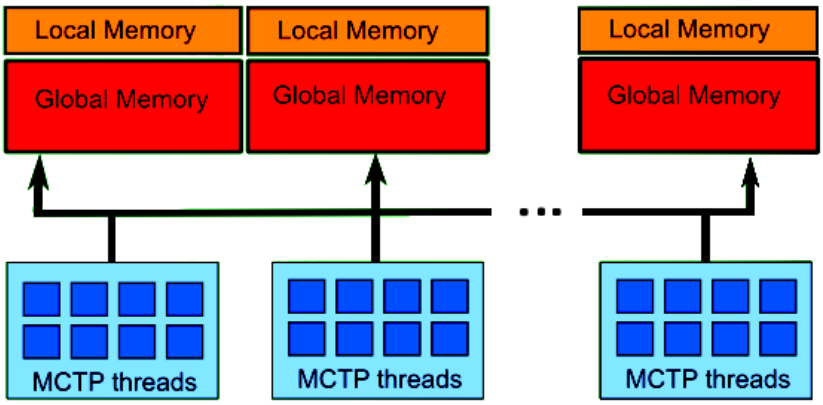
Fraunhofer
ITWM

THE BUILDING BLOCKS FOR HPC:

GPI and MCTP

EFFICIENT SCALABLE MULTICORE



Fraunhofer Institut für Techno-
und Wirtschaftsmathematik ITWM
Fraunhofer-Platz 1
67663 Kaiserslautern
Germany

Dr. Franz-Josef Pfreundt
Phone: +49(0)631 31600 4459
Email: pfreundt@itwm.fraunhofer.de

http://www.itwm.fraunhofer.de

Introduction

Industrial and scientific applications are processing more and more data requiring increasingly larger machines. Modern computer architectures are now multi-core: processing systems composed of independent cores.

This implies a paradigm shift on the development of software where developers must exploit the inherent parallelism of these architectures to be able to scale their applications.

In the high-performance computing area, large computer systems are built with these modern architectures together with high-performance interconnects such as Infiniband. In this petaflop era the peak performance and communication latency are key points to take advantage of such large computer systems.

It is essential to feed all the cores with a contiguous data stream, just at that time when the computation actually needs them and to efficiently share the work among the cores. Most suitable, data transfers should not charge the CPU. New solutions are needed to overcome the scalability limitations with current approaches such as MPI.

We face these problems with two *easy to use*, *robust* and *scalable* APIs staying consistently in the same programming model. On the cluster level, our Global address space Programming Interface (GPI) provides a low latency communication library which works at full Infiniband wirespeed. On the node level, our Multi Core Threading Package (MCTP) library supplements GPI with an optimized threading library which has *full hardware awareness*.

GPI and MCTP have already proven their applicability and their outstanding performance capabilities in industrial applications provided by the Fraunhofer ITWM such as the generalized radon transform method in seismic imaging (GRT) or the seismic workflow and visualization suite PSPRO. The GRT is highly irregular and the only production ready version in the world.

Industrial customers are overwhelmed by the GPI and MCTP features and performance. For example, Åre Osen from Statoil AS says

> "Our parallel seismic imaging code is based on GPI, since it requires random access to distributed data. We are really satisfied with the performance, the multicore scalability and the robustness of GPI."

Christian Simmendinger from T-Systems means

> "For the scalable parallel version of DLR's TAU code we used GPI since MPI did not deliver the required scalability on our large multicore cluster."

GPI

The Global address space Programming Interface (GPI) is a low latency communication library and a runtime system for scalable real-time parallel applications running on cluster systems. It provides a partitioned global address space (PGAS) to the application which in turn has direct and full access to a remote data location. The whole functionality includes communication primitives, environment runtime checks, synchronization primitives such as fast barriers or global atomic counters, all which allow the development of parallel programs for large scale.

Focused on performance, by leveraging on the network interconnect hardware (wire-speed), it minimizes the communication overhead with overlap of communication and computation (asynchronous communications). The GPI provides a simple, reliable runtime system to handle large datasets, dynamic and irregular applications that are I/O and compute intensive. It has reached production quality for x86 and IBM Cell/B.E architectures.

DMA communication

GPI provides one-sided asynchronous read and write operations from and into the partioned global address space from arbitrary locations. The important point is that those operations are non-blocking, allowing the program to continue its execution and hence take better advantage of CPU cycles. If the application needs to make sure the data was transfered, it just calls a wait operation.

This way, it is possibile to implement a perfect overlap of communication with computation, one of the basic requirements for scalable applications.

