US008676877B2

(12) **United States Patent**
Tannenbaum et al.

(10) **Patent No.: US 8,676,877 B2**
(45) **Date of Patent: Mar. 18, 2014**

(54) **CLUSTER COMPUTING USING SPECIAL PURPOSE MICROPROCESSORS**

(75) Inventors: **Zvi Tannenbaum**, Newport Beach, CA (US); **Dean E. Dauger**, Huntington Beach, CA (US)

(73) Assignee: **Advanced Cluster Systems, Inc.**, Aliso Viejo, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 174 days.

(21) Appl. No.: **13/423,063**

(22) Filed: **Mar. 16, 2012**

(65) **Prior Publication Data**

US 2013/0097406 A1 Apr. 18, 2013

Related U.S. Application Data

(63) Continuation of application No. 12/040,519, filed on Feb. 29, 2008, now Pat. No. 8,140,612, which is a continuation-in-part of application No. 11/744,461, filed on May 4, 2007, now Pat. No. 8,082,289.

(60) Provisional application No. 60/813,738, filed on Jun. 13, 2006, provisional application No. 60/850,908, filed on Oct. 11, 2006.

(51) **Int. Cl.**
G06F 9/44 (2006.01)
G06F 15/16 (2006.01)

(52) **U.S. Cl.**
USPC **709/201**; 717/177; 719/320

(58) **Field of Classification Search**
USPC 709/201; 719/320; 717/177
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,423,046	A	6/1995	Nunnelley et al.
5,881,315	A	3/1999	Cohen
6,108,699	A	8/2000	Moiin
6,202,080	B1	3/2001	Lu et al.
6,546,403	B1	4/2003	Carlson, Jr. et al.
6,578,068	B1	6/2003	Bowman-Amuah
6,751,698	B1	6/2004	Deneroff et al.
6,782,537	B1	8/2004	Blackmore et al.
6,968,335	B2	11/2005	Bayliss et al.
7,093,004	B2	8/2006	Bernardin et al.

(Continued)

FOREIGN PATENT DOCUMENTS

JP	08-87473 A	2/1996
JP	2002117010	4/2002

OTHER PUBLICATIONS

"gridMathematica 1.1: Grid Computing Gets a Speed Boost from Mathematica 5", The Mathematica Journal, vol. 9, No. 2, 2004.
"Wolfram gridMathematica", Wolfram Research, Inc., 2007.

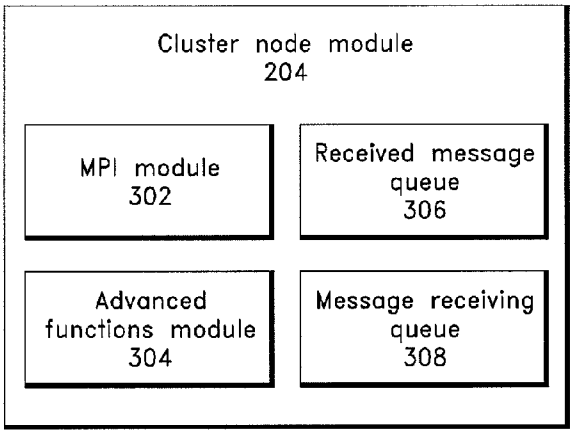
(Continued)

Primary Examiner — Larry Donaghue
(74) *Attorney, Agent, or Firm* — Knobbe, Martens, Olson & Bear, LLP

(57) **ABSTRACT**

In some embodiments, a computer cluster system comprises a plurality of nodes and a software package comprising a user interface and a kernel for interpreting program code instructions. In certain embodiments, a cluster node module is configured to communicate with the kernel and other cluster node modules. The cluster node module can accept instructions from the user interface and can interpret at least some of the instructions such that several cluster node modules in communication with one another and with a kernel can act as a computer cluster.

14 Claims, 5 Drawing Sheets



(56) **References Cited**

U.S. PATENT DOCUMENTS

7,136,924	B2	11/2006	Dauger
7,174,381	B2	2/2007	Gulko et al.
7,249,357	B2	7/2007	Landman et al.
7,334,232	B2	2/2008	Jacobs et al.
7,437,460	B2	10/2008	Chidambaran et al.
7,472,193	B2	12/2008	Dauger
7,554,949	B2	6/2009	Chen
7,937,455	B2	5/2011	Saha et al.
8,082,289	B2	12/2011	Tannenbaum et al.
8,140,612	B2	3/2012	Tannenbaum et al.
2002/0049859	A1	4/2002	Bruckert et al.
2003/0005266	A1	1/2003	Akkary et al.
2003/0051062	A1	3/2003	Circenis
2003/0135621	A1	7/2003	Romagnoli
2003/0195931	A1	10/2003	Dauger
2003/0195938	A1	10/2003	Howard et al.
2004/0110209	A1	6/2004	Yokota et al.
2004/0157203	A1	8/2004	Dunk
2005/0021751	A1	1/2005	Block et al.
2005/0038852	A1	2/2005	Howard
2005/0060237	A1	3/2005	Barsness et al.
2005/0076105	A1	4/2005	Keohane et al.
2005/0108394	A1	5/2005	Braun et al.
2005/0180095	A1	8/2005	Ellis
2006/0053216	A1	3/2006	Deokar et al.
2006/0106931	A1	5/2006	Richoux
2007/0073705	A1	3/2007	Gray
2008/0250347	A1	10/2008	Gray et al.
2008/0281997	A1	11/2008	Archer et al.
2009/0222543	A1	9/2009	Tannenbaum et al.

OTHER PUBLICATIONS

Carns et al., "An Evaluation of Message Passing Implementations on Beowulf Workstations", Aerospace Conference, Mar. 6-13, 1999, IEEE 1999, vol. 5, pp. 41-54.

Dauger et al., "Plug-and-Play Cluster Computing using Mac OS X", IEEE International Conference, Dec. 1-4, 2003, Paper appears in Cluster Computing Jan. 8, 2004, pp. 430-435.

Dauger et al., Plug-and Play Cluster Computing: High-Performance Computing for the Mainstream, IEEE Computing in Science and Engineering, Mar./Apr. 2005, pp. 27-33.

Hamscher et al., "Evaluation of Job-Scheduling Strategies for grid Computing", LNCS: Lecture Notes in Computer Science, 2000, pp. 191-202.

International Search Report dated Sep. 11, 2008, International Application No. PCT/US07/70585.

Jahanzeb et al., "Libra: a computational economy-based job scheduling system for clusters", Software Practice and Experience, Feb. 24, 2004, vol. 34, pp. 573-590.

Jain et al., "Data Clustering: A Review", ACM Computing Surveys, Sep. 1999, vol. 31, No. 3.

Maeder, R., "Mathematica Parallel Computing Toolkit: Unleash the Power of Parallel Computing", Wolfram Research, Jan. 2005.

Tepeneu and Ida, "MathGridLink—A bridge between Mathematica and the Grid", Nippon Sofutowea Kagakkai Taikai Ronbunshu, 2003, vol. 20, pp. 74-77.

Wolfram, Stephen: The Mathematica Book 5th Edition; Wolfram Research, Inc. 2003.

Kepner, Jeremy et al., "Parallel Matlab: The Next Generation", Aug. 20, 2004, MIT Lincoln Laboratory, Lexington, MA.

Kim, Hahn et al., "Introduction to Parallel Programming and pMatlab v2.0", 2011, MIT Lincoln Laboratory, Lexington, MA.

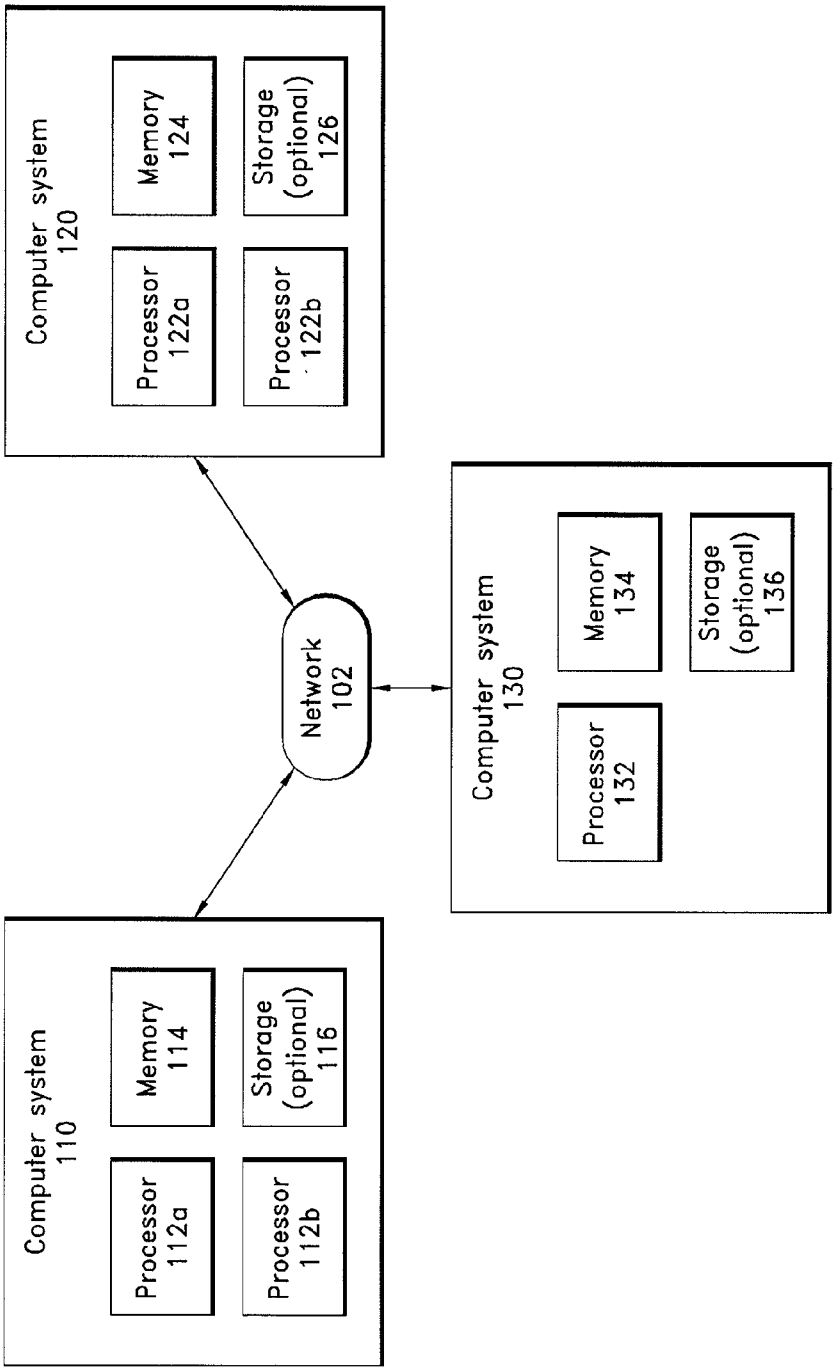

Computer system 120
Processor 122a
Memory 124
Processor 122b
Storage (optional) 126
Network 102
Computer system 110
Processor 112a
Memory 114
Processor 112b
Storage (optional) 116
Computer system 130
Processor 132
Memory 134
Storage (optional) 136

