 600. For example, memory controller 618 may control communication between processor 614, system memory 616, and I/O controller 620 via communication infrastructure 612. Memory controller may perform and/or be a means for performing, either alone or in combination with other elements, one or more of the operations or features described herein.

[0064] I/O controller 620 generally represents any type or form of module capable of coordinating and/or controlling the input and output functions of a computing device. For example, I/O controller 620 may control or facilitate transfer of data between one or more elements of computing system 600, such as processor 614, system memory 616, communication interface 622, display adapter 626, input interface 630, and storage interface 634. I/O controller 620 may be used, for example, to perform and/or be a means for performing, either alone or in combination with other elements, one or more of the operations described herein. I/O controller 620 may also be used to perform and/or be a means for performing other operations and features set forth in the instant disclosure.

[0065] Communication interface 622 broadly represents any type or form of communication device or adapter capable of facilitating communication between example computing system 600 and one or more additional devices. For example, communication interface 622 may facilitate communication between computing system 600 and a private or public network including additional computing systems. Examples of communication interface 622 include, without limitation, a wired network interface (such as a network interface card), a wireless network interface (such as a wireless network interface card), a modem, and any other suitable interface. In one embodiment, communication interface 622 provides a direct connection to a remote server via a direct link to a network, such as the Internet. Communication interface 622 may also indirectly provide such a connection through, for example, a local area network (such as an Ethernet network), a personal area network, a telephone or cable network, a cellular telephone connection, a satellite data connection, or any other suitable connection.

[0066] Communication interface 622 may also represent a host adapter configured to facilitate communication between computing system 600 and one or more additional network or storage devices via an external bus or communications channel. Examples of host adapters include, without limitation, SCSI host adapters, USB host adapters, IEEE (Institute of Electrical and Electronics Engineers) 694 host adapters, Serial Advanced Technology Attachment (SATA) and External SATA (eSATA) host adapters, Advanced Technology Attachment (ATA) and Parallel ATA (PATA) host adapters, Fibre Channel interface adapters, Ethernet adapters, or the like. Communication interface 622 may also allow computing system 600 to engage in distributed or remote computing. For example, communication interface 622 may receive instructions from a remote device or send instructions to a remote device for execution. Communication interface 622 may perform and/or be a means for performing, either alone or in combination with other elements, one or more of the operations disclosed herein. Communication interface 622 may also be used to perform and/or be a means for performing other operations and features set forth in the instant disclosure.

[0067] As illustrated in Figure 6, computing system 600 may also include at least one display device 624 coupled to communication infrastructure 612 via a display adapter 626. Display device 624 generally represents any type or form of device capable of visually displaying information forwarded by display adapter 626. Similarly, display adapter 626 generally represents any type or form of device configured to forward graphics, text, and other data from communication infrastructure 612 (or from a frame buffer, as known in the art) for display on display device 624.

[0068] As illustrated in Figure 6, computing system 600 may also include at least one input device 628 coupled to communication infrastructure 612 via an input interface 630. Input device 628 generally represents any type or form of input device capable of providing input, either computer- or human-generated, to computing system 600. Examples of input device 628 include, without limitation, a keyboard, a pointing device, a speech recognition device, or any other input device. In one embodiment, input device 628 may perform and/or be a means for performing, either alone or in combination with other elements, one or more of the operations disclosed herein. Input device 628 may also be used to perform and/or be a means for performing other operations and features set forth in the instant disclosure.

[0069] As illustrated in Figure 6, computing system 600 may also include a primary storage device 632 and a backup storage device 633 coupled to communication infrastructure 612 via a storage interface 634. Storage devices 632 and 633 generally represent any type or form of storage device or medium capable of storing data and/or other computer-readable instructions. For example, storage devices 632 and 633 may be a magnetic disk drive (e.g., a so-called hard drive), a solid state disk, a floppy disk drive, a magnetic tape drive, an optical disk drive, a FLASH drive, or the like. Storage interface 634 generally represents any type or form of interface or device for transferring data between storage devices 632 and 633 and other components of computing system 600.

[0070] Continuing with reference to Figure 6, storage devices 632 and 633 may be configured to read from and/or write to a removable storage unit configured to store computer software, data, or other computer-readable information. Examples of suitable removable storage units include, without limitation, a floppy disk, a magnetic tape, an optical disk, a FLASH memory device, or the like. Storage devices 632 and 633 may also include other similar structures or devices for allowing computer software, data, or other computer-readable instructions to be loaded into computing system 600. For example, storage devices 632 and 633 may be configured to read and write software, data, or other computer-readable information. Storage devices 632 and 633 may also be a part of computing system 600 or may be separate devices accessed through other interface systems.

[0071] Storage devices 632 and 633 may be used to perform, and/or be a means for performing, either alone or in combination with other elements, one or more of the operations disclosed herein. Storage devices 632 and 633 may also be used to perform, and/or be a means for performing, other operations and features set forth in the instant disclosure.

[0072] Many other devices or subsystems may be connected to computing system 600. Conversely, all of the components and devices illustrated in Figure 6 need not be present to practice the embodiments described herein. The devices and subsystems referenced above may also be interconnected in different ways from that shown in Figure 6. Computing system 600 may also employ any number of software, firmware, and/or hardware configurations. For example, the example embodiments disclosed herein may be encoded as a computer program (also referred to as computer software, software applications, computer-readable instructions, or computer control logic) on a computer-readable medium.

[0073] The computer-readable medium containing the computer program may be loaded into computing system 600. All or a portion of the computer program stored on the computer-readable medium may then be stored in system memory 616 and/or various portions of storage devices 632 and 633. When executed by processor 614, a computer program loaded into computing system 600 may cause processor 614 to perform and/or be a means for performing the functions of the example embodiments described and/or illustrated herein. Additionally or alternatively, the example embodiments described and/or illustrated herein may be implemented in firmware and/or hardware. For example, computing system 600 may be configured as an ASIC adapted to implement one or more of the embodiments disclosed herein.