One aspect related to both communication operations is the possibility to use different queues for handling the requests. The user provides which queue he would like to use to handle the request. These queues allow more scalability and can be used as channels for different types of requests where similar types of requests are queued and then get synchronized together but independently from the other ones. (separation of concern)

The GPI also allows a more Send/Receive style of programming, where nodes exchange messages or commands. In this case, a non-blocking send has to match a blocking receive. A special variant of this Send/Receive style is the passive communication. Here, the receive is a blocking call that burns no CPU cycles at all and is woken up directly by the Infiniband network layer. This passive communication allows for fair distributed updates of globally shared parts of data and is beyond the capabilities of other solutions such as MPI.

Despite the main focus being the use of the Infiniband fabric, it is also possible to use communication with RDMA enabled 10 GE. This provides a second and alternative way of communication.

Collectives

GPI provides an efficient method to synchronize all nodes in a cluster. The GPI barrier takes less than 35 μs on 128 nodes using Infiniband multicast. Even with older hardware that doesn't support reliable multicast, the GPI barrier is typically faster than other barriers and scales much better.

In addition to the barrier, GPI provides an allreduce operation for up to 255 elements of some standard types on each node. Allreduce is one of the most used collective operation in HPC application. The send buffer and the receive buffer do not have to be in the partioned global address space regions in this case. Heap or stack memory locations can also be used. Among the supported reduce operations, there is a minimum, a maximum and a sum function sufficient for all practical needs.

Global atomic counters

In addition to the pure communication routines, GPI provides global atomic counters, i. e. integral types that can be manipulated through atomic functions. These atomic functions are guaranteed to execute from start to end without fear of preemption causing corruption. There are two basic operations on atomic counters: fetch and add and fetch, compare and swap. The counters can be used as global shared variables used to synchronize nodes or events.

On top of global atomic counters it is quite straight forward to implement different load balancing mechanisms as well as more complex synchronization primitives.

Execution model

A GPI daemon is running on all nodes, waiting for requests to run a program binary on its node. The node where the user starts the binary becomes the master node. When the user executes the binary, a given configuration file on the master node is read where the nodes that should be started as worker nodes are listed. The daemon on the master node will then request to the other daemons (running on the other nodes) to start the application. The daemon loads the program only after several checks related to the user and infrastructure. For the user it checks for permissions and limits such as

memory size, file size or file permissions. As for infrastructure, the daemon makes sure everything is in place in order to load the program: checks for Infiniband hardware and whether or not all the nodes on the configuration file are up and have a running daemon. The application can also query the daemon and make sure the environment is sane.

The latest version of the GPI daemon interacts with the PBS management system to discover the nodes assigned to a user, removing the otherwise necessary step of creating or modifying the needed machine file.

The initial process creation by the daemon is static. One process per node is created. To allow multithreading and to take advantage of the new multi-core architectures, the user should use MCTP. However, other typical threading APIs like pthreads or even OpenMP can be used.

Some processes might need to execute different code. To allow this flexibility, the GPI uses a similar concept to MPI: ranks. Each node has a rank which can be used to select portions of code to be executed by a single or some ranks.

MCTP

The MCTP supplements GPI on the node level. Software developers can no longer rely on an increasing clock speed. Multithreaded design is neccessary in order to achieve scalable applications.

The MCTP is a threading package to make multi-threading programming slightly more programmer friendly. It abstracts the native threads of the platform and provides complete state-of-the-art functionality to work with threads, threadpools, critical section and related topics such as synchronization or high frequency timer.

One of the key features of MCTP, clearly distinguishing it from other threading libraries, is its full hardware awareness including NUMA layout or cache to core mapping.

Thread pools

The MCTP has the functionality to create threadpools, i. e. to introduce a group of threads. Those pools are a priori independent from each other. The threads in a pool can be synchronized with each other, and a pool can be synchronized with another pool.

In addition to that, it is possible to start, stop, suspend and resume the threads in a pool or complete pools. The suspension of a given thread is immediate and independent from any other synchronization primitives as

mutexes, semaphores, etc., i. e. one can also put threads currently holding mutexes to sleep.

Hardware awareness

The MCTP library provides full access to the hardware information, i. e. the NUMA layout, the cache to core mapping, complete core and socket information, the number of cores (physical as well as logical), supported SSE level. It can get and set the affinity mask to map threads to particular cores.

Not only it is possible to *get* this information, the MCTP is the only threading library, that is able to *use* it. Having these tools at hand, one has full runtime control over the execution resources.

Synchronization

The MCTP provides several methods to synchronize the threads in a pool (barrier), namely active, relaxed and passive synchronization. Any of the threads in a group can only continue after all threads of the pool have reached the barrier.

