8

Hierarchy-Aware Message-Passing in the Upcoming Many-Core Era

Carsten Clauss, Simon Pickartz, Stefan Lankes and Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University
Germany

1. Introduction

The demands of large parallel applications often exceed the computing and memory resources a local computing site offers. Therefore, by combining distributed computing resources as provided by Grid environments can help to satisfy these resource demands. However, since such an environment is a heterogeneous system by nature, there are some drawbacks that, if not taken into account, are limiting its applicability. Especially the inter-site communication often constitutes a bottleneck in terms of higher latencies and lower bandwidths than compared to the site-internal case. The reason for this is that the inter-site communication is typically handled via wide-area transport protocols and respective networks; whereas the internal communication is conducted via fast local-area networks or even via dedicated high-performance cluster interconnects. That in turn means that an efficient utilization of such a hierarchical and heterogeneous infrastructure demands a Grid middleware that provides support for all these different kinds of communication facilities (Clauss et al., 2008). Moreover, with the upcoming Many-core era a further level of hierarchy gets introduced in terms of *Cluster-on-Chip* processor architectures. The Single-chip Cloud Computer (SCC) experimental processor is a *concept vehicle* created by Intel Labs as a platform for Many-core software research (Intel Corporation, 2010). This processor is indeed a very recent example for such a Cluster-on-Chip architecture. In this chapter, we want to discuss the challenges of hierarchy-aware message-passing in distributed Grid environments in the upcoming Many-core era by taking the example of the SCC. The remainder of this chapter is organized as follows: Section 2 initially reviews the basic knowledge about parallel processing and message-passing. In Section 3, the demands for parallel processing and message-passing especially in Grid computing environments are detailed. Section 4 focuses on the Intel SCC Many-core processor and how message-passing can be conducted with respect to this chip. Afterwards, Section 5 discusses how the world of chip-embedded Many-core communication can be integrated into the macrocosmic world of Grid computing. Finally, Section 6 concludes this chapter.

2. Parallel processing using message-passing

With a rising amount of cores in today's processors, parallel processing is a prevailing field of research. One approach is the *message-passing paradigm*, where parallelization is achieved by having processes with the capability of exchanging messages with other processes. Instead

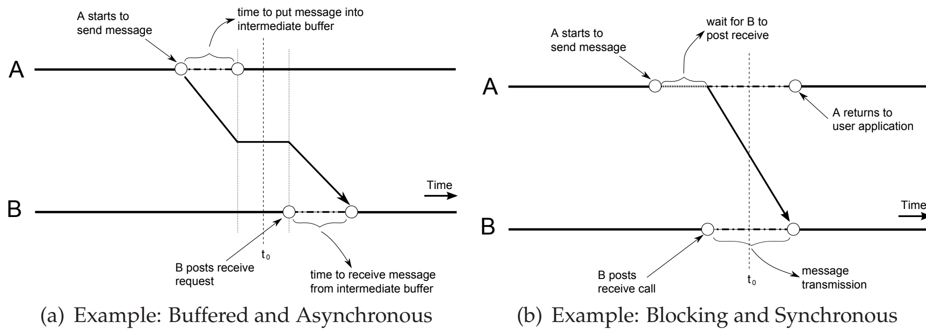
of sharing common memory regions, processes perform send and receive operations for data and information transfer. In high-performance computing the message-passing paradigm is well established. However, this programming model gets more and more interesting also for the consumer sector. The message-passing model is mostly architecture independent, but it may profit from underlying hardware that supports the shared-memory model in terms of more performance. It is accompanied by a strictly separated address space. Therefore erroneous memory reads and writes are easier to locate than it would be with shared memory programming (Gropp et al., 1999).

2.1 Communication modes

The inter-process communication for synchronization and data exchanges has to be performed by calling send and received functions in an explicit manner. In doing so, the parallelization strategy is to divide the algorithm into independent subtasks and to assign these tasks to parallel processes. However, at the end of these independent subtasks intermediate results need commonly to be exchanged between the processes in order to compute the overall result.

2.1.1 Point-to-point communication

In point-to-point communication several different communication modes have to be distinguished: *buffered* and *non-buffered*, *blocking* and *non-blocking*, *interleaved* and *overlapped*, *synchronous* and *asynchronous* communication. First of all, *non-buffered* and *buffered* communication has to be differentiated. The latter requires an intermediate data buffer through which sender and receiver perform communication. A send routine will send a message to a buffer that may be accessed by both the sending and the receiving side. Calling the respective receive function, the message will be copied out of that buffer and stored locally at the receiving process. Figure 1(a) shows sender A transmitting a message to an intermediated buffer and returning after completion. The buffer holds the message until receiver B posts the respective receive call and completes the data transfer to its local buffer. In addition to that, the terms *blocking* and *non-blocking* related to message-passing have to be defined. They relate to the semantics of the respective send and receive function calls. A process that calls a blocking send function remains in this function until the transfer is completed. Whether this is associated with the arrival of the according message at the receiving side or only with the completion of the transmission on the sender side, has to be defined in the context where the function is used. Figure 1(b) shows an example where the completion of a blocking send call is defined as the point in time after the whole message arrived at the receiver and is stored in a local buffer. For this period, A is blocked even if B has not posted the respective receive call yet. On the contrary, a non-blocking send routine returns immediately regardless whether the message arrived at the receiver or not. Thus, the sender has to ensure with other mechanisms that a message was successfully transmitted before reusing its local send buffer. With non-blocking routines it is possible to perform *interleaved* but also *overlapped* communication and computation. Overlapped communication results in real parallelism where the data delivery occurs autonomously after being pushed by the sending process. Meanwhile, the sender is able to perform computation that is independent from the transmitted data. The same applies to the receiving side. With interleaved communication, message dependencies may be broken up, but there is still a serialized processing which requires a periodical alternating between computation and communication.



(a) Example: Buffered and Asynchronous (b) Example: Blocking and Synchronous

Fig. 1. Comparison of Asynchronous and Synchronous Communication

This may increase the application's performance if a resource is currently not available. Instead of waiting for the opponent to be ready, time is used to perform other tasks. Thus, the application itself has to check from time to time if the communication process is still stuck, by calling functions to query the status of the respective *request handle*. These are objects which are commonly being passed back for this purpose by a non-blocking function call. Although often used as a synonym for blocking/non-blocking function calls (Tanenbaum, 2008), *synchronous* and *asynchronous* communication primitives should be further distinguished. Facilitated by buffered communication, asynchronous message-passing enables the sender to complete the send routine without having the receiver posted a respective receive call. Thus, it is not necessary to have a global point in time when sender and receiver are coevally within their respective function calls. In Figure 1(a) *A* returns from the send call and *locally* completes the message transfer before the matching receive routine is posted. In contrast to that, in non-buffered mode where no intermediate communication buffer is available, it is not possible to perform asynchronous message-passing. That is because data transfer only occurs when both, sender and receiver, are situated in the communication routines at the same time. Referring to Figure 1(b) it becomes clear what is meant by *one point in time*. At time t_0 both, sender and receiver, are in the respective communication routines what is necessary in order to complete them.

2.1.2 Collective communication operations

Collective operations are communication functions that have to be called by all participating processes. These functions help the programmer to realize more complex communication patterns than simple point-to-point communication within a single function call. Moreover, it must be emphasized that using such collective operations not only simplifies the application programming, but also enables the lower communication layer to implement the collective communication patterns in the most efficient way. For that reason, application programmers should utilize offered collective operations instead of implementing the patterns by means of point-to-point communication whenever possible. However, a possible drawback of collective operations is that they may be synchronizing what means that the respective function may only return when all participating processes have called it. In case of unbalanced load, processes possibly have to wait a long time within the function, not being able to progress with the computation. In the following, some important examples of collective communication

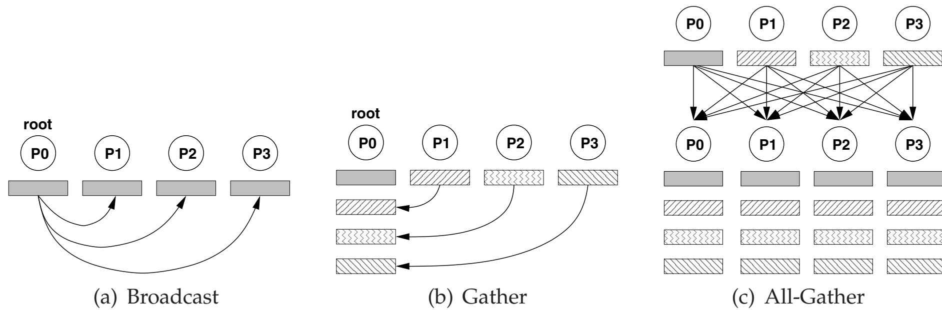


Fig. 2. Examples of Collective Communication Patterns

patterns are shown. Although important, not nearly all communication libraries provide collective functions for these patterns. If a process needs to send a message to all the other processes, a *broadcast* function (if provided by the communication library) can be utilized. In doing so, all participating processes have to call this function and have to state who is the initial sender (the so-called *root*, see Figure 2(a)) among them. In turn, all the others realize that they are eventual receivers so that the communication pattern can be conducted. However, during the communication progress of the pattern every process can become a sender and/or receiver. That means that the internal implementation of the pattern is up to the respective library. For example, internally this pattern may be conducted in terms of a loop over all receivers, or even better tree-like achieving higher performance. In many parallel algorithms, a so-called *master* process is used to distribute subtasks among the other processes (the so-called *worker*) and to coordinate the collection of partial results later on. Therefore, such a master may initially act as the root process of a broadcast operation distributing subtask-related data. Afterwards, a *gather* operation then may be used at the master process to collect partial results generating the final result with the received data. Figure 2(b) shows the pattern of such a gather operation. Besides this asymmetric master/worker approach, symmetric parallel computation (and hence communication) schemes are common, too. This means, regarding collective operations, that for example during a gather operation *all* processes obtain *all* partial datasets (a so-called *all-gather* operation). Internally, this may be for example implemented in terms of an initial gather operation to one process, followed by a subsequent broadcast to all processes. However, the internal implementation of such communication patterns can also be realized in a symmetric manner, as Figure 2(c) shows for the all-gather example.