[0074] Figure 7 is a block diagram of an exemplary operating environment 700 in which client systems 710, 720, and 730 and servers 740 and 745 may be coupled to a network 750. Client systems 710, 720, and 730 generally represent any type or form of computing device or system, such as computing system 600 of Figure 6.

[0075] Similarly, servers 740 and 745 generally represent computing devices or systems, such as application servers or database servers, configured to provide various database services and/or run certain software applications. In some embodiments, the servers 740 may include and/or execute an executable 742 configured for compiling code or allocating memory, as described herein. In some embodiments, the servers 745 may include and/or execute an executable 746 configured for compiling code or allocating memory, as described herein. For example, one or more of the executables 742 and 746 can be a compiler for compiling code as described with respect to Figure 4. As another example, one or more of the executables 742 and 746 can be a program compiled as described with respect to Figure 4 and when executed allocate memory based on a class and/or topology basis as described with respect to Figure 5. Network 750 generally represents any telecommunication or computer network including, for example, an intranet, a WAN, a LAN, a PAN, or the Internet.

[0076] As illustrated in Figure 7, one or more storage devices 760(1)-(L) may be directly attached to server 740. Similarly, one or more storage devices 770(1)-(N) may be directly attached to server 745. Storage devices 760(1)-(L) and storage devices 770(1)-(N) generally represent any type or form of storage device or medium capable of storing data and/or other computer-readable instructions. Storage devices 760(1)-(L) and storage devices 770(1)-(N) may represent NAS devices configured to communicate with servers 740 and 745 using various protocols, such as NFS, SMB, or CIFS.

[0077] Servers 740 and 745 may also be connected to a SAN fabric 780. SAN fabric 780 generally represents any type or form of computer network or architecture capable of facilitating communication between storage devices. SAN fabric 780 may facilitate communication between servers 740 and 745 and storage devices 790(1)-(M) and/or an intelligent storage array 795. SAN fabric 780 may also facilitate, via network 750 and servers 740 and 745, communication between client systems 710, 720, and 730 and storage devices 790(1)-(M) and/or intelligent storage array 795 in such a manner that devices 790(1)-(M) and array 795 appear as locally attached devices to client systems 710, 720, and 730. As with storage devices 760(1)-(L) and storage devices 770(1)-(N), storage devices 790(1)-(M) and intelligent storage array 795 generally represent any type or form of storage device or medium capable of storing data and/or other computer-readable instructions.

[0078] With reference to computing system 600 of Figure 6, a communication interface, such as communication interface 622, may be used to provide connectivity between each client system 710, 720, and 730 and network 750 of Figure 7. Client systems 710, 720, and 730 may be able to access information on server 740 or 745 using, for example, a Web browser or other client software. Such software may allow client systems 710, 720, and 730 to access data hosted by server 740, server 745, storage devices 760(1)-(L), storage devices 770(1)-(N), storage devices 790(1)-(M), or intelligent storage array 795. Although Figure 7 depicts the use of a network (such as the Internet) for exchanging data, the embodiments described herein are not limited to the Internet or any particular network-based environment.

[0079] The above described embodiments may be used, in whole or in part, in systems that process large amounts of data and/or have tight latency constraints, and, in particular, with systems using one or more of the following protocols and formats: Key-Value (KV) Store, Memcached, Redis, Neo4J (Graph), Fast Block Storage, Swap Device, and Network RAMDisk. In addition, the above described embodiments may be used, in whole or in part, in systems employing virtualization, Virtual Desktop Infrastructure (VDI), distributed storage and

distributed processing (e.g., Apache Hadoop), data analytics cluster computing (e.g., Apache Spark), Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and other cloud computing platforms (e.g., Vmware vCloud, Open Stack, and Microsoft Azure). Further, the above described embodiments may be used, in whole or in party, in systems conducting various types of computing, including Scale Out, Disaggregation, Multi-Thread/Distributed Processing, RackScale, Data Center Scale Computing, Elastic Memory Provisioning, Memory as a Service, page migration and caching and Application Offloading/Acceleration and Integration, using various types of storage, such as Non-Volatile Memory Express, Flash, Multi-Tenancy, Internet Small Computer System Interface (iSCSI), Object Storage, Scale Out storage, and using various types of networking, such as 10/40/100GbE, Software-Defined Networking, Silicon Photonics, Rack TOR Networks, and Low-Latency networking.

[0080] While the foregoing disclosure sets forth various embodiments using specific block diagrams, flowcharts, and examples, each block diagram component, flowchart step, operation, and/or component described and/or illustrated herein may be implemented, individually and/or collectively, using a wide range of hardware, software, or firmware (or any combination thereof) configurations. In addition, any disclosure of components contained within other components should be considered as examples because many other architectures can be implemented to achieve the same functionality.

[0081] The foregoing description, for purpose of explanation, has been described with reference to specific embodiments. However, the illustrative discussions above are not intended to be exhaustive or to limit the disclosure to the precise forms disclosed. Many modifications and variations are possible in view of the above teachings. The embodiments were chosen and described in order to best explain the principles of the disclosure and its practical applications, to thereby enable others skilled in the art to best utilize the disclosure and various embodiments with various modifications as may be suited to the particular use contemplated.

[0082] Embodiments according to the present disclosure are thus described. While the present disclosure has been described in particular embodiments, it should be appreciated that the disclosure should not be construed as limited by such embodiments, but rather construed according to the below claims.

CLAIMS

What is claimed:

1. A method comprising:

accessing code to be compiled, wherein the code comprises a plurality of instances of an allocation operator for memory allocation;

compiling the accessed code into a set of compiled instructions that are configured to:

associate a first memory pool of a non-uniform memory architecture (NUMA) memory with a first thread executing on a first processing element;

associate a second memory pool of the NUMA memory with a second thread executing on a second processing element, wherein the first memory pool has a different memory architecture than the second memory pool;

allocate memory from the first memory pool for instances of the allocation operator on a per object basis for the first thread; and

allocate memory from the second memory pool for instances of the allocation operator on a per object basis for the second thread.