The active barrier is implemented using busy waiting, i. e. the barrier is fast but charges the CPU.

In contrast to that, the passive barrier is implemented differently. It also uses busy waiting, but each waiting thread issues PAUSE CPU instructions, which cause that CPU core to do nothing for several cycles, and by that not occupy the memory bus by constantly polling the shared variable.

The relaxed barrier is implemented using mutexes (futexes), i. e. without busy waiting and is useful when the arriving times differ by more than say 10 ms.

The brand new NUMA barrier takes care of the NUMA layout and first synchronizes the threads on a single socket and then synchronizes the sockets. This ensures scalability also on actual machines with 4 or more sockets and 8 or more cores per socket. Again, this is far beyond the capabilities of any other existing thread package.

Benchmark results

To visualize the outstanding performance capabilities of GPI together with MCTP, benchmarks on the Fraunhofer ITWM cluster (unless stated otherwise) have been performed. The ITWM cluster consists of 260 compute nodes, each hosting a Dual Intel Xeon 5148LV ("Woodcrest") CPU with 4

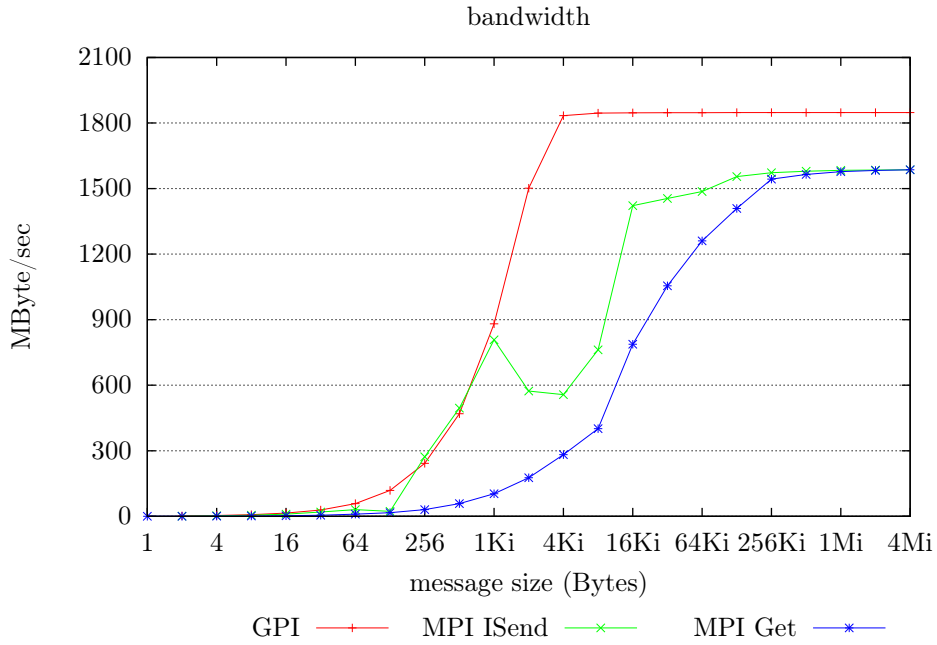


Figure 1: Data transfer bandwidth for two node communication as a function of message size for GPI and MPI. The tests were conducted on two machines with Intel QuadCore Xeon 5460 (2 Cores), 3.16 GHz and a ConnectX Infiniband Host Channel Adapter. We compare the results obtained with the MVAPICH2 1.2p1.

cores running at 2.33 GHz and having 8 GiB RAM. The nodes are connected by a Mellanox MHGS18 Infiniband interface (single port double data rate).

Beside the usual micro benchmarks such as latency, bandwidth etc., also real world application benchmarks are presented. Obtaining performance numbers solely with micro benchmarks might fail to reflect some aspects that are only present in real application.

GPI micro benchmarks

Fig. 1 shows the data transfer bandwidth between two nodes as a function of message size for GPI, MPI ISend and MPI Get (note the logarithmic scale on the abscissa). GPI reaches the full hardware bandwidth of 1.8 GiB/s at a message size of 4 KiB already. However, MPI ISend as well as MPI Get only reach a plateau at 1.6 GiB/s. This is less than the full hardware bandwidth and is reached at a message size of 1 MiB, which is quite large. GPI shows a considerably higher bandwidth than MPI over the entire range of