2.2 Process topologies

A process topology describes the logical and/or physical arrangement of parallel processes within the communication environment. Thus, the *logical* arrangement represents the communication pattern of the parallel algorithm, whereas the *physical* arrangement constitutes the assignment of processes to physical processors. Of course, in hierarchical (or even heterogeneous) systems, the logical process topology should be mapped onto the underlying physical topology in such a way that they are as congruent as possible. For example, and as already noted in the last section, collective communication patterns should be adapted to the underlying hardware topologies. This may be done, for instance, by an optimized

communication library as it is described for hierarchical systems in the later Section 3.2.2. Moreover, even an adaptation of the parallel algorithm itself to the respective hardware topology may become necessary in order to avoid unnecessary network contention. Therefore, a likewise hierarchical algorithm design would accommodate such systems. However, in homogeneous environments, the algorithm design can still be kept flat and process topologies are mapped almost transparently onto the hardware.

2.2.1 Programming paradigms

Based on the consideration where to place the processes and which part of a parallel task each of them should process, two programming paradigms can be distinguished (Wilkinson & Allen, 2005): the *Multiple-Program Multiple-Data* (MPMD) and the *Single-Program Multiple-Data* (SPMD) paradigm. According to the MPMD paradigm, each process working on a different subtask within a parallel session processes an individual program. Therefore, in an extreme case, all parallel processes may run different programs. However, usually this paradigm is not that distinctive. A very common example for MPMD is the master/worker approach where just the master runs a different program than the workers. In contrast to this, in a session according to the SPMD paradigm, all processes run only one single program. That in turn implies that the processes must be able to identify themselves[1] because otherwise all of them would work on the same subtask.

2.2.2 Session startup and process spawning

Considering the question which process should work on which subtask leads to a further question: When shall the processes of a session be created? Regarding this problem, two approaches can be distinguished: In the case of a *static* startup, all the processes are created at the beginning of a parallel run and are normally bound to their respective processors during runtime. Such a static startup is usually conducted and supported by a job scheduler detecting and assigning idle processors. However, in the case of a *dynamic* process startup, further processes can be spawned by already running processes even at runtime. This approach is commonly combined with the MPMD paradigm. So for example, when a master process running a master program spawns worker processes running subtask-related subprograms. However, this approach demands for an additional interaction between spawning processes and the runtime environment during execution, in order to place spawned processes onto free processors. That is the reason why this approach is more complicated in most cases.

2.3 Programming interfaces

The actual handling of a message transfer, that is the execution of the respective communication protocols through the different networking layers, is much too complex and too hardware-oriented to be done at application level. Therefore, the application programmer is usually provided with appropriate communication libraries that hide the hardware-related part of message transfers and hence allow the development of platform-independent parallel applications.

[1] for example by means of *process identifiers*

2.3.1 The Berkeley Socket API

A very common communication interface is the Berkeley Socket API, also known as BSD Sockets. A *socket* is a communication termination endpoint that facilitates the access to various transport layer protocols, such as TCP and UDP (Winett, 1971). Although usable to communicate between processes on the same machine, their intention is to enable the inter-process communication over computer networks. This can be done either first establishing a connection via creating a stream socket for the TCP protocol, or connectionless using datagram sockets for the UDP protocol. The sockets are managed by the operating system which organizes the access to the underlying network. They are used in a Client-Server manner, what means that the connection establishment between a pair of processes must be triggered in an asymmetric way, starting from the client side. Afterwards, messages may be exchanged bidirectional via the socket by using simple send and receive functions (Stevens et al., 2006).

2.3.2 Communication libraries for parallel environments

Besides simple send and receive functions, communication libraries especially for parallel environments do not only offer simple Client-Server relations, but rather provide support for a session management covering all parallel processes, including process startup and an all-to-all connection establishment. Commonly such libraries also offer additional features, as for example, for conducting collective operations or for transparent data conversion. In the course of time, several of such communication libraries had been developed, usually driven by the demand for new libraries in connection with new hardware platforms. Examples are: *NX*, *NX/2* and *NX/M* that are libraries developed by Intel for a past generation of dedicated multi-computers (Pierce, 1988), *Zipcode* is a software system for message-passing developed by the California Institute of Technology (Skjellum & Leung, 1990), *P4: Portable Programs for Parallel Processors* is a socket-based communication library by Argonne National Laboratory (Butler & Lusk, 1994), *Chameleon* is no communication library by itself but rather a macro-based interface to several underlying communication libraries (Gropp & Smith, 1993), and *PVM: Parallel Virtual Machine* is still a very common communication library (Dongarra et al., 1993) that has also been extended by the ability to be runnable in Grid environments (Geist, 1998).

2.3.3 The Message-Passing Interface (MPI)

When looking at the diversity of communication libraries listed in the last section, it becomes obvious that writing portable parallel applications was hardly possible in those days. Hence, there was a strong demand for the creation of a unified interface standard for parallel communication libraries in the early 1990s. This demand for an easy portability of parallel applications to always new generations of parallel machines eventually led in 1993 to the definition of such a unified library interface by the so-called Message-Passing Interface Forum. The goal was to define a communication standard that is hardware and programming language independent but still meets the requirements of high-performance computing. The result was the Message-Passing Interface Standard (MPI), which is a specification of a library interface (Message Passing Interface Forum, 2009). The main objective is that users do not need to compromise among efficiency, portability and functionality without having to abstain

from advantages of specialized hardware (Gropp et al., 1999). Although MPI is, in contrast to the libraries mentioned in the last section, *not* a specific implementation but just an interface standard, the standardizing processes was accompanied by the development of a prototype and reference implementation: MPICH (Gropp et al., 1996).[2]

Today, two different compatibility levels can be distinguished:[3] Compatibility with MPI-1 means that an MPI implementation provides support for all features specified in the MPI standard Version 1.3. And compatibility with MPI-2 means as opposed to MPI-1 that the respective MPI implementation also provides support for the extensions specified up to the MPI standard Version 2.2. Altogether, these two levels incorporate a function set of about 280 MPI functions. However, many MPI applications just use a handful of them, mostly focusing on the actual message handling. To begin with the term of a message, the tuple (*address*, *count*, *datatype*) defines an MPI message buffer, in which *count* describes the amount of elements of *datatype* beginning at *address*. Thus, it is ensured that the receiving side obtains the same data even if it uses another data format than the sending side. To distinguish between several messages, a *tag* is introduced that represents the message type defined by the user application. Furthermore, MPI defines the concepts of the *context* and *groups* aggregated in a so-called *communicator*. Only messages with a valid context (that is in terms of a matching communicator) will be received and processes may be combined to logical groups by means of the communicator. In addition to these basic concepts, a wide range of further mechanism like non-blocking, buffered or synchronizing communication, as well as collective operations and a particular error handling is provided. Although most MPI applications are written according to the SPMD paradigm, MPI-2 also features process spawning and support for programs written according to the MPMD paradigm.

2.3.4 The Multicore Communications API (MCAPI)

The MCAPI, recently developed by the Multicore Association, resembles an interface for message-passing like MPI. However, in contrast to MPI and sockets which were primely designed for inter-computer communication, the MCAPI intends to facilitate lightweight inter-core communication between cores on one chip (Multicore Association, 2011). These may be even those which execute code from chip internal memory. Therefore the MCAPI tries to avoid the von Neumann bottleneck[4] using as less memory as it is necessary to realize communication between the cores. According to this, the two main goals of this API are extremely high-performance and low memory footprint of its implementations. In order to achieve these principals, the specification sticks to the KISS[5] principal. Only a small number of API calls are provided that allow efficient implementations on the one hand, and the opportunity to build other APIs that have more complex functionality on top of it, on the other hand. For an inter-core communications API, such as MCAPI, it is much easier to realize these goals because an implementation does not have to concern issues like reliability

[2] Nowadays, two more popular and also freely available MPI implementations exist: Open MPI (Gabriel et al., 2004) and MPICH2 (Gropp, 2002).

[3] Currently, the specifications of the upcoming MPI-3 standard are under active development by the working groups of the Message-Passing Interface Forum.

[4] It describes the circumstance that program memory and data memory share the same bus and thus result in a shortage in terms of throughput.

[5] Keep It Small and Simple

and packet loss which is the case in computer networks for example. In addition to that, the interconnect between cores on a chip offered by the hardware facilitates high-performance data transfer in terms of latency and throughput. Although designed for communication and synchronization between cores on a chip in embedded systems, it does not require the cores to be homogeneous. An implementation may realize communication between different architectures supported by an arbitrary operating system or even bare-metal. The standard purposely avoids having any demands to the underlying hardware or software layer. An MCAPI program that only makes use of functions offered by the API should be able to run in thread-based systems as well as in process-based systems. Thus, existing MCAPI programs should be easily ported from one particular implementation to another without having to adapt the code. This is facilitated by the specification itself. Only semantics of the function calls are described without any implementation concerns. Although MCAPI primarily focuses on on-chip core-to-core communication, when embedded into large-scale but hierarchy-aware communication environments, it can also be beneficial for distributed systems (Brehmer et al., 2011).

