Georg-Simon-Ohm Hochschule Nürnberg
Fakultät Informatik

Regionales Rechenzentrum Erlangen
High Performance Computing

Masterarbeit

Paralleles Programmieren auf hybrider Hardware: Modelle und Anwendungen

Betreuender Hochschullehrer:	Prof. Dr. Reinhard Eck
Betreuer RRZE:	Dr. Georg Hager
Autor:	Holger Stengel
Matrikelnummer:	2084013

Eingereicht am:	15. März 2010

Zusammenfassung

Diese Arbeit hat zum Ziel, vorherrschende Programmiermodelle auf Parallelrechnern mit hybrider Architektur zu bewerten. Der Schwerpunkt liegt dabei auf der Kombination von MPI mit OpenMP und deren Anwendung auf Clustern von Shared-Memory-Knoten.

Im theoretischen Teil werden zunächst anhand eines parallelen Jacobi-Lösers die prinzipiellen Unterschiede in der Parallelisierung mittels MPI, OpenMP und MPI+OpenMP erläutert. Dabei wird der entscheidende Vorteil hybrider Programmierung deutlich, der im möglichen Überlapp zwischen Kommunikation und Rechnung liegt.

Die im Hinblick auf hybrides Programmieren in vielen Publikationen vorherrschende Sichtweise ist, dass OpenMP vorrangig dazu dient, den Einzelprozess zu beschleunigen bzw. weitere Parallelisierungsebenen zugänglich zu machen. Diese eingeengte Sichtweise wird dem in MPI+OpenMP liegenden Potenzial nicht gerecht. Deswegen sollen in dieser Arbeit drei verschiedene Ansätze verglichen werden:

- "Masteronly"-Stil ohne Überlappung von Kommunikation und Rechnung; kommuniziert wird nur im seriellen Teil
- Überlapp von Kommunikation und Rechnung, wobei ein Thread kommuniziert; auf die anderen wird die Rechenarbeit manuell aufgeteilt, da durch das Abspalten eines Threads i.W. die komplette Worksharing-Funktionalität von OpenMP verloren geht
- Überlapp von Kommunikation und Rechnung unter Anwendung des neuen OpenMP "Task" Konstrukts. Damit lassen sich die Einschränkungen des zweiten Zuganges sehr elegant umgehen.

Anhand von Benchmarks wird die Performance der verschiedenen Zugänge verglichen, wobei auch auf eine effiziente Implementierung der reinen MPI-Versionen Wert gelegt wird. Als anwendungsnaher Testfall dient ein dreidimensionaler Jacobi-Löser.

Contents

1 Introduction

1.1 Motivation

Moore's Law [1], which has been valid for 45 years now, states that the number of transistors per chip that are required to hit the "sweet spot" of minimal manufacturing cost would increase exponentially at a constant factor of two every 24 months. Amazingly this tremendous growth in chip complexity has led to an almost proportional boost in computational performance for microprocessor chips. This far from trivial connection caused many application programmers to believe that single-processor performance will continue to grow at this speed. However, problems with heat dissipation and architectural issues stimulated a fundamental paradigm shift in the middle of the past decade: chip manufacturers started to offer *multi-core* microprocessors, which carried several computational cores on a single die or at least in a single package. The clock speed and overall performance of the single core begin to stagnate, and the only way to keep pace with Moore's Law with respect to computational performance is to utilize multiple cores by code parallelization. Programmers are confronted with the unpleasant fact that the responsibility for "fast" code is being shifted from hardware to software.

Large compute resources like clusters and supercomputers were also affected by this fundamental turnover. Although the elementary principles and problems of parallel programming have not changed with the introduction of multi-cores, the question arises what the best programming model for clustered multi-core multi-socket compute nodes is today. The combination of a shared-memory programming model like OpenMP [2] with distributed-memory parallelization via the Message Passing Interface (MPI) [3] appears to be a natural candidate. It is, however, far from obvious how exactly the complex hardware hierarchy should be exploited for best performance, even if the fundamental paradigm "hybrid MPI+OpenMP" is taken for granted. Moreover there seems to be a general lore stating that a hybrid code can never be faster than a well-written pure MPI program that utilizes every core as if it were not part of a multi-core chip.

This work tries to shed some light onto the discussion about pure MPI versus hybrid MPI+OpenMP for modern clustered multi-socket multi-core HPC systems. It puts heavy emphasis on the idea that one should compare the "best possible" MPI code with the "best possible" hybrid code in order to arrive at a meaningful performance comparison. To this end it analyzes important factors like intra-node versus inter-node message-passing, the behavior of communication paths supporting multiple point-to-point connections, the influence of shared caches on communication performance, and the impact of typical OpenMP pitfalls. As a simple yet nontrivial benchmark case, a three-dimensional Jacobi solver was chosen. It shows data transfer and communication patters similar to more advanced algorithms, but is very approachable in terms of performance analysis and modeling.

This work is organized as follows: The first chapter gives an overview of the testing environment. Technical specifications for the computer systems and network interconnects are provided as well as basic parallelization methods for the Jacobi benchmark algorithm. Chapter 2 introduces a parallel performance model for the Jacobi algorithm. Based on this model theoretical maximum performance for several MPI and hybrid parallel approaches is estimated. MPI optimizations relevant for hybrid computer systems are covered by the third chapter. Optimizations include appropriate subdomain placement and analysis of potential overlap effects. Finally, MPI/OpenMP hybrid programming techniques and their application to the Jacobi algorithm are evaluated.

1.2 Testbed

1.2.1 Benchmark systems

This section gives an overview of the computer systems used for benchmarking. They are located at RRZE[1] and NERSC[2]. The following descriptions have been adapted from the respective websites.

Woodcrest

The Woodcrest cluster at RRZE (termed “Woody”) is intended for distributed-memory (MPI) or hybrid parallel programs with medium to high communication requirements. It consists of 217 compute nodes, each with two Xeon 5160 “Woodcrest” chips (4 cores) running at 3.0 GHz with 4 MB Shared Level 2 Cache per dual core and 8 GB of RAM. Nodes are connected with a double data rate (DDR) InfiniBand network with a theoretical bandwidth of 10 GBit/s per link and direction. For details regarding interconnect see section 1.2.2. The machine is described in detail in [4].

Tinyblue

RRZE’s Tinyblue cluster [5] is intended for distributed-memory (MPI) or hybrid parallel programs with medium to high communication requirements. It consists of 84 compute nodes, each with two Xeon 5550 “Nehalem” chips (8 cores) running at 2.66 GHz with 8 MB Shared Cache per chip, 12 GB of RAM. Benchmarks show that at least three processes or threads are required to fully utilize memory bandwidth on one socket. Nodes are connected by a quad data rate (QDR) InfiniBand network with a theoretical bandwidth of 40 GBit/s per link and direction. MPI point-to-point bandwidth measured with Intel’s IMB benchmark suite (see section 1.2.2) is 3.2 GB/s at a latency of 1.6 μs.

[1]`http://www.rrze.uni-erlangen.de/`
[2]`http://www.nersc.gov/`

Townsend

The Townsend cluster [6] at RRZE consists of 66 compute nodes, each with one Xeon 3070 Dual Core chip running at 2.67 GHz and 4 GB of RAM. Intel's "Port Townsend" mainboard which is deployed in this machine is the reason for its name. Nodes are connected with DDR InfiniBand network, but the also available Gigabit Ethernet network turned out to be the only reliable Gigabit Ethernet implementation among the available benchmark systems. Gigabit Ethernet was used for benchmarking communication-intensive code (introduced in chapter 2).

Cray XT4

The NERSC Cray XT4 system [7] is ranked number 15 on the current list[3] of the 500 most powerful computers in the world. Each of its 9,572 nodes has a 2.3 GHz single socket quad-core AMD Opteron processor ("Budapest") and 8 GB of memory (2 GB of memory per core). Nodes are connected via HyperTransport to a three-dimensional torus network. This interconnect offers MPI point-to-point bandwidth of 1.6 GB/s and latency of 7 μs.

1.2.2 Communication characteristics

The PingPong and PingPing benchmarks included in Intel's MPI Benchmark collection (IMB) are a convenient way to investigate a system's inter-process communication behavior. They return duration and network bandwidth for MPI communications with configurable amounts of payload. PingPong covers one consecutive send and receive, whereas PingPing's communication pattern is made up of two simultaneous sends of both parties. A short introduction to those benchmarks is given below, for details see the IMB User Guide [8].

Subject of investigation are communication partners located

cross-node (CN) on different nodes,

cross-socket (CS) on different sockets of the same node, and

intra-socket (IS) on different cores of the same socket.

The latter two are also referred to by the more generic term **intra-node (IN)**.

PingPong

As the name implies, the IMB PingPong benchmark measures round-trip time for a message between two processes. The second process will not send an answer (Pong) until

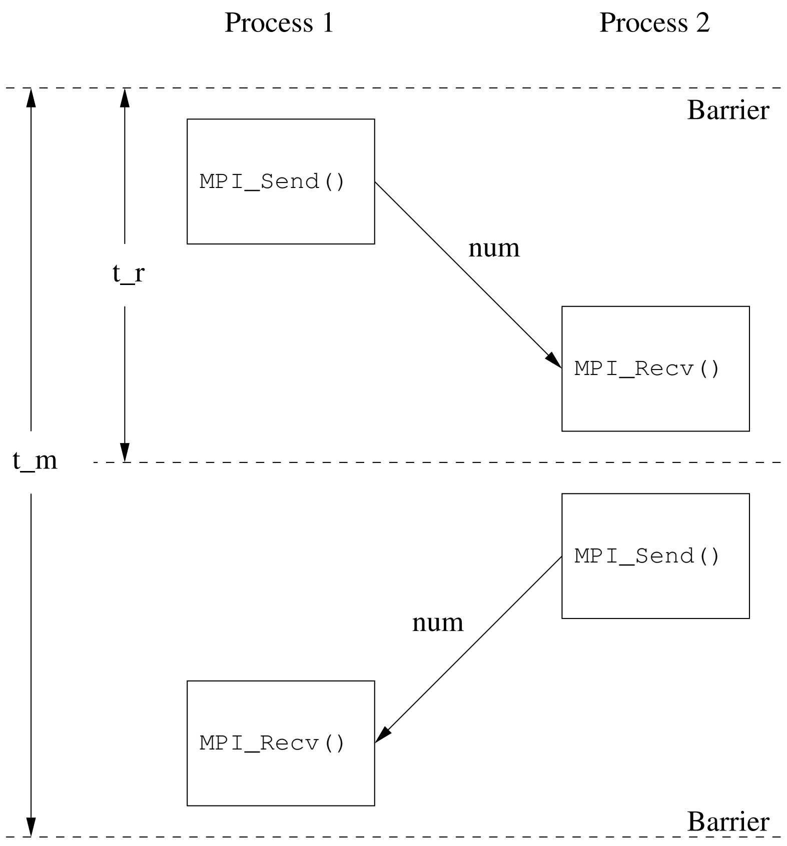

Figure 1.1: IMB Ping-Pong communication pattern. Reported time (t_r) is half the measured round-trip time (t_m). Time and bandwidth are reported for various buffer sizes (num).

it has completely received the message (Ping) from the first process (see figure 1.1 for illustration).

Time measurement ends after the first process has received the message. The benchmark reports half the measured round-trip time. This value is used to derive the unidirectional bandwidth for the communication path, which can be regarded as the maximum bandwidth obtainable for an application.

Bandwidth values plotted over message size show shapes that are characteristic for Ping-Pong and PingPing benchmarks for almost any type of network (figures 1.2 to 1.6).

For small message lengths communication time is dominated by latency. With growing message size these overhead effects become less influential and obtained bandwidth grows continuously. At a message size of about one megabyte, bandwidth reaches a maximum for our test system (two Tinyblue nodes connected with InfiniBand network) and stagnates. This is the maximum obtainable bandwidth for the communication path.