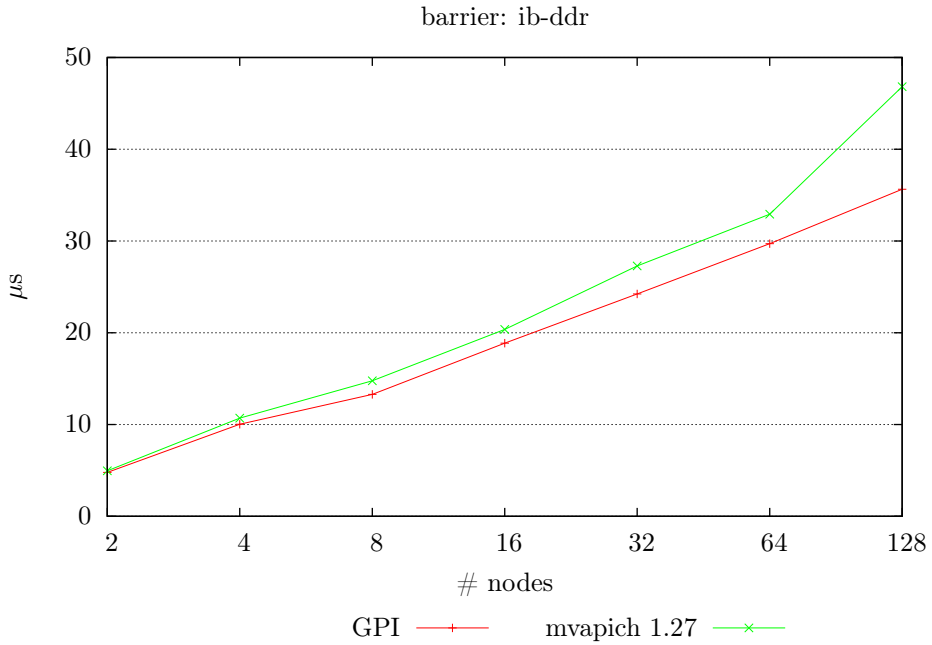


Figure 2: Node synchronisation latency as a function of the number of nodes for GPI and MPI.

message sizes. MPI ISend also shows different behavior for different regimes of message sizes. This is due to a change in the implementation of MPI ISend which depends on message size.

Fig. 2 shows the barrier latency as a function of the number of involved nodes. The GPI barrier is always faster than the MPI barrier. Furthermore, the GPI barrier shows a perfect scaling behavior whereas the MPI barrier deviates from scaling for more than 64 nodes.

As for the barrier, GPI allreduce is not only faster than MPI allreduce, it scales much better at the same time. (Fig. 3) The GPI allreduce scales perfect whereas the mpi deviates from scaling for more than 64 cores. Note that MPI needs 70% more time than GPI on 512 cores.

MCTP micro benchmarks

The MCTP micro benchmarks are performed on an Intel Xeon X7560 CPU running at 2.27 GHz in order to have a considerable number of CPU cores.

Fig. 4 shows the synchronisation time between several threads (cores) as a function of the number of threads (cores) for different threading libraries (note the logarithmic scale on the abscissa and on the ordinate). Each thread

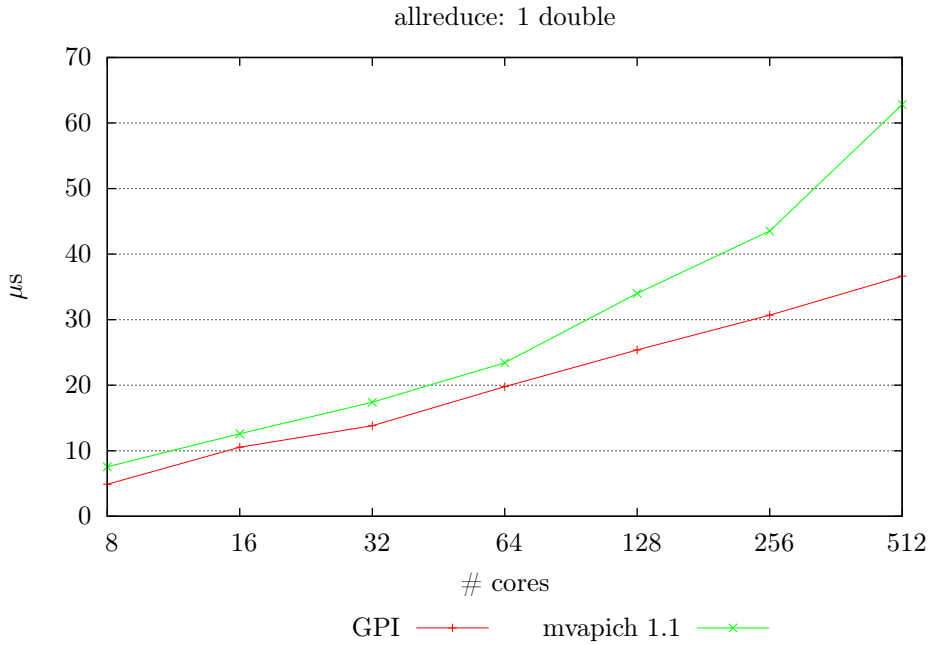


Figure 3: Allreduce latency for 1 double as a function of the number of cores involved, both for GPI and MPI.

runs on a separate core.