3. Message-passing in the grid

When running large parallel applications with demands for resources that exceed the capacity the local computing site offers, the deployment in a distributed Grid environment may help to satisfy these demands. Advances in wide-area networking technology have fostered this trend towards geographically distributed high-performance parallel computing in the recent years. However, as Grid resources are usually heterogeneous by nature, this is also true for the communication characteristics. Especially the inter-site communication often constitutes a bottleneck in terms of higher latencies and lower bandwidths than compared to the site-internal case. The reason for this is that the inter-site communication is typically handled via wide area transport protocols and respective networks, whereas the internal communication is conducted via fast local-area networks or even via dedicated high-performance interconnections. That in turn means that an efficient utilization of such a hierarchical and heterogeneous infrastructure demands a communication middleware providing support for all these different kinds of networks and transport protocols (Clauss et al., 2008).

3.1 Clusters of clusters

The basic idea of cluster computing is to link multiple independent computers by means of a network in such a way that this system can then be used for efficient parallel processing. Practically, such a cluster of computers constitutes a system that exhibits a NoRMA[6] architecture where each network node possesses its own private memory and where messages must be passed explicitly across the network. However, a major advantage of such systems is that they are much more affordable than dedicated supercomputers because they are usually composed of standard hardware. For this reason, cluster systems built of common *components off the shelf* (COTS) have already become prevalent even in the area of high-performance computing and datacenters. Moreover, this trend has been fostered in the last decades also by the fact that common desktop or server CPUs have already reached the performance class

[6] No Remote Memory Access

of former dedicated but expensive supercomputer CPUs.[7] This idea of linking common computing resources in such a way that the resulting system forms a new machine with an even higher degree of parallelism just leads to the next step (Balkanski et al., 2003): building a *Cluster of Clusters* (CoC). Such systems often arise inherently when, for example in a datacenter, new cluster installations are combined with older ones. This is because datacenters usually upgrade their cluster portfolio periodically by new installations, while not necessarily taking older installations out of service. On the one hand, this approach has the advantage that the users can chose that cluster system out of the portfolio that fits best for their application, for example, in terms of efficiency. On the other hand, when running large parallel applications, older and newer computing resources can be bundled in terms of cluster of clusters in order to maximize the obtainable performance. However, at this point also a potential disadvantage becomes obvious: While a single cluster installation usually constitutes a homogeneous system, a coupled system built from clusters of different generations and/or technologies exhibits a heterogeneous nature which is much more difficult to be handled.

3.1.1 Wide area computing

When looking at coupled clusters *within* a datacenter, the next step to an even higher degree of parallelism suggests itself: linking clusters (or actually cluster of clusters) in *different* datacenters in a wide area manner. However, it also becomes obvious that the interlinking wide area network poses a potential bottleneck with respect to inter-process communication. Therefore, the interlinking infrastructure as well as its interfaces and protocols of such a wide area Grid environment play a key role regarding the overall performance. Obviously, TCP/IP is the standard transport protocol used in the Internet, and due to its general design, it is also often employed in Grid environments. However, it has been proven that TCP has some performance drawbacks especially when being used in high-speed wide area networks with high-bandwidth but high-latency characteristics (Feng & Tinnakornsrisuphap, 2000). Hence, Grid environments, which are commonly based on such dedicated high-performance wide area networks, often require customized transport protocols that take the Grid-specific properties into account (Welzl & Yousaf, 2005). Since a significant loss of performance arises from TCP's window-based congestion control mechanism, several alternative communication protocols like FOBS (Dickens, 2003), SABUL (Gu & Grossman, 2003), UDT[8] (Gu & Grossman, 2007) or PSockets (Sivakumar et al., 2000) try to circumvent this drawback by applying their own transport policies and tactics at application level. That means that they are implemented in form of user-space libraries which in turn have to rely on standard kernel-level protocols like TCP or UDP, again. An advantage of this approach is that there is no need to modify the network stack of the operating systems being used within the Grid. The disadvantage is, of course, the overhead of an additional transport layer on top of an already existing network stack. Nevertheless, a further advantage of such user-space communication libraries is the fact that they can offer a much more comprehensive and customized interface to the Grid applications than the general purpose OS socket API does. However, in the recent years, a third kernel-level transport protocol has become common and available (at least within the

[7] The other way around, this trend can also be recognized when looking at today's multicore CPUs, making most common desktops or even laptops already being a parallel machine.
[8] UDT: a UDP-based Data Transfer Protocol

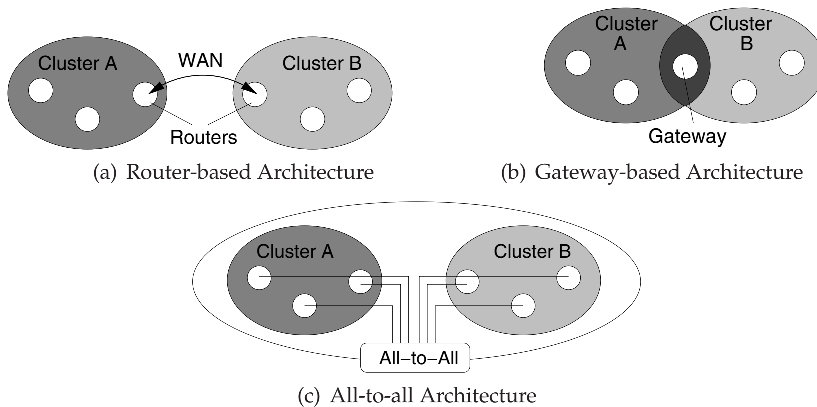
Linux kernel): the Stream Control Transmission Protocol (SCTP) which provides, similar to TCP, a reliable and in-sequence transport service (Stewart et al., 2007). Additionally, SCTP offers several features not present in TCP, as for example the *multihoming* support. This means that an endpoint of a SCTP association (SCTP uses the term *association* to refer to a connection) can be bound to more than one IP address at the same time. Thus, a transparent fail-over between redundant network paths becomes possible. Furthermore, it can be shown that SCTP *may* also perform much better than TCP especially in heterogeneous wide area networks due to a faster congestion control recovery mechanism (Nagamalai et al., 2005). For that reasons, employing SCTP also in Grid environments can be beneficial compared to common TCP (Kamal et al., 2005).

3.1.2 Grid-services and session layers

When looking at this diversity of alternative transport protocols, the question arises which one should be used by the bridging session layer of a message-passing library in Grid computing environments? The answer is that this depends on the properties of the actual environment. In fact, the best solution may differ even within the Grid, due to its heterogeneous nature. Moreover, since Grid resources can be volatile, the optimal protocol to be used may also vary in the course of time, as an initially assigned bandwidth does not necessarily be granted during a whole session for example. For that reason, an efficient session layer for message-passing-based Grid computing should be capable of supporting more than one transport facility at the same time. Nevertheless, such a session layer should also be aware of the inter-site communication overhead by being and acting as resource-friendly as possible in this respect. In order to exploit a Grid environment at its full potential, the underlying network must be a managed resource, just like computing and storage resources usually are. As such, it should be manages by an intelligent and autonomic Grid middleware (Hessler & Welzl, 2007). Such a middleware, like a Grid scheduler, needs to retrieve runtime information about the current capacity and quality of the communication infrastructure, as well as information about the communication patterns and characteristics of the running Grid applications. For that purpose, the possibility of a dynamic interaction between this scheduling middleware and the respective application would be very desirable. Therefore, a session layer for message-passing in Grid environments should also provide *Grid service interfaces* in order to make such information inquirable at runtime. Moreover, a dedicated interface that also allows to access and even to reconfigure the session settings at runtime would help to exploit the Grid's heterogeneous network capabilities at their best. Consequently, a session layer for an actual efficient message-passing should provide such integrated services to the Grid environment.

3.2 Grid-enabled message-passing interfaces

Since MPI is the most important API for implementing parallel programs for large-scale environments, also some MPI libraries have already been extended meeting these demands of distributed and heterogeneous computing. Those libraries are often called *Grid-enabled* because they do not only use plain TCP/IP (which is obviously the lowest common denominator) for all inter-process communication, but are also capable of exploiting fast but local networks and interconnect facilities accommodating the hierarchy of the Grid. Hence, for being able to provide support for the various high-performance cluster networks and



(a) Router-based Architecture

(b) Gateway-based Architecture

(c) All-to-all Architecture

Fig. 3. Different Topology Approaches regarding the Interlinking Network

their specific communication protocols, most of those libraries in turn rely on other high-level communication libraries (like site-*native* MPI libraries), rather than implementing this support inherently. Therefore, Grid-enabled MPI libraries can be understood as a kind of a *meta-layer* bridging the distributed computing sites. For that reason their application area is also referred to as a so-called *meta-computing* environment. The most common Grid-enabled MPI libraries are MPICH-G2 (Karonis et al., 2003), PACX-MPI (Gabriel et al., 1998), GridMPI (Matsuda et al., 2004), StaMPI (Imamura et al., 2000), MPICH/Madeleine (Aumage et al., 2001) and MetaMPICH (Pöppe et al., 2003), which are all proven to run large-scale applications in distributed environments. Although these meta-MPI implementations usually use native MPI support for site-internal communication, as for example provided by a site-local vendor MPI, they must also be based on at least a transport layer being capable of wide area communication for bridging and forwarding messages also to the remote sites. However, since regular transport protocols like TCP/IP are commonly point-to-point-oriented, it is a key task of such a bridging layer to setup all the required inter-site connections and thus acting as a session layer for the wide area communication.