So far only a single process pair utilized the communication path. With the multi-mode versions of PingPong and PingPing several pairs communicate simultaneously. This gives information about bandwidth scalability for a connection. The QDR (quad data rate) InfiniBand network offers sustained bandwidth of about 3 GB/s for a single pair (figure 1.2). A second pair can lead to a slight improvement, but further pairs do not lead to

[3] `http://www.top500.org/list/2009/11/100`

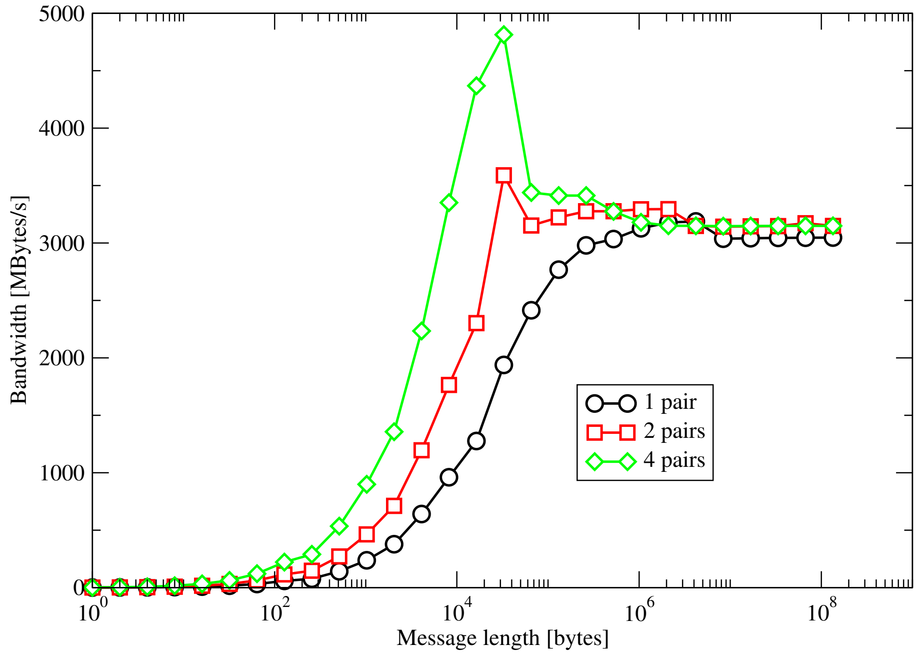

Figure 1.2: Bandwidth versus message length for the IMB PingPong benchmark. One, two or four PingPong pairs communicate simultaneously between two nodes of the Tinyblue cluster.

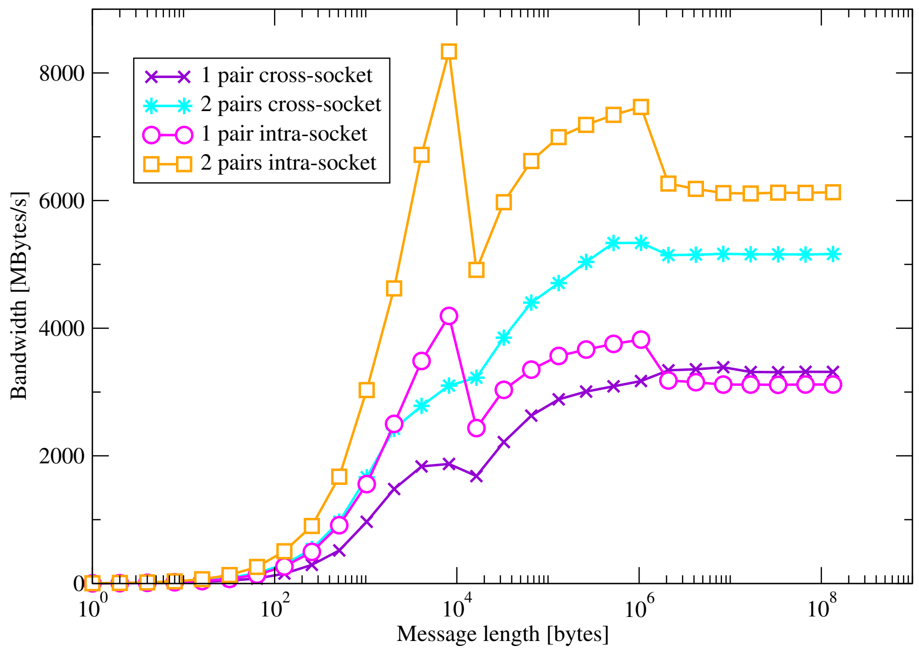

Figure 1.3: IMB PingPong for both intra-node scenarios on Tinyblue.

additional bandwidth. Note that the performance boost for multi-mode measurements at medium sized messages is a benchmark effect and does not represent actual bandwidth growth. It can be shown ([9]) that this effect is caused by an overlap of send and receive operations, possible for certain message sizes.

Intra-node PingPong measurements with one process pair show characteristics and bandwidths similar to cross-node on the test system (figure 1.3). The QuickPath interconnect present at Xeon 5550 “Nehalem” processors offers sound scalability for multi-mode PingPong. Vanilla cross-socket measurements might yield unrealistically high bandwidths. This is caused by a reuse of communication buffer for all repetitions performed for a message size [10]. Instead of actually transferring the buffer it is reused from cache as memory location does not change. Buffer reuse can be prevented with the `-off_cache` option to IMB. As a shared cache is actually existing for processes communicating within the cores of one socket, buffer reuse can be considered as native behavior and therefore do not have to be bypassed for intra-socket measurements.

Contrary to the widespread belief that intra-node communication comes at no cost, these results show that, although latency is much smaller between cores on the same node, asymptotic bandwidth is comparable to what can be achieved with state-of-the-art cluster interconnects, especially for a single point-to-point connection.

PingPing

With IMB PingPong being a tool for measuring unidirectional bandwidth, PingPing is intended to fathom bidirectional or full-duplex bandwidth between two processes. Therefore both processes issue an MPI send operation with identically sized buffers simultaneously (see figure 1.4).

The timespan from emitting the message to receiving the counterpart is measured and utilized for bandwidth calculation. Analog to multi-mode PingPong a multi-mode version of PingPing exists. Here multiple process pairs communicate simultaneously over one path. This pattern of concurrent transfers mimics halo exchange communication in typical stencil codes. In general, multiple synchronous transfers to and from neighboring subdomains proceed using a single network interface. Determined benchmark values are thus suitable as input for Jacobi performance models (chapter 2). Cross-node measurements offer quite similar properties to PingPong for the test system. In particular, sustained bandwidth for large messages is almost identical (figure 1.5). This indicates that the InfiniBand network is capable of real full-duplex transfer. Again, the multi-mode benchmark yields a slight step-up in performance.

Intra-node PingPing measurements show that for the QuickPath network full-duplex communication implies a performance impact. Compared to according PingPong values bandwidth is halved (figure 1.3). Hence, the considerations for comparing cross-node and intra-node communication performance are even more relevant here than for the PingPong case.

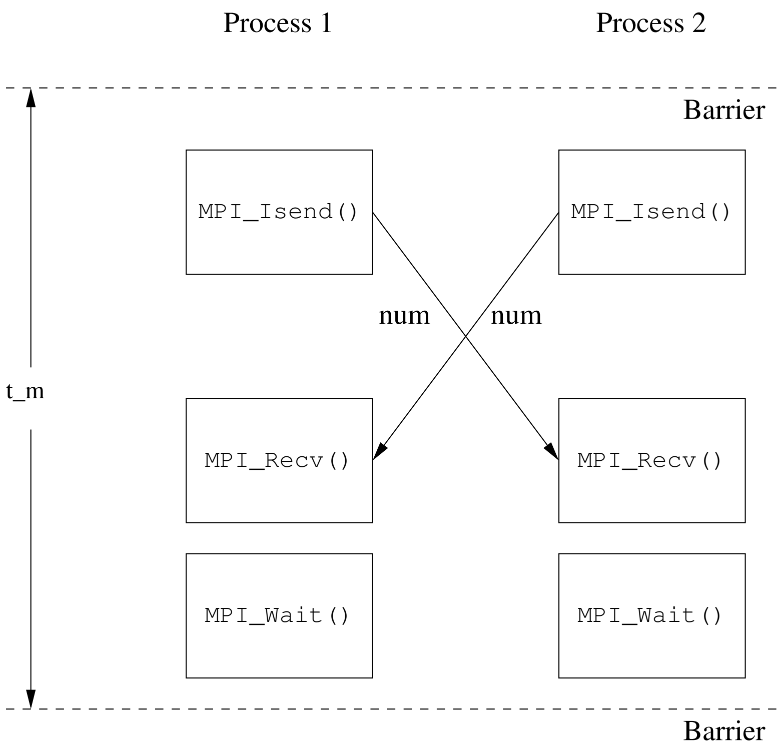

Figure 1.4: IMB Ping-Ping communication pattern. Duration of two simultaneous send operations in opposing directions is measured (t_m). Time and bandwidth are reported for various buffer sizes (num).

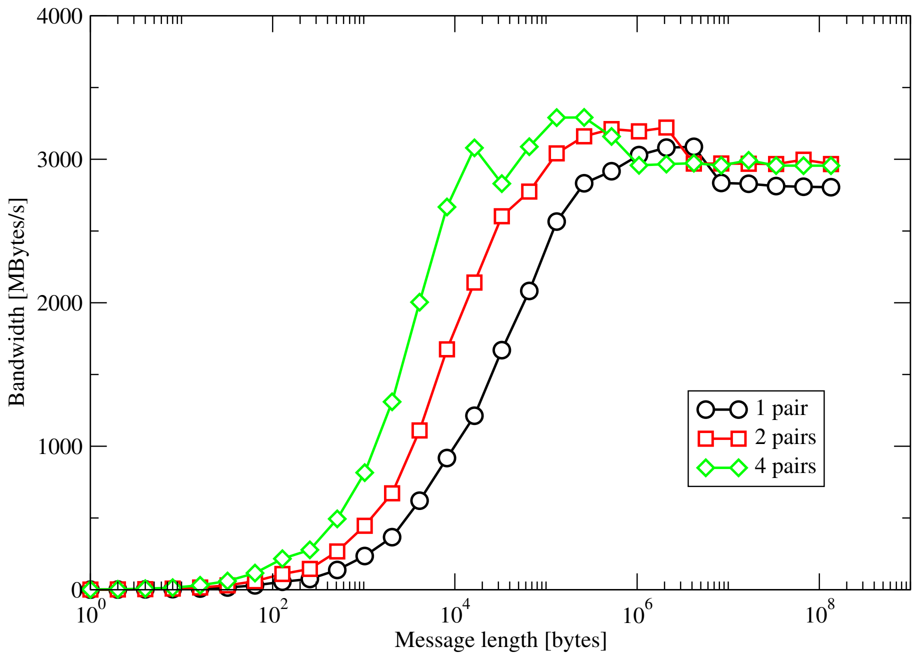

Figure 1.5: IMB PingPing between two nodes of the Tinyblue cluster.