2. The method of Claim 1, wherein the first processing element is a processor.

3. The method of Claim 1, wherein the first processing element is a processor core.

4. The method of Claim 1, wherein the first memory pool is allocated based on proximity of the first memory pool to the first processing element.

5. The method of Claim 1, wherein the first memory pool is associated with an instance of a class of the first thread.

6. The method of Claim 1, wherein the first memory pool is associated with a thread scoped variable.

7. The method of Claim 1, wherein the NUMA memory comprises a plurality of memory modules and the first and second processing elements are reconfigurable.

8. The method of Claim 1, wherein the code is substantially compatible with C++ code.

9. The method of Claim 1, wherein the allocation operator is a modified new operator.

10. The method of Claim 9, wherein the modified new operator is based on a template configured for overloading allocation.

11. The method of Claim 1, wherein the allocation operator is a modified delete operator.

12. The method of Claim 11, wherein the modified delete operator is based on a template configured for overloading allocation.

13. A non-transitory computer-readable storage medium comprising instructions stored thereon, the instructions configured to cause a computer system to:

execute, on a first processing element, compiled code corresponding to an operator enabling NUMA aware allocation of memory;

determine that an execution occurs on the first processing element and identify a first NUMA node as local to the first processing element, in response to execution of the compiled code corresponding to the operator enabling NUMA aware allocation of memory;

initialize a first memory pool on the first NUMA node in response to the operator enabling the NUMA aware allocation of memory;

execute, on a second processing element, compiled code corresponding to the operator enabling NUMA aware allocation of memory;

determine that an execution occurs on the second processing element and identify a second NUMA node as local to the second processing element, in response to execution of the compiled code corresponding to the operator enabling NUMA aware allocation of memory; and

initialize a second memory pool on the second NUMA node in response to the operator enabling the NUMA aware allocation of memory.

14. The non-transitory computer-readable storage medium of Claim 13, wherein the plurality of NUMA nodes comprises a plurality of memory modules and the first and second processing elements are reconfigurable.

15. The non-transitory computer-readable storage medium of Claim 13, wherein the compiled code is compiled C++ code.

16. The non-transitory computer-readable storage medium of Claim 13, wherein the operator enabling NUMA aware allocation of memory is a modified new operator.

17. The non-transitory computer-readable storage medium of Claim 16, wherein the modified new operator is based on a template configured for overloading allocation.

18. A method comprising:

receiving code to be compiled, the code comprising: a first class; a second class; and a plurality of instances of an operator for memory allocation with respect to at least one of the first class and the second class;

compiling the received code into a set of compiled instructions configured to:

associate a first set of memory pools of a first non-uniform memory architecture (NUMA) memory with the first class;

associate a second set of memory pools of a second NUMA memory with the second class, wherein the first memory pool has a different memory architecture than the second memory pool;

allocate memory from a memory pool within the first set of memory pools for instances of the operator for memory allocation with respect to the first class; and

allocate memory from a memory pool within the second set of memory pools for instances of the operator for memory allocation with respect to the second class.

19. The method of Claim 18, wherein the first set of memory pools is within a respective processor's local cache while the second set of memory pools is within a respective processor's shared cache.

20. The method of Claim 18, wherein the allocation of memory from the memory pool within the first set of memory pools is based on NUMA aware allocation comprising a determination of which part of the memory pool within the first set of memory pools to allocate.

21. The method of Claim 20, wherein the first set of memory pools comprises a local cache and non local cache relative to a respective processor core.

22. The method of Claim 18, wherein the code is C++ code.

23. The method of Claim 18, wherein the operator enabling class aware allocation of memory is a modified new operator.

24. The method of Claim 23, wherein the modified new operator is based on a template configured for overloading allocation.

25. A non-transitory computer-readable storage medium comprising instructions stored thereon, the instructions configured to cause a computer system to:

determine that a first plurality of non-uniform memory architecture (NUMA) nodes from within a first set of memory pools are to be used for memory allocations with respect to a first class and a second plurality of NUMA nodes from within a second set of memory pools are to be used for memory allocations with respect to a second class;

execute, on a processing element, compiled code corresponding to an operator enabling class aware allocation of memory;

in response to execution of the compiled code corresponding to the operator enabling class aware allocation of memory, determine that the allocation occurs with respect to the first class and identify a first memory pool from within the first set of memory pools to be used for memory allocation with respect to the first class;

initialize the NUMA node from with the first set of memory pools in response to the operator enabling class aware allocation of memory;

execute, on the processing element, compiled code corresponding to the operator enabling class aware allocation of memory;

in response to the execution of compiled code corresponding to the operator enabling class aware allocation of memory, determine that the allocation of the memory occurs with respect to the second class and identify a second memory pool within the second set of memory pools to be used for memory allocation with respect to the second class; and

initialize the NUMA node from within the second set of memory pools in response to the operator enabling class aware allocation of memory.

26. The non-transitory computer-readable storage medium of Claim 25, wherein the first plurality of NUMA nodes comprises a plurality of memory modules and the processing element is reconfigurable.

27. The non-transitory computer-readable storage medium of Claim 25, wherein the compiled code is compiled C++ code.

28. The non-transitory computer-readable storage medium of Claim 25, wherein the operator enabling class aware allocation of memory is a modified new operator.

29. The non-transitory computer-readable storage medium of Claim 28, wherein the modified new operator is based on a template configured for overloading allocation.