3.2.1 Hardware topologies

When establishing the inter-site connections, a session layer has to take the actual hardware topologies into account in order to enable an efficient message-passing later on. With respect to topologies, three different linking approaches can be differentiated: *router*-based architectures, *gateway*-based architectures and finally a so-called *all-to-all* structures (Bierbaum, Clauss, Pöppe, Lankes & Bemmerl, 2006). In a router-based architecture, only certain cluster nodes have a direct access to the interlinking network. That means that all inter-site messages have to be routed through these special cluster nodes which then forward the messages to the remote clusters (see Figure 3(a)). This routing can either be done *transparently* concerning the MPI library, for example by means of the underlying transport protocol like TCP/IP. Or the MPI library itself has to perform this message routing, for example due to an incompatibility between the cluster internal and the external transport layer. In a gateway-based architectural approach, one or more cluster nodes are part of two or more clusters (see Figure 3(b)). That way, these nodes can act as gateways for

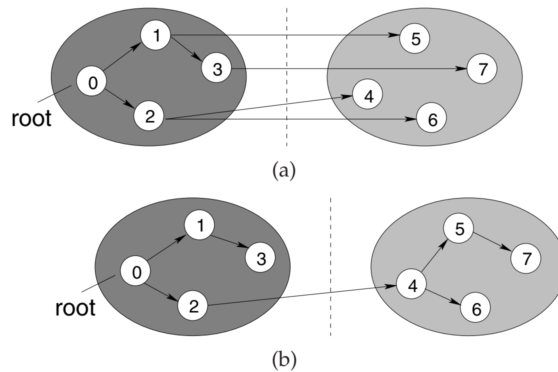


Fig. 4. Bad (a) and Good (a) Implementation of a Broadcast Operation on Coupled Clusters

messages to be transferred from one cluster to another. However, this approach is only suitable for locally coupled clusters, due to a missing wide area link. Finally, when using a fully connected interlinking network, all nodes in one cluster can directly communicate with all nodes in the other clusters. Actually, such a all-to-all topology only needs to be *logically* full connected, for example realized by means of switches (see Figure 3(c)). Not all Grid-enabled MPI libraries provide support for all these topologies. While router-based architectures are supported e.g. by PACX-MPI and MetaMPICH, the gateway approach is only supported by MPICH/Madeleine, whereas all-to-all topologies are supported by almost all above mentioned libraries.

3.2.2 Collective communication patterns

An efficient routing of messages through hierarchical topologies needs to take the underlying hardware structures accordingly into account. Moreover, this is especially true for collective communication operations because bottlenecks and congestion may arise, due to a high number of participating nodes. As already mentioned in Section 2.1.2, there exist a lot of collective communication operations and it is up to the respective communication library to map their patterns onto the hardware topologies in a most optimal way. So a broadcast operation for example may be optimally conducted in a *homogenous* system in terms of a binomial tree. However, in a *hierarchical* system, using just the same pattern would lead to redundant inter-site messages, as shown in Figure 4. Therefore, to avoid unnecessary inter-site communication, the following two rules should be observed: Send no messages with the same content more than once from one to another cluster, and each message must take at most one inter-site hop. The first rule helps to save inter-site bandwidth, whereas the second rule limits the impact of the inter-site latency on the overall communication time. An auxiliary communication library, especially designed for supporting optimized collective operations in hierarchical environments, is the so-called MagPIe library (Kielmann et al., 1999). This library is just an auxiliary library in this respect that is an extension to the well-known MPI implementation MPICH. Figure 5 shows as an example the broadcast pattern implemented by MagPIe for a system of four coupled clusters.

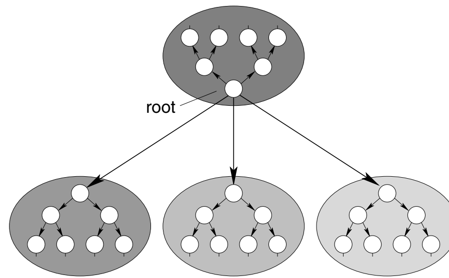


Fig. 5. Communication Pattern implemented by the MagPIe Library for a Broadcast Operation

3.3 The architecture of MetaMPICH

In this section, we detail the architecture of the Grid-enabled MPI implementation developed at the Chair for Operating Systems of the RWTH Aachen University: MetaMPICH which is derived, like many other MPI implementations, from the original MPICH implementation by Argonne National Laboratory (Gropp et al., 1996).

3.3.1 Session configuration

One key strength of MetaMPICH is that it can be configured in a very flexible manner. For that purpose, MetaMPICH relies on a dedicated configuration file that is parsed before each session startup. This configuration file contains information about the communication topologies as well as user-related information about the requested MPI session. The information must be coded in a special description language customized to coupled clusters in Grid environments. Such a configuration file is structured into three parts: a header part with basic information about the session, a part describing the different clusters and a part specifying the overall topology. The header part gives, for example, information about the number of clusters and the number of nodes per cluster and thus the total number of nodes. The second part describes each participating cluster in terms of access information, environment variables, node and router lists as well as information about type and structure of the cluster-internal network. In the third part, the individual links between router nodes in case of a router-based architecture are described in terms of protocols and addresses. The same applies to clusters that are connected in an all-to-all manner: Here, a transport protocol must be specified[9] and additional netmasks may be stated, too. Moreover, MetaMPICH even supports mixed configurations, where some clusters are connected via an all-to-all network, whereas others are simultaneously connected via router nodes. Figure 6 shows an example for such a mixed session configuration.

3.3.2 Internal message handling

Since MetaMPICH is derived from MPICH, it also inherits major parts of its layered architecture, which is shown here in Figure 7. Both supported interlinking approaches of

[9] MetaMPICH provides support for TCP, UDT and SCTP as the interlinking transport layers.

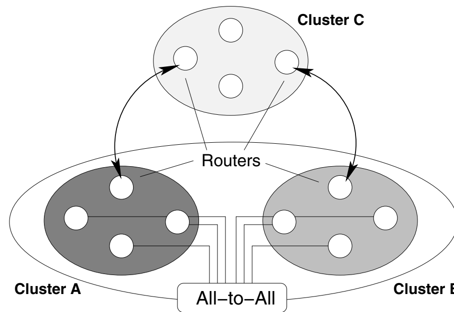


Fig. 6. Example for a Mixed Configuration Supported by MetaMPICH

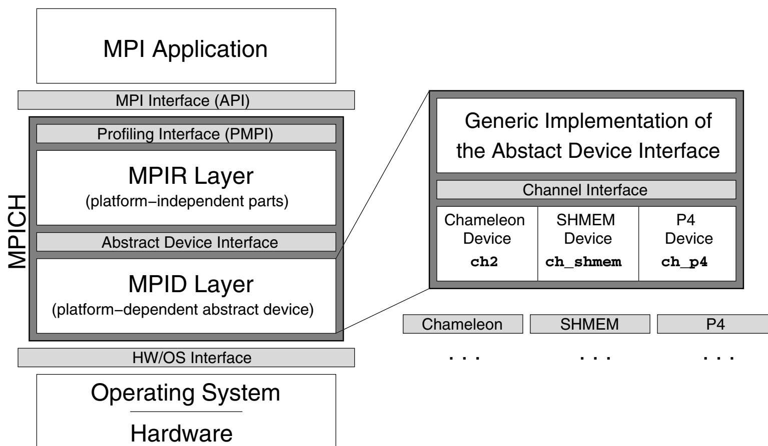


Fig. 7. The Layer Model of MPICH that enables the Multi-Device Support of MetaMPICH

MetaMPICH (the all-to-all approach as well as the router-based approach) in turn, rely on the so-called *multi-device* feature of MPICH. This feature allows the utilization of multiple *abstract communication devices*, which are data structures representing the actual interfaces to lower level communication layers, at the same time. That way, for example, communication via both TCP and shared memory within one MPI session becomes possible. MetaMPICH in turn uses this feature to directly access the interfaces of cluster-internal high-speed interconnects like SCI, Myrinet or InfiniBand via customized devices, while other devices are used to link the clusters via TCP, UDT[10] or SCTP. However, when running a router-based configuration, certain cluster nodes need to act as routers. That means that messages to remote clusters are at first forwarded via the cluster-native interconnect (and thus by means of a customized communication device) to a router node. The router node then sends the message to a corresponding router node at the remote site that finally tunnels the message via that cluster-native interconnect to the actual receiver.

[10] UDT: a UDP-based Data Transfer Protocol, see Section 3.1.1.

3.3.3 The integrated service interface

A further key strength of MetaMPICH is an integrated service interface that can be accessed within the Grid environment via *remote procedure calls* (RPC). Although there exist several approaches for implementing RPC facilities in Grid environments, we have decided to base our implementation on the raw XML-RPC specification (Winer, 1999). Therefore, all service queries have to be handled via XML-coded remote method invocations. Simple services just provide the caller with status information about the current session, as for instance whether a certain connection has already been established, which transport protocol is in use, or how many bytes of payload have already been transfered on this connection. However, also quality-of-service metrics like latency and bandwidth of a connection can be inquired. All these information can then be evaluated by an external entity like a Grid monitoring daemon in order to detect bottlenecks or deadlocks in communication. Besides such query-related services, MetaMPICH also offers RPC interfaces that allow external entities actually to control session-related settings. In doing so, external monitoring or scheduling instances are given the ability to reconfigure an already established session even at runtime. Besides such external control capabilities, also *self-referring* monitoring services are supported by MetaMPICH. These services react automatically on session-internal events, as for instance the detection of a bottleneck or the requirement of a cleanup triggered by a timeout (Clauss et al., 2008).