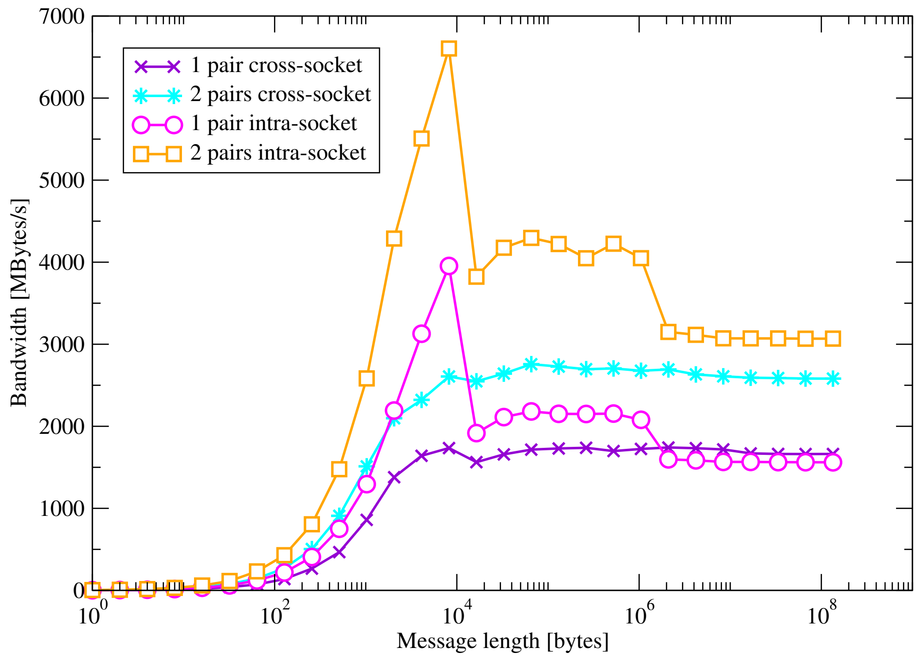

Figure 1.6: IMB PingPing for both intra-node scenarios on Tinyblue.

1.3 Parallel approaches to the Jacobi algorithm

In scientific computing, stencil-based computations are a widespread class of applications. They are utilized as iterative solvers for partial differential equations (PDE) in numerical analysis and simulation. We will now examine the Jacobi method, a simple yet instructive algorithm prototypical for the class of solvers using stencil computations. The following code fragment shows a typical six point Jacobi kernel for a 3D grid.

```
for (int n = 0; n < iterations; ++n)
{
  for (int z = 1; z < dim_z; ++z)
  {
    for (int y = 1; y < dim_y; ++y)
    {
      for (int x = 1; x < dim_x; ++x)
      {
        t1[z][y][x] = ( t0[z-1][y][x] + t0[z][y-1][x] + t0[z][y][x-1] +
                   t0[z+1][y][x] + t0[z][y+1][x] + t0[z][y][x+1] ) / 6;
      }
    }
  }  // switch grids
}
```

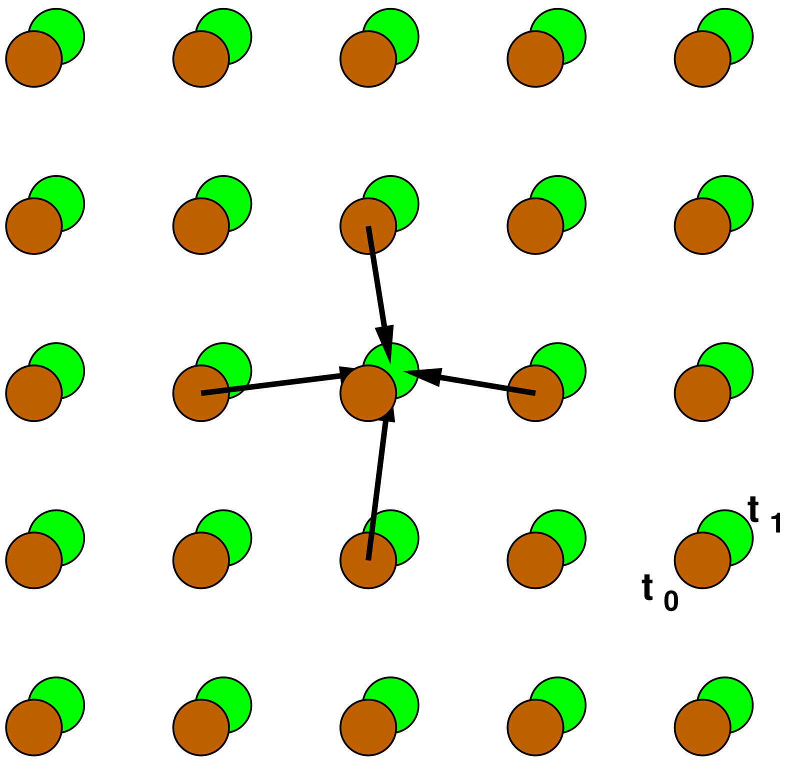

Figure 1.7: Stencil update for one cell in a 2D grid [11]. Neighbor elements from the current grid (t_0) are aggregated to update a cell in the grid representing the following time step (t_1).

This code implements a relaxation algorithm that is, e.g., capable of solving boundary value problems like the Laplace equation $\Delta T = 0$. Since we are interested in the performance properties alone, convergence issues (and the proper criteria for detecting it) are ignored, and a fixed number of iterations is performed.

The central data structures are two arrays of equal size. They represent the state of the simulated surface at the current ($t1$) and previous ($t0$) time step. During one sweep (iteration over the entire grid) every lattice point of the current grid is updated with the average value of its adjacent neighbors in the old grid. This is illustrated for two dimensions in figure 1.7.

After each sweep the grids are exchanged, so that the former destination grid becomes the source grid for the upcoming iteration.

When updating boundary cells that have neighbors "outside" the grid, two strategies can be applied. The straightforward approach is to provide static (also called open) boundary values. For some applications it is reasonable to implement periodic boundary conditions. Here, the cell at the opposing position in the grid is used as neighbor, resulting in pipe- or even torus-shaped topologies.

Spatial blocking

The Jacobi solver implementation used for benchmarking incorporates spatial blocking, a technique to improve data locality. With growing domain size data elements of logically neighboring cells drift apart in memory. It is thus getting unlikely that data needed for a stencil update can be retrieved from cache. Probability for expensive memory access rises on the other hand, hence performance unnecessarily depends on problem size. To circumvent this, sweeps are no longer performed over the whole domain like depicted in the code sample above, but over partitions of the domain [12]. Those partitions, usually called blocks, are spatial areas with adjustable size. Optimal block dimensions are given if two layers of the block fit into cache. This ensures maximum re-use of data from local cache, since only a single element has to be fetched from memory per stencil update. It is usually reasonable to choose a relatively large block size in the dimension that correlates with "memory direction" in order to take advantage of automatic prefetching mechanisms.

Parallelizing the Jacobi algorithm for shared and distributed memory computers is the subject of the following sections.

1.3.1 OpenMP parallel

Shared-memory parallel programming for science and engineering is currently dominated by OpenMP [2]. Given a serial code, parallel regions can be defined by simply inserting OpenMP directives, and major code restructuring is usually not required. Instructions within a parallel region will be executed by every thread of the currently active thread team.

An OpenMP parallel Jacobi solver can be derived form the example code in the previous section by simply inserting an OpenMP parallel loop construct [13] like the following prior to one of the loops. (Note that compilers without OpenMP support simply ignore those directives or emit a warning.)

```
#pragma omp parallel for schedule(static)
```

This causes the following loop to be executed in parallel, i.e. iterations are spread across the thread team. Which loop iterations actually are assigned to which thread, depends on the `schedule` clause of the OpenMP parallel loop construct. A plain `static` schedule, for example, divides the iteration space into `n` approximately equally sized chunks (`n` being the number of threads. See figure 1.8).

This distribution in combination with parallelizing the outermost loop (z direction) enhances data locality and therefore improves cache reuse. As a positive side-effect, the resulting large work-packages minimize overhead incurred by startup, synchronization and administrating threads.

Especially on ccNUMA systems, where memory is logically shared but physically distributed, data locality can yield a decent performance gain. Besides the arrangements

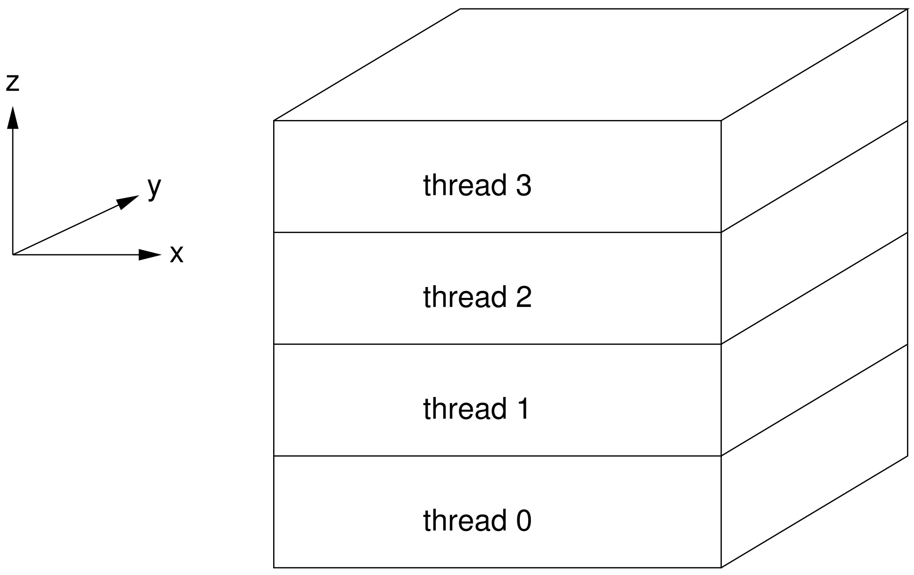

Figure 1.8: Distribution of iterations to OpenMP threads for a parallelization of the outermost loop (z direction) and static schedule. The overall number of iterations (i.e. the number of elements in this direction) is split up into equal chunks for the threads.

mentioned above, subsidiary techniques like pinning threads to cores and parallel initialization (page placement) should be taken into account [11].

1.3.2 MPI parallel

Writing a parallel Jacobi solver for a distributed memory system using MPI [3] holds far more effort than the OpenMP approach from the previous section. Since work no longer is done by a team of threads that share one address space but by normal processes, synchronization and communication among the latter has to be done explicitly.

It begins with the check for convergence after each sweep, which can not be omitted for productive applications relying on numerical correctness. The maximum deviation between values from the current and the previous time step can be easily calculated by each process for its local domain, but for a global maximum a reduction (i.e. synchronization of all processes) is required.

The second challenge arising when turning from shared-memory to distributed memory programming incorporates handling boundary areas. It is no longer sufficient to just segment the computational domain logically by splitting a loop, but it rather has to be physically divided and spread to the private memory of each process. After this procedure (coined domain decomposition), each worker performs a sweep on its local patch. To update cells at domain boundaries, values of neighboring patches are required. However, access to those boundary cells does not work automatically like in the shared-memory scenario above. Instead, up-to-date boundary values have to be transferred (i.e. using

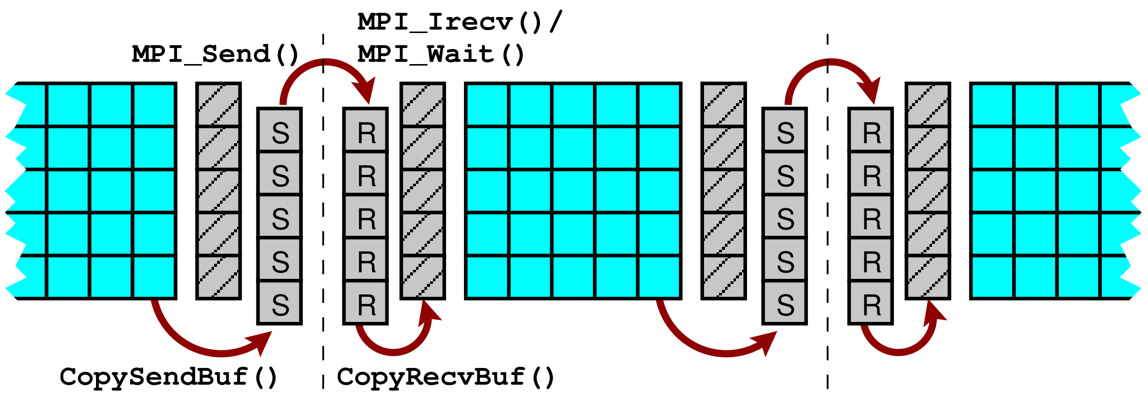

Figure 1.9: Halo communication for the Jacobi solver (illustrated in two dimensions here) along one of the coordinate directions. Hatched cells are ghost layers, and cells labeled “R” (“S”) belong to the intermediate receive (send) buffer. The latter is being reused for all other directions. Note that halos are always provided for the grid that gets read (not written) in the upcoming sweep. Fixed boundary cells are omitted for clarity. [11]

message passing via MPI) between all participating processes before each sweep. Data structures containing boundary values of neighboring processes are usually called *halo* or *ghost layers*. Halo exchange is illustrated in figure 1.9

Which ranks have to exchange halo data depends on the grid’s underlying topology and the mapping of physical coordinates to ranks. Management of this administrative data can be either implemented manually by the developer or be left to MPI’s *virtual topology* support functions. See chapter 3 for further discussion of this topic.