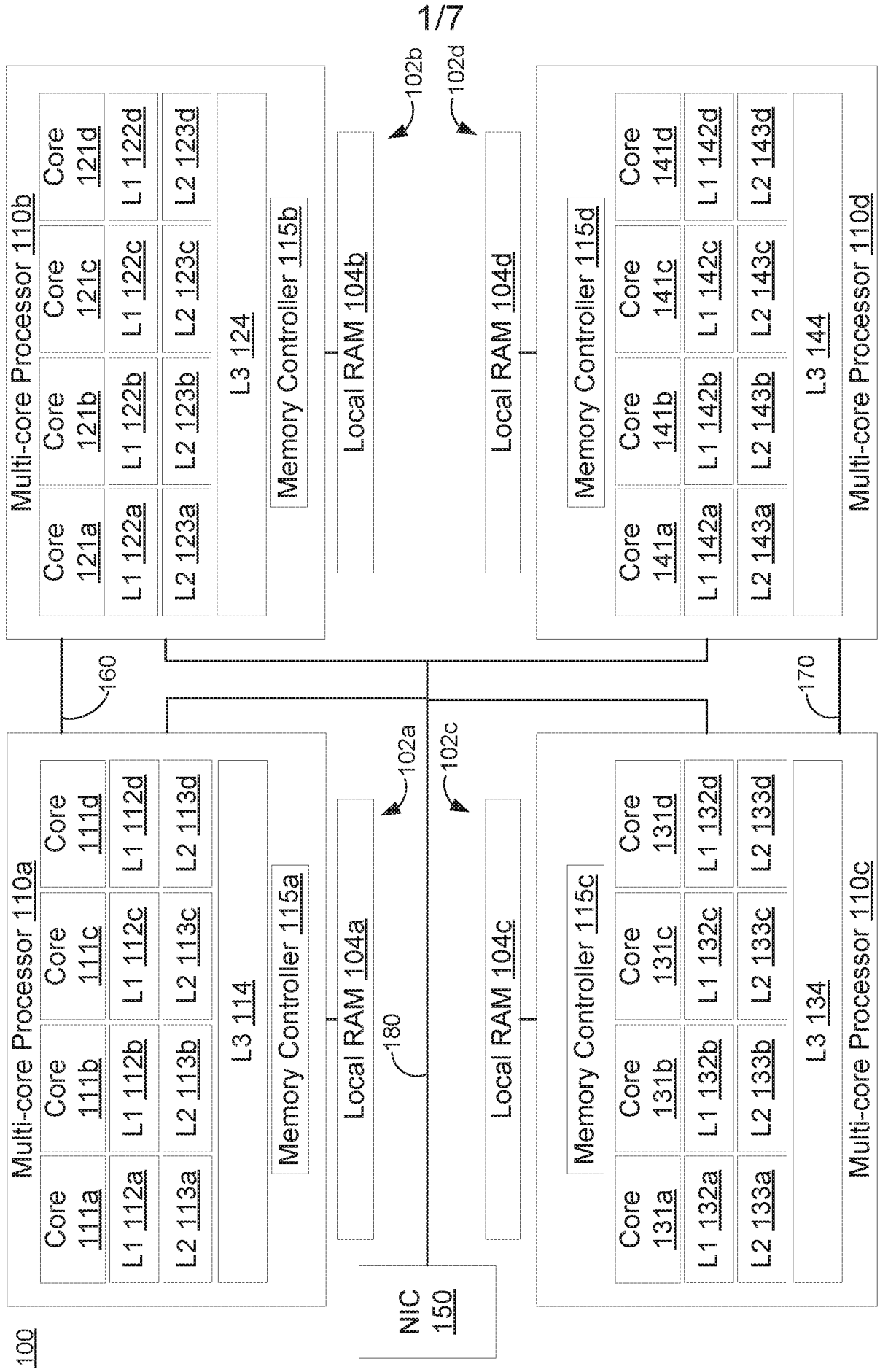

100
Multi-core Processor 110a
Core 111a
Core 111b
Core 111c
Core 111d
L1 112a
L1 112b
L1 112c
L1 112d
L2 113a
L2 113b
L2 113c
L2 113d
L3 114
Memory Controller 115a
Local RAM 104a
Multi-core Processor 110b
Core 121a
Core 121b
Core 121c
Core 121d
L1 122a
L1 122b
L1 122c
L1 122d
L2 123a
L2 123b
L2 123c
L2 123d
L3 124
Memory Controller 115b
Local RAM 104b
Multi-core Processor 110c
Core 131a
Core 131b
Core 131c
Core 131d
L1 132a
L1 132b
L1 132c
L1 132d
L2 133a
L2 133b
L2 133c
L2 133d
L3 134
Memory Controller 115c
Local RAM 104c
Multi-core Processor 110d
Core 141a
Core 141b
Core 141c
Core 141d
L1 142a
L1 142b
L1 142c
L1 142d
L2 143a
L2 143b
L2 143c
L2 143d
L3 144
Memory Controller 115d
Local RAM 104d
NIC 150
160
170
180
102a
102b
102c
102d