4. Message-passing on the chip

Since the beginning of the multicore era, parallel processing has become prevalent across-the-board. While previously parallel working processors almost exclusively belonged to the domain of datacenters, today nearly every common desktop PC is already a multiprocessor system. And according to Moore's Law, the number of compute cores per system will continue to grow on both the low end and the high end. Already at this stage, there exist multicore architectures with up to twelve entire cores. However, this high degree of parallelism poses an enormous challenge in particular for the software layers.

4.1 Cluster-on-chip architectures

On a traditional multicore system, a single operating system manages all cores and schedules threads and processes among them with the objective of load balancing. Since there is no distinction between the cores of a chip, this architecture type is also referred to as symmetric multiprocessing (SMP). In such a system, the memory management can be handled nearly similar to a single-core but multi-processing system because the processor hardware already undertakes the crucial task of cache coherency management. However, a further growth of the number of cores per system also implies an increasing chip complexity, especially with respect to the cache coherence protocols; and this in turn may cause a loss of the processors' capability and verifiability. Therefore, a very attractive alternative is to waive the hardware-based cache coherency and to introduce a software-oriented message-passing based architecture instead: a so-called *Cluster-on-Chip* architecture. In turn, this architecture can be classified into two types: The first resembles a homogeneous cluster where all cores are identically, whereas the second exhibits a heterogeneous design. Therefore, the second type is commonly referred to as asymmetric multiprocessing (AMP).

4.2 The Intel SCC Many-core processor

The Intel SCC is a 48-core experimental processor built to study Many-core architectures and how to program them, concerning parallelization capabilities (Intel Corporation, 2010). With this architecture, Many-core systems may be investigated that do not make use of a hardware based cache coherent *shared-memory* programming model but use the *message-passing* paradigm instead. For this purpose, a new memory type is introduced that is located on the chip itself.

4.2.1 Top level view

The 48 cores are arranged in a 6x4 array of 24 tiles with two cores on each of them. They are connected by an on-die 2D mesh that is used for inter-core data transfer but also to access the four on-die memory controllers. These address up to 64 GiB of DDR3 memory altogether which can be used as private but also shared among the cores. The SCC system contains a *Management Console PC* (MCPC) that is used to control the SCC being connected to an FPGA[11] on the SCC board using the PCIe bus. The FPGA controls all off-die data traffic and provides a method to extend the SCC system by new features. Programs may be loaded by the MCPC into the SCC's memory. The same applies to operating systems that shall be booted. The MCPC can be used to read the content of the memory. For this purpose the SCC's memory regions may be mapped into the MCPC's address space. Figure 8 gives a schematic view of the architecture described above. Furthermore the SCC introduces a concept to govern the energy consumption of the chip. It is divided into 7 voltage and 24 frequency domains that can be adjusted independently. Thus, the programmer has the opportunity to influence the software's power consumption. This may be achieved for example by throttling down a core that currently does not have any tasks.

4.2.2 Tile level view

The cores are based on the Intel P54C architecture, an x86 design used for the Intel Pentium I. They contain 16 KiB integrated L1 data and instruction cache each. Apart from the two cores, a tile holds an additional L2 cache of 256 KiB per core to cache the off-die private memory. In addition to that, the so-called *message-passing buffer* (MPB) is provided, a fast on-die shared memory of 16 KiB per tile whereby 8 KiB may logically be assigned to each core. Since the SCC does not provide any cache coherency between the cores, the MPB is intended to realize explicit message-passing among the cores. The so-called *Mesh Interface Unit* (MIU) on each tile handles all memory requests which may be those for message-passing via MPB or accesses to the off-die memory. According to Figure 8, the MIU is the only instance that interacts with the router constituting the connection to the mesh and therefore to the other tiles. For synchronization purposes each tile provides two *Test-and-Set registers*. They are accessible by all cores competitively and guarantee an atomic access. In addition to that, configuration registers are supplied that may be used to modify the operating modes of the on-tile hardware elements.

[11] Field-Programmable Gate Array

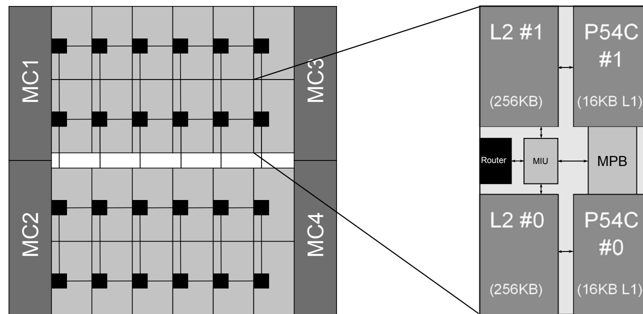


Fig. 8. Block Diagram of the SCC Architecture: a 6x4 mesh of tiles with two cores per tile

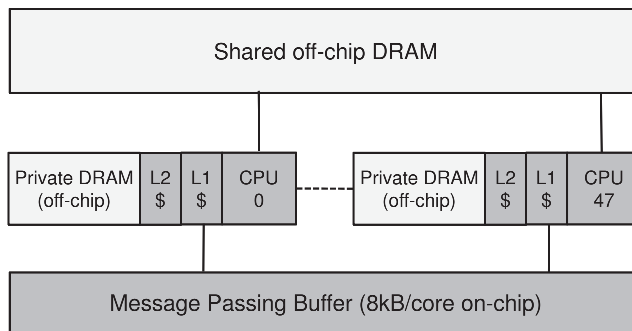


Fig. 9. Logical Software View onto the SCC's Memory System

4.2.3 Software level view

In order to avoid problems due to the missing cache coherency, the off-die memory of the SCC is logically divided into 48 private regions (one per core) plus one global region for all cores. Since for all cores an exclusive access to their private regions is guaranteed, the caches can be enabled for these regions per default. In doing so, each core can then boot its own operating system, usually a Linux kernel (Mattson et al., 2010). Therefore, the SCC is able to run 48 Linux instances simultaneously, actually resembling a cluster on the chip. Moreover, it is also possible to share data between the cores, since all cores have concurrent access to the additional global memory region. However, because of the missing cache coherency, the caches are disabled for this shared region per default. This logical software view onto the memory is illustrated in Figure 9.

4.3 SCC-customized message-passing interfaces

The memory architecture of SCC facilitates various programming models (Clauss, Lankes, Reble & Bemmerl, 2011), where the cores may interact either via the on-die MPBs or via the off-die shared memory. However, due to the lack of cache coherency, message-passing seems to be the most efficient model for the SCC.

4.3.1 RCCE: the SCC-native communication layer

The software environment provided with the SCC, called RCCE (Mattson & van der Wijngaart, 2010), is a lightweight communication API for explicit message-passing on the SCC. For this purpose, basic send and receive routines are provided that support *blocking* point-to-point communication which are based on one-sided primitives (put/get). They access the MPBs and are synchronized by the send and receive routines using flags, introduced with this API. Although used library internal in this case, the flags are also available to the user application. They can be accessed with respective write and read functions and may be used to realize critical sections or synchronization between the cores. Both at the sending and at the receiving side matching destination/source and size parameters have to be passed to the send and receive routine. Otherwise this will lead to unpredictable results. Communication occurs in a *send local, receive remote* scheme. This means that the local MPB, situated at the sending core, is used for the message transfer. The communication API is used in a static SPMD manner. So-called *Units of Execution* (UEs) are introduced that may be associated with a thread or process. Being assigned to one core each, with an ID out of 0 to #cores−1, all UEs form the program. As it is not sure when a UE exactly starts the execution, the programmer may not expect any order within the program. To encounter this, one may use functions to synchronize the UEs, like a barrier for example. Inspired by MPI, there is a number of collective routines (see Section 2.1.2). For example a program, in which each UE makes a part of a calculation, may use a all-reduce to update the current result on all UEs instead of using send/receive routines. A wider range of collectives is provided with the additional library *RCCE_comm* (Chan, 2010) that includes functions like *scatter*, *gather*, etc. With RCCE a fully synchronized communication environment is made available to the programmer. It is possible to gain experience in message-passing in a very simple way. However, if one wants to have further control over the MPB, the so-called *non-gory* interface of RCCE described above is not sufficient anymore. Thus, Intel supplies a *gory* version which offers the programmer more flexibility in using the SCC. Asynchronous message-passing using the one-sided communication functions is now possible, however it has to be considered that cache coherency must not be expected. Therefore the programmer has to make sure by himself that the access to shared memory regions is organized by the software. Although, a very flexible interface for one-sided communication is made available with the gory version, the lack of non-blocking functions concerning two-sided communication forces to look for alternatives.

4.3.2 iRCCE: a non-blocking extension to RCCE

At the Chair of Operating Systems of RWTH Aachen University an extension to the RCCE communications API called *iRCCE* has been developed (Clauss, Lankes, Bemmerl, Galowicz & Pickartz, 2011). It offers a *non-blocking* communication interface for point-to-point communication. Now interleaved communication and computation is possible. Due to the fact that the SCC does not supply asynchronous hardware to perform the message exchange, functions to push the pending requests are provided. To make sure the communication progress has completed, a test or wait function has to be called. To be able to process multiple communication requests, a queuing mechanism is implemented that handles posted requests in a strict FIFO manner. According to the definitions made in Chapter 2.1.1, iRCCE offers a non-blocking but still synchronized communication interface. Since messages may exceed

the available MPB space, it can only be used to transfer data chunk-wise from sender to the receiver. Furthermore, the library itself does not perform overlapped but just interleaved message transfer from sender to receiver. Therefore, the transfer progress has to be actively fostered by the user application. Hence, even with this approach it is not possible to realize real asynchronous message-passing between the cores. While offering a wide range of functions that facilitate non-blocking communication between the cores of the SCC, iRCCE just as RCCE is still a low-level communications API which allows other APIs, like MPI for example, to be built on top of it. That is also reflected in the application programmer interface. It is kept very simple and one who is experienced in message-passing will not have any problems working with it. Due to the simplicity, the functionality is limited, compared to MPI for example.[12] No buffer management has been implemented with the consequence that the send and receive buffers have to be provided by the programmer. Furthermore, there is no mechanism to differ between different message types, like it is possible with tags in MPI.