Since communication is a frequent performance bottleneck for MPI parallel applications, several techniques exist to reduce halo traffic. Communication effort directly corresponds to the surface of a subdomain. A basic approach is therefore to minimize subdomain faces by a thorough domain decomposition. Figure 1.10 shows the influence of domain decomposition on subdomain faces. According to this, quadratic (or cubic, for 3-dimensional domains) subdomains are ideal, as they exhibit the least surface for a given volume. MPI offers support functions that implement this partitioning (introduced in section 3.1).

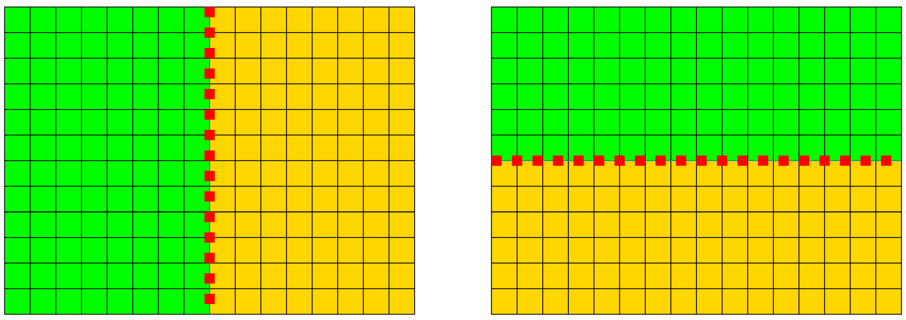

Figure 1.10: Two possible decompositions of one domain into two subdomains of equal volume. The right version requires more communication due to a larger surface (dashed line) between the neighbors. Communication volume is 16 elements for the right decomposition, and only 12 elements for that on the left.

2 Jacobi parallel performance model

In order to check performance measurements for plausibility and to estimate possible benefits of further code improvements it makes sense to calculate the expected maximum performance for a given environment by setting up a performance model. A performance model tries to predict performance behavior either qualitatively or quantitatively using a set of parameters. Those parameters can be measured values but are often estimated by some other model in turn. Typical parameters are low-level properties of the hardware at hand, like bandwidths, latencies, or derived quantities like single-node performance for some elementary operation.

This section describes bandwidth-based performance models for pure MPI and hybrid Jacobi solvers. After determining the necessary input values, performance for various scenarios will be predicted. Actual performance measurements are presented in subsequent chapters.

The measurements in this chapter were performed on the Townsend cluster using Gigabit Ethernet. This network is slow compared to the InfiniBand network, which is also available in the benchmark machine (DDR InfiniBand offers more than ten times higher sustained bandwidth at only about 10 % of the latency). Of course InfiniBand would be preferred for productive applications. In order to emphasize the influence of communication, however, it is sensible to use Gigabit Ethernet. One should also keep in mind that Gigabit Ethernet is a very popular interconnect in cluster environments: As of November 2009, more than 50 % of all Top500 systems are equipped with Gigabit Ethernet as the primary network [14].

2.1 Pure MPI

The workflow of an MPI parallel Jacobi solver mainly consists of two steps: sweeping the subdomain and exchanging halo with neighbors. Usually they are executed in a consecutive manner. Therefore durations of both add up to the time required for one Jacobi time step. Given the number of participating nodes N and a node's subdomain size in one direction L, a weak scaling performance model can be set up to

$$P(N) = \frac{NL^3}{t_{sweep} + t_{comm}} \tag{2.1}$$

We restrict to *weak scaling* (workload grows with node count) for this introduction, but a *strong scaling* model (overall workload is constant for all node counts) is just as well possible.

Since all processes simultaneously sweep a subdomain of equal size (weak scaling), t_{sweep} is constant for all numbers of nodes, whereas overall communication effort depends on the number of nodes or neighbors. Problem size grows with the number of processors, leaving single thread performance constant. On the other hand, communication plays a larger role with growing N, since more messages must be transmitted through the network and the maximum number of neighbors per subdomain increases. Note that on a fully non-blocking network the communication overhead is dominated by the process that works on the subdomain with the largest surface; see below for more details.

In the following sections values for t_{sweep} and t_{comm} are derived and applied to the model. With this instrument theoretical maximum performance is predicted for several parallelization approaches.

2.1.1 Computation

Based on the common definition of performance $P = \frac{work}{time}$, t_{sweep} can be derived from P_{sweep} using the following equation:

$$t_{sweep} = \frac{L^3}{P_{sweep}} \tag{2.2}$$

According to [11], P_{sweep} can be calculated by

$$P_{\text{sweep}} = \min\left(P_{\max}, \frac{b_{\text{S}}}{B_{\text{c}}}\right), \tag{2.3}$$

where b_S, the *STREAM bandwidth*, is the maximum memory bandwidth retrievable by an application. This value can be determined by the STREAM benchmark [15]. B_c, the code balance, describes the code's relation of memory accesses to floating point operations and can be used to estimate an upper bound for performance. One site update of a 3D Jacobi solver requires five additions, one multiplication, six loads (seven, if RFO is required) and one store. RFO is the transfer of a cache line into the cache after a write miss. It is required because the cache can only communicate with memory in packets of the cache line size. Assuming that *nontemporal stores*[1] are utilized (i.e. no RFO) and that only one element has to be loaded from memory, code balance is 1/3 W/F. Given a reasonable block size (see section 1.3) all but one element reside in cache as they were used before.

In many situations it is not necessary to tinker with the STREAM benchmark and code balance in order to formally derive P_{sweep}. It is usually sufficient to determine it with single-node Jacobi benchmark runs. This is particularly true in a situation where focus does not lie on computational but on communication effort. Tests with a mature Jacobi

[1]Only available in x86 processor architectures

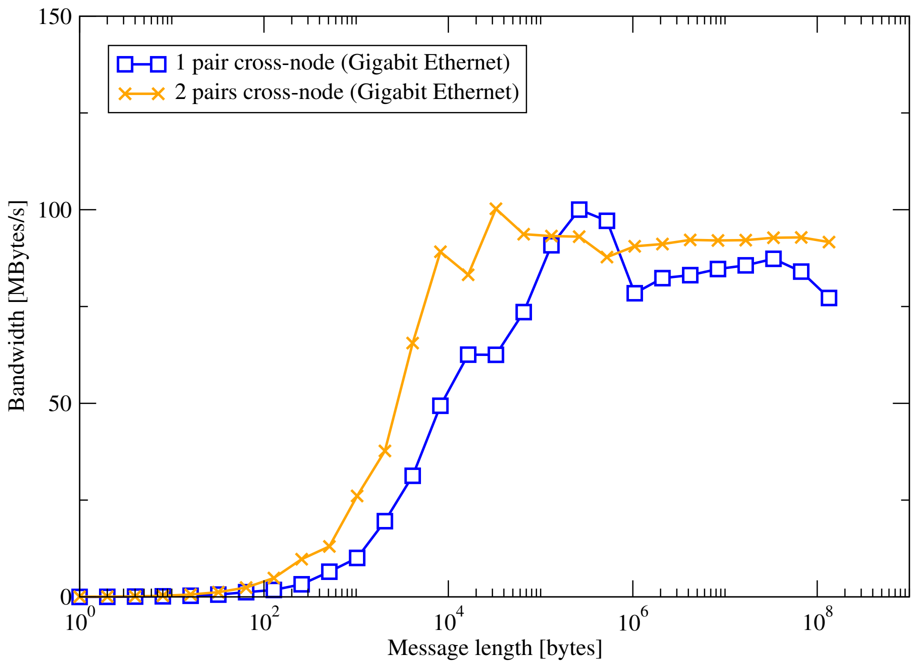

Figure 2.1: IMB PingPing (cross-node) using Gigabit Ethernet on Townsend. Reported bandwidths apply for one direction. Multi-mode values are accumulated over both pairs.

benchmark code on the Townsend cluster yield P_{sweep} values of 177 MLUPs/s (1 core) to 198 MLUPs/s (2 cores), leading to, e.g., t_{sweep} of 9.76 ms and 8.73 ms for a domain with 120^3 elements.

2.1.2 Communication

Network communication is made up of two phases, setup and transfer. The duration of the latter is determined by the network's achievable bandwidth $b_{network}$ and the communication volume V, whereas the former is represented by latency (t_l). Both t_l and $b_{network}$ can be measured with the IMB PingPing benchmark introduced in section 1.2.2. The benchmark consists of one or more (multi-mode) process pairs communicating simultaneously. Multi-mode PingPing has similar communication characteristics to Jacobi halo exchange. Measurements also show how network bandwidth scales for multiple communication partners.

According to figure 2.1 bandwidth reaches its maximum at about 85 MB/s per direction. A single communication pair is not able to utilize the network to full capacity, and additional pairs can gain a slight bandwidth rise to 90 MB/s. With a subdomain size of 120^3 elements per direction, buffer size for one halo transfer is $120^2 * 8$ Byte $= 112.5$ kB. As figure 2.1 shows, full bandwidth can be assumed for this communication volume.

Along with bandwidth PingPing also reports communication time. In order to estimate network latency, PingPing measurements with small message sizes are examined, as communication overhead is dominant there.

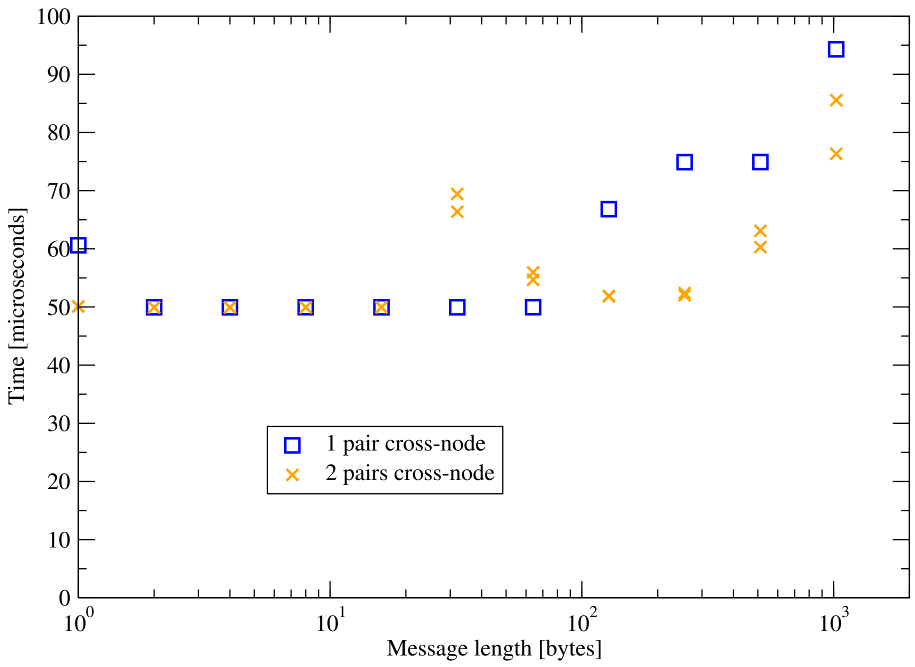

Figure 2.2: Communication time versus message size for IMB Ping-Ping (cross-node) using Gigabit Ethernet on Townsend. Communication time equals network latency for small message sizes. Communication time is shown for single mode (blue square) and both multi-mode pairs (orange cross).