FIGURE 1

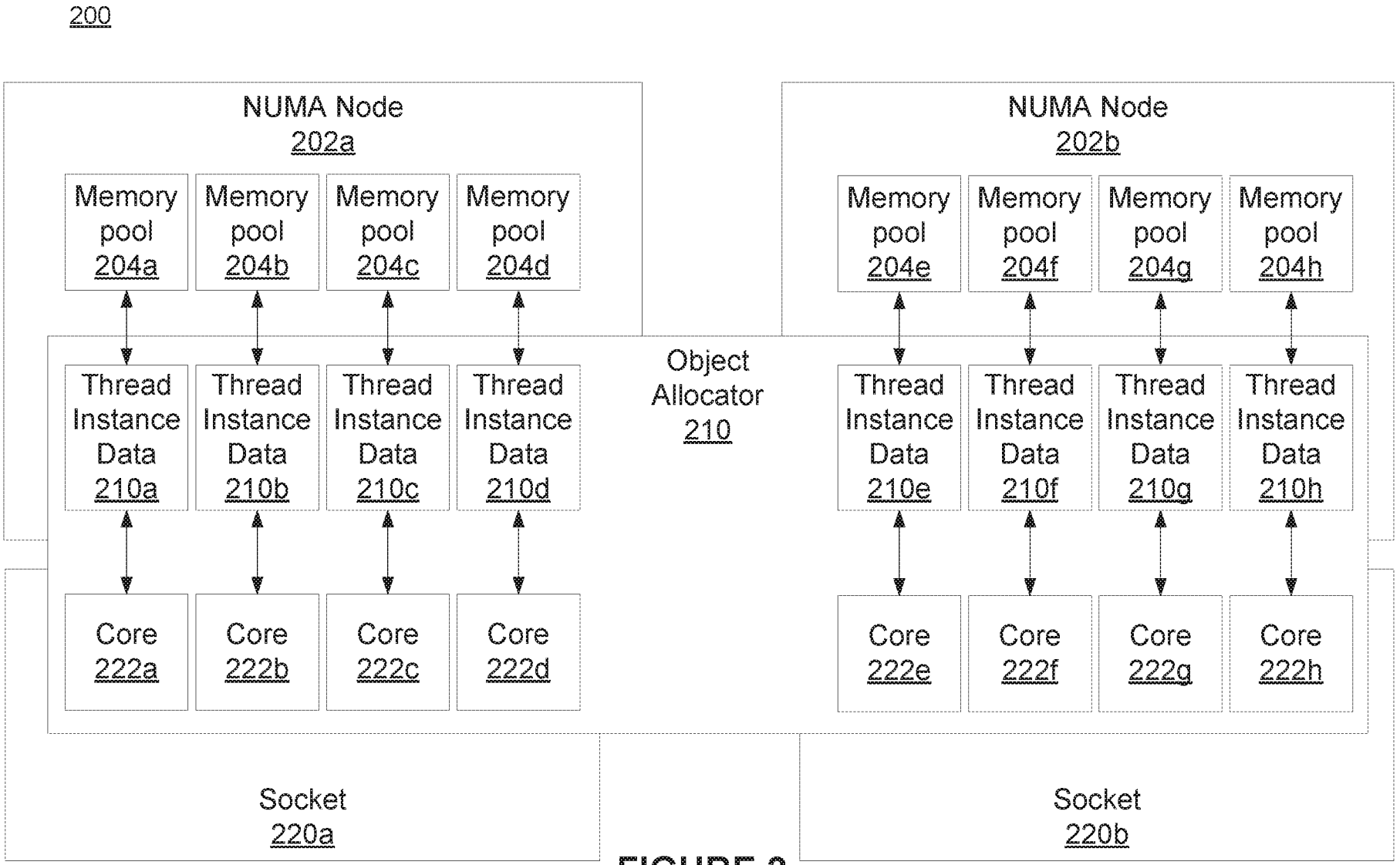

200
NUMA Node
202a
NUMA Node
202b
Memory pool 204a
Memory pool 204b
Memory pool 204c
Memory pool 204d
Memory pool 204e
Memory pool 204f
Memory pool 204g
Memory pool 204h
Thread Instance Data 210a
Thread Instance Data 210b
Thread Instance Data 210c
Thread Instance Data 210d
Thread Instance Data 210e
Thread Instance Data 210f
Thread Instance Data 210g
Thread Instance Data 210h
Object Allocator 210
Core 222a
Core 222b
Core 222c
Core 222d
Core 222e
Core 222f
Core 222g
Core 222h
Socket 220a
Socket 220b