FIG. 1

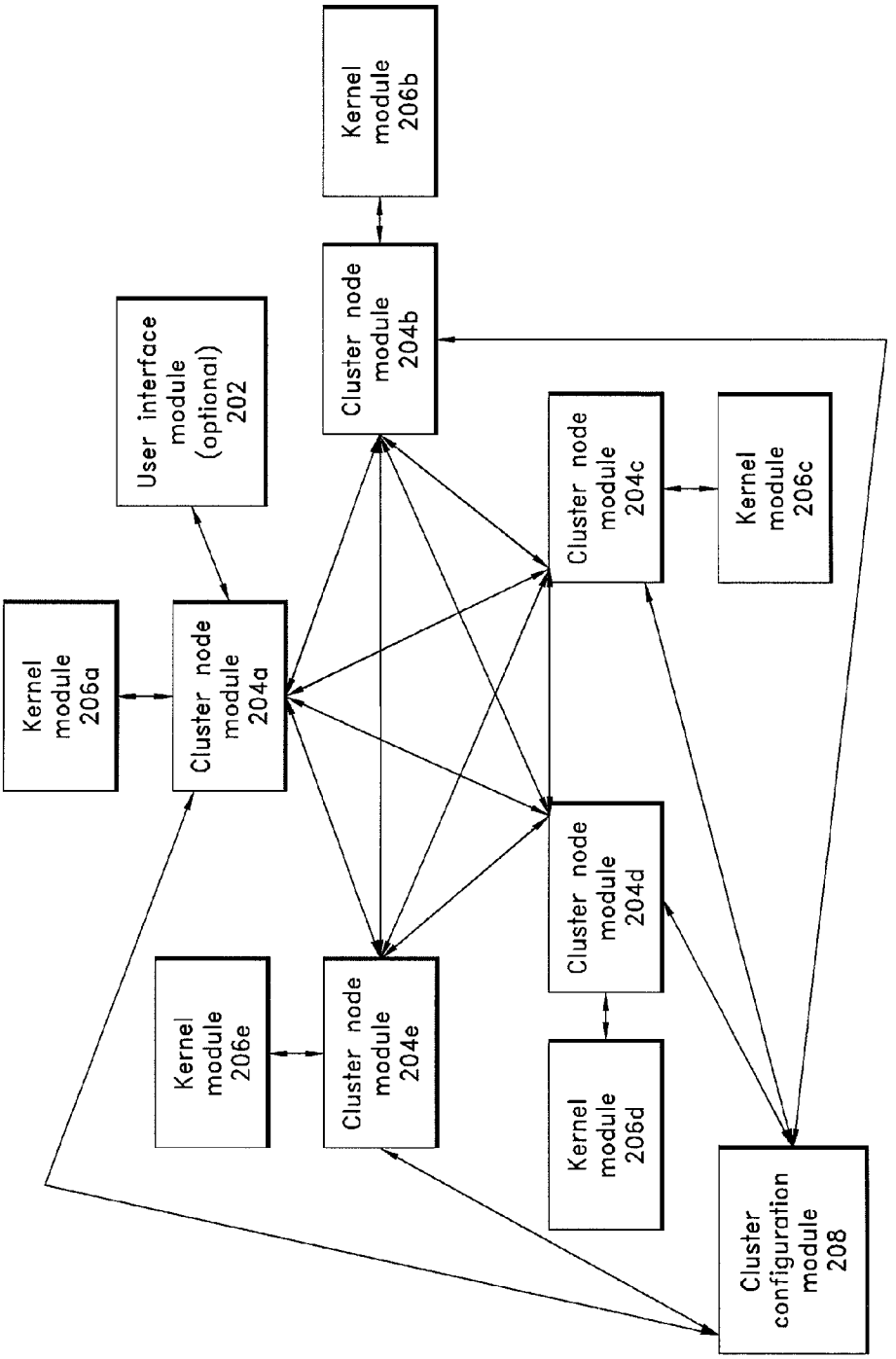

Kernel module 206a
Cluster node module 204a
User interface module (optional) 202
Kernel module 206b
Cluster node module 204b
Cluster node module 204c
Kernel module 206c
Cluster node module 204d
Kernel module 206d
Kernel module 206e
Cluster node module 204e
Cluster configuration module 208

FIG. 2

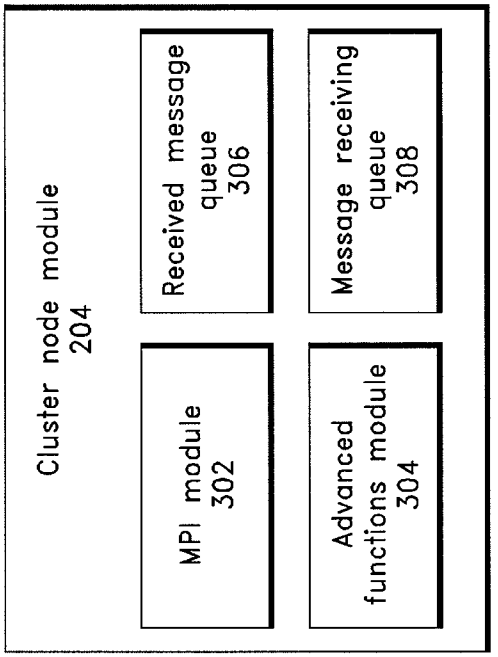

Cluster node module
204
MPI module
302
Received message
queue
306
Advanced
functions module
304
Message receiving
queue
308

FIG. 3

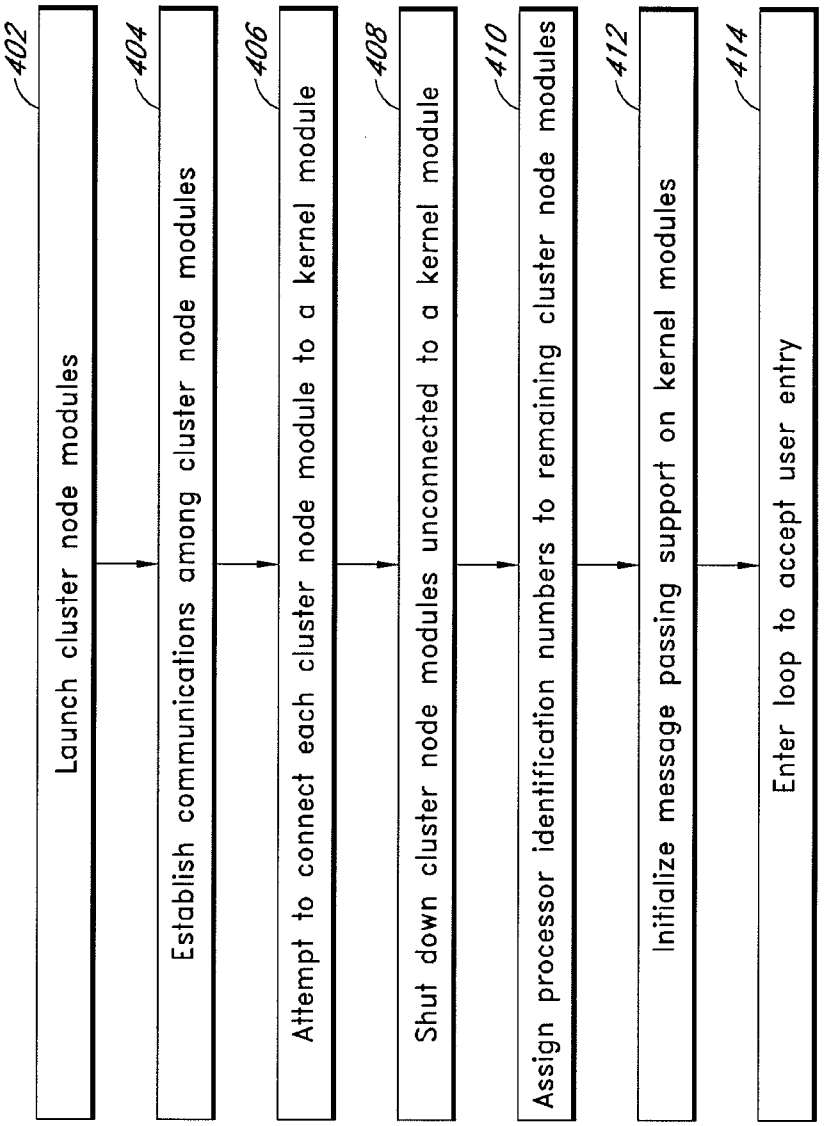

402
Launch cluster node modules
404
Establish communications among cluster node modules
406
Attempt to connect each cluster node module to a kernel module
408
Shut down cluster node modules unconnected to a kernel module
410
Assign processor identification numbers to remaining cluster node modules
412
Initialize message passing support on kernel modules
414
Enter loop to accept user entry

FIG. 4

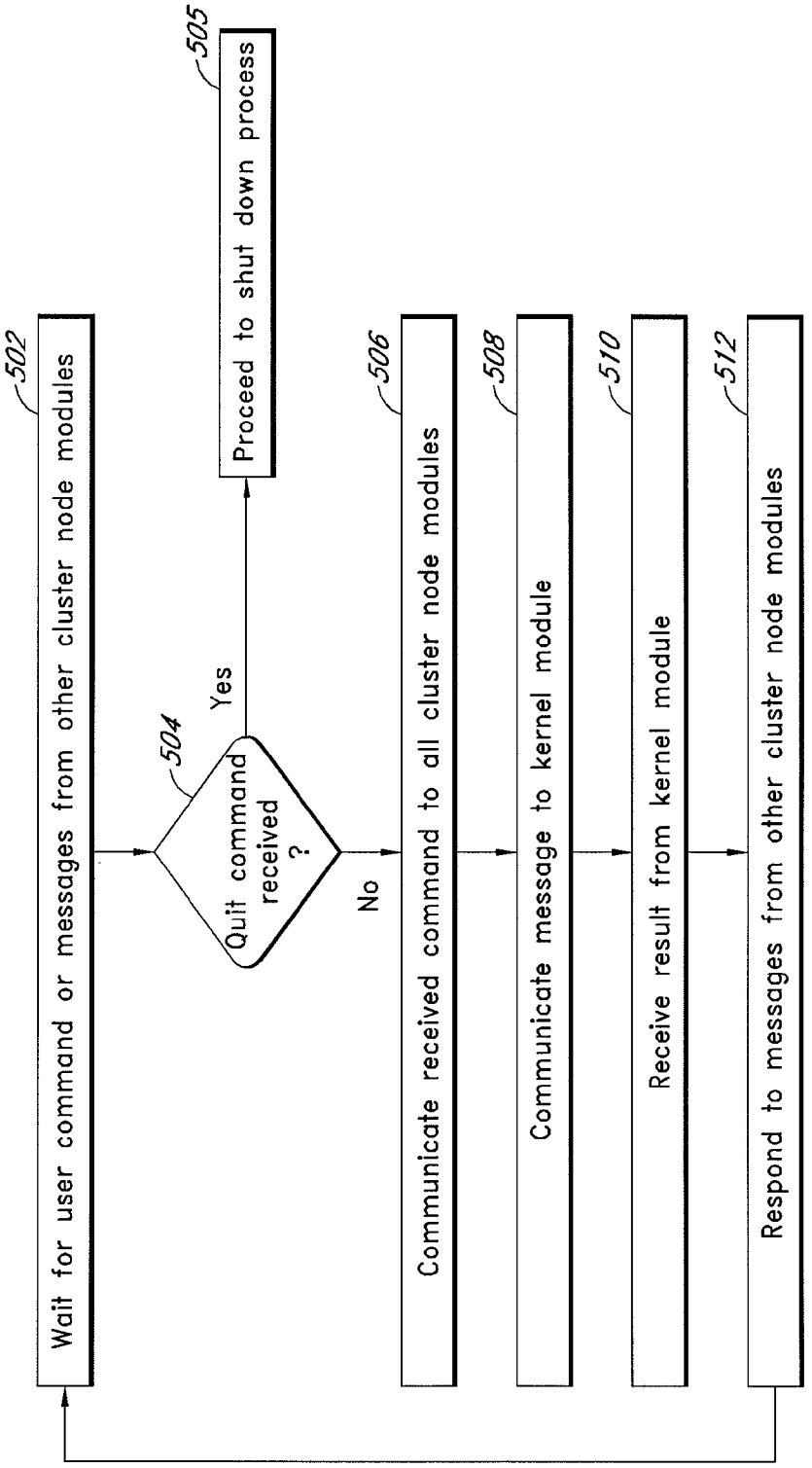

502
Wait for user command or messages from other cluster node modules
504
Quit command received ?
Yes
No
505
Proceed to shut down process
506
Communicate received command to all cluster node modules
508
Communicate message to kernel module
510
Receive result from kernel module
512
Respond to messages from other cluster node modules

FIG. 5

CLUSTER COMPUTING USING SPECIAL PURPOSE MICROPROCESSORS

RELATED APPLICATIONS

This application is a continuation of U.S. patent application Ser. No. 12/040,519, filed Feb. 29, 2008 now U.S. Pat. No. 8,140,612, which is a continuation-in-part of U.S. patent application Ser. No. 11/744,461, filed May 4, 2007, now U.S. Pat. No. 8,082,289, which claims the benefit under 35 U.S.C. §119(e) of U.S. Provisional Patent Application No. 60/813,738, filed Jun. 13, 2006, and U.S. Provisional Patent Application No. 60/850,908, filed Oct. 11, 2006. The entire contents of each of the above-referenced applications are incorporated by reference herein and made a part of this specification.

BACKGROUND

1. Field

The present disclosure relates to the field of cluster computing generally and to systems and methods for adding cluster computing functionality to a computer program, in particular.

2. Description of Related Art

Computer clusters include a group of two or more computers, microprocessors, and/or processor cores ("nodes") that intercommunicate so that the nodes can accomplish a task as though they were a single computer. Many computer application programs are not currently designed to benefit from advantages that computer clusters can offer, even though they may be running on a group of nodes that could act as a cluster. Some computer programs can run on only a single node because, for example, they are coded to perform tasks serially or because they are designed to recognize or send instructions to only a single node.