Figure 2.2 shows a constant duration of $50\mu s$ for buffer sizes up to 64 kB. This implies a dominance of communication overhead over actual transfer time. Reported values can thus be regarded as latency.

Each multi-mode PingPing pair reports latency values equal to single-mode latency ($50\mu s$). This indicates that overall latency scales linearly with the number of communication pairs and hence latencies do not overlap for small messages. Measurements with message sizes in the scale of halo volumes (about 100 kB in our tests) show, however, that latencies do overlap to a certain degree. Therefore only a fraction of the latency penalty has to be taken into account for each communication.

2.1.3 Model

Now that all necessary technical parameters are available, the final step towards a performance model is to bring them into due proportion. As a start we look at a scenario where a single process runs on each node. For N participating nodes, overall domain size must be in a shape that domain decomposition yields N equal cubic subdomains. Subdomains are cubic to ease calculation of communication volume in each direction. Domain decomposition determines which processes have adjoining subdomains and have to exchange halo information after each sweep. Note that open boundary conditions were used, so halo exchange is not necessary at domain boundaries. Since all nodes work in

nodes	nodes per dim. X / Y / Z	domain size	max. CN neighbors	CN communications 1ppn	2ppn
1	1 / 1 / 1	120 / 120 / 120	0	0	0
2	2 / 1 / 1	240 / 120 / 120	1	1	2
3	3 / 1 / 1	360 / 120 / 120	2	1	2
4	2 / 2 / 1	240 / 240 / 120	2	2	3
6	3 / 2 / 1	360 / 240 / 120	3	2	3
8	2 / 2 / 2	240 / 240 / 240	3	3	5
12	3 / 2 / 2	360 / 240 / 240	4	3	5
16	4 / 2 / 2	480 / 240 / 240	4	3	5
24	3 / 4 / 2	360 / 480 / 240	5	3	5
27	3 / 3 / 3	360 / 360 / 360	6	3	5
32	4 / 4 / 2	480 / 480 / 240	5	3	5

Table 2.1: This table shows for each number of nodes the result of the domain decomposition and the required overall domain size required to yield subdomains of equal size (120^3) for each node. The last three columns contain the maximum number of possible neighbors for one node and the resulting number of cross-node communications required for halo exchange for the case that only one (1ppn) or both (2ppn) cores per node are utilized. Note that each rank sends to neighbors in positive and negative direction of one dimension simultaneously. Therefore the number of communications is not equal to the number of neighbors. See text for details.

parallel and the communication network is non-blocking, performance is limited by the node with the highest communication demand, i.e. with the largest number of neighbors. Table 2.1 shows the maximum number of neighbors a node can have resulting from a typical domain decomposition.

Data volume exchanged with one neighbor is determined by halo size and halo width.

- Halo size is the number of elements at the particular subdomain face multiplied by the size of one element.
- Halo width specifies the thickness of the halo layer. It is usually 1, but if e.g. more than one sweep is performed before a halo exchange, it has to be higher [16].

We have seen before that latencies of multiple simultaneous communications overlap. Since the benchmark code issues communication operations in positive and negative direction of each dimension (e.g. up and down) simultaneously, only one latency has to be taken into account per dimension. The following equation incorporates previous results. It allows performance estimations for N nodes running one process each.

$$P_{1ppn}(N) = \frac{N * L^3}{t_{sweep1ppn} + \frac{8\text{Byte} * L^2 * maxneighbors}{b_{gbit}} + n_{comm} * l_{gbit}} \tag{2.4}$$

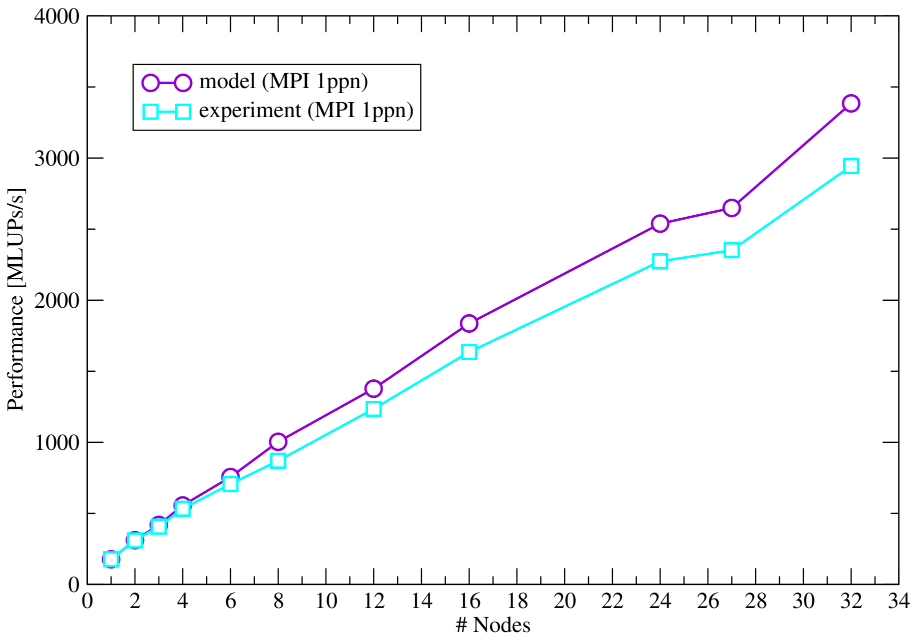

Figure 2.3: MPI parallel 3D Jacobi solver with spatial blocking on 1 to 32 nodes of the Townsend Cluster using 1 core per node and Gigabit Ethernet interconnect. Domain size for each node is 120^3 `double` elements (weak scaling).

In figure 2.3 predicted performance is compared to measurements gathered in a weak scaling study with 1 to 32 nodes of the Townsend cluster. A single process computes a subdomain of 120^3 double elements on each node. Halo size is therefore constant for all directions and node counts. Communication volume solely depends on the number of neighbors. Values from model and experiment are almost identical for small node numbers. A small gap can be observed starting at eight nodes, as measurements do not exactly reach predicted performance. This difference might have its source in the additional communication direction required from eight nodes on. However, the characteristic shape, e.g. the ditch at 27 nodes due to higher communication volume, is reproduced by the model.

So far, only one of the two available cores per node were used by benchmarks. In order to fully exploit computational resources it is obvious to use all cores. This can be achieved without any source code modifications by just running the MPI parallel solver with the double number of processes. Provided that MPI processes are thoroughly distributed to nodes, a node accommodates two processes, each running on one core and processing one half of the subdomain. The additional core on each node does not lead to a duplication of computational performance due to limited memory bandwidth, but to a step-up of about 20 % (from 177 MLUPs/s to 198 MLUPs/s). However, intra-node fragmentation of the subdomain causes a change in communication profile. The domain decomposition algorithm ensures identical subdomain sizes per node, no matter if one or two processes are active on the node. Therefore faces to neighbors and with that communication volume remains equal. But actual halo transfer is split up to two processes for two of the three

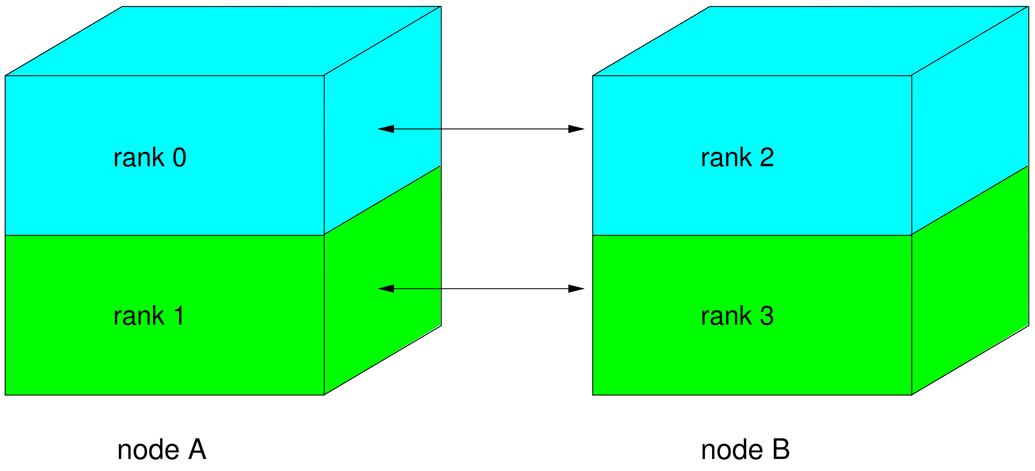

Figure 2.4: Halo exchange between two nodes running two MPI processes each. A node's subdomain is further subdivided when using two ranks instead of one. Communication volume is constant, but the number of latencies increases as two communications are required for halo exchange in two of the three dimensions.

dimensions (see figure 2.4). This implies a larger number of communications and thus latencies to be taken into account (see table 2.1). Equation 2.4 applies for the this situation, parameters n_{comm} and t_{sweep} have to be adjusted. Note that buffer size for "split" halo communication is still large enough ($120 * 60 * 8$ Byte $= 56$ kB) to gain full network bandwidth.

Figure 2.5 summarizes performance predictions for this and the following parallelization strategies.

2.2 MPI/OpenMP hybrid

In this section we refine the basic performance models developed in the previous sections, in order to predict performance for Jacobi solvers parallelized using both MPI and OpenMP. This approach is called MPI/OpenMP programming (introduced in chapter 4).

General specifications from the previous section (e.g. the test system and domain sizes) apply. In each of the following sections a hybrid programming scenario is briefly described. Based on measured data from the previous section computational and communication effort are estimated. Performance predictions are summarized in figure 2.5.

Vector mode hybrid

A single process per node spawns two OpenMP threads for computation. After each sweep, halo communication is done in a serial region. Since a node's subdomain is not split up for two processes, communication equals an MPI implementation with one process per node. Therefore equation 2.4 applies.

Assuming that OpenMP overhead does not affect performance for the given subdomain size, an estimate for t_{sweep} with two threads can be derived from the previously measured performance value for two MPI processes on one node (198 MLUPs/s).

Task mode hybrid (manual)

A single process per node spawns two OpenMP threads, one dedicated to halo computation and communication, the other dedicated to computation of bulk blocks. Communication and computation are assigned to threads manually by thread ID. This introduces a major difference to the solutions considered so far: Communication and computation are no longer alternating but are executed simultaneously. Explicit overlap requires fundamental modification to our basic performance model (equation 2.1), where overall execution time is the sum of communication and computation time. Due to concurrent execution, execution time is the maximum of both durations. A basic performance model for task mode hybrid applications can thus be set up:

$$P_{taskmode}(N) = \frac{NL^3}{max(t_{sweep}, t_{comm})} \tag{2.5}$$

As one of the two threads is dedicated to communication, only a single thread is available for computation. Consequently, measured single MPI process performance (177 MLUPs/s) is the appropriate input value for deriving t_{sweep}.

Again, the communication pattern equals that of the MPI solver with one process per node. Communication time can thus be calculated according to equation 2.4.

Task mode hybrid (OpenMP tasks)

Manual work scheduling has one obvious drawback. Static dedication of one thread to communication might lead to load imbalance, especially in the single node case, where no communication is required. This can be solved with the following modification: The bulk thread creates an OpenMP task from each loop iteration. After finishing communication, the halo thread joins task processing. Scheduling of computational work to threads is load dependent.