4.3.3 An MCAPI implementation for the SCC

A *proof of concept* for an MCAPI implementation for the SCC has been developed at the Chair of Operating Systems of RWTH Aachen University, too. The approach that was made is to layer it on top of iRCCE including the features offered by an additional *mailbox system*[13]. This approach does not endeavor to be a highly optimized communication interface. However, it should be sufficient to investigate the usability of the communication interface offered by the MCAPI for future Many-core systems. The MCAPI defines a communication topology that consists of *domains*, *nodes* and *endpoints*. A domain contains an arbitrary number of nodes. The specification does not oblige what to associate with a domain. However, in this SCC-specific implementation, a core is defined as a node and the whole SCC chip as a domain. For now only one domain is supported, however further versions may connect different SCCs and thus offering more than one domain (see also Section 5). An endpoint is a communication interface that may be created at all nodes. Each node internally holds two endpoint lists, one for the local endpoints and one for the remote ones. As the specification requires, the tuple (*domain,node,port*) defining an endpoint is globally unique within the communication topology. The iRCCE communication interface only provides one physical channel for sending purpose (that is the local MPB). In contrast to that the MCAPI allows an arbitrary amount of endpoints to be created at each node. Thus, the approach made by this implementation has to supply a multiplex mechanism as well as a demultiplex mechanism that organizes the message transfer over the channel provided by iRCCE at each node.

5. Message-passing in future Many-core Grids

In the previous sections, we have considered the demands of message-passing in Grid environments as well as in Many-core systems. However, we have done this each apart from the other. Now, in this section we want to discuss how these two worlds can eventually be combined.

[12] MPI actually provides functions for real asynchronous two-sided communication.

[13] This is an asynchronous extension to iRCCE that may be used additionally to the functionality offered by iRCCE itself.

5.1 Bringing two worlds together

When looking at Section 3 and Section 4, it becomes obvious that the two worlds there described will inevitably merge in the near future. The world of chip-embedded Many-core systems will have to be incorporated into the hierarchies of distributed Grid computing. However, the question is how this integration step can be conducted in such a way that both worlds can benefit from this fusion. In doing so, especially the communication interfaces between these two worlds will play a key role regarding their interaction.

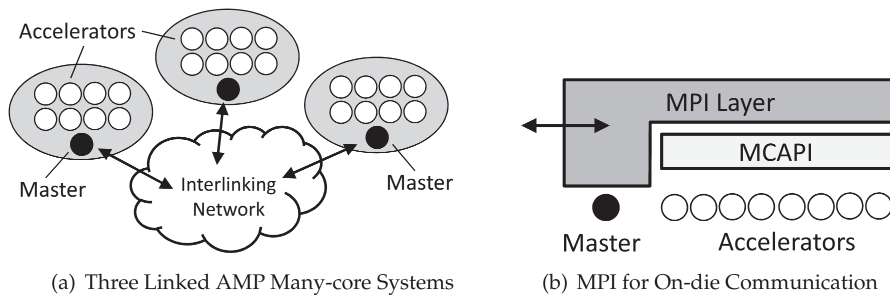
5.1.1 The macrocosmic world

In the world of Grid computing as well as in the domain of High-Performance Computing (HPC), MPI has become the prevalent standard for message-passing. The range of functions defined by the MPI standard is very large and an MPI implementation that aims to provide the sheer magnitude has to implement way above 250 functions.[14] In turn, this implies that such an MPI library tends to become heavyweight in terms of resource consumption, as for example in terms a large memory footprint. However, this is definitely tolerable concerning the HPC domain; at least as long as the resources deployed lead to a high performance as for example in the form of low latencies, high bandwidth and optimal processor utilization. Furthermore, a Grid-enabled MPI implementation must also be capable of dealing with heterogeneous structures and it must be able to provide support for the varying networking technologies and protocols of hierarchical topologies. Moreover, it becomes clear how complex and extensive the implementation of such a library might get if besides the MPI API additional service interfaces for an interaction between the MPI session and the Grid environment come into play. However, when looking at the application level it can be noticed that many MPI applications just make use of less than 10 functions from the large function set offered. On the one hand, this is due to the fact that already a handful of core functions are sufficient in order to write a vast number of useful and efficient MPI programs (Gropp et al., 1999). On the other hand, knowledge of and experience with *all* offered MPI functions are not very common even within the community of MPI users. Therefore, many users rely on a subset of more common functions and implement less common functionalities at application level on their own.[15]

5.1.2 The microcosmic world

Currently, there does not exist one uniform and dominant standard for message-passing in the domain of Many-cores and cluster-on-chip architectures. Although MPI can basically also be employed in such embedded systems, customized communication interfaces, as for example RCCE for the SCC, are predominant in this domain. The reason for this is that MPI is frequently too heavyweight for pure on-chip communication because major parts of an MPI implementation would be superfluous in such systems, as for example the support for unreliable connections, different networking protocols or heterogeneity in general. However, a major drawback of customized libraries with proprietary interfaces is that applications get bound to their specific library and thus become less portable to other platforms. Therefore, a unified and widespread *interface standard* for on-chip communication in multicore and

[14] That is an MPI implementation compliant with the compatibility level MPI-2.
[15] An example for this is the comprehensive set of *collective operations* offered by MPI (see Section 2.1.2).



(a) Three Linked AMP Many-core Systems (b) MPI for On-die Communication

Fig. 10. Many-core Systems According to the AMP Approach

Many-core systems, as the MCAPI promises, would certainly be a step in the right direction. Actually, the MCAPI specification aims to facilitate lightweight implementations that can handle the core-to-core communication in embedded systems with limited resource but less requirements (Brehmer et al., 2011),

5.1.3 The best of both worlds

Several approaches for Many-core architectures follow the asymmetric multiprocessing approach (AMP) where one core is designated to be a master core whereas the other cores act as accelerator cores (see Section 4.1). Examples for this approach are the Cell Broadband Engine or Intel's Knights Corner (Vajda, 2011). One way to combine the macrocosmic world of Grid computing with the microcosmic world of Many-cores in terms of message-passing is using MPI for the off-die communication between multiple master cores and customized communication interfaces, as for example MCAPI, for the on-die communication between the masters and their respective accelerator cores. In this *Multiple-Master Multiple-Worker* approach, the master cores not only act as dispatchers for the local workload but must also act as routers for messages to be sent from one accelerator core to another one on a remote die (see Figure 10(a)). This approach can be arranged with the MPMD paradigm (see Section 2.2.1), where the master cores run a different program (based on MPI plus e.g. MCAPI calls for the communication) than the accelerator cores (running e.g. a pure MCAPI-based parallel code). In addition, the master cores may spawn the processes on the local accelerator cores at runtime and in an iterative manner, in order to assign dedicated subtasks to them. However, one drawback of this approach is the need for processes running on the master cores to communicate via two (or even more) application programming interfaces. A further drawback is the fact, that the master cores are prone to become bottlenecks in terms of inter-site communication. Another approach would be to base all communication calls of the applications upon the MPI API so that all cores become part of one large MPI session. However, also this approach has some drawbacks that, if not taken into account, threaten to limit the overall performance.

5.1.4 The demand for hierarchy-awareness

First of all, when running one large MPI session that covers all participating cores in a Many-core Grid, one has to apply a Grid-enabled MPI library that is not only capable of

routing messages to remote sites but that also must be able to handle the internal on-die communication in a fast, efficient and thus lightweight manner. Hence, in a first instance such an MPI library must be able to differentiate between on-die messages and messages to remote destinations. Moreover, in case of an AMP system, the MPI library should be available in two versions: a lightweight one (possibly just featuring a subset of the comprehensive MPI functionalities) customized to the respective Many-core infrastructure, and a fully equipped one (but possibly quite heavyweight) running on the master cores that also offers, for example, Grid-related service interfaces (see Section 3.1.2). That way, on-die messages can be passed fast and efficient via a *thin* MPI layer that in turn may be based upon another Many-core related communication interface like MCAPI (see Figure 10(b)). At the same time, messages to remote sites can be transferred via appropriate local or wide area transport protocols (see Section 3.1.1) and Grid-related service inquiries can be served. So far, the considered hierarchy is just two-tiered in terms of communication: on-die and off-die. However, with respect to hierarchy-awareness, a further level can be recognized when building local clusters of Many-cores and then interlinking multiple of those via local and/or wide area networks.[16] In that case, the respective MPI library has to distinguish between three (or even four) types of communication: on-die, cluster-internal, (local area) and wide area. Additionally, at each of these levels, hierarchy-related information should be exploited in order to reduce the message traffic and to avoid the congestion of bottlenecks. So, for example, the implementation of *collective operations* has to take the information about such a deep hierarchy into account (see Section 3.2.2).

5.2 SCC-MPICH: A hierarchy-aware MPI library for the SCC

Considering the Intel SCC as a prototype for future Many-core processors, the question is: How can we build clusters of SCCs and deploy them in a Grid environment? In this section, we want to introduce SCC-MPICH that is a customized MPI library for the SCC developed at the Chair for Operating Systems of RWTH Aachen University.[17] Since SCC-MPICH is derived from the original MPICH library, just the same as MetaMPICH, it is possible to plug the SCC-related part of SCC-MPICH[18] into MetaMPICH. That way, the building of a prototype for evaluating the opportunities, the potentials as well as the limitations of future Many-core Grids should become possible.