FIGURE 2

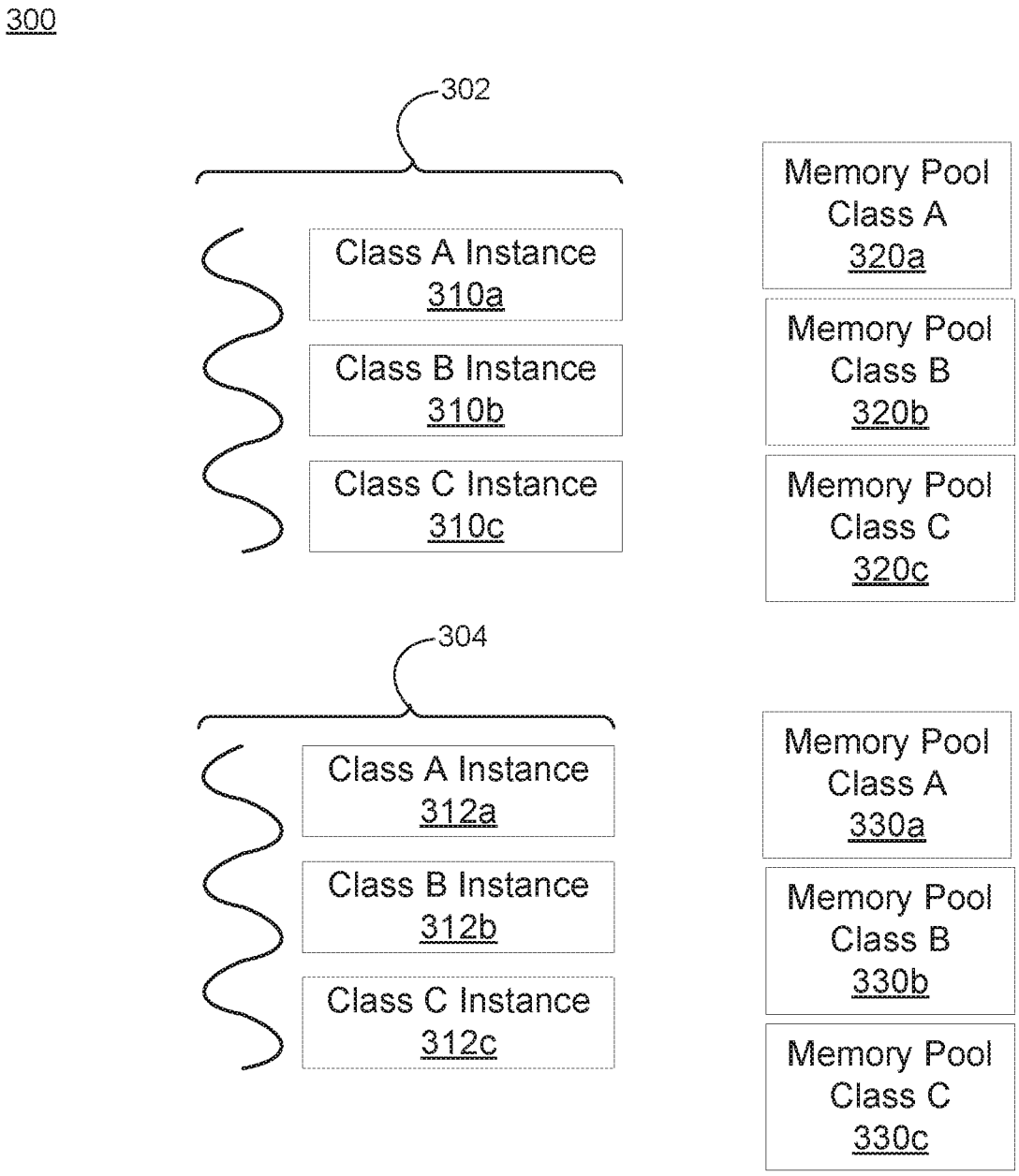

300
302
Class A Instance
310a
Class B Instance
310b
Class C Instance
310c
Memory Pool
Class A
320a
Memory Pool
Class B
320b
Memory Pool
Class C
320c
304
Class A Instance
312a
Class B Instance
312b
Class C Instance
312c
Memory Pool
Class A
330a
Memory Pool
Class B
330b
Memory Pool
Class C
330c

FIGURE 3

400

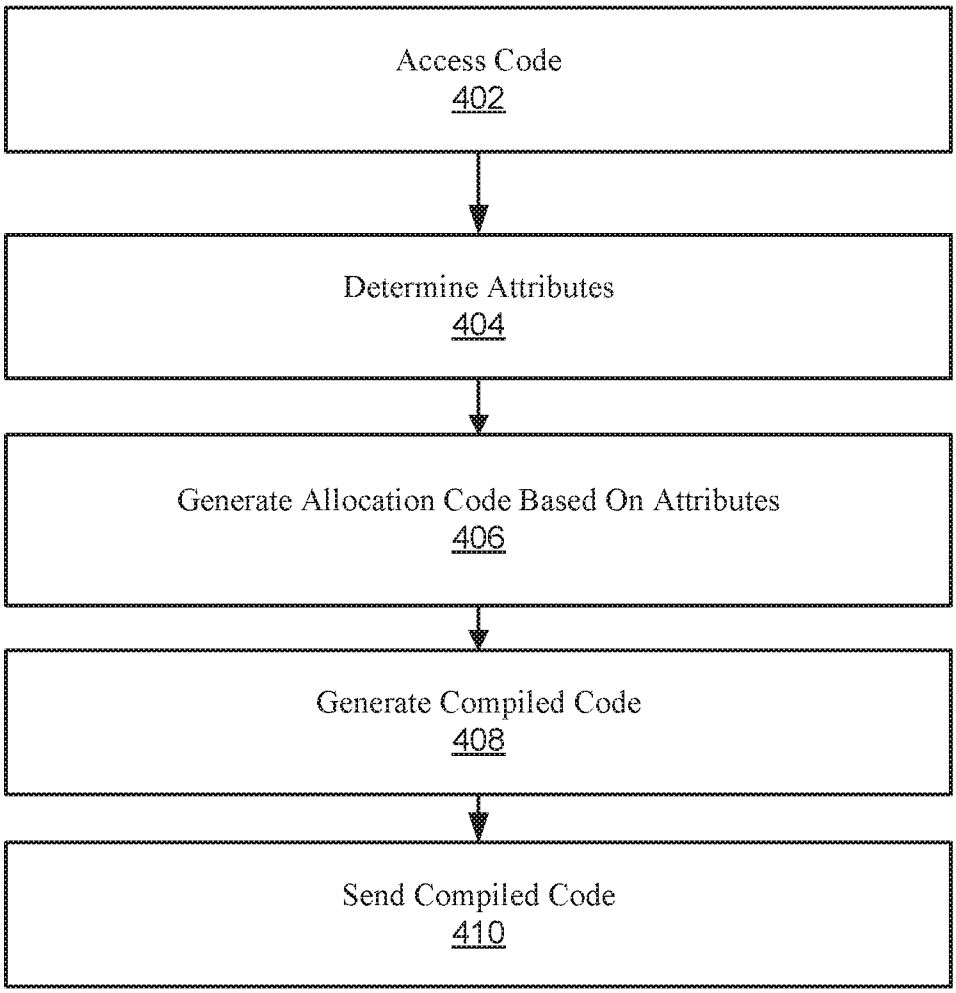

Access Code
402
Determine Attributes
404
Generate Allocation Code Based On Attributes
406
Generate Compiled Code
408
Send Compiled Code
410