Among the analyzed threading libraries, there is an OpenMP implementation provided by the Intel 11.1 compiler, the Posix threads library and the MCTP 1.52 library. In addition to these threading libraries, we also compare to MVAPICH-1.27 MPI started on a single node. For a fair comparison, a separate MPI process is started on each core.

MCTP comes in two flavors: First, there is the standard variant and second, there is the variant using the NUMA extension.

Looking at Fig. 4, one observes, that MCTP provides by far the fastest implementation of the synchronisation mechanism. If one compares MCTP with the best synchronisation mechanism among the other threading libraries, one finds the following performance increases: For a small number of cores, MCTP is faster by a *factor* of at least 16. For a larger number of cores, MCTP is even faster by a *factor* of at least 64.

Up to 16 cores, there is no difference between the MCTP NUMA variant and the standard variant. However, for core numbers larger than 16, the NUMA variant outperforms the standard variant showing the importance of the explicit knowledge of the NUMA layout.

Fig. 5 shows the time spent in a critical section directive as a function

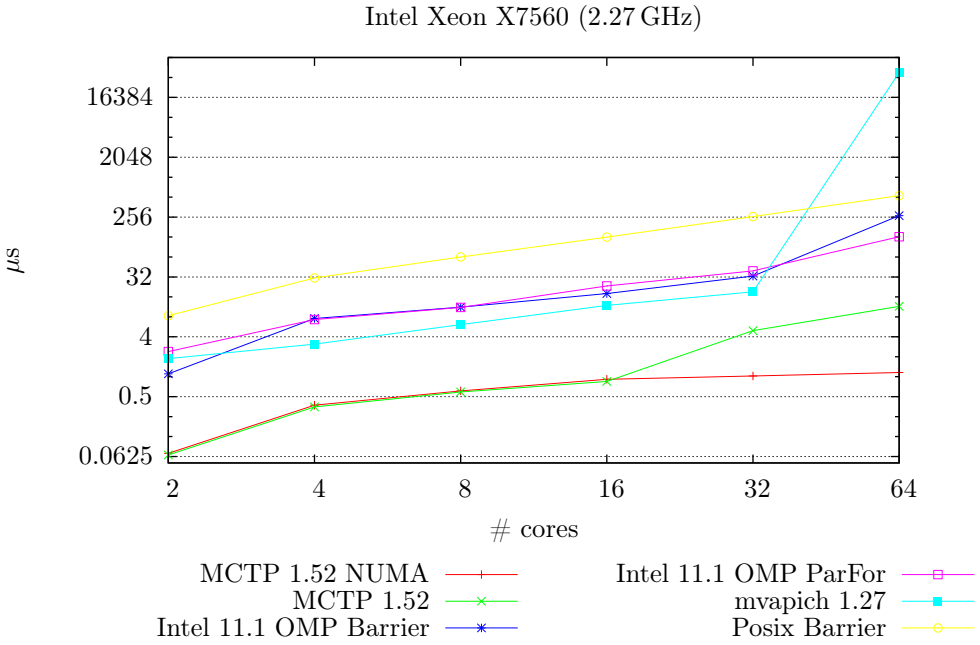


Figure 4: Thread synchronization latency as a function of the number of cores for several threading libraries.

of the number of threads (cores) for different threading libraries (note the logarithmic scale on the abscissa and on the ordinate). Here as well, each thread runs on a separate core. Inside the critical section, an integer variable is incremented by one.

Among the analyzed threading libraries, there is an OpenMP implementation delivered by the Intel 11.1 compiler, the Posix threads library and the MCTP 1.52 library.

Here as well, MCTP shows the best performance over the entire range of cores. For small core numbers, MCTP is faster than the other threading libraries by a factor of 4 at least. For large core numbers, MCTP is still faster by a factor of approximately 2.