5.2.1 An SCC-customized abstract communication device

Although the semantics of RCCE's communication functions are obviously derived from the MPI standard, the RCCE API is far from implementing all MPI-related features (see Section 4.3). And even though iRCCE extends the range of supported functions (and thus the provided communication semantics), a lot of users are familiar with MPI and hence want to use its well-known functions also on the SCC. A very simple way to use MPI functions on the SCC is just to port an existing TCP/IP-capable MPI library to this new target platform. However, since the TCP/IP driver of the Linux operating system image

[16] The resulting architecture may be called a *Cluster-of-Many-core-Clusters*, or just a true *Many-core Grid*.
[17] At this point we want to mention that by now there also exists another MPI implementation for the SCC: RCKMPI by Intel (Urena et al., 2011).
[18] This is actually an implementation of the non-generic part of an *abstract communication device* customized to the SCC (see Section 3.3).

for the SCC does not utilize the fast on-die message-passing buffers (MPBs), the achievable communication performance of such a ported TCP/IP-based MPI library lags far behind the MPB-based communication performance of RCCE and iRCCE. For this reason, we have implemented SCC-MPICH as an SCC-optimized MPI library which in turn is based upon the iRCCE extensions of the original RCCE communication library.[19] In doing so, we have added to the original MPICH a new abstract communication device (refer back to Figure 7 in Section 3) that utilizes the fast on-die MPBs of the SCC as well as the off-die shared-memory for the core-to-core communication. In turn, this new SCC-related communication device provides four different communication protocols: *Short*, *Eager*, *Rendezvous* and a second Eager protocol called *ShmEager* (Clauss, Lankes & Bemmerl, 2011). The Short protocol is optimized in order to obtain low communication latencies. It is used for exchanging message headers as well as header-embedded short payload messages via the MPBs. Bigger messages must be sent either via one of the two Eager protocols or via the Rendezvous protocol. The main difference between Eager and Rendezvous mode is that Eager messages must be accepted on the receiver side even if the corresponding receive requests are not yet posted by the application. Therefore, a message sent via Eager mode can implicate an additional overhead by copying the message temporarily into an intermediate buffer. However, when using the ShmEager protocol, the off-die shared-memory is used to pass the messages between the cores. That means that this protocol does not require the receiver to copy unexpected messages into additional *private* intermediate buffers unless there is no longer enough *shared* off-die memory. The decision which of these protocols is to be used depends on the message length as well as on the ratio of expected to unexpected messages (Gropp & Lusk, 1996).

5.2.2 Integration into MetaMPICH

By integrating the SCC-related communication device of SCC-MPICH into MetaMPICH, multiple SCCs can now be linked together according to the all-to-all approach as well as to the router-based approach (see Section 3.3.1). However, at this point the question arises how the SCC's frontend, that is the so-called *Management Console PC* (MCPC), has to be considered regarding the session configuration (see Section 4.2.1). In fact, the cores of the MCPC[20] and the 48 cores of the SCC are connected via an FPGA in such a way that they are contained within a private TCP address space. That means that all data traffic between the SCC cores and the outside world has to be routed across the MCPC. In this respect, an SCC system can be considered as an AMP system where the CPUs of the MCPC represent the master cores while the SCC cores can be perceived as 48 accelerator cores. In turn, a session of two (or more) coupled SCC systems must be based on a three-tiered mixed configuration: 1. On-die communication via the customized communication device of SCC-MPICH; 2. System-local communication within the private TCP domain; 3. Router-based communication via TCP, UDT or SCTP as local or wide area transport protocols to remote systems. Figure 11 shows an example configuration of two linked SCC systems.

[19] In fact, the development of iRCCE was driven by the demand for a non-blocking communication substrate for SCC-MPICH because one cannot layer non-blocking semantics (as supported by MPI) upon just blocking communication functions (as provided by RCCE), see also Section 2.1.1.

[20] Actually, the MCPC is just a common server equipped with common multicore CPUs.

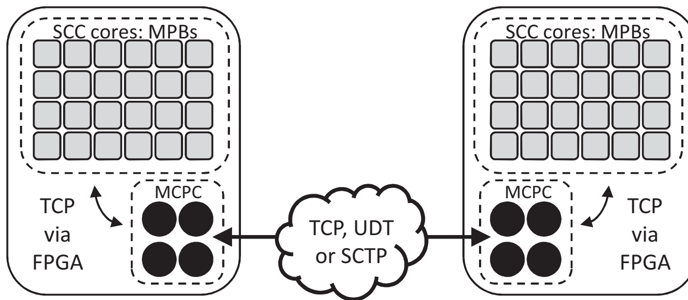


Fig. 11. Two Linked SCC Systems: Each Consisting of 4 MCPC Cores and 48 SCC Cores

5.2.3 Future prospects

The next step would be to link more than two SCC systems to a real cluster of SCCs and to deploy them in a Grid environment. However, one major problem is that currently the SCC cores are not able to communicate directly with the outside world. That means that all messages must be routed across the MCPC nodes which in turn may become bottlenecks in terms of communication. Although MetaMPICH supports configurations with more than one router node per site and allows for a static balancing of the router load, a second major problem is the link between the MCPC and the SCC cores: Because due to this link, each message to a remote site has to take two additional process-to-process hops. Therefore, a smarter solution might be to enable the SCC cores to access the interlinking network in a direct manner. However, this in turn implies that the processes running on the SCC cores have then to handle also the wide area communication. So, for example, when performing collective communication operations, a router process running on the MCPC can be used to collect and consolidate data locally before forwarding messages to remote sites, thereby relieving the SCC cores from this task. Without a router process, the SCC cores have to organize the local part of the communication pattern on their own. Hence, a much smarter approach might be hybrid: allow for direct point-to-point communication between remote SCC cores and use additional processes running on the MCPCs to perform collective operations and/or to handle Grid-related service inquiries. All the more so because such hierarchy-related interaction with other Grid applications will play an important part towards a successful merge of both worlds. Although the runtime system of MetaMPICH has already been extended by the ability to interact with a *meta-scheduling service* in UNICORE-based Grids (Bierbaum, Clauss, Eickermann, Kirtchakova, Krechel, Springstubbe, Wäldrich & Ziegler, 2006), the integration into other existing or future Grid middleware needs to be considered.

6. Conclusion

It is quite obvious that the world of chip-embedded Many-core systems on the one hand and the world of distributed Grid computing on the other hand will merge in the near future. With the Intel SCC as a prototype for future Many-core processors, we have even today the opportunity to investigate the requirements of such Many-core equipped Grid environments. In this chapter, we have especially focused on the challenges of message-passing in this upcoming new computing era. In doing so, we have presented the two MPI libraries

MetaMPICH and SCC-MPICH and we have shown how both can be combined in order to build a Grid-enabled message-passing library for coupled Many-core systems. By means of this prototype implementation, the evaluation of opportunities, potentials as well as limitations of future Many-core Grids becomes possible. We have especially pointed out the demand for hierarchy-awareness to be included into the communication middleware in order to reduce the message traffic and to avoid the congestion of bottlenecks. As a result, this approach requires knowledge about the hardware structures and thus related information and service interfaces. Moreover, even a likewise hierarchical algorithm design for the parallel applications will probably become necessary. However, in order to keep this chapter focused, a lot of other very interesting and important aspects could not be covered here. So, for examples, it is still quite unclear how such hardware-related information can be handled and passed in a standardized manner. Although the MPI Forum is currently fostering the upcoming MPI-3 standard, it looks quite unlikely that already the next standard will give answers to these questions.

7. References

Aumage, O., Mercier, G. & Namyst, R. (2001). MPICH/Madeleine: A True Multi-Protocol MPI for High Performance Networks, *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, IEEE CS Press, San Francisco, CA, USA.

Balkanski, D., Trams, M. & Rehm, W. (2003). Heterogeneous Computing With MPICH/Madeleine and PACX MPI: a Critical Comparison, *Chemnitzer Informatik-Berichte* CSR-03-04: 1–20.

Bierbaum, B., Clauss, C., Eickermann, T., Kirtchakova, L., Krechel, A., Springstubbe, S., Wäldrich, O. & Ziegler, W. (2006). Reliable Orchestration of distributed MPI-Applications in a UNICORE-based Grid with MetaMPICH and MetaScheduling, *Proceedings of the 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'06)*, Vol. 4192 of *Lecture Notes in Computer Science*, Springer-Verlag, Bonn, Germany.

Bierbaum, B., Clauss, C., Pöppe, M., Lankes, S. & Bemmerl, T. (2006). The new Multidevice Architecture of MetaMPICH in the Context of other Approaches to Grid-enabled MPI, *Proceedings of the 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'06)*, Vol. 4192 of *Lecture Notes in Computer Science*, Springer-Verlag, Bonn, Germany.

Brehmer, S., Levy, M. & Moyer, B. (2011). Using MCAPI to Lighten an MPI Load, EE Times Design Article (online).

Butler, R. & Lusk, E. (1994). Monitors, Messages and Clusters: The P4 Parallel Programming System, *Parallel Computing* 20(4): 547–564.

Chan, E. (2010). RCCE_comm: A Collective Communication Library for the Intel Single-Chip Cloud Computer, *Technical report*, Intel Corporation.

Clauss, C., Lankes, S. & Bemmerl, T. (2008). Design and Implementation of a Service-integrated Session Layer for Efficient Message Passing in Grid Computing Environments, *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC'08)*, IEEE CS Press, Krakow, Poland.