Some application programs include an interpreter that executes instructions provided to the program by a user, a script, or another source. Such an interpreter is sometimes called a "kernel" because, for example, the interpreter can manage at least some hardware resources of a computer system and/or can manage communications between those resources and software (for example, the provided instructions, which can include a high-level programming language). Some software programs include a kernel that is designed to communicate with a single node. An example of a software package that includes a kernel that is designed to communicate with a single node is Mathematica® from Wolfram Research, Inc. ("Mathematica"). Mathematics software packages from other vendors and other types of software can also include such a kernel.

A product known as gridMathematica, also from Wolfram Research, Inc., gives Mathematica the capability to perform a form of grid computing known as "distributed computing." Grid computers include a plurality of nodes that generally do not communicate with one another as peers. Distributed computing can be optimized for workloads that consist of many independent jobs or packets of work, which do not need to share data between the jobs during the computational process. Grid computers include at least one node known as a master node that manages a plurality of slave nodes or computational nodes. In gridMathematica, each of a plurality of kernels runs on a single node. One kernel is designated the master kernel, which handles all input, output, and scheduling of the other kernels (the computational kernels or slave kernels). Computational kernels receive commands and data only from the node running the master kernel. Each computational kernel performs its work independently of the other computational kernels and intermediate results of one job do not affect other jobs in progress on other nodes.

SUMMARY

Embodiments described herein have several features, no single one of which is solely responsible for their desirable attributes. Without limiting the scope of the invention as expressed by the claims, some of the advantageous features will now be discussed briefly.

Some embodiments described herein provide techniques for conveniently adding cluster computing functionality to a computer application. In one embodiment, a user of a software package may be able to achieve higher performance and/or higher availability from the software package by enabling the software to benefit from a plurality of nodes in a cluster. One embodiment allows a user to create applications, using a high-level language such as Mathematica, that are able to run on a computer cluster having supercomputer-like performance. One embodiment provides access to such high-performance computing through a Mathematica Front End, a command line interface, one or more high-level commands, or a programming language such as C or FORTRAN.

One embodiment adapts a software module designed to run on a single node, such as, for example, the Mathematica kernel, to support cluster computing, even when the software module is not designed to provide such support. One embodiment provides parallelization for an application program, even if no access to the program's source code is available. One embodiment adds and supports Message Passing Interface ("MPI") calls directly from within a user interface, such as, for example, the Mathematica programming environment. In one embodiment, MPI calls are added to or made available from an interactive programming environment, such as the Mathematica Front End.

One embodiment provides a computer cluster including a first processor, a second processor, and a third processor. The cluster includes at least one computer-readable medium in communication at least one of the first processor, the second processor, or the third processor. A first kernel resides in the at least one computer-readable medium and is configured to translate commands into code for execution on the first processor. A first cluster node module resides in the at least one computer-readable medium. The first cluster node module is configured to send commands to the first kernel and receives commands from a user interface. A second kernel resides in the at least one computer-readable medium. The second kernel is configured to translate commands into code for execution on the second processor. A second cluster node module resides in the at least one computer-readable medium. The second cluster node module is configured to send commands to the second kernel and communicates with the first cluster node module. A third kernel resides in the at least one computer-readable medium. The third kernel is configured to translate commands into code for execution on the third processor. A third cluster node module resides in the at least one computer-readable medium. The third cluster node module is configured to send commands to the third kernel and configured to communicate with the first cluster node module and the second cluster node module. The first cluster node module comprises a data structure in which messages originating from the second and third cluster node modules are stored.

Another embodiment provides a computer cluster that includes a plurality of nodes and a software package including a user interface and a single-node kernel for interpreting program code instructions. A cluster node module is config-

ured to communicate with the single-node kernel and other cluster node modules. The cluster node module accepts instructions from the user interface and interprets at least some of the instructions such that several cluster node modules in communication with one another act as a cluster. The cluster node module appears as a single-node kernel to the user interface. In one embodiment, the single-node kernel includes a Mathematical kernel. In some embodiments, the user interface can include at least one of a Mathematica front end or a command line. In some embodiments, the cluster node module includes a toolkit including library calls that implement at least a portion of MPI calls. In some embodiments, the cluster node module includes a toolkit including high-level cluster computing commands. In one embodiment, the cluster system can include a plurality of Macintosh® computers ("Macs"), Windows®-based personal computers ("PCs"), and/or Unix/Linux-based workstations.

A further embodiment provides a computer cluster including a plurality of nodes. Each node is configured to access a computer-readable medium comprising program code for a user interface and program code for a single-node kernel module configured to interpret user instructions. The cluster includes a plurality of cluster node modules. Each cluster node module is configured to communicate with a single-node kernel and with one or more other cluster node modules, to accept instructions from the user interface, and to interpret at least some of the user instructions such that the plurality of cluster node modules communicate with one another in order to act as a cluster. A communications network connects the nodes. One of the plurality of cluster node modules returns a result to the user interface.

Another embodiment provides a method of evaluating a command on a computer cluster. A command from at least one of a user interface or a script is communicated to one or more cluster node modules within the computer cluster. Each of the one or more cluster node modules communicates a message based on the command to a respective kernel module associated with the cluster node module. Each of the one or more cluster node modules receives a result from the respective kernel module associated with the cluster node module. At least one of the one or more cluster node modules responds to messages from other cluster node modules.

Another embodiment provides a computing system for executing Mathematica code on multiple nodes. The computing system includes a first node module in communication with a first Mathematica kernel executing on a first node, a second node module in communication with a second Mathematica kernel executing on a second node, and a third node module in communication with a third Mathematica kernel executing on a third node. The first node module, the second node module, and the third node module are configured to communicate with one another using a peer-to-peer architecture. In some embodiments, each of the first node module, the second node module, and third node module includes a data structure for maintaining messages originating from other node modules and a data structure for maintaining data specifying a location to which an message is expected to be received and an identifier for a node from which the message is expected to be sent.

BRIEF DESCRIPTION OF THE DRAWINGS

A general architecture that implements the various features are described with reference to the drawings. The drawings and the associated descriptions are provided to illustrate embodiments and not to limit the scope of the disclosure. Throughout the drawings, reference numbers are re-used to indicate correspondence between referenced elements.

FIG. **1** is a block diagram of one embodiment of a computer cluster.

FIG. **2** is a block diagram showing relationships between software modules running on one embodiment of a computer cluster.

FIG. **3** is a block diagram of one embodiment of a cluster node module.

FIG. **4** is a flow chart showing one embodiment of a cluster initialization process.

FIG. **5** is a flow chart showing one embodiment of the operation of a cluster node module.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

For purposes of illustration, some embodiments are described herein in the context of cluster computing with Mathematica software. The present disclosure is not limited to a single software program; the systems and methods can be used with other application software such as, for example, Maple®, MATLAB®, MathCAD®, Apple Shake®, Apple® Compressor, IDL®, other applications employing an interpreter or a kernel, Microsoft Excel®, Adobe After Effects®, Adobe Premiere®, Adobe Photoshop®, Apple Final Cut Pro®, and Apple iMovie®. Some figures and/or descriptions, however, relate to embodiments of computer clusters running Mathematica. The system can include a variety of uses, including but not limited to students, educators, scientists, engineers, mathematicians, researchers, and technicians. It is also recognized that in other embodiments, the systems and methods can be implemented as a single module and/or implemented in conjunction with a variety of other modules. Moreover, the specific implementations described herein are set forth in order to illustrate, and not to limit, the disclosure.

I. Overview

The cluster computing system described herein generally includes one or more computer systems connected to one another via a communications network or networks. The communications network can include one or more of a local area network ("LAN"), a wide area network ("WAN"), an intranet, the Internet, etc. In one embodiment, a computer system comprises one or more processors such as, for example, a microprocessor that can include one or more processing cores ("nodes"). The term "node" refers to a processing unit or subunit that is capable of single-threaded execution of code. The processors can be connected to one or more memory devices such as, for example, random access memory ("RAM"), and/or one or more optional storage devices such as, for example, a hard disk. Communications among the processors and such other devices may occur, for example, via one or more local buses of a computer system or via a LAN, a WAN, a storage area network ("SAN"), and/or any other communications network capable of carrying signals among computer system components. In one embodiment, one or more software modules such as kernels, run on nodes within the interconnected computer systems. In one embodiment, the kernels are designed to run on only a single node. In one embodiment, cluster node modules communicate with the kernels and with each other in order to implement cluster computing functionality.

FIG. **1** is a block diagram of one embodiment of a computer cluster **100** wherein computer systems **110**, **120**, **130** communicate with one another via a communications network

102. Network 102 includes one or more of a LAN, a WAN, a wireless network, an intranet, or the Internet. In one embodiment of the computer cluster, computer system 110 includes processors 112*a*, 112*b*, memory 114, and optional storage 116. Other computer systems 120, 130 can include similar devices, which generally communicate with one another within a computer system over a local communications architecture such as a local bus (not shown). A computer system can include one or more processors, and each processor can contain one or more processor cores that are capable of single-threaded execution. Processor cores are generally independent microprocessors, but more than one can be included in a single chip package. Software code designed for single-threaded execution can generally run on one processor core at a time. For example, single-threaded software code typically does not benefit from multiple processor cores in a computer system.

FIG. 2 is a block diagram showing relationships among software modules running on one embodiment of a computer cluster 100. In the embodiment shown in FIG. 2, the kernel modules 206*a-e* are designed for single-threaded execution. For example, if each of the processors 112*a*, 112*b*, 122*a*, 122*b*, 132 shown in FIG. 1 includes only one processor core, two kernel modules (for example, kernel modules 206*a*, 206*b*) loaded into the memory 114 of computer system 110 could exploit at least some of the processing bandwidth of the two processors 112*a*, 112*b*. Similarly, two kernel modules 206*c*, 206*d* loaded into the memory 124 of computer system 120 could exploit at least some of the processing bandwidth of the two processors 122*a*, 122*b*. Likewise, the bandwidth of processor 132 of computer system 130 could be utilized by a single instance of a cluster node module 204*e* loaded into the computer system's memory 134.

In the embodiment shown in FIG. 2, each of the kernel modules 206*a-e* is in communication with a single cluster node module 204*a-e*, respectively. For example, the kernel module 206*a* is in communication with the cluster node module 204*a*, the kernel module 206*b* is in communication with the cluster node module 206*b*, and so forth. In one embodiment, one instance of a cluster node module 204*a-e* is loaded into a computer system's memory 114, 124, 134 for every instance of a kernel module 206*a-e* running on the system. As shown in FIG. 2, each of the cluster node modules 204*a-e* is in communication with each of the other cluster node modules 204*a-e*. For example, one cluster node module 204*a* is in communication with all of the other cluster node modules 204*b-e*. A cluster node module 204*a* may communicate with another cluster node module 204*b* via a local bus (not shown) when, for example, both cluster node modules 204*a-b* execute on processors 112*a*, 112*b* within the same computer system 110. A cluster node module 204*a* may also communicate with another cluster node module 204*c* over a communications network 102 when, for example, the cluster node modules 204*a, c* execute on processors 112*a*, 122*a* within different computer systems 110, 120.

As shown in FIG. 2, an optional user interface module 202 such as, for example, a Mathematica front end and/or a command line interface, can connect to a cluster node module 204*a*. The user interface module can run on the same computer system 110 and/or the same microprocessor 112*a* on which the cluster node module 204*a* runs. The cluster node modules 204*a-e* provide MPI calls and/or advanced cluster functions that implement cluster computing capability for the single-threaded kernel modules. The cluster node modules 204*a-e* are configured to look and behave like a kernel module 206*a* from the perspective of the user interface module 202. Similarly, the cluster node modules 204*a-e* are configured to look and behave like a user interface module 202 from the perspective of a kernel module 206*a*. The first cluster node module 204*a* is in communication with one or more other cluster node modules 204*b*, 204*c*, and so forth, each of which provides a set of MPI calls and/or advanced cluster commands. In one embodiment, MPI may be used to send messages between nodes in a computer cluster.

Communications can occur between any two or more cluster node modules (for example, between a cluster node module 204*a* and another cluster node module 204*c*) and not just between "adjacent" kernels. Each of the cluster node modules 204*a-e* is in communication with respective kernel modules 206*a-e*. Thus, the cluster node module 204*a* communicates with the kernel module 206*a*. MPI calls and advanced cluster commands are used to parallelize program code received from an optional user interface module 208 and distribute tasks among the kernel modules 206*a-e*. The cluster node modules 204*a-e* provide communications among kernel modules 206*a-e* while the tasks are executing. Results of evaluations performed by kernel modules 206*a-e* are communicated back to the first cluster node module 204*a* via the cluster node modules 204*a-e*, which communicates them to the user interface module 208.