GPI FFT

The fast Fourier transformation (FFT) is a fundamental method appearing in a large variety of scientific applications. The 2-D FFT operates on a two dimensional complex array of size $N \times N$. In principle, the 2-D FFT can be considered as being constructed out of two subsequent one 1-D FFTs with a matrix transpose in between in order to exchange the fast and the slow

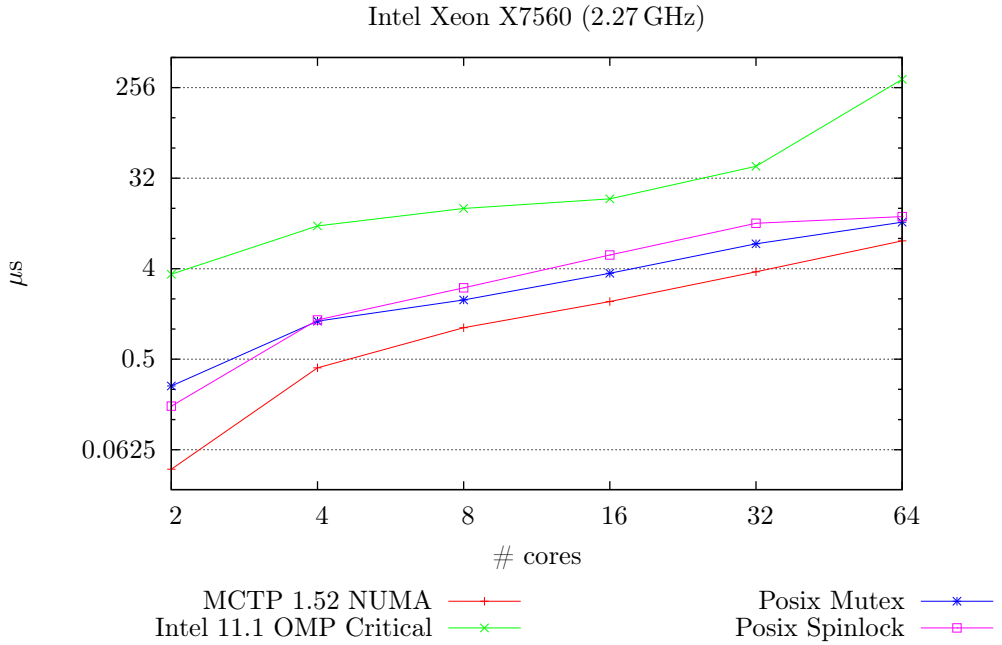


Figure 5: Critical section latency as a function of the number of cores for several threading libraries.

array direction. The transpose step is advantageous and necessary in order to exploit cache locality for both of the 1-D FFTs.

Fig. 6 shows the execution time of a two dimensional standard Numerical Recipes fast Fourier transformation (FFT) with $N = 2^{14}$ as a function of the number of involved cores.

In order to overlap the communication by the computation, the shown implementation of the 2-D FFT distributes the data along the slow array direction among the nodes, such that each node hosts a subset of complete fast array direction data. Then, one loops over the slow array direction on each node and computes the 1-D FFT along the fast direction. Here, the workload is distributed among the cores. Immediately, when one FFT is finished, the result is sent to the other nodes with non-blocking writes. In this way, the transpose step and the first 1-D FFT are intertwined and overlap. Before the second 1-D FFT starts, a wait is performed in order to make sure that all the data is transfered.

The GPI and the MPI Put algorithms are implemented exactly in the same way as described above. Note, that in order to achieve the presented performance in the case of MPI Put, the eager protocol had to be switched off. Otherwise, the performance is much worse. This is due to the fact, that