Estimation of t_{sweep} is a little more involved than for the previous situations. Looking at the single node case, where no communication is required, both threads compute and performance roughly equals single node MPI performance with two processes. However, an interdependence between communication effort and computing performance exists. With growing communication effort, the communicating thread will take less computational

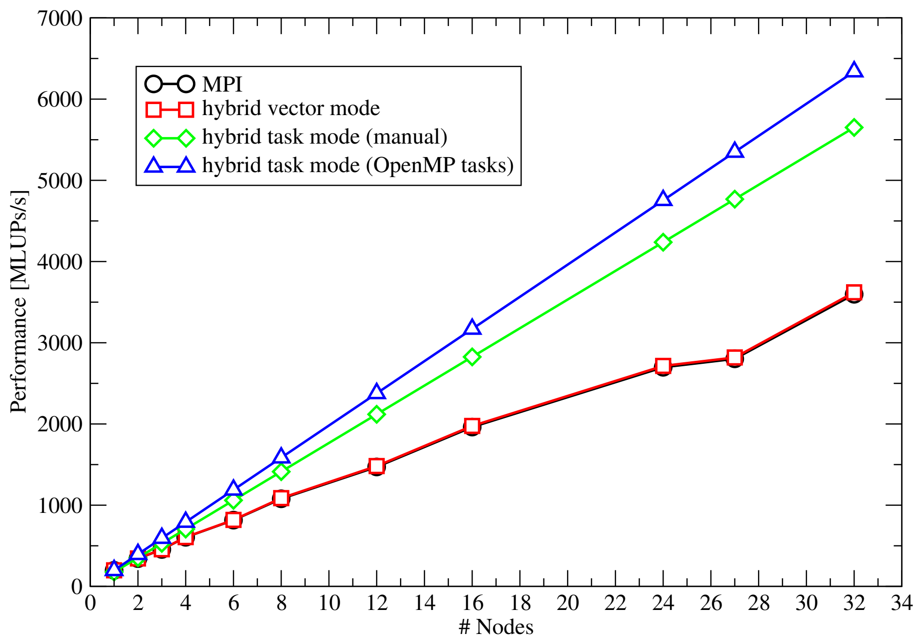

Figure 2.5: Prediction of maximum performance for various parallelization approaches of a 3D Jacobi solver, based on simple models developed in this chapter. Parameters reflect a weak scaling study with 1 to 32 nodes of the Townsend cluster using Gigabit Ethernet and a subdomain size of 120^3 elements per dimension.

work and thus computational performance will converge towards single process MPI performance. In order to render the theoretical maximum and to restrain from manual task mode, the 2 process MPI value is used for the projections pictured in figure 2.5.

Performance predictions show a distinct predominance of the task mode hybrid approach. Explicit overlap of communication and computation pays off even in the manual task mode case where only one core per node is used for computation. Perfect scaling from 1 to 32 nodes indicates that communication time is of no influence, i.e. t_{sweep} is larger than t_{comm} for all numbers of nodes. However, for 27 nodes, where one node has the maximum number of 6 neighbors, t_{sweep} and t_{comm} are quite similar (8.7 vs 7.8 ms). It is most likely that communication gains influence in a less optimistic environment.

Vector mode hybrid and pure MPI performance are almost identical, with a slight advance for the former. The only difference in the underlying models is the number of accounted latencies, which differs in 1 or 2. In spite of transfer times from 1 to 8 ms, latencies of 50 μs are irrelevant.

Benchmark tests show that predicted performance can not be claimed in a realistic situation (see chapter 4). However, the qualitative hierarchy among methods is reproduced.

3 MPI optimizations

It is necessary to fully utilize the possibilities of pure MPI programming in order to gain insight into the real benefits of its hybrid challenger. Therefore a number of techniques are introduced or evaluated here to optimize MPI communication.

3.1 Mapping of ranks to subdomains

An MPI parallel program generally starts a process on every available core in order to fully utilize the computing power of a machine. However, in some situations it may also be advisable to make use of only a fraction of the cores of each node. For multi-core systems, the number of processes p to start is the product of node count and the number of cores per node. Thus the computing domain must be decomposed into p subdomains. Intra-node partitioning will intensify in the future, as the number of cores per node is growing. Which subdomains are computed by which processor is determined by a two-level mapping (see figure 3.1 for an illustration). MPI processes, identified by their rank, are assigned to physical processors. This can be made deterministic by pinning libraries like PLPA [19], which bind processes or threads to CPUs, and the specific startup procedures implemented in the MPI library used. Mapping of subdomains to ranks on the other hand can be left to MPI's library function `MPI_Cart_create()`. It is defined as follows [11]:

```
MPI_Comm comm_cart;
int ndims, reorder;
int* dims, periods;

int MPI_Cart_create(MPI_Comm comm_old,    // input communicator
                    int ndims,            // number of dimensions
                    int *dims,            // number of ranks in each dim.
                    int *periods,         // periodicity per dimension
                    int reorder,          // true = allow rank reordering
                    MPI_Comm *comm_cart); // Cartesian communicator
```

The function creates a communicator that contains topology information applicable to a Cartesian grid. This includes proximity relationships among ranks, the number of dimensions and, of course, a map between ranks and coordinates addressing subdomains. Coordinates merely describe a logical decomposition ("topology") of the domain. Calculation of actual subdomain addresses, i.e. offsets along the coordinate directions, is left to the programmer.

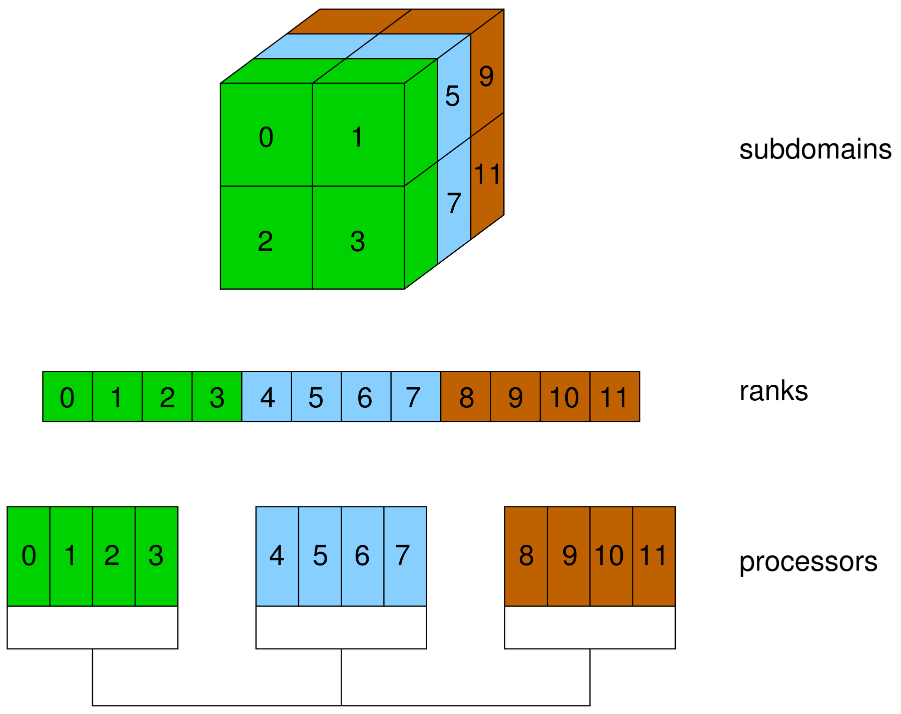

Figure 3.1: Illustration of the two-level mapping of subdomains to processors. For the given number of 12 processors, the domain is decomposed into $2*2*3$ subdomains $(X*Y*Z)$. Subdomain coordinates are assigned to MPI ranks, which are pinned to processor cores in turn.

The input parameter `dims` determines the number of processes in each dimension and therewith the number of cuts through the domain. Given the number of processes and the number of dimensions, several decompositions may be possible. It is often sensible to cut the domain into an equal number of slices along each dimension. For this task the "convenience function" `MPI_Dims_create()` can be used. It is parametrized by the number of nodes and dimensions and returns the number of processes per dimension. Output can be influenced by specifying the number of processes for some dimensions in advance, which might be necessary to trim the shape of the topology. See the following definition for reference [11]:

```
int nprocs, ndims;
int* dims;
int MPI_Dims_create(int nprocs,  // number of processes in grid
                    int ndims,   // number of Cartesian dimensions
                    int* dims);  // in:   /=0 # procs. fixed in this dir.
                                 //       ==0 calculate # nodes
                                 // out: number of processes in each dir.
```

Whether the placement of ranks given by the input communicator may be altered by `MPI_Cart_create()` for the new communicator can be specified via the parameter `reorder`.

The following code example uses `MPI_Cart_create()` and `MPI_Dims_create()` to investigate how ranks are mapped to partitions of a 3-dimensional domain by the MPI library. This is of interest as it gets important, especially with growing numbers of cores per node, to correctly arrange subdomains across MPI ranks in order to minimize cross-node in favor of intra-node communication.

```
int main(int argc, char *argv[])
{
  int dims[3]={0,0,0}, periods[3]={0,0,0}, ndims=3;
  int reorder=1, nprocs, rank;
  MPI_Comm commCart3D;
  int nb[6], coords[3];
  int hostnamelen;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc,&argv);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);  // get number of processes
  MPI_Dims_create (nprocs, ndims, dims);
  MPI_Cart_create (MPI_COMM_WORLD, ndims, dims,
                   periods, reorder, &commCart3D);

  MPI_Get_processor_name(hostname,&hostnamelen);
  MPI_Comm_rank(commCart3D, &rank);
  MPI_Cart_coords(commCart3D, rank, 3, coords);  // get coordinates
```

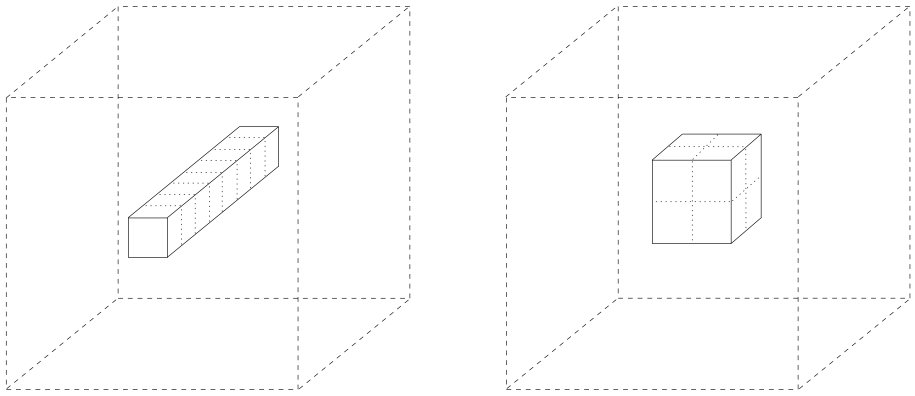

Figure 3.2: Distribution of subdomains to multi-core nodes for current $MPI\_Cart\_create()$ implementations (left) and a communication aware approach (right). The number of cross-node communications for this 8 core per node example add up to $8*4+2 = 34$ (left) and $8*3 = 24$ (right), resulting in 30 % less communication.

```
  MPI_Cart_shift(commCart3D, 0, 1, &nb[0], &nb[1]); // upper / lower
  MPI_Cart_shift(commCart3D, 1, 1, &nb[2], &nb[3]); // left / right
  MPI_Cart_shift(commCart3D, 2, 1, &nb[4], &nb[5]); // front / back
                                                    // neighbor
  // output
  MPI_Finalize();
}
```

By running this program, a thorough insight into the absolute and relative location of ranks within the domain can be gained. Experiments with several MPI libraries show however, that they are not as aware of the hardware environment as they could be.

In many MPI implementations, `MPI_Cart_create()` assigns ranks to consecutive subdomains beginning with the fastest running dimension (z in our case). One node with n cores therefore works on a “stick” consisting of n subdomains (see left of figure 3.2). The aggregated surface of a node and therefore the amount of cross-node halo communication depends on the arrangement of subdomains. The goal should be to create cubic domains for each node, as a cube has a smaller surface than a stick at a given volume (see right of figure 3.2).