Intercommunication among kernel modules 206*a-e* during thread execution, which is made possible by cluster node modules 204*a-e*, provides advantages for addressing various types of mathematic and scientific problems, for example. Intercommunication provided by cluster computing permits exchange of information between nodes during the course of a parallel computation. Embodiments of the present disclosure provide such intercommunication for software programs such as Mathematica, while grid computing solutions can implement communication between only one master node and many slave nodes. Grid computing does not provide for communication between slave nodes during thread execution.

For purposes of providing an overview of some embodiments, certain aspects, advantages, benefits, and novel features of the invention are described herein. It is to be understood that not necessarily all such advantages or benefits can be achieved in accordance with any particular embodiment of the invention. Thus, for example, those skilled in the art will recognize that the invention can be embodied or carried out in a manner that achieves one advantage or group of advantages as taught herein without necessarily achieving other advantages or benefits as can be taught or suggested herein.

II. Computer Cluster 100

As shown in FIG. 1, one embodiment of a cluster system 100 includes computer systems 110, 120, 130 in communication with one another via a communications network 102. A first computer system 110 can include one or more processors 112*a-b*, a memory device 114, and an optional storage device 116. Similarly, a second computer system 120 can include one or more processors 122*a-b*, a memory device 124, and an optional storage device 126. Likewise, a third computer system 130 can include one or more processors 132, a memory device 134, and an optional storage device 136. Each of the computer systems 110, 120, 130 includes a network interface (not shown) for connecting to a communications network 102, which can include one or more of a LAN, a WAN, an intranet, a wireless network, and/or the Internet.

A. Computer System 110

In one embodiment, a first computer system 110 communicates with other computer systems 120, 130 via a network 102 as part of a computer cluster 100. In one embodiment, the

computer system **110** is a personal computer, a workstation, a server, or a blade including one or more processors **112***a-b*, a memory device **114**, an optional storage device **116**, as well as a network interface module (not shown) for communications with the network **102**.

1. Processors **112***a-b*

In one embodiment, the computer system **110** includes one or more processors **112***a-b*. The processors **112***a-b* can be one or more general purpose single-core or multi-core microprocessors such as, for example, a Pentium® processor, a Pentium® II processor, a Pentium® Pro processor, a Pentium® III processor, Pentium® 4 processor, a Core Duo® processor, a Core 2 Duo® processor, a Xeon® processor, an Itanium® processor, a Pentium® M processor, an x86 processor, an Athlon® processor, an 8051 processor, a MIPS® processor, a PowerPC® processor, an ALPHA® processor, etc. In addition, one or more of the processors **112***a-b* can be a special purpose microprocessor such as a digital signal processor. The total number of processing cores (for example, processing units capable of single-threaded execution) within all processors **112***a-b* in the computer system **110** corresponds to the number of nodes available in the computer system **110**. For example, if the processors **112***a-b* were each Core 2 Duo® processors having two processing cores, computer system **110** would have four nodes in all. Each node can run one or more instances of a program module, such as a single-threaded kernel module.

2. Network Interface Module

The computer system **110** can also include a network interface module (not shown) that facilitates communication between the computer system **110** and other computer systems **120**, **130** via the communications network **102**.

The network interface module can use a variety of network protocols. In one embodiment, the network interface module includes TCP/IP. However, it is to be appreciated that other types of network communication protocols such as, for example, Point-to-Point Protocol ("PPP"), Server Message Block ("SMB"), Serial Line Internet Protocol ("SLIP"), tunneling PPP, AppleTalk, etc., may also be used.

3. Memory **114** and Storage **116**

The computer system **110** can include memory **114**. Memory **114** can include, for example, processor cache memory (such as processor core-specific or cache memory shared by multiple processor cores), dynamic random-access memory ("DRAM"), static random-access memory ("SRAM"), or any other type of memory device capable of storing computer data, instructions, or program code. The computer system **110** can also include optional storage **116**. Storage **116** can include, for example, one or more hard disk drives, floppy disks, flash memory, magnetic storage media, CD-ROMs, DVDs, optical storage media, or any other type of storage device capable of storing computer data, instructions, and program code.

4. Computer System **110** Information

The computer system **110** may be used in connection with various operating systems such as: Microsoft® Windows® 3.X, Windows 95®, Windows 98®, Windows NT®, Windows 2000®, Windows XP®, Windows CE®, Palm Pilot OS, OS/2, Apple® MacOS®, MacOS X®, MacOS X Server®, Disk Operating System (DOS), UNIX, Linux®, VxWorks, or IBM® OS/2®, Sun OS, Solaris OS, IRIX OS operating systems, etc.

In one embodiment, the computer system **110** is a personal computer, a laptop computer, a Blackberry® device, a portable computing device, a server, a computer workstation, a local area network of individual computers, an interactive kiosk, a personal digital assistant, an interactive wireless communications device, a handheld computer, an embedded computing device, or the like.

As can be appreciated by one of ordinary skill in the art, the computer system **110** may include various sub-routines, procedures, definitional statements, and macros. Each of the foregoing modules are typically separately compiled and linked into a single executable program. However, it is to be appreciated by one of ordinary skill in the art that the processes that are performed by selected ones of the modules may be arbitrarily redistributed to one of the other modules, combined together in a single module, made available in a shareable dynamic link library, or partitioned in any other logical way.

B. Computer System **120**

In one embodiment, a second computer system **120** communicates with other computer systems **110**, **130** via a network **102** as part of a computer cluster **100**. In one embodiment, the computer system **120** is a personal computer, a workstation, a server, or a blade including one or more processors **122***a-b*, a memory device **124**, an optional storage device **126**, as well as a network interface module (not shown) for communications with the network **102**.

1. Processors **112***a-b*

In one embodiment, the computer system **120** includes one or more processors **122***a-b*. The processors **122***a-b* can be one or more general purpose single-core or multi-core microprocessors such as a Pentium® processor, a Pentium® II processor, a Pentium® Pro processor, a Pentium® III processor, Pentium® 4 processor, a Core Duo® processor, a Core 2 Duo® processor, a Xeon® processor, an Itanium® processor, a Pentium® M processor, an x86 processor, an Athlon® processor, an 8051 processor, a MIPS® processor, a PowerPC® processor, an ALPHA® processor, etc. In addition, the processors **122***a-b* can be any special purpose microprocessors such as a digital signal processor. The total number of processing cores (for example, processing units capable of single-threaded execution) within all processors **122***a-b* in the computer system **120** corresponds to the number of nodes available in the computer system **120**. For example, if the processors **122***a-b* were each Core 2 Duo® processors having two processing cores, computer system **120** would have four nodes in all. Each node can run one or more instances of a program module, such as a single-threaded kernel module.

2. Network Interface Module

The computer system **120** can also include a network interface module (not shown) that facilitates communication between the computer system **120** and other computer systems **110**, **130** via the communications network **102**.

The network interface module can use a variety of network protocols. In one embodiment, the network interface module includes TCP/IP. However, it is to be appreciated that other types of network communication protocols such as, for example, Point-to-Point Protocol ("PPP"), Server Message Block ("SMB"), Serial Line Internet Protocol ("SLIP"), tunneling PPP, AppleTalk, etc., may also be used.

3. Memory **124** and Storage **126**

The computer system **120** can include memory **124**. Memory **124** can include, for example, processor cache memory (such as processor core-specific or cache memory shared by multiple processor cores), dynamic random-access memory ("DRAM"), static random-access memory ("SRAM"), or any other type of memory device capable of storing computer data, instructions, or program code. The computer system **120** can also include optional storage **126**. Storage **126** can include, for example, one or more hard disk drives, floppy disks, flash memory, magnetic storage media,

CD-ROMs, DVDs, optical storage media, or any other type of storage device capable of storing computer data, instructions, and program code.

4. Computer System **120** Information

The computer system **120** may be used in connection with various operating systems such as: Microsoft® Windows® 3.X, Windows 95®, Windows 98®, Windows NT®, Windows 2000®, Windows XP®, Windows CE®, Palm Pilot OS, OS/2, Apple® MacOS®, MacOS X®, MacOS X Server®, Disk Operating System (DOS), UNIX, Linux®, VxWorks, or IBM® OS/2®, Sun OS, Solaris OS, IRIX OS operating systems, etc.

In one embodiment, the computer system **120** is a personal computer, a laptop computer, a Blackberry® device, a portable computing device, a server, a computer workstation, a local area network of individual computers, an interactive kiosk, a personal digital assistant, an interactive wireless communications device, a handheld computer, an embedded computing device, or the like.

As can be appreciated by one of ordinary skill in the art, the computer system **120** may include various sub-routines, procedures, definitional statements, and macros. Each of the foregoing modules are typically separately compiled and linked into a single executable program. However, it is to be appreciated by one of ordinary skill in the art that the processes that are performed by selected ones of the modules may be arbitrarily redistributed to one of the other modules, combined together in a single module, made available in a shareable dynamic link library, or partitioned in any other logical way.

C. Computer System **130**

In one embodiment, a third computer system **130** communicates with other computer systems **110**, **120** via a network **102** as part of a computer cluster **100**. In one embodiment, the computer system **130** is a personal computer, a workstation, a server, or a blade including one or more processors **132**, a memory device **134**, an optional storage device **136**, as well as a network interface module (not shown) for communications with the network **102**.

1. Processors **112***a-b*

In one embodiment, the computer system **130** includes a processor **132**. The processor **132** can be a general purpose single-core or multi-core microprocessors such as a Pentium® processor, a Pentium® II processor, a Pentium® Pro processor, a Pentium® III processor, Pentium® 4 processor, a Core Duo® processor, a Core 2 Duo® processor, a Xeon® processor, an Itanium® processor, a Pentium® M processor, an x86 processor, an Athlon® processor, an 8051 processor, a MIPS® processor, a PowerPC® processor, or an ALPHA® processor. In addition, the processor **132** can be any special purpose microprocessor such as a digital signal processor. The total number of processing cores (for example, processing units capable of single-threaded execution) within processor **132** in the computer system **130** corresponds to the number of nodes available in the computer system **130**. For example, if the processor **132** was a Core 2 Duo® processor having two processing cores, the computer system **130** would have two nodes. Each node can run one or more instances of a program module, such as a single-threaded kernel module.

2. Network Interface Module

The computer system **130** can also include a network interface module (not shown) that facilitates communication between the computer system **130** and other computer systems **110**, **120** via the communications network **102**.

The network interface module can use a variety of network protocols. In one embodiment, the network interface module includes TCP/IP. However, it is to be appreciated that other types of network communication protocols such as, for example, Point-to-Point Protocol ("PPP"), Server Message Block ("SMB"), Serial Line Internet Protocol ("SLIP"), tunneling PPP, AppleTalk, etc., may also be used.

3. Memory **134** and Storage **136**

The computer system **130** can include memory **134**. Memory **134** can include, for example, processor cache memory (such as processor core-specific or cache memory shared by multiple processor cores), dynamic random-access memory ("DRAM"), static random-access memory ("SRAM"), or any other type of memory device capable of storing computer data, instructions, or program code. The computer system **130** can also include optional storage **136**. Storage **136** can include, for example, one or more hard disk drives, floppy disks, flash memory, magnetic storage media, CD-ROMs, DVDs, optical storage media, or any other type of storage device capable of storing computer data, instructions, and program code.

4. Computer System **130** Information

The computer system **130** may be used in connection with various operating systems such as: Microsoft® Windows® 3.X, Windows 95®, Windows 98®, Windows NT®, Windows 2000®, Windows XP®, Windows CE®, Palm Pilot OS, OS/2, Apple® MacOS®, MacOS X®, MacOS X Server®, Disk Operating System (DOS), UNIX, Linux®, VxWorks, or IBM® OS/2®, Sun OS, Solaris OS, IRIX OS operating systems, etc.

In one embodiment, the computer system **130** is a personal computer, a laptop computer, a Blackberry® device, a portable computing device, a server, a computer workstation, a local area network of individual computers, an interactive kiosk, a personal digital assistant, an interactive wireless communications device, a handheld computer, an embedded computing device, or the like.

As can be appreciated by one of ordinary skill in the art, the computer system **130** may include various sub-routines, procedures, definitional statements, and macros. Each of the foregoing modules are typically separately compiled and linked into a single executable program. However, it is to be appreciated by one of ordinary skill in the art that the processes that are performed by selected ones of the modules may be arbitrarily redistributed to one of the other modules, combined together in a single module, made available in a shareable dynamic link library, or partitioned in any other logical way.