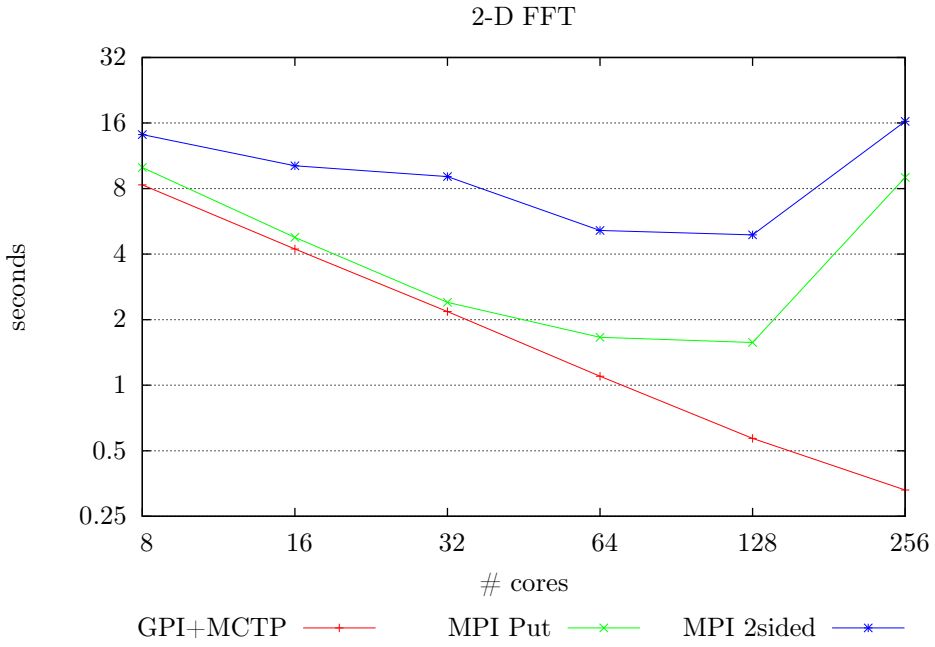


Figure 6: 2-D FFT execution time as a function of the number of cores for GPI and MVAPICH2-1.4.1 MPI.

switching the eager protocol on, MPI polls the message queues all the time which consumes CPU cycles. Obviously, these CPU cycles are wasted and cannot be used for the computation.

In the case of the MPI ISend/IRecv implementation, all the IRecv instructions are launched before the first 1-D FFT loop. Then, the algorithm which was described above applies.

The GPI implementation is always faster than both of the MPI implementations and shows perfect scaling with the number of cores. The MPI implementations start to deviate from perfect scaling at 32 cores.

GPI molecular dynamics

Molecular dynamics (MD) is a simulation method in which atoms and molecules are allowed to interact for a period of time by approximations of known physics, giving a view of the motion of the particles. This kind of simulation is frequently used in the study of proteins and biomolecules, as well as in materials science.

Usually, during the simulation the N particles of interest are distributed among the nodes of a cluster. The most time critical part of the simulation is

the computation of the interaction force between the particles which is used to propagate the entire system forward in time. In order to compute the force on a given particle, one needs to know the position of each of the other particles. Hence, the particle positions have to be broadcasted among the nodes before the computation can start. In standard molecular dynamics applications based on MPI, there is a separate communication and computation step. Fig. 7 shows the computed time steps per second of such a standard

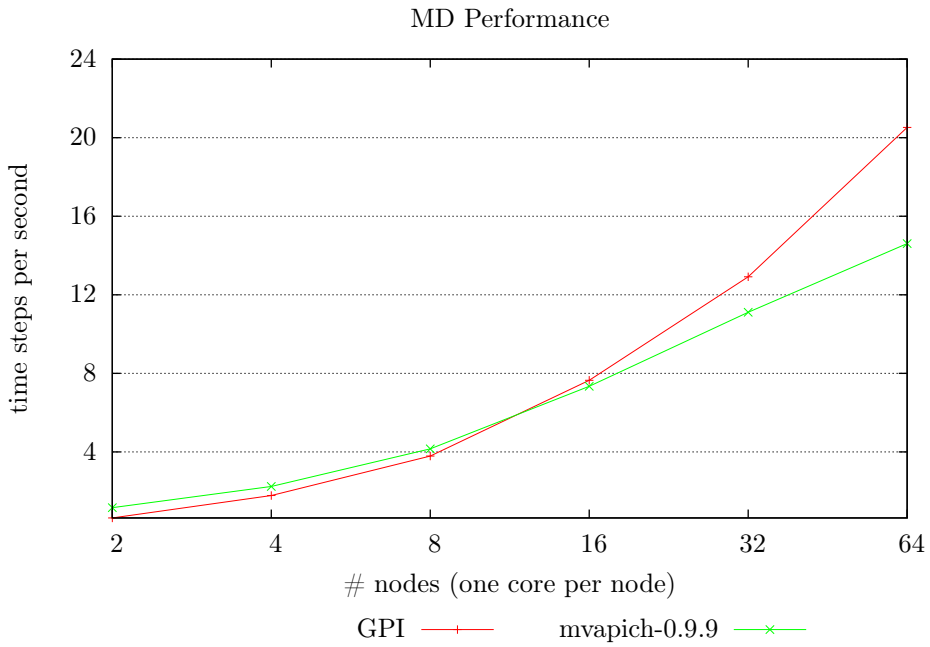


Figure 7: MD time steps per second as a function of nodes for MPI

MPI application in comparison to a GPI implementation of the same algorithm in which the communication and computation steps do overlap. This produces an additional computational effort which vanishes asymptotically as the number of nodes becomes large. The computed time steps per second are shown as a function of the number of nodes. For less than 16 nodes the additional effort in computation cannot be compensated by the hidden communication. However, for more than 16 nodes the additonal computational effort becomes less important and the strategy pays off. The GPI implementation outperforms the MPI implementation showing the importance of overlapping communication and computation patterns.