Figure 3.3 shows that the importance of optimizing subdomain surface grows as the number of cores per node increases.

With `MPI_Cart_create()` not delivering optimal results it is due to the application developer to establish an adequate mapping. This however implies to dispense with this and other convenience functions delivered with MPI. The following section describes an approach to a topology-aware distribution of ranks to subdomains.

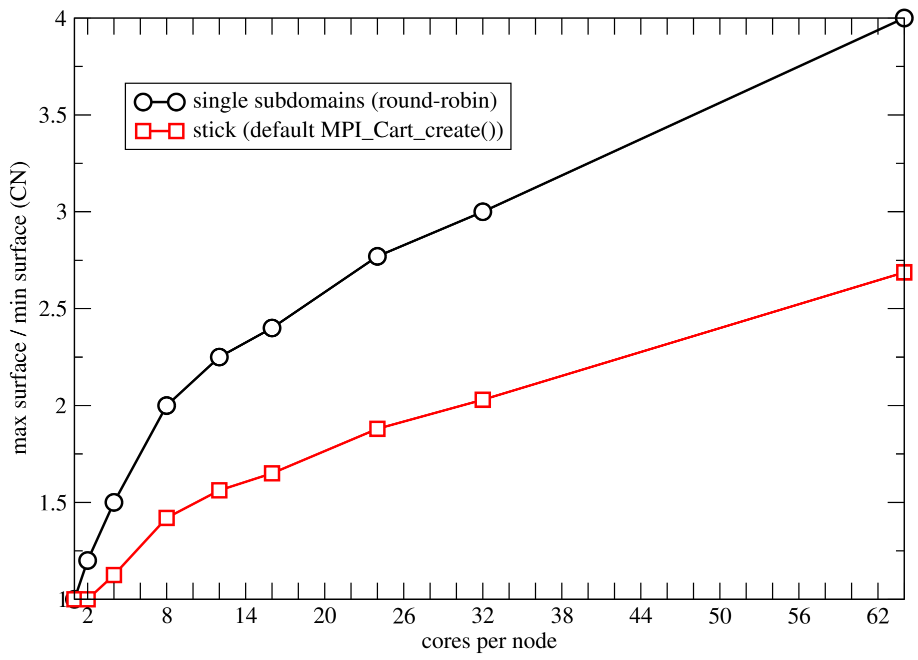

Figure 3.3: Aggregated cross-node communication volume for cubic subdomains. Penalty for standard MPI (stick) compared to optimal (cubic) distribution of subdomains to nodes (red squares). The black graph (circles) compares optimal and worst case distribution, where all neighbors of subdomains are located on different nodes (round robin distribution) resulting in cross-node transfer for any next-neighbor communication in the process topology.

3.1.1 Topology-aware mapping

The starting point is a 3-dimensional computing domain and a set of multi-core nodes. To simplify matters we assume that the total number of processors is suitable to spawn a 3-dimensional process grid, i.e. it can be factorized into three integer numbers larger than 1. It is furthermore presumed that ranks are bound to cores. A function like `MPI_Dims_create()` was utilized to determine the number of processes per coordinate direction (`dims`) and all ranks are part of the current communicator. For illustrating the concept we restrict the number of cores per node to 8, the smallest value with a perceptible benefit from correct subdomain placement and also the number of cores present in on node of our test system. In order to reduce cross-node communication the surface of the subdomains assigned to the cores of a node has to be minimized. Instead of the "stick" of subdomains created by `MPI_Cart_create()` a more "cubic" shape has to be mapped to a node. For a node with 8 cores a cube with 2 subdomains per direction is ideal (see figure 3.2). Similar to `dims` (the array containing the number of subdomains per direction from the domain decomposition), an array `nodeDims` is established to hold the number of ranks per direction for a node. Note that `nodeDims` can be created with `MPI_Dims_create()` as well as `dims`, since the task is in both cases to equally distribute a number of processes across a number of dimensions. Equal distribution is desirable only for cubic domains that are to be partitioned into cubic subdomains, which can be presumed for common situations.

The following function generates a map between subdomains and ranks in consideration of intra-node structure represented by `nodeDims`. Therefore a blocked loop nest describes how ranks have to be placed in the grid one by one. In the style of `MPI_Cart_create()`, the function takes return parameters (`rankCoord` and `coordRank`). They contain maps of ranks to subdomain coordinates for bidirectional lookup. Input parameters are `dims` (which implies the number of ranks) and `nodeDims`. The number of dimensions (`ndims`, as used in `MPI_Cart_create()`) is restricted to three due to the structure of the loop, but note that a 2D mapping can be generated by predefining the number of elements in one dimension to 1.

```
struct coord
{
  int c0;
  int c1;
  int c2;
};

#define cidx(c0,c1,c2,d1,d2) ((c2)+((c1)+(c0)*(d1))*(d2))

void subdomainMap(int* dims,        // in: ranks per dimension (global)
                  int* nodeDims,    // in: ranks per dimension (per node)
                  coord* rankCoord, // out: get coordinates to given rank
                  int* coordRank)   // out: get rank to given coordinates
{
  int rank=0;
  // iterate over nodes in process grid
  for(int nc0 = 0; nc0 < dims[0]; nc0 += nodeDims[0]){
  for(int nc1 = 0; nc1 < dims[1]; nc1 += nodeDims[1]){
  for(int nc2 = 0; nc2 < dims[2]; nc2 += nodeDims[2]){
    // distribution of ranks within node
    for(int c0 = nc0; c0 < min(nc0 + nodeDims[0], dims[0]); c0++){
    for(int c1 = nc1; c1 < min(nc1 + nodeDims[1], dims[1]); c1++){
    for(int c2 = nc2; c2 < min(nc2 + nodeDims[2], dims[2]); c2++){

      // rankCoord to retrieve coordinates for a rank
      coord co = {c0,c1,c2};
      rankCoord[rank] = co;
      // coordRank to retrieve a rank for coordinates
      int cindex = cidx(c0,c1,c2,dims[1],dims[2]);
      coordRank[cindex] = rank;

      rank++;
    }
    }
    }
  }
  }
  }
  return;
}
```

When using this method instead of `MPI_Cart_create()`, some of the convenience functions provided by MPI are no longer sufficient. Two of those functions, `MPI_Cart_coords` and `MPI_Cart_rank`, can be replaced by simple access to one of the maps created by `subdomainMap()`. For halo exchange in MPI parallel stencil codes it is essential for each rank to know its neighbors. Therefore the following method acts as a substitute for `MPI_Cart_shift()` by applying the maps created above. It is parametrized by the initiators rank and a list containing a coordinate offset for each direction. These offsets are positive or negative integer numbers that are added to the current coordinates. A check for the sanity of coordinates or dimensions and boundary handling is omitted for brevity. To fully substitute the functionality of `MPI_Cart_create()`, this function would be suitable to implement periodicity of the grid. Explicitly handing over parameter `dims` to `neighborRank` is necessary for determining a rank from coordinates. In a production environment this value and both maps will of course be encapsulated by more sophisticated data structures like an object or, corresponding to MPI's implementation, a communicator.

```
int neighborRank(int myRank,              // initiator's rank
                 int* coordinateOffsets,  // neighbor's distance
```

```
                                          // per dimension
                 int* dims)               // ranks per dimension
{
  // determine coordinates of current rank
  coord myCoord = rankCoord[myRank];

  // determine coordinates of neighbor rank
  coord nbCoord;
  nbCoord.c0 = myCoord.c0 + coordinateOffsets[0];
  nbCoord.c1 = myCoord.c1 + coordinateOffsets[1];
  nbCoord.c2 = myCoord.c2 + coordinateOffsets[2];

  // determine neighbor's rank
  int cindex = cidx(nbCoord.c0, nbCoord.c1,
                    nbCoord.c2, dims[1], dims[2]);
  int nbRank = coordRank[cindex];

  return nbRank;
}
```

Benchmark The benefit of topology-aware mapping of subdomains and ranks can be demonstrated with a strong scaling study of the Jacobi solver benchmark (see figure 3.4), if the comparatively slow Gigabit Ethernet network is used. For the InfiniBand network, however, differences in communication overhead on this scale are of no consequence.

A 3-dimensional MPI parallel Jacobi solver was used to investigate the influence of subdomain mapping. Topologies generated by `MPI_Cart_create()` and `subdomainMap()` were compared. Domain size started at 50 and was increased to 1200 elements per direction. Substantially larger domains would exceed available memory. Performance reaches a plateau at a domain size of about 500 elements per direction and does not rise significantly for larger domains. For smaller domains (and with that smaller subdomain faces) performance is limited by communication overhead.

On four nodes of the Tinyblue system with 8 cores per node, performance could be improved for about 50 %. This conforms to the corresponding theoretical benefit value from figure 3.3.

Some vendors offer mechanisms to the user for monitoring and manipulating distribution of MPI ranks on the nodes [20].

A OpenMP/MPI hybrid approach (as introduced in chapter 4) could be an alternative solution to the somewhat "pedestrian-style" subdomain mapping.

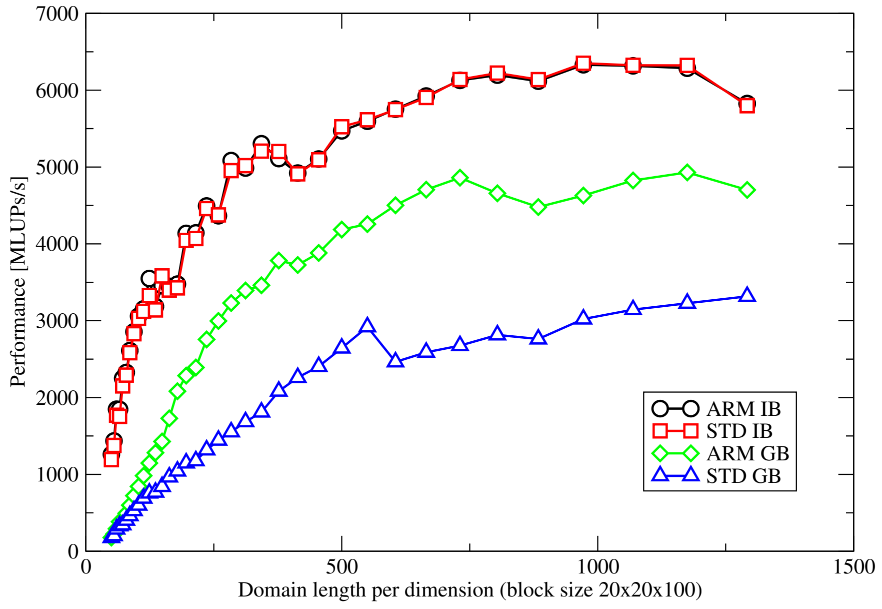

Figure 3.4: 3D Jacobi with 32 processes (4 Tinyblue nodes with 8 cores each). Mapping of ranks to subdomains done by MPI (STD) and manually (ARM - advanced rank mapping) on InfiniBand (IB) and Gigabit (GB) networks.

3.2 Overlap of communication and computation

MPI features, among others, the non-blocking communication functions `MPI_Isend()` and `MPI_Irecv()`. Unlike their blocking counterparts, which return not before buffers can be used/overwritten safely (not: have been transferred), calls to those functions return immediately. The initiator can proceed while communication may take place in background. This behavior avoids deadlocks in some situations and allows the MPI library to reschedule outstanding communications. It is commonly assumed that it can be utilized to overlap computation and communication without the effort of OpenMP/MPI hybrid programming (section 4.2.2). This is, however, by no means default behavior as experiments show. To which degree concurrency can be achieved depends on the MPI implementation.