E. Communications Network **102**

In one embodiment, computer systems **110**, **120**, **130** are in communication with one another via a communications network **102**.

The communications network **102** may include one or more of any type of electronically connected group of computers including, for instance, the following networks: a virtual private network, a public Internet, a private Internet, a secure Internet, a private network, a public network, a value-added network, a wired network, a wireless network, an intranet, etc. In addition, the connectivity to the network can be, for example, a modem, Ethernet (IEEE 802.3), Gigabit Ethernet, 10-Gigabit Ethernet, Token Ring (IEEE 802.5), Fiber Distributed Datalink Interface (FDDI), Frame Relay, InfiniBand, Myrinet, Asynchronous Transfer Mode (ATM), or another interface. The communications network **102** may connect to the computer systems **110**, **120**, **130**, for example, by use of a modem or by use of a network interface card that resides in each of the systems.

In addition, the same or different communications networks **102** may be used to facilitate communication between the first computer system **110** and the second computer sys-

tem **120**, between the first computer system **110** and the third computer system **130**, and between the second computer system **120** and the third computer system **130**.

III. Software Modules

As shown in FIGS. **1** and **2**, one embodiment of a cluster system **100** includes a user interface module **202** that is able to access a plurality of kernel modules **206***a-e* by communicating with a first cluster node module **204***a*. User interface module can be stored in a memory **114**, **124**, **134** while running, for example, and/or can be stored in a storage device **116**, **126**, **136**. The first cluster node module **204***a* is in communication with each of the other cluster node modules **204***b-e*. The kernel modules **206***a-e* can reside in the memory of one or more computer systems on which they run. For example, the memory **114** of the first computer system **110** can store instances of kernel modules **206***a-b*, the memory **124** of the second computer system **120** can store instances of kernel modules **206***c-d*, and the memory **134** of the third computer system **130** can store an instance of kernel module **206***e*. The kernel modules **206***a-e*, which include single-threaded program code, are each associated with one of the processors **112***a*, **112***b*, **122***a*, **122***b*, **132**. A cluster configuration module stored on one or more of the computer systems **110**, **120**, **130** or on a remote computer system, for example, can establish communication with the cluster node modules **204***a-e*. In one embodiment, communication between the cluster configuration module **208** and the cluster node modules **204***a-e* initializes the cluster node modules **204***a-e* to provide cluster computing support for the computer cluster **100**.

A. Cluster Node Module **204**

In one embodiment, the cluster node modules **204***a-e* provide a way for many kernel modules **206***a-e* such as, for example, Mathematica kernels, running on a computer cluster **100** to communicate with one another. A cluster node module **204** can include at least a portion of an application programming interface ("API") known as the Message-Passing Interface ("MPI"), which is used in several supercomputer and cluster installations. A network of connections (for example, the arrows shown in FIG. **2**) between the cluster node modules **204***a-e* can be implemented using a communications network **102**, such as, for example, TCP/IP over Ethernet, but the connections could also occur over any other type of network or local computer bus.

A cluster node module **204** can use an application-specific toolkit or interface such as, for example, Mathematica's MathLink, Add-Ons, or packets, to interact with an application. Normally used to connect a Mathematica kernel to a user interface known as the Mathematica Front End or other Mathematica kernels, MathLink is a bidirectional protocol to sends "packets" containing messages, commands, or data between any of these entities. MathLink does not allow direct cluster computing-like simultaneous communication between Mathematica kernels during execution of a command or thread. MathLink is also not designed to perform multiple simultaneous network connections. In some embodiments, a cluster node module **204** can use an application-specific toolkit such as, for example, MathLink, for connections between entities on the same computer.

When speaking about procedures or actions on a cluster or other parallel computer, not all actions happen in sequential order, nor are they required to. For example, a parallel code, as opposed to a single-processor code of the classic "Turing machine" model, has multiple copies of the parallel code running across the cluster, typically one for each processor (or "processing element" or "core"). Such parallel code is written in such a way that different instances of the same code can communicate, collaborate, and coordinate work with each other. Multiple instances of these codes can run at the same time in parallel.

If the count of the code instances is an integer N, each instance of code execution can be labeled 0 through N−1. For example, a computer cluster can include N connected computers, each containing a processor. The first has cluster node module 0 connected with kernel module 0 running on processor 0. The next is cluster node module 1 and kernel module 1, on processor 1, and so forth for each of the N connected computers. Some steps of their procedure are collaborative, and some steps are independent. Even though these entities are not necessarily in lock-step, they do follow a pattern of initialization, main loop behavior (for example, cluster node module operation), and shut down.

In contrast, a parallel computing toolkit (PCT) that is provided as part of the gridMathematica software package does not provide a means for instances of the same code running on different nodes to communicate, collaborate, or coordinate work among the instances. The PCT provides commands that connect Mathematica kernels in a master-slave relationship rather than a peer-to-peer relationship as enabled by some embodiments disclosed herein. A computer cluster having peer-to-peer node architecture performs computations that can be more efficient, easier to design, and/or more reliable than similar computations performed on grid computers having master-slave node architecture. Moreover, the nature of some computations may not allow a programmer to harness multi-node processing power on systems that employ master-slave node architecture.

FIG. **3** shows one embodiment of a cluster node module **204** implementing MPI calls and advanced MPI functions. In the embodiment shown in FIG. **3**, cluster node module **204** includes MPI module **302**, advanced functions module **304**, received message queue **306**, and message receiving queue **308**.

1. MPI Module **302**

In one embodiment, the cluster node module **204** includes an MPI module **302**. The MPI module **302** can include program code for one or more of at least five kinds of MPI instructions or calls. Selected constants, instructions, and/or calls that can be implemented by the MPI module **302** are as follows:

MPI Constants

Node identifiers are used to send messages to nodes or receive messages from them. In MPI, this is accomplished by assigning each node a unique integer ($IdProc) starting with 0. This data, with a knowledge of the total count ($NProc), makes it possible to programmatically divide any measurable entity.

TABLE A

Constant	Description
$IdProc	The identification number of the current processor
$NProc	The number of processors in the current cluster
$mpiCommWorld	The communicator world of the entire cluster (see MPI Communicator routines, below)
mpiCommWorld	The default communicator world for the high-level routines.

Basic MPI Calls

In one embodiment, the MPI module **302** can include basic MPI calls such as, for example, relatively low-level routines that map MPI calls that are commonly used in other languages

(such as C and Fortran), so that such calls can be available directly from the Mathematica user interface **204**. In some embodiments, basic MPI calls include calls that send data, equations, formulas, and/or other expressions.

Simply sending expressions from one node to another is possible with these most basic MPI calls. One node can call to send an expression while the other calls a corresponding routine to receive the sent expression. Because it is possible that the receiver has not yet called mpiRecv even if the message has left the sending node, completion of mpiSend is not a confirmation that it has been received.

TABLE B

Call	Description
mpiSend[expr, target, comm, tag]	Sends an expression expr to a node with the ID target in the communicator would comm, waiting until that expression has left this kernel
mpiRecv [expr, target, comm, tag]	Receives an expression into expr from a node with the ID target in the communicator world comm, waiting until the expression has arrived
mpiSendRecv[sendexpr, dest, recvexpr, source, comm]	Simultaneously sends the expression sendexpr to the node with the ID target and receives an expression into recvexpr from the node with the ID source in the communicator world comm, waiting until both operations have returned.

Asynchronous MPI Calls

Asynchronous calls make it possible for the kernel to do work while communications are proceeding simultaneously. It is also possible that another node may not be able to send or receive data yet, allowing one kernel to continue working while waiting.

TABLE C

Call	Description
mpiISend[expr, target, comm, tag, req]	Sends an expression expr to a processor with the ID target in the communicator world comm, returning immediately. It can be balanced with calls to mpiTest[req] until mpiTest[req] returns True.
mpiIRecv[expr, target, comm, tag, req]	Receives an expression expr from a proessor with the ID target in the communicator world comm, returning immediately. It can be balanced with calls to mpiTest[req] until mpiTest[req] returns True. The expr is not safe to access until mpiTest[req] returns True.
mpiTest[req]	Completes asynchronous behavior of mpiISend and mpiIRecv
mpWait[req]	Calls mpiTest until it returns True.
mpiWaitall[reglist]	Calls mpiWait all on every element of reqlist
mpiWaitany[reqlist]	Calls mpiTest on each element of realist until one of them returns True

The mpiISend[ ] command can be called from within a kernel module **206** (for example, a Mathematica kernel). It creates a packet containing the Mathematica expression to be sent as payload and where the expression should be sent. The packet itself is destined only for its local cluster node module. Once received by its local cluster node module, this packet is decoded and its payload is forwarded on to the cluster node module specified in the packet.

The mpiIRecv[ ] command can also be called from within a kernel module **206**. It creates a packet specifying where it expects to receive an expression and from which processor this expression is expected. Once received by its local cluster node module, this packet is decoded and its contents are stored in a message receiving queue (MRQ) **308** (FIG. **3**).

The mpiTest[ ] command can be called from within a kernel module **206**. It creates a packet specifying which message to test for completion, then waits for a reply expression to evaluate. Once received by the kernel module's associated cluster node module **204**, this packet is decoded and its message specifier is used to search for any matching expressions listed as completed in its received message queue (RMQ) **306**. If such completed expressions are found, it is sent to its local kernel module as part of the reply in mpiTest[ ]. The kernel module receives this reply expression and evaluates it, which updates the kernel module's variables as needed.

Other MPI calls are built on the fundamental calls mpiISend, mpiIRecv, and mpiTest. For example, mpiBcast, a broadcast, creates instructions to send information from the broadcast processor to all the others, while the other processors perform a Recv. Similarly, high-level calls of the toolkit can be built on top of the collection of MPI calls.

Collective MPI Calls

In one embodiment, the MPI module **302** can include program code for implementing collective MPI calls (for example, calls that provide basic multi-node data movement across nodes). Collective MPI calls can include broadcasts, gathers, transpose, and other vector and matrix operations, for example. Collective calls can also provide commonly used mechanisms to send expressions between groups of nodes.

TABLE D

Call	Description
mpiBcast[expr, root, comm]	Performs a broadcast of expr from the root processor to all the others in the communicator world comm. An expression is expected to be supplied by the root processor, while all the others expect expr to be overwritten by the incoming expression.
mpiGather[sendexpr, recvexpr, root, comm]	All processors (including root) in the communicator comm send their expression in sendexpr to the root processor, which produces a list of these expressions, in the order according to comm, in recvexpr. On the processors that are not root, recvexpr is ignored.
mpiAllgather[sendexpr, recvexpr, comm]	All processors in the communicator comm send their expression in sendexpr, which are organized into a list of these expressions, in the order according to comm, in recvexpr on all processors in comm.
mpiScatter[sendexpr, recvexpr, root, comm]	Processor root partitions the list in sendexpr into equal parts (if possible) and places each piece in recvexpr on all the processors (including root) in the communicator world comm, according the order and size of comm.
mpiAlltoall[sendexpr, recvexpr, comm]	Each processor sends equal parts of the list in sendexpr to all other processors in the communicator world comm, which each collects from all other processors are organizes into the order according to comm.

In one embodiment, the MPI module **302** includes program code for implementing parallel sums and other reduction operations on data stored across many nodes. MPI module **302** can also include program code for implementing simple parallel input/output calls (for example, calls that allow cluster system **200** to load and store objects that are located on a plurality of nodes).

TABLE E

Call	Description
mpiReduce[sendexpr, recvexpr, operation, root, comm]	Performs a collective reduction operation between expressions on all processors in the communicator world comm for every element in the list in sendexpr returning the resulting list in recvexpr on the processor with the ID root.
mpiAllreduce[sendexpr, recvexpr, operation, comm]	Performs a collective reduction operation between expressions on all processors in the communicator world comm for every element in the list in sendexpr returning the resulting list in recvexpr on every processor.
mpiReduceScatter [sendexpr, recvexpr, operation, comm]	Performs a collective reduction operation between expressions on all processors in the communicator world comm for every element in the list in sendexpr, partitioning the resulting list into pieces for each processor's recvexpr.

These additional collective calls perform operations that reduce the data in parallel. The operation argument can be one of the constants below.