FIGURE 4

500

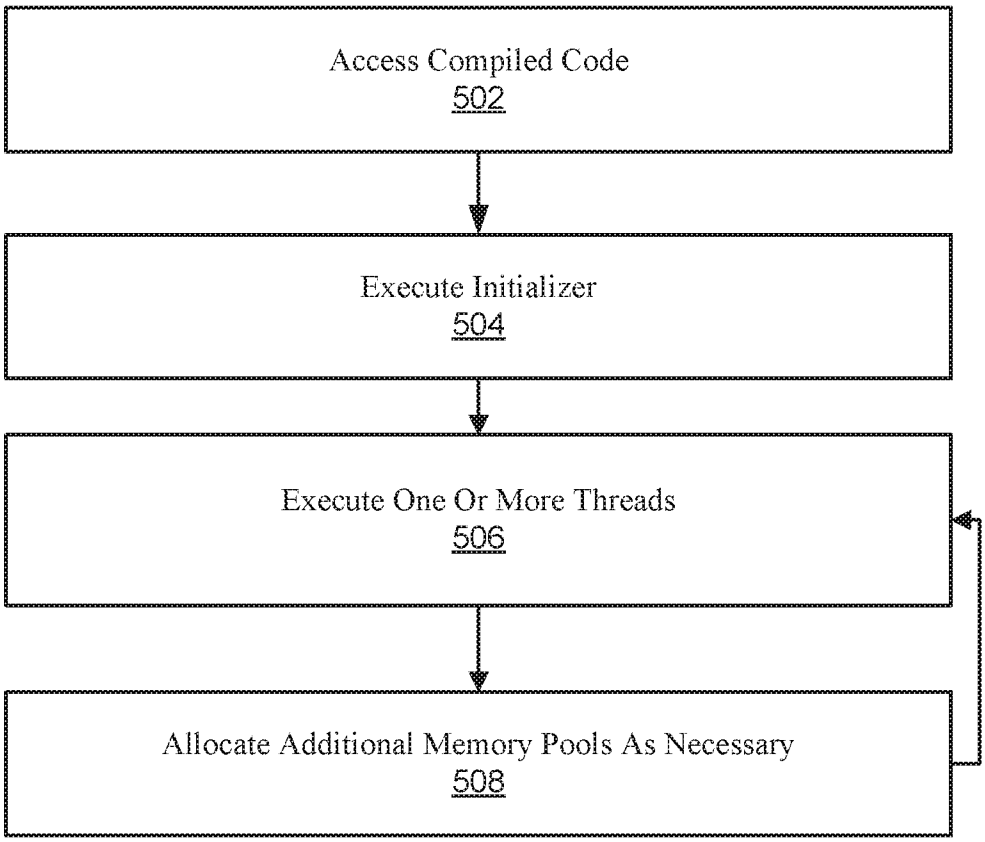

Access Compiled Code
502
Execute Initializer
504
Execute One Or More Threads
506
Allocate Additional Memory Pools As Necessary
508

FIGURE 5

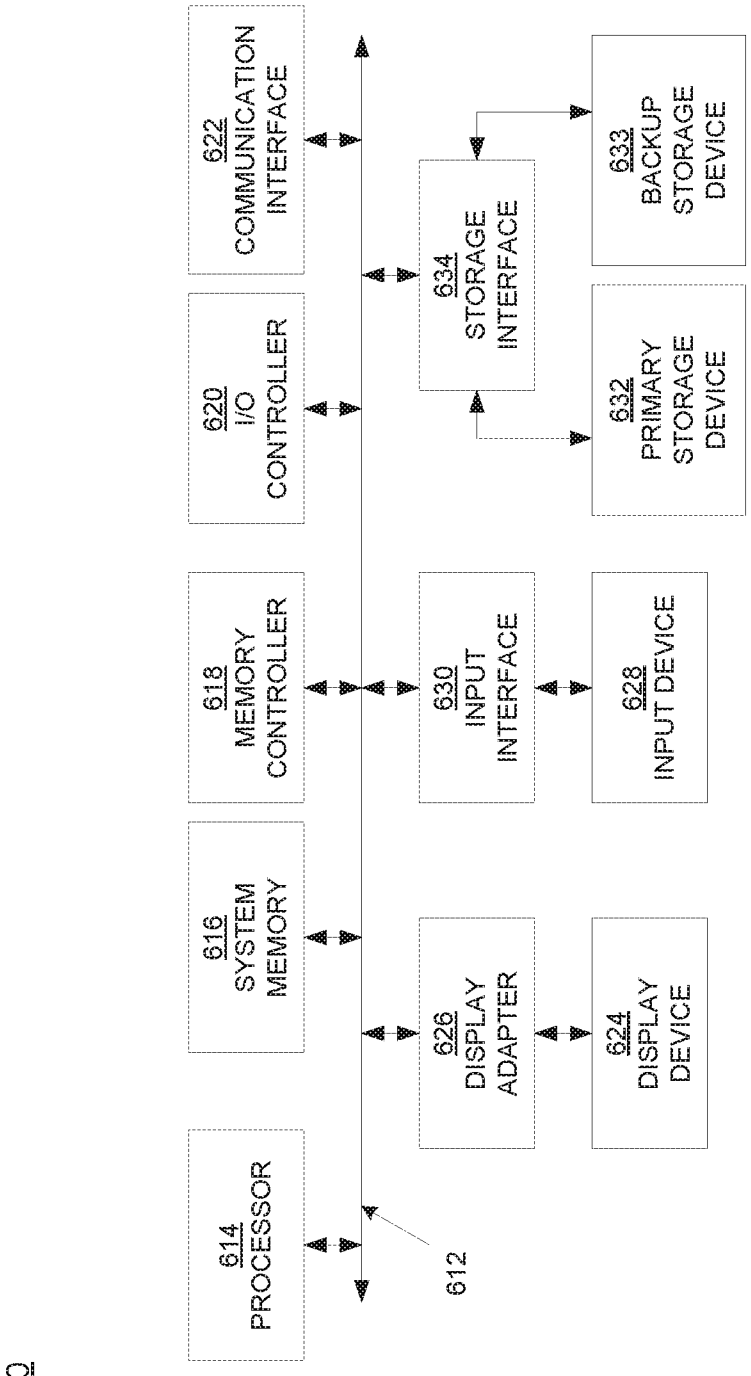

600
614 PROCESSOR
616 SYSTEM MEMORY
618 MEMORY CONTROLLER
620 I/O CONTROLLER
622 COMMUNICATION INTERFACE
612
626 DISPLAY ADAPTER
624 DISPLAY DEVICE
630 INPUT INTERFACE
628 INPUT DEVICE
634 STORAGE INTERFACE
632 PRIMARY STORAGE DEVICE
633 BACKUP STORAGE DEVICE