Clauss, C., Lankes, S. & Bemmerl, T. (2011). Performance Tuning of SCC-MPICH by means of the Proposed MPI-3.0 Tool Interface, *Proceedings of the 18th European MPI Users Group Meeting (EuroMPI 2011)*, Vol. 6960, Springer, Santorini, Greece.

Clauss, C., Lankes, S., Bemmerl, T., Galowicz, J. & Pickartz, S. (2011). iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer, *Technical report*, Chair for Operating Systems, RWTH Aachen University. Users' Guide and API Manual.

Clauss, C., Lankes, S., Reble, P. & Bemmerl, T. (2011). Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor, *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011), Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, Istanbul, Turkey.

Dickens, P. M. (2003). FOBS: A Lightweight Communication Protocol for Grid Computing, *Processing of the 9th International Euro-Par Conference (Euro-Par'03)*, Austria.

Dongarra, J., Geist, A., Mancheck, R. & Sunderam, V. (1993). Integrated PVM Framework Supports Heterogeneous Network Computing, *Computers in Physics* 7(2): 166–175.

Feng, W. & Tinnakornsrisuphap, P. (2000). The Failure of TCP in High-Performance Computational Grids, *Proceedings of the Supercomputing Conference (SC2000)*, ACM Press and IEEE CS Press, Dallas, TX, USA.

Gabriel, E., Fagg, G., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R., Daniel, D., Graham, R. & Woodall, T. (2004). Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation, *Proceedings of the 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04)*, Vol. 3241 of *Lecture Notes in Computer Science*, Springer-Verlag, Budapest, Hungary.

Gabriel, E., Resch, M., Beisel, T. & Keller, R. (1998). Distributed Computing in a Heterogeneous Computing Environment, *Proceedings of the 5th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'97)*, Vol. 1497 of *Lecture Notes in Computer Science*, Springer-Verlag, Liverpool, UK.

Geist, A. (1998). Harness: The Next Generation Beyond PVM, *Proceedings of the 5th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'98)*, Vol. 1497 of *Lecture Notes in Computer Science*, Springer-Verlag, Liverpool, UK.

Gropp, W. (2002). MPICH2: A New Start for MPI Implementations, *Proceedings of the 9th European PVM/MPI Users Group Meeting (EuroPVM/MPI'02)*, Vol. 2474 of *Lecture Notes in Computer Science*, Springer-Verlag, Linz, Austria.

Gropp, W. & Lusk, E. (1996). MPICH Working Note: The Implementation of the Second-Generation MPICH ADI, *Technical Report* , Mathematics and Computer Science Division, Argonne National Laboratory (ANL).

Gropp, W., Lusk, E., Doss, N. & Skjellum, A. (1996). A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, *Parallel Computing* 22(6): 789–828.

Gropp, W., Lusk, E. & Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, Scientific and engineering computation, second edn, The MIT Press.

Gropp, W. & Smith, B. (1993). Chameleon Parallel Programming Tools – User's Manual, *Technical Report ANL-93/23*, Argonne National Laboratory.

Gu, Y. & Grossman, R. (2003). SABUL: A Transport Protocol for Grid Computing, *Journal of Grid Computing* 1(4): 377–386.

Gu, Y. & Grossman, R. (2007). UDT: UDP-based Data Transfer for High-Speed Wide Area Networks, *Computer Networks* 51(7): 1777–1799.

Hessler, S. & Welzl, M. (2007). Seamless Transport Service Selection by Deploying a Middleware, *Computer Communications* 30(3): 630–637.

Imamura, T., Tsujita, Y., Koide, H. & Takemiya, H. (2000). An Architecture of STAMPI: MPI Library on a Cluster of Parallel Computers, *Proceedings of the 7th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'00)*, Vol. 1908 of *Lecture Notes in Computer Science*, Springer-Verlag, Balatonfüred, Hungary.

Intel Corporation (2010). SCC External Architecture Specification (EAS), *Technical report*, Intel Corporation.

Kamal, H., Penoff, B. & Wagner, A. (2005). SCTP versus TCP for MPI, *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'05)*, ACM Press and IEEE CS Press, Seattle, WA, USA.

Karonis, N., Toonen, B. & Foster, I. (2003). MPICH-G2: A Grid-enabled implementation of the Message Passing Interface, *Journal of Parallel and Distributed Computing* 63(5): 551–563.

Kielmann, T., Hofmann, R., Bal, H., Plaat, A. & Bhoedjang, R. (1999). MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems, *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM Press, Atlanta, GA, USA.

Matsuda, M., Ishikawa, Y., Kaneo, Y. & Edamoto, M. (2004). Overview of the GridMPI Version 1.0 (*in Japanese*), *Proceedings of Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP'04)*.

Mattson, T. & van der Wijngaart, R. (2010). RCCE: a Small Library for Many-Core Communication, *Technical report*, Intel Corporation. Users' Guide and API Manual.

Mattson, T., van der Wijngaart, R., Riepen, M., Lehnig, T., Brett, P., Haas, W., Kennedy, P., Howard, J., Vangal, S., Borkar, N., Ruhl, G. & Dighe, S. (2010). The 48-core SCC Processor: The Programmer's View, *Proceedings of the 2010 ACM/IEEE Conference on Supercomputing (SC10)*, New Orleans, LA, USA.

Message Passing Interface Forum (2009). *MPI: A Message-Passing Interface Standard – Version 2.2*, High-Performance Computing Center Stuttgart (HLRS).

Multicore Association (2011). *Multicore Communications API (MCAPI) Specification*, The Multicore Association.

Nagamalai, D., Lee, S.-H., Lee, W. G. & Lee, J.-K. (2005). SCTP over High Speed Wide Area Networks, *Proceedings of the 4th International Conference on Networking (ICN'05)*, Vol. 3420, Springer-Verlag, Reunion, France.

Pierce, P. (1988). The NX/2 Operating System, *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, ACM Press, Pasadena, CA, USA.

Pöppe, M., Schuch, S. & Bemmerl, T. (2003). A Message Passing Interface Library for Inhomogeneous Coupled Clusters, *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, IEEE CS Press, Nice, France.

Sivakumar, H., Bailey, S. & Grossman, R. L. (2000). PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks, *Proceedings of the High Performance Networking and Computing Conference (SC2000)*, ACM Press and IEEE CS Press, Dallas, TX, USA.

Skjellum, A. & Leung, A. (1990). Zipcode: a Portable Multicomputer Communication Library atop the Reactive Kernel, *Proceedings of the 5th Distributed Memory Concurrent Computing Conference*, IEEE CS Press, Charleston, SC, USA.

Stevens, R., Fenner, B. & Rudoff, A. (2006). *UNIX Network Programming – The Socket Networking API*, Vol. 1, third edn, Addison-Wesley.

Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L. & Paxson, V. (2007). Stream Control Transmission Protocol, *Request for Comments (RFC) 4960*, Network Working Group.

Tanenbaum, A. (2008). *Modern Operating Systems*, third edn, Prentice-Hall.

Urena, I. A. C., Riepen, M. & Konow, M. (2011). RCKMPI - Lightweight MPI Implementation for Intel's Single-Chip Cloud Computer (SCC), *Proceedings of the 18th European MPI Users Group Meeting (EuroMPI 2011)*, Vol. 6960, Springer, Santorini, Greece.

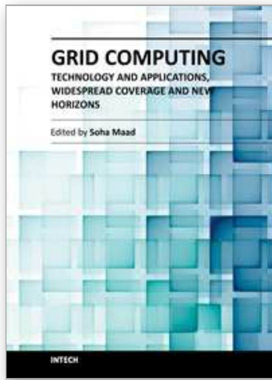
Vajda, A. (2011). *Programming Many-Core Chips*, Springer.

Welzl, M. & Yousaf, M. (2005). Grid-Specific Network Enhancements: A Research Gap?, *International Workshop on Autonomic Grid Networking and Management (AGNM'05)*, IEEE CS Press, Spain.

Wilkinson, B. & Allen, M. (2005). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, second edn, Prentice-Hall.

Winer, D. (1999). *XML-RPC Specification*, UserLand, Inc.

Winett, J. M. (1971). The Definition of a Socket, *Request for Comments (RFC) 147*, Massachusetts Institute of Technology, USA.



Grid Computing - Technology and Applications, Widespread Coverage and New Horizons
Edited by Dr. Soha Maad

ISBN 978-953-51-0604-3
Hard cover, 354 pages
Publisher InTech
Published online 16, May, 2012
Published in print edition May, 2012

Grid research, rooted in distributed and high performance computing, started in mid-to-late 1990s. Soon afterwards, national and international research and development authorities realized the importance of the Grid and gave it a primary position on their research and development agenda. The Grid evolved from tackling data and compute-intensive problems, to addressing global-scale scientific projects, connecting businesses across the supply chain, and becoming a World Wide Grid integrated in our daily routine activities. This book tells the story of great potential, continued strength, and widespread international penetration of Grid computing. It overviews latest advances in the field and traces the evolution of selected Grid applications. The book highlights the international widespread coverage and unveils the future potential of the Grid.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Carsten Clauss, Simon Pickartz, Stefan Lankes and Thomas Bemmerl (2012). Hierarchy-Aware Message-Passing in the Upcoming Many-Core Era, Grid Computing - Technology and Applications, Widespread Coverage and New Horizons, Dr. Soha Maad (Ed.), ISBN: 978-953-51-0604-3, InTech, Available from: http://www.intechopen.com/books/grid-computing-technology-and-applications-widespread-coverage-and-new-horizons/hierarchy-aware-message-passing-in-the-upcoming-many-core-era

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2012 The Author(s). Licensee IntechOpen. This is an open access article distributed under the terms of the Creative Commons Attribution 3.0 License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.