TABLE F

Constant	Description
mpiSum	Specifies that all the elements on different processors be added together in a reduction call
mpiMax	Specifies that the maximum of all the elements on different processors be chosen in a reduction call
mpiMin	Specifies that the minimum of all the elements on different processors be chosen in a reduction call

MPI Communicator Calls

In one embodiment, the MPI module **302** includes program code for implementing communicator world calls (for example, calls that would allow subsets of nodes to operate as if they were a sub-cluster). Communicators organize groups of nodes into user-defined subsets. The communicator values returned by mpiCommSplit[ ] can be used in other MPI calls instead of mpiCommWorld.

TABLE G

Call	Description
mpiCommSize[comm]	Returns the number of processors within the communicator comm
mpiCommRank[comm]	Returns the rank of this processor in the communicator comm
mpiCommDup[comm]	Returns a duplicate communicator of the communicator comm
mpiCommSplit[comm, color, key]	Creates a new communicator into several disjoint subsets each identified by color. The sort order within each subset is first by key, second according to the ordering in the previous communicator. Processors not meant to participate in any new communicator indicates this by passing the constant mpiUndefined. The corresponding communicator is returned to each calling processor.
mpiCommMap[comm] mpiCommMap[comm, target]	Returns the mapping of the communicator comm to the processor indexed according to $mpiCommWorld. Adding a second argument returns just the ID of the processor with the ID target in the communicator comm.
mpiCommFree[comm]	Frees the communicator comm

Other MPI Support Calls

Other calls that provide common functions include:

TABLE H

Call	Description
mpiWtime[ ]	Provides wall-dock time since some fixed time in the past. There is no guarantee that this time will read the same on all processors.
mpWtick[ ]	Returns the time resolution of mpiWtime[ ]
MaxByElement[in]	For every nth element of each list of the list in, chooses the maximum according to Max[ ], and returns the result as one list. Used in the mpiMax reduction operation.
MinByElement[in]	For every nth element of each list of the list in, chooses the minimum according to Min[ ], and returns the result as one list. Used in the mpiMin reduction operation.

2. Advanced Functions Module **304**

In one embodiment, the cluster node module **204** includes an advanced functions module **304**. The advanced functions module **304** can include program code that provides a toolkit of functions inconvenient or impractical to do with MPI instructions and calls implemented by the MPI module **302**. The advanced functions module **304** can rely at least partially on calls and instructions implemented by the MPI module **302** in the implementation of advanced functions. In one embodiment, the advanced functions module **304** includes a custom set of directives or functions. In an alternative embodiment, the advanced functions module **304** intercepts normal Mathematica language and converts it to one or more functions optimized for cluster execution. Such an embodiment can be easier for users familiar with Mathematica functions to use but can also complicate a program debugging process. Some functions implemented by the advanced functions module **304** can simplify operations difficult or complex to set up using parallel computing. Several examples of such functions that can be implemented by the advanced functions module **304** are shown below.

Built on the MPI calls, the calls that are described below provide commonly used communication patterns or parallel versions of Mathematica features. Unless otherwise specified, these are executed in the communicator mpiCommWorld, whose default is $mpiCommWorld, but can be changed to a valid communicator at run time.

Common Divide-and-Conquer Parallel Evaluation

In one embodiment, the advanced functions module **304** includes functions providing for basic parallelization such as, for example, routines that would perform the same operations on many data elements or inputs, stored on many nodes. These functions can be compared to parallelized for-loops and the like. The following calls address simple parallelization of common tasks. In the call descriptions, "expr" refers to an expression, and "loopspec" refers to a set of rules that determine how the expression is evaluated. In some embodiments, the advanced functions module **304** supports at least three forms of loopspec, including {var, count}, where the call iterates the variable var from 1 to the integer count; {var, start, stop}, where the call iterates the variable var every integer from start to stop; and {var, start, stop, increment}, where the call iterates the variable var from start adding increment for each iteration until var exceeds stop, allowing var to be a non-integer.

TABLE I

Call	Description
ParallelDo[expr, loopspec]	Like Do[ ] except that it evaluates expr across the cluster, rather than on just one processor. The rules for how expr is evaluated is specified in loopspec, like in Do[ ].

TABLE I-continued

Call	Description
ParallelFunctionToList[f, count] ParallelFunctionToList[f, count, root]	Evaluates the function f[i] from 1 to count, but across the cluster, and returns these results in a list. The third argument has it gather this list into the processor whose ID is root.
ParallelTable[expr, loopspec] ParallelTable[expr, loopspec, root]	Like Table[ ] except that it evaluates expr across the cluster, rather than on just one processor, returning the locally evalulated portion. The third argument has it gather this table in to the processor whose ID is root.
ParallelFunction[f, inputs, root]	Like f[inputs] except that it evaluates f on a subset of inputs scattered across the cluster from processor root and gathered back to root.
ParallelNintegrate[expr, loopspec] ParallelNintegrate[expr, loopspec, digits]	Like Nintegrate[ ] except that it evaluates a numerical integration of expr over domains partitioned into the number of processors in the cluster, then returns the sum. The third argument has each numerical integration execute with at least that many digits of precision.

Guard-Cell Management

In one embodiment, the advanced functions module **304** includes functions providing for guard-cell operations such as, for example, routines that perform nearest-neighbor communications to maintain edges of local arrays in any number of dimensions (optimized for 1-, 2-, and/or 3-D). Typically the space of a problem is divided into partitions. Often, however, neighboring edges of each partition can interact, so a "guard cell" is inserted on both edges as a substitute for the neighboring data. Thus the space a processor sees is two elements wider than the actual space for which the processor is responsible. EdgeCell helps maintain these guard cells.

TABLE J

Call	Description
EdgeCell[list]	Copies the second element of list to the last element of the left processor and the second-to-last element of list to the first element of the right processor while simultaneously receiving the same from its neighbors.

Matrix and Vector Manipulation

The advanced functions module **304** can also include functions providing for linear algebra operations such as, for example, parallelized versions of basic linear algebra on structures partitioned on many nodes. Such linear algebra operations can reorganize data as needed to perform matrix and vector multiplication or other operations such as determinants, trace, and the like. Matrices are partitioned and stored in processors across the cluster. These calls manipulate these matrices in common ways.

TABLE K

Call	Description
ParallelTranspose[matrix]	Like Transpose[ ] except that it transposes matrix that is in fact represented across the cluster, rather than on just one processor. It returns the portion of the transposed matrix meant for that processor.
ParallelProduct[matrix, vector]	Evaluates the product of matrix and vector, as it would on one processor, except that matrix is represented across the cluster.
ParallelDimensions[matrix]	Like Dimensions[ ] except that matrix is represented across the cluster, rather than on just one processor. It returns a list of each dimension.
ParallelTr[matrix]	Like Tr[ ] except that the matrix is represented across the cluster, rather than on just one processor. It returns the trace of this matrix.
ParallelIdentity[rank]	Like Identity[ ], it generates a new identity matrix, except that the matrix is represented across the cluster, rather than on just one processor. It returns the portion of the new matrix for this processor.
ParallelOuter[f, vector1, vector2]	Like Outer[f, vector1, vector2] except that the answer becomes a matrix represented across the cluster, rather than on just one processor. It returns the portion of the new matrix for this processor.
ParallelInverse[matrix]	Like Inverse[ ] except that the matrix is represented across the cluster, rather than on just one processor. It returns the inverse of the matrix.

Element Management

In one embodiment, the advanced functions module **304** includes element management operations. For example, a large bin of elements or particles cut up in space across the nodes may need to migrate from node to node based on rules or criteria (such as their spatial coordinate). Such operations would migrate the data from one node to another. Besides the divide-and-conquer approach, a list of elements can also be partitioned in arbitrary ways. This is useful if elements need to be organized or sorted onto multiple processors. For example, particles of a system may drift out of the space of one processor into another, so their data would need to be redistributed periodically.

TABLE L

Call	Description
ElementManage[list, switch]	Selects which elements of list will be sent to which processors according to the function switch[ ] is evaluated on each element of list.

TABLE L-continued

Call	Description
ElementManage[list]	If switch is a function, switch[ ] should return the ID of the processor that element should be sent. If switch is an integer, the call assumes that each elements is itself a list, whose first element is a number ranging from 0 to the passed argument. This call returns a list of the elements, from any processor, that is switch selected for this processor. Each element of list can be a list of two elements, the first being the ID of the processor where the element should be sent, while the second is arbitrary data to send. This call returns those list elements, from any and all processors, whose first element is this processors ID in a list. This call is used internally by the two-argument version of ElementManage[ ].

Fourier Transform

In one embodiment, the advanced functions module **304** includes program code for implementing large-scale parallel fast Fourier transforms ("FFTs"). For example, such functions can perform FFTs in one, two, and/or three dimensions on large amounts of data that are not stored on one node and that are instead stored on many nodes. Fourier transforms of very large arrays can be difficult to manage, not the least of which is the memory requirements. Parallelizing the Fourier transform makes it possible to make use of all the memory available on the entire cluster, making it possible to manipulate problem sizes that no one processor could possibly do alone.

TABLE M

Call	Description
ParallelFourier[list]	Like Fourier[ ] except that list is a two- or three-dimensional list represented across the cluster, like for matrices, above. It returns the portion of the Fourier-transformed array meant for that processor.

Parallel Disk I/O

In one embodiment, the advanced functions module **304** includes parallel disk input and output calls. For example, data may need to be read in and out of the cluster in such a way that the data is distributed across the cluster evenly. The calls in the following table enable the saving data from one or more processors to storage and the retrieval data from storage.

TABLE N

Call	Description
ParallelPut[expr, filename] ParallelPut[expr, filename root] ParallelPut[expr, filename, root, comm]	Puts expr into the file with the name filename in order on processor 0. The third argument specifies that the file be written on the processor whose ID is root. The fourth uses the communicator world comm.
ParallelGet[filename] ParallelGet[filename, root] ParallelGet[filename, root, comm]	Reads and returns data from the file with the name filename on processor 0 partitioned into each processor on the cluster. The second argument specifies that the file is to be read on the processor whose ID is root. The third uses the communicator world comm.
ParallelBinaryPut[expr, type, filename] ParallelBinaryPut[expr, filename, root] ParallelBinaryPut[expr, filename, root, comm]	Puts expr into the file with the binary format type with the name filename in order on processor 0. The fourth argument specifies that the file be written on the processor whose ID is root. The fifth uses the communicator world comm.
ParallelBinaryGet[type, filename] ParallelBinaryGet[type, filename, root] ParallelBinaryGet[type, filename, root, comm]	Reads and returns data in the binary format type from the file with the name filename on processor 0 partitioned into each processor on the cluster. The third argument specifies that the file is to be read on the processor whose ID is root. The fourth uses the communicator world comm.
ParallelGetPerProcessor [expr, filename] ParallelGetPerProcessor [filename, root] ParallelGetPerProcessor [filename, root, comm]	Puts expr into the file with the name filename in order on processor 0, one line per processor. The third argument specifies that the file be written on the processor whose ID is root. The fourth uses the communicator world comm.
ParallelGetPerProcessor [filename] ParallelGetPerProcessor [filename, root] ParallelGetPerProcessr [filename, root, comm]	Reads and returns data from the file with the name filename on processor 0, one line for each processor. The second argument specifies that the file is to be read on the processor whose ID is root. The third uses the communicator world comm.

Automatic Load Balancing

Some function calls can take an inconsistent amount of processing time to complete. For example, in Mathematica, the call f[20] could in general take much longer to evaluate than f[19]. Moreover, if one or more processors within the cluster are of different speeds (for example, if some operate at a core frequency of 2.6 GHz while other operate at less than one 1 GHz), one processor may finish a task sooner than another processor.

In some embodiments, the advanced functions module **304** includes a call that can improve the operation of the computer cluster **100** in such situations. In some embodiments, the root processor assigns a small subset of the possible calls for a function to each processor on the cluster **100**. Whichever processor returns its results first is assigned a second small subset of the possible calls. The root processor will continue to assign small subsets of the possible calls as results are received until an evaluation is complete. The order in which the processors finish can vary every time an expression is evaluated, but the root processor will continue assigning additional work to processors as they become available.

In one illustrative example, there are 4 processors and f[1] to f[100] to evaluate. One could implement this by assigning f[1], f[2], f[3], f[4] to each of processors 0 (the root can assign to oneself) through 3. If the f[2] result came back first, then processor 1 would be assigned f[5]. If the f[4] result is returned next, f[6] would be assigned to processor 3. The assignments continue until all results are calculated. The results are organized for output back to the user.

In alternative embodiments, the subsets of possible calls can be assigned in any order, rather than sequentially, or in batches (for example, f[1], f[5], f[9] assigned to processor **1**, etc.). Also, the subsets could be organized by delegation. For example, one processor node may not necessarily be in direct control of the other processors. Instead, a large subset could be assigned to a processor, which would in turn assign subsets of its work to other processors. The result would create a hierarchy of assignments like a vast army.