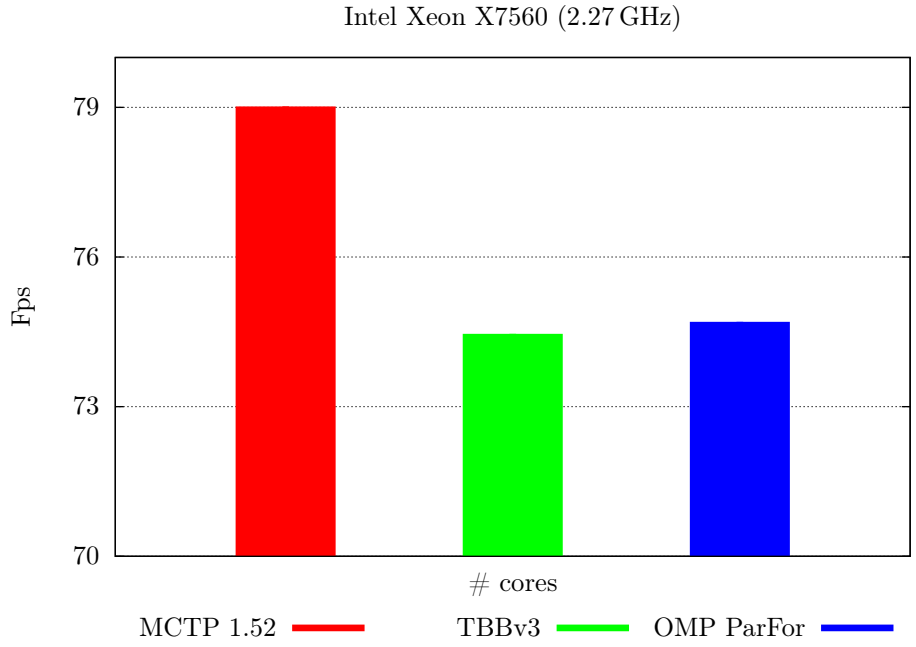


Figure 8: Frames per second achieved by the tile based rendering algorithm implemented with different threading libraries.

MCTP tile based rendering

A common technique to speed up 3-D rendering tasks or 2-D image processing algorithms is screen space parallelization, where a simple but effective task parallelization and load balancing scheme can be applied. One approach is to subdivide the image space into quadrangular regions (tiles) and assign those to different worker threads.

By assigning a unique integer number to each tile, one can use global counters to distribute the workload over all threads during runtime. If a worker thread gets idle, it immediately increments the counter to get a new work package (tile). This global counter update can be done at constant time with the GPI.

Before a thread starts working on a new tile, it writes back a previously calculated tile directly into the GPUs pixel buffer or texture object.

Fig. 8 shows the performance of the rendering algorithm measured in frames per second (Fps) depending on different threading libraries. Among these, there is an OpenMP implementation delivered by the Intel 11.1 compiler and the Intel threading building blocks library TBBv3.

Although the threading overhead is only 5% of the total execution time,

MCTP achieves a considerable increase in performance of approximately 10% in comparison to the other threading libraries. The performance benefit which has been observed in the micro benchmarks directly translates to real world applications.

References

[1] The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model, Computer Science – Research and Development, Volume 23, Numbers 3–4 / June, 2009

GPI and MCTP feature summary

The GPI and the MCTP together allow the exploitation of parallelism at different levels – cluster and multi-core. They are the building blocks for the development of efficient and scalable parallel applications.

GPI (cluster level)

- ▷ noncoherent partioned global shared memory
- ▷ asynchronous access to remote memory and files
- ▷ low latency
- ▷ full wirespeed communication
- ▷ independent message queues
- ▷ fast synchronisation
- ▷ fast collectives
- ▷ global atomic counters
- ▷ migration path from MPI
- ▷ robust and proven in industrial applications

MCTP (node level)

- ▷ abstracts native threads
- ▷ supports thread pools
- ▷ start/stop/suspend/resume threads in a pool or complete pools
- ▷ fast synchronization methods (active, relaxed and passive)
- ▷ full hardware awareness (NUMA layout/cache/core/socket/SSE level)
- ▷ get/set affinity mask
- ▷ easy to use lock/unlock (critical sections)
- ▷ thread safe aligned malloc
- ▷ file cache manipulation (linux)
- ▷ high frequency timer