FIGURE 6

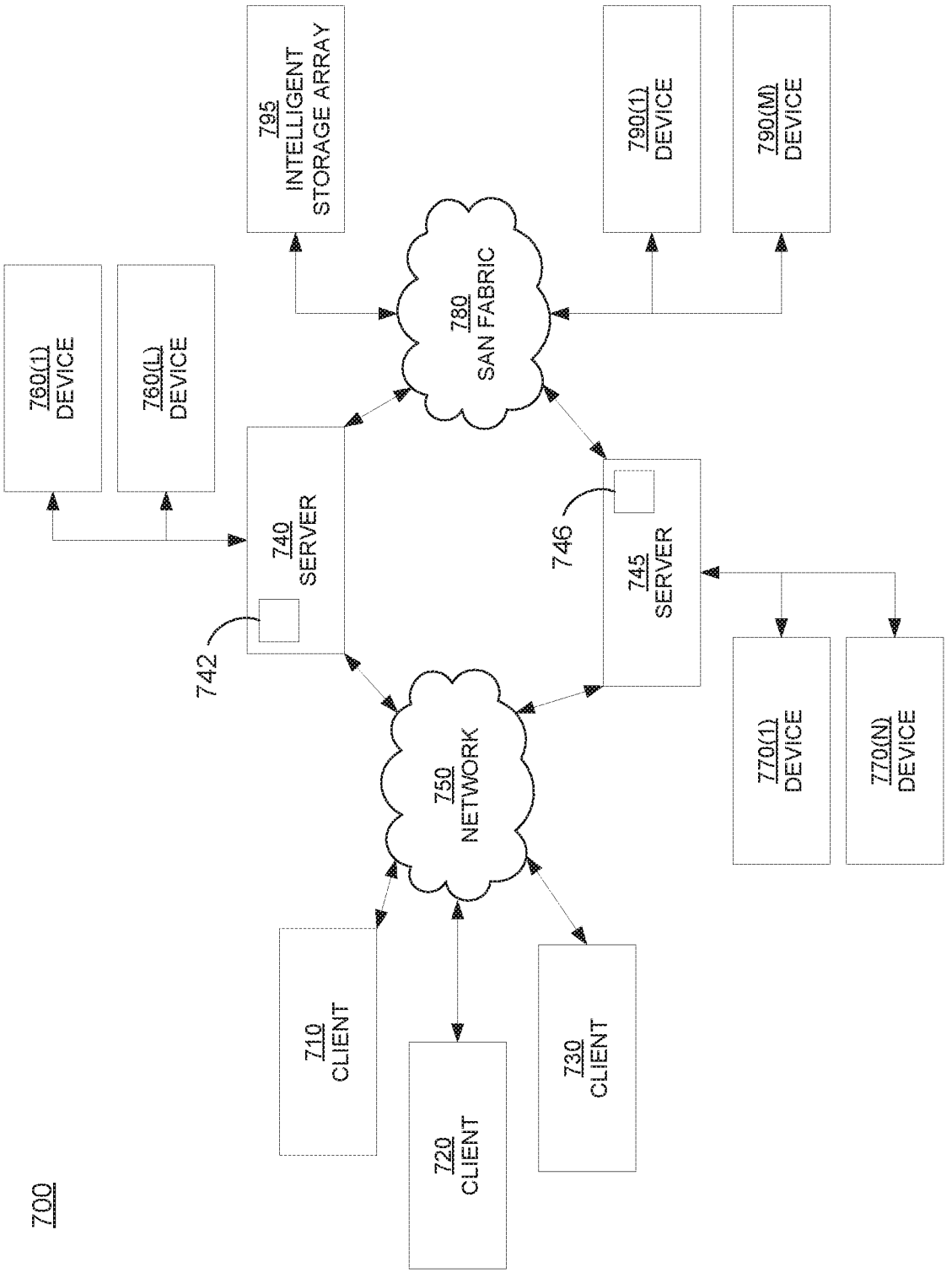

700
710 CLIENT
720 CLIENT
730 CLIENT
750 NETWORK
740 SERVER
742
745 SERVER
746
760(1) DEVICE
760(L) DEVICE
770(1) DEVICE
770(N) DEVICE
780 SAN FABRIC
790(1) DEVICE
790(M) DEVICE
795 INTELLIGENT STORAGE ARRAY

FIGURE 7

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US16/40970

A. CLASSIFICATION OF SUBJECT MATTER
IPC(8) - G06F 9/50 (2016.01)
CPC - G06F 9/5088
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC(8) Classifications: G06F 9/50, 9/46, 9/48 (2016.01)
CPC Classifications: G06F 9/5088, 9/46, 9/48, 9/50, 9/5061

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
PatSeer (US, EP, WO, JP, DE, GB, CN, FR, KR, ES, AU, IN, CA, Other Countries (INPADOC), RU, AT, CH, TH, BR, PH); IEEE/IEEEXplore; Google/Google Scholar; IP.com; Keywords: multiple, thread, allocation, memory, non-uniform, NUMA, attribute, proximity, near, distance, pool, modules, scope, objects, modify, access, aware, compiler, processor

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2012/0102501 A1 (WADDINGTON, D et al.) 26 April 2012; paragraphs [0023], [0027], [0030]-[0033], [0039], [0041], [0045], [0070]	1-29
A	US 8635626 B2 (JULA, A et al.) 21 January 2014; entire document	1-29
A	US 2012/0317371 A1 (GOUNARES, A) 13 December 2012; entire document	1-29

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
24 August 2016 (24.08.2016)	14 SEP 2016
Name and mailing address of the ISA/ Mail Stop PCT, Attn: ISA/US, Commissioner for Patents P.O. Box 1450, Alexandria, Virginia 22313-1450 Facsimile No. 571-273-8300	Authorized officer Shane Thomas PCT Helpdesk: 571-272-4300 PCT OSP: 571-272-7774

Form PCT/ISA/210 (second sheet) (January 2015)