TABLE O

LoadBalanceFunctionToList[f, count] LoadBalanceFunctionToList[f, count, root]	Evaluates the function f[i] from 1 to count, but across the cluster using load-balancing techniques, and returns these results in a list. The third argument has it gather this list into the processor whose ID is root.

3. Received Message Queue **306**

In one embodiment, the cluster node module **204** includes a received message queue **306**. The received message queue **306** includes a data structure for storing messages received from other cluster node modules. Related data pertaining to the messages received, such as whether an expression has been completed, may also be stored in the received message queue **306**. The received message queue **306** may include a queue and/or another type of data structure such as, for example, a stack, a linked list, an array, a tree, etc.

4. Message Receiving Queue **308**

In one embodiment, the cluster node module **204** includes a message receiving queue **308**. The message receiving queue **308** includes a data structure for storing information about the location to which an expression is expected to be sent and the processor from which the expression is expected. The message receiving queue **308** may include a queue and/or another type of data structure such as, for example, a stack, a linked list, an array, a tree, etc.

B. Cluster Configuration Module **208**

Cluster configuration module **208** includes program code for initializing a plurality of cluster node modules to add cluster computing support to computer systems **110**, **120**, **130**. U.S. Pat. No. 7,136,924, issued to Dauger (the "'924 patent"), the entirety of which is hereby incorporated by reference and made a part of this specification, discloses a method and system for parallel operation and control of computer clusters. One method generally includes obtaining one or more personal computers having an operating system with discoverable network services. In some embodiments, the method includes obtaining one or more processors or processor cores on which a kernel module can run. As described in the '924 patent, a cluster node control and interface (CNCI) group of software applications is copied to each node. When the CNCI applications are running on a node, the cluster configuration module **208** can permit a cluster node module **204**, in combination with a kernel module **206**, to use the node's processing resources to perform a parallel computation task as part of a computer cluster. The cluster configuration module **208** allows extensive automation of the cluster creation process in connection with the present disclosure.

C. User Interface Module **202**

In some embodiments, computer cluster **100** includes a user interface module **202**, such as, for example a Mathematica Front End or a command line interface, that includes program code for a kernel module **206** to provide graphical output, accept graphical input, and provide other methods of user communication that a graphical user interface or a command-line interface provides. To support a user interface module **202**, the behavior of a cluster node module **204***a* is altered in some embodiments. Rather than sending output to and accepting input from the user directly, the user interface module **202** activates the cluster node module **204***a* to which it is connected and specifies parameters to form a connection, such as a MathLink connection, between the cluster node module **204***a* and the user interface module **202**. The user interface module's activation of the cluster node module **204***a* can initiate the execution of instructions to activate the remaining cluster node modules **204***b-e* on the cluster and to complete the sequence to start all kernel modules **206***a-e* on the cluster. Packets from the user interface module **202**, normally intended for a kernel module **206***a*, are accepted by the cluster node module **204***a* as a user command. Output from the kernel module **206***a* associated with the cluster node module **204***a* can be forwarded back to the user interface module **202** for display to a user. Any of the cluster node modules **204***a-e* can be configured to communicate with a user interface module **202**.

D. Kernel Module **206**

A kernel module **206** typically includes program code for interpreting high-level code, commands, and/or instructions supplied by a user or a script into low-level code, such as, for example, machine language or assembly language. In one embodiment, each cluster node module **204***a-e* is connected to all other cluster node modules, while each kernel module **206***a-e* is allocated and connected only to one cluster node module **204**. In one embodiment, there is one cluster node module-kernel module pair per processor. For example, in an embodiment of a computer cluster **100** including single-processor computer systems, each cluster node module-kernel module pair could reside on a single-processor computer. If a computer contains multiple processors or processing cores, it may contain multiple cluster node module-kernel module pairs, but the pairs can still communicate over the cluster node module's network connections.

IV. Cluster Computing Methods

In one embodiment, the computer cluster **100** includes a cluster initialization process, a method of cluster node module operation, and a cluster shut down process.

A. Cluster Initialization Process

In one embodiment, a cluster configuration module **202** initializes one or more cluster node modules **204** in order to provide cluster computing support to one or more kernel modules **206**, as shown in FIG. **4**.

At **402**, cluster node modules are launched on the computer cluster **100**. In one embodiment, the cluster node module **204***a* running on a first processor **112***a* (for example, where the user is located) accesses the other processors **112***b*, **122***a-b*, **132** on the computer cluster **100** via the cluster configuration module **208** to launch cluster node modules **204***b-e* onto the entire cluster. In an alternative embodiment, the cluster configuration module **208** searches for processors **112***a-b*, **122***a-b*, **132** connected to one another via communications network **102** and launches cluster node modules **204***a-e* on each of the processors **112***a-b*, **122***a-b*, **132**.

The cluster node modules **204***a-e* establish communication with one another at **404**. In one embodiment, each of the cluster node modules **204***a-e* establish direct connections using the MPI_Init command with other cluster node modules **204***a-e* launched on the computer cluster **100** by the cluster configuration module **208**.

At **406**, each cluster node module **204** attempts to connect to a kernel module **206**. In one embodiment, each instance of the cluster node modules **204***a-e* locates, launches, and connects with a local kernel module via MathLink connections and/or similar connection tools, for example, built into the kernel module **206**.

At **408**, the cluster node modules **204** that are unconnected to a kernel module **206** are shut down. In one embodiment, each cluster node module **204** determines whether the local kernel module cannot be found or connected to. In one embodiment, each cluster node module **204** reports the failure to connect to a kernel module **206** to the other cluster node modules on computer cluster **100** and quits.

Processor identification numbers are assigned to the remaining cluster node modules **204** at **410**. In one embodiment, each remaining cluster node module **204** calculates the total number of active processors (N) and determines identification numbers describing the remaining subset of active cluster node modules **204***a-e* and kernel modules **206***a-e*. This new set of cluster node module-kernel module pairs may be numbered 0 through N−1, for example.

Message passing support is initialized on the kernel modules **206***a-e* at **412**. In one embodiment, each cluster node module **204** supplies initialization code (for example, Mathematica initialization code) to the local kernel module **206** to support message passing.

Finally, at **414**, the cluster node modules **204***a-e* enter a loop to accept user entry. In one embodiment, a main loop (for example, a cluster operation loop) begins execution after the cluster node module **204***a* on the first processor **112***a* returns to user control while each of the other cluster node modules **204** waits for messages from all other cluster node modules **204***a-e* connected to the network **102**.

The initialization process creates a structure enabling a way for the kernel modules **206***a-e* to send messages to one another. In some embodiments, any kernel module can send data to and receive data from any other kernel module within the cluster when initialization is complete. The cluster node module creates an illusion that a kernel module is communicating directly with the other kernel modules. The initialization process can create a relationship among kernel modules on a computer cluster **100** such as the one shown by way of example in FIG. **2**.

B. Cluster Node Module Operation

In one embodiment, a cluster node module **204** implements cluster computing support for a kernel module **206** during a main loop, as shown in FIG. **5**.

At **502**, cluster node modules **204** wait for user commands or messages from other cluster node modules. In one embodiment, the cluster node module **204***a* connected to the user interface module **202** waits for a user command, while the other cluster node modules **204***b-e* continue checking for messages.

Once a command or message is received, the method proceeds to **504**. At **504**, the cluster node module **204***a* determines whether the message received is a quit command. If a quit command is received, the cluster node module **204***a* exits the loop and proceeds to a cluster node module shut down process at **505**. If the message received is not a quit command, the process continues to **506**.

At **506**, received commands are communicated to all cluster node modules **204***a-e* on the computer cluster **100**. In one embodiment, when a user enters a command in the user interface module **202**, the cluster node module **204***a* connected to the user interface module **202** submits the user command to all other cluster node modules **204***b-e* in the computer cluster **100**. The user commands can be simple (for example, "1+1"), but can also be entire subroutines and sequences of code (such as, for example, Mathematica code), including calls to MPI from within the user interface module **202** (for example, the Mathematica Front End) to perform message passing between kernel modules **206***a-e* (for example, Mathematica kernels). These include the fundamental MPI calls, which are implemented using specially identified messages between a cluster node module **204** and its local kernel module **206**.

The message (or user command) is communicated to the kernel modules **206***a-e* at **508**. In one embodiment, the cluster node module **204***a* connected to the user interface module **202** submits the user command to the kernel module **206***a* to which it is connected. Each of the other cluster node modules **204***b-e*, after receiving the message, submits the command to the respective kernel module **206***b-e* to which it is connected.

At **510**, a cluster node module **204** receives a result from a kernel module **206**. In one embodiment, once the kernel module **206** completes its evaluation, it returns the kernel module's output to the cluster node module **204** to which it is connected. Depending on the nature of the result from the kernel module, the cluster node module **204** can report the result to a local computer system or pass the result as a message to another cluster node module **204**. For example, the cluster node module **204***a* running on the first processor **112***a* reports the output on its local computer system **110**. For example, on the first processor **112***a*, cluster node module **204***a* only directly reports the output of kernel module **206***a*.

Messages from other cluster node modules **204** are responded to at **512**. In one embodiment, each cluster node module (for example, the cluster node module **204***a*) checks for and responds to messages from other cluster node modules **204***b-e* and from the kernel module **206***a* repeatedly until those are exhausted. In one embodiment, output messages from the kernel module **206** are forwarded to output on the local computer system. Messages from other cluster node modules **204** are forwarded to a received message queue **306** ("RMQ"). Data from each entry in the message receiving queue **308** ("MRQ") is matched with entries in the RMQ **306** (see, for example, description of the mpiIRecv[ ] call, above). If found, data from the MRQ **308** are combined into those in the RMQ **306** and marked as "completed" (see, for example, description of the mpiTest[ ] call, above). This process provides the peer-to-peer behavior of the cluster node modules **204***a-e*. Via this mechanism, code running within multiple, simultaneously running kernel modules (for example, Mathematica kernels) can interact on a pair-wise or collective basis, performing calculations, processing, or other work on a scale larger and/or faster than one kernel could have done alone. In this manner, user-entered instructions and data

specifying what work will be done via user commands can be executed more quickly and/or reliably. Once responding to messages has completed, the process returns to **502**.

In some embodiments, a computer system includes software, such as an operating system, that divides memory and/or other system resources into a user space, a kernel space, an application space (for example, a portion of the user space allocated to an application program), and/or an operating system space (for example, a portion of the user space allocated to an operating system). In some embodiments, some or all of the cluster node modules **204***a-e* are implemented in the application space of a computer system. In further embodiments, at least some of the cluster node modules **204***a-e* are implemented in the operating system space of a computer system. For example, some cluster node modules in a computer cluster may operate in the application space while others operate in the operating system space.

In some embodiments, some or all of the functionality of the cluster node modules **204***a-e* is incorporated into or integrated with the operating system. The operating system can add cluster computing functionality to application programs, for example, by implementing at least some of the methods, modules, data structures, commands, functions, and processes discussed herein. Other suitable variations of the techniques described herein can be employed, as would be recognized by one skilled in the art.

In some embodiments, the operating system or components of the operating system can identify and launch the front end **202** and the kernels **206**. The operating system or its components can connect the front end **202** and kernels **206** to one another in the same manner as a cluster node module **204** would or by a variation of one of the techniques described previously. The operating system can also be responsible for maintaining the communications network **102** that connects the modules to one another. In some embodiments, the operating system implements at least some MPI-style calls, such as, for example, collective MPI-style calls. In some embodiments, the operating system includes an application programming interface (API) library of cluster subroutine calls that is exposed to application programs. Applications programs can use the API library to assist with launching and operating the computer cluster.

C. Cluster Shut Down Process

In one embodiment, a computer cluster **100** includes a procedure to shut down the system. If the operation process (or main loop) on the cluster node module **204***a* connected to the user interface module **202** detects a "Quit" or "Exit" command or otherwise receives a message from the user indicating a shut down, the sequence to shut down the cluster node modules **204***a-e* and the kernel modules **206***a-e* is activated. In one embodiment, the cluster node module **204***a* connected to the user interface module **202** sends a quit message to all other cluster node modules **204***b-e*. Each cluster node module **204** forwards the quit command to its local kernel module **206**. Once its Mathematica kernel has quit, each cluster node module **204** proceeds to tear down its communication network with other cluster node modules (for example, see description of the MPI_Finalize command, above). At the conclusion of the process, each cluster node module **204** exits execution.

V. Example Operation

For purposes of illustration, sample scenarios are discussed in which the computer cluster system is used in operation. In these sample scenarios, examples of Mathematica code are given, and descriptions of how the code would be executed by a cluster system are provided.

Basic MPI

Fundamental data available to each node includes the node's identification number and total processor count.

```
In[1]:= {$IdProc, $NProc}
Out[1]:= {0, 2}
```

The first element should be unique for each processor, while the second is generally the same for all. Processor 0 can see what other values are using a collective (see below) communications call such as mpiGather[ ].

```
In[2]:= mpiGather[{$IdProc, $NProc},list,0]; list
Out[2]:= {{0, 2}, {1, 2}}
```

Peer-to-Peer MPI

The mpiSend and mpiRecv commands make possible basic message passing, but one needs to define which processor to target. The following defines a new variable, targetProc, so that each pair of processors will point to each other.

```
In[3]:= targetProc=If[1==Mod[$IdProc, 2 ],$IdProc−1,$IdProc+l]
Out[3]:= 1
```

In this example, the even processors target its "right" processor, while the odd ones point its "left." For example, if the processors were lined up in a row and numbered in order, every even-numbered processor would pair with the processor following it in the line, and every odd-numbered processor would pair with the processor preceding it. Then a message can be sent:

```
In[4]:= If [ 1==Mod[ $IdProc , 2],mpiSend[N[Pi,22],targetProc,
    mpiCommWorld,d], mpiRecv[a,targetProc,mpiCommWorld,d]]
```

The If [ ] statement causes the processors to evaluate different code: the odd processor sends 22 digits of Pi, while the even receives that message. Note that these MPI calls return nothing. The received message is in the variable a:

```
In[5]:= a
Out[5]:= 3.1415926535897932384626
In[6]:= Clear[a]
```

The variable a on the odd processors would have no definition. Moreover, if $NProc is 8, processor 3 sent Pi to processor 2, processor 5 sent Pi to processor 4, and so on. These messages were not sent through processor 0, but they communicated on their own.

The mpiISend and mpiIRecv commands have a letter "I" to indicate asynchronous behavior, making it possible to do other work while messages are being sent and received, or if the other processor is busy. So, the above example could be done asynchronously:

```
In[7]:= If[1==Mod[$IdProc, 2],mpiISend[N[Pi,22],targetProc,
    mpiCommWorld,d,e],
    mpiIRecv[a,targetProc,mpiCommWorld,d,e]]
```

The variable e has important data identifying the message, and mpiTest[e] can return True before the expressions are to be accessed. At this point, many other evaluations can be performed. Then, one can check using mpiTest when the data is needed:

```
In[29]:= mpiTest[e]
Out[29]:= True
In[30]:= a
Out[30]:= 3.14159265358979323846 26
In[31]:= Clear[a,e]
```

The mpiWait[e] command could have also have been used, which does not return until mpiTest[e] returns True. The power of using these peer-to-peer calls is that it becomes possible to construct any message-passing pattern for any problem.

Collective MPI

In some cases, such explicit control is not required and a commonly used communication pattern is sufficient. Suppose processor 0 has an expression in b that all processors are meant to have. A broadcast MPI call would do:

In[8]:=mpiBcast[b, 0, mpiCommWorld]

The second argument specifies which processor is the "root" of this broadcast; all others have their b overwritten. To collect values from all processors, use mpiGatherD:

In[9]:=mpiGather[b, c, 0, mpiCommWorld]

The variable c of processor 0 is written with a list of all the b of all the processors in mpiCommWorld. The temporal opposite is mpiScatter:

```
In[10]:= Clear[b]; a = {2, 4, 5, 6}; mpiScatter[a, b, 0,
    mpiCommWorld]; b
Out[10]:= {2, 4}
```

The mpiScatter command cuts up the variable a into even pieces (when possible) and scatters them to the processors. This is the result if $NProc=2, but if $NProc=4, b would only have {2}.

MPI provides reduction operations to perform simple computations mixed with messaging. Consider the following:

```
In[11]:= a = {{2 + $IdProc, 45[ ],3,{1 + $IdProc,$NProc[ ][ ];
    mpiReduce [a, d, mpiSum, 0, mpiCommWorld ]
In[12]:= d
Out[12]:= {{5, 90}, 6, {3, 4}}
```

The mpiSum constant indicates that variable a of every processor will be summed. In this case, $NProc is 2, so those elements that were not identical result in odd sums, while those that were the same are even.

Most of these calls have default values if not all are specified. For example each of the following calls will have the equivalent effect as the above mpiGather[ ] call:

```
mpiGather[b, c, 0]
mpiGather[b, c]
c = mpiGather[b]
```

High-Level Calls

High-level calls can include convenient parallel versions of commonly used application program calls (for example, Mathematica calls). For example, ParallelTable[ ] is like Table[ ], except that the evaluations are automatically performed in a distributed manner:

```
In[13]:= ParallelTable[i,{i,100},0]
Out[13]:= {1,2,3,4,5, . . . ,99,100}
```

The third argument specifies that the answers are collated back to processor 0. This is a useful, simple way to parallelize many calls to a complex function. One could define a complicated function and evaluate it over a large range of inputs:

```
In[14]:= g[x__] := Gamma[2 + 0.5*(x−1)];
    ParallelTable[g[i],{i,100},0]
Out[14]:= {1, 1.32934, 2., 3.32335, 6., 11.6317, 24., 52.3428,
    120., 287.885, 720}
```

ParallelFunctionToList[ ] also provides a simplified way to perform this form of parallelism.

Operations with Non-Trivial Communication

Matrix Operations

In some embodiments, one or more functions can help solve matrix calculations in parallel:

```
In[15]:= a = Table[i+ 3* $IdProc + 2 j, {i, 2}, {j,4}]
Out[15] := {{3, 5, 7, 9}, {4, 6, 8, 10}}
In[16]:= t = ParallelTranspose[a]
Out[16]:= {{3, 4, 6, 7}, {5, 6, 8, 9}}
```

Fourier Transforms

A Fourier transform of a large array can be solved faster in parallel, or made possible on a cluster because it can all be held in memory. A two-dimensional Fourier transform of the above example follows:

```
In[17]:= f = ParallelFourier[a]
Out[17]:= {{32. + 0. I, −4. − 4. I, −4., −4. + 4. I}, {−3. −
    3. I, 0. + 0. I, 0., 0. + 0. I}}
```

Edge Cell Management

Many problems require interactions between partitions, but only on the edge elements. Maintaining these edges can be performed using EdgeCell[ ].

```
In[18]:= a = {2, 4, 5, 6, 7 }+8*$IdProc
Out[18]:= {2, 4, 5, 6, 7}
In[19]:= EdgeCell[a]; a
Out[19]:= {14, 4, 5, 6, 12}
```

Element Management

In particle-based problems, items can drift through space, sometimes outside the partition of a particular processor. This can be solved with ElementManage[1:

```
In[20]:= list={{0,4},{1,3},{1,4},{0,5}}; fcn[x_]:=x[[1]]
In[21]:= ElementManage[list, fcn]
Out[21]:= {{0, 4}, {0, 5}, {0, 4}, {0, 5}}
In[21]:= ElementManage[list, 2]
Out[21]:= {{0, 4}, {0, 5}, {0, 4}, {0, 5}}
```

The second argument of ElementManage describes how to test elements of a list. The fcn identifier returns which processor is the "home" of that element. Passing an integer assumes that each element is itself a list, whose first element is a number ranging from 0 to the passed argument.

While the examples above involve Mathematica software and specific embodiments of MPI calls and cluster commands, it is recognized that these embodiments are used only to illustrate features of various embodiments of the systems and methods.

VI. Additional Embodiments

Although cluster computing techniques, modules, calls, and functions are disclosed with reference to certain embodiments, the disclosure is not intended to be limited thereby. Rather, a skilled artisan will recognize from the disclosure herein a wide number of alternatives for the exact selection of cluster calls, functions, and management systems. For example, single-node kernels can be managed using a variety of management tools and/or can be managed manually by a user, as described herein. As another example, a cluster node module can contain additional calls and procedures, including calls and procedures unrelated to cluster computing, that are not disclosed herein.

Other embodiments will be apparent to those of ordinary skill in the art from the disclosure herein. Moreover, the described embodiments have been presented by way of example only, and are not intended to limit the scope of the disclosure. Indeed, the novel methods and systems described herein can be embodied in a variety of other forms without departing from the spirit thereof. Accordingly, other combinations, omissions, substitutions and modifications will be apparent to the skilled artisan in view of the disclosure herein. Thus, the present disclosure is not intended to be limited by the disclosed embodiments, but is to be defined by reference to the appended claims. The accompanying claims and their equivalents are intended to cover forms or modifications as would fall within the scope and spirit of the inventions.

What is claimed is:

1. A system for performing an instruction received from a front end by executing commands on one or more special purpose microprocessors, the system comprising:

a plurality of nodes, wherein each node is configured to access a computer-readable memory system comprising program code for a single-node kernel module, and wherein each single-node kernel module is configured to interpret instructions received by the single-node kernel module into commands that are executable by a special purpose microprocessor;

a plurality of cluster node modules, wherein each cluster node module is stored in a computer-readable memory system and configured to communicate with a single-node kernel and with one or more other cluster node modules, to accept instructions, and to interpret at least some of the instructions such that the plurality of cluster node modules communicate with one another in order to act as a cluster in executing commands using one or more hardware processors; and

a communications system configured to connect the plurality of nodes;

wherein the plurality of cluster node modules cooperate to interpret and translate, as needed, the instruction for execution by a plurality of single-node kernel modules, and wherein at least one of the plurality of cluster node modules returns a result to the front end.

2. The system of claim **1**, wherein the special purpose microprocessor comprises a digital signal processor.

3. The system of claim **1**, wherein the plurality of nodes are organized into two or more groups of node subsets.

4. The system of claim **3**, wherein at least one of the two or more groups of node subsets exchange data with the special purpose microprocessor.

5. The system of claim **1**, wherein the special purpose microprocessor comprises multiple processor cores.

6. The system of claim **1**, wherein at least one of the plurality of cluster node modules resides in processor cache memory.

7. The system of claim **1**, wherein each single-node kernel modules resides in processor cache memory.

8. A system for performing an instruction received from a front end by executing commands on one or more hardware processors, the system comprising:

a plurality of nodes, wherein each node is configured to access a computer-readable memory system comprising program code for a single-node kernel module, and wherein each single-node kernel module is configured to interpret instructions received by the single-node kernel module into commands that are executable by a hardware processor, wherein the commands are configured to perform computations one or more elements from a list of elements;

a plurality of cluster node modules, wherein each cluster node module is stored in a computer-readable memory system and configured to communicate with a single-node kernel and with one or more other cluster node modules, to accept instructions, and to interpret at least some of the instructions such that the plurality of cluster node modules communicate with one another in order to act as a cluster in performing computations on the list of elements; and

a communications system configured to connect the plurality of nodes;

wherein the plurality of cluster node modules cooperate to interpret and translate, as needed, the instruction for execution by a plurality of single-node kernel modules, wherein the list of elements is partitioned for execution on the plurality of nodes, and wherein at least one of the plurality of cluster node modules returns a result to the front end.

9. The system of claim **8**, wherein one or more elements of the list of elements migrates to a different node.

10. A method of performing an instruction received from a front end by executing commands on one or more special purpose microprocessors, the method comprising:

communicating an instruction from a front end to one or more cluster node modules connected to one another by a communications system;

for each of the one or more cluster node modules, communicating a message based on the instruction to a respective kernel module associated with the cluster node mod-

ule, wherein the respective kernel module is configured to interpret the message into commands that are executable by a special purpose microprocessor;

for each of the one or more cluster node modules, receiving a result from the respective kernel module associated with the cluster node module; and

for at least one of the one or more cluster node modules, responding to messages from other cluster node modules.

11. The method of claim **10**, wherein communicating a message based on the command to a respective kernel module associated with the cluster node module comprises communicating a specially identified message to the respective kernel module.

12. The method of claim **10**, wherein responding to messages from other cluster node modules comprises:

forwarding messages from the other cluster node modules to a received message queue;

matching data from each entry in a message receiving queue with entries in the received message queue;

combining data from the message receiving queue with matching data in the received message queue; and

marking the matching data as completed.

13. The method of claim **10**, wherein communicating a command from at least one of a user interface or a script to one or more cluster node modules within the computer cluster comprises communicating an instruction from a Mathematica Front End to a first cluster node module, wherein the first cluster node module forwards the instruction to other cluster node modules running on the computer cluster.

14. The method of claim **10**, wherein each of the one or more cluster node modules communicates with a respective kernel module using MathLink.

* * * * *