The following code fragment shows how the overlapping capabilities of a system can be probed. One process issues a non-blocking `MPI_Irecv()` followed by a call to a working routine, which is either a simple loop with purely computational load or a memory bandwidth bound streaming code. The other process acts as communication partner and executes a blocking `MPI_Send()`. It turned out that the somewhat unintuitive arrangement of MPI calls in the code - receive before send - runs slightly faster on all used systems, especially Cray XT4, and therefore is preferred. For XT4 this is plausible as it is explicitly optimized for message preposting [20]. Finally, the call to `MPI_Wait()` ensures that MPI communication completed, i.e. the buffer has actually been transferred. So time measurement ends after both, computation and communication, have finished. The benchmark executes this code for a range of work amounts and a fixed communication buffer size.

Listing 3.1: Computation/communication overlap benchmark

```
MPI_Barrier(MPI_COMM_WORLD);
if(rank==0)
{
  stime = MPI_Wtime();
  MPI_Irecv(buf,bufsize,MPI_DOUBLE,1,0,MPI_COMM_WORLD,request);
  delayTime = do_work(amount);    // stream or div
  MPI_Wait(request,status);
  etime = MPI_Wtime();
  cout << "X= " << delayTime << " Y= " << etime-stime << endl;
}
else
{
  MPI_Send(buf,bufsize,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
}
MPI_Barrier(MPI_COMM_WORLD);
```

A formal overlap of communication and computation occurs, if the measured overall time is lower than the sum of both executed separately. An easy graphical interpretation of figure 3.5 is therefore that values resulting from an overlapping measurement are below a machine specific limit. This limit is a straight with a gradient of 1 and an offset of pure communication time. MPI point-to-point bandwidth (determined with IMB benchmarks, see section 1.2.2) can be used to calculate the offset. It perfectly matches measured values. The following table gives an overview to measured point-to-point bandwidths and resulting transfer times.

Machine	Interconnect	MPI Bandwidth	Transfer Time (80 MB)
Woody	DDR InfiniBand	1.5 GB/s	0.053 sec
XT4	SeaStar2	1.6 GB/s	0.050 sec
Tinyblue	QDR InfiniBand	3.0 GB/s	0.027 sec

Now it becomes obvious which MPI libraries offer actual non-blocking communication. First of all it is Cray's MPI library (based on MPICH-2 [21]) used in the XT4 system (blue line) that showed overlapping behavior for all scenarios tested. Overall time is constant until the time for computation exceeds communication time, resulting in a characteristic kink at $x = y = 0.05s$. From here, overall time grows directly with computation time. Prior to that point computation did not account to overall time, it could be overlapped completely by communication.

For OpenMPI (version 1.4) [22] on Tinyblue (violet line), only the non-blocking send is performed asynchronously, whereas there is no overlap for non-blocking receive. With an

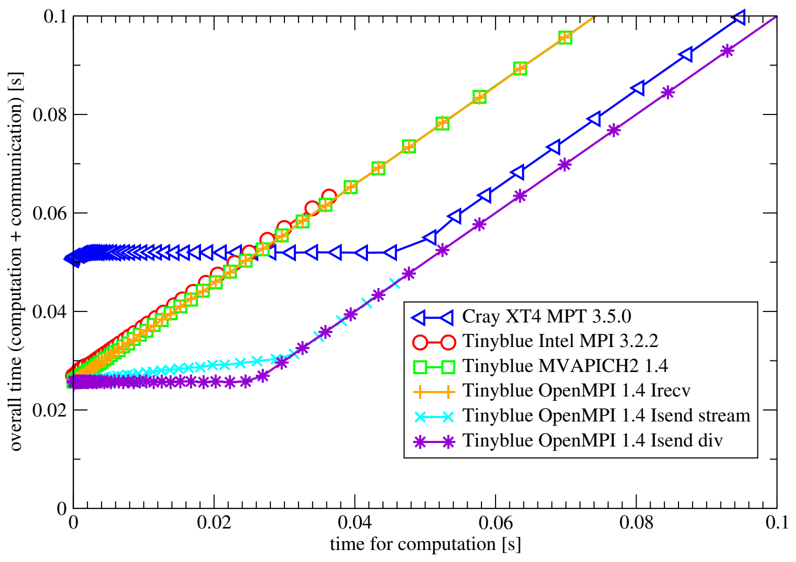

Figure 3.5: Overall execution time versus computational time for the computation/communication overlap benchmark as shown in listing 3.1. Buffer size for non-blocking MPI communication is 80 MB. Computation is either a compute-bound division (div) or memory-bound array streaming, both with workload varying in a sensible interval. In order to fully exhaust memory bus 4 threads are started for streaming.

interconnect twice as fast as the Cray network the break is certainly much earlier for this curve, but at the analog position. When changing computation from a simple division to streaming an array from memory a slight difference turns up (turquoise line). Due to increasing size of the streaming array and therewith pressure on the memory subsystem, duration of the MPI communication rises weakly.

For Intel MPI (version 3.2.2) [24] and MVAPICH2 (version 1.4) [23] overlap of communication and computation could not be found. The kink-less graph for this measurement reflects this. Overall time is directly related to computational effort. Measurement data for the Woodcrest cluster is analog to Tinyblue and thus omitted in the figure for clarity.

The advantage of actual non-blocking communication is depicted in the figure: Despite only half network bandwidth compared to Tinyblue, XT4 shows similar values for overall time when computation time exceeded communication time. If overlapped communication can be employed in a real algorithm, one may get away with a slow network and still reach performance levels similar to what can be achieved with a much faster network and no overlap.

3.3 Overlap of intra-node and cross-node communication

In this section we examine, under which circumstances intra-node (IN) and cross-node (CN) MPI communications can overlap. This kind of concurrency would be beneficial e.g. for pure MPI stencil codes on multi-core systems, since halo exchanges between (CS and CN) neighbors are less expensive if they overlap. Note that intra-node communication is implemented solely as cross-socket (CS) communication throughout this chapter, without loss of generality. The following code snippet shows the benchmark used.

Listing 3.2: CS/CN communication overlap benchmark

```
MPI_Barrier(MPI_COMM_WORLD);
output_rank = CN_recv_rank;
start = MPI_Wtime();

// cross-socket
if(rank == CS_recv_rank)
{
  MPI_Recv(buf1,bufsize,MPI_DOUBLE,CS_send_rank,0,MPI_COMM_WORLD,status);
}
else if(rank == CS_send_rank)
{
  MPI_Send(buf1,bufsize,MPI_DOUBLE,CS_recv_rank,0,MPI_COMM_WORLD);
}

// cross-node
```

```
else if(rank == CN_recv_rank)
{
  stime = MPI_Wtime();
  MPI_Recv(buf2,var_buf,MPI_DOUBLE,CN_send_rank,
           0,MPI_COMM_WORLD,status+1);
  etime = MPI_Wtime();
  CN_time = etime-stime;
}
else if(rank == CN_send_rank)
{
  MPI_Send(buf2,var_buf,MPI_DOUBLE,CN_recv_rank,0,MPI_COMM_WORLD);
}

MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();
if(rank == output_rank)
  cout << "X= " << CN_time << " Y= " << end-start << endl;
```

Non-blocking MPI calls are used to transfer one buffer between two sockets of the same node and another buffer between two different nodes. The size of the CS buffer is static, whereas CN buffer size is varied in order to obtain an easy graphical interpretation of the benchmark results (see figure 3.6). The overall number of MPI processes started equals the number of physical cores available on the two nodes, but only four ranks are actually involved in benchmarking, the remaining ranks wait at a global barrier. This allows convenient selection of communication partners within the code. Standard rank placement and process pinning are used, so the location of a process can be determined by its rank number.

In figure 3.6, the sum of cross-node and intra-node communication time is plotted over cross-node buffer size. As cross-node communication underlies various influences, not duration but actual buffer size is the metric used for the x-axis. This ensures a comprehensible and well-defined presentation results. Cross-node buffer grows from 80 kilobyte to several hundred megabytes, intra-node buffer is of constant size. Again, changes in the slope of a curve provides information whether overlap is at hand.

Similar to the results from the previous section, the MPI library on the Cray XT4 system exhibits overlap of intra-node and cross-node communication (black line), albeit not that explicitly as in the previous section. The reason for this somewhat partial overlap is yet to be investigated.

Intel MPI was tested on Woody and Tinyblue with analog results. Overlap of cross-socket and cross-node communication is possible with Intel MPI for the scenario depicted above. However, one prerequisite has to be fulfilled. Rank 0 must not be involved in cross-socket communication, participation in cross-node communication is uncritical however. There is a possible relation to a phenomenon observed during multi-mode PingPing tests: One PingPing pair communicated cross-socket and the other cross-node. Whenever

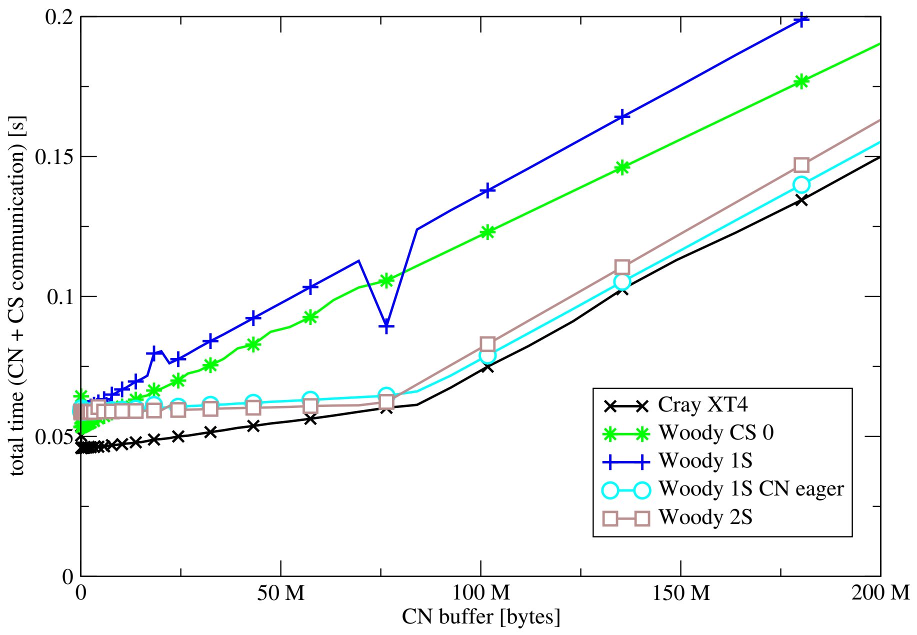

Figure 3.6: Overall execution time versus cross-node buffer size for the CS/CN communication overlap benchmark as shown in listing 3.2. Buffer size for cross-socket communication is 80 MB. Overlap occurs for all probed situations on the Cray system. For the Woodcrest cluster overlap depends on the number of communicating ranks, i.e. one sender (1S) or two separate senders (2S). No overlap could be observed with rank 0 participating in cross-socket communication.

both, rank 0 and 1, ran on the master node and therefore communicated cross-socket, a noticeable latency growth for each pair was discovered. Whatever the reason, the disadvantageous issue with rank 0 forms a severe restriction that can not generally be bypassed for real applications.

Previous measurements where performed using four processes, 2 senders and 2 receivers. As a subdomain has both, intra- and cross-node neighbors for common MPI parallel stencil codes, a scenario where a single process communicates with two partners deserves study. Therefore the benchmark code from above has to be adapted. Send operations are replaced by the non-blocking counterpart (including a `MPI_Waitall()` statement) and concentrated for a single rank.

```
MPI_Barrier(MPI_COMM_WORLD);
output_rank = CN_recv_rank;
start = MPI_Wtime();

// cross-socket receive
if(rank == CS_recv_rank)
{
  MPI_Recv(buf1,bufsize,MPI_DOUBLE,CS_send_rank,0,MPI_COMM_WORLD,status);
}

// send
else if(rank == send_rank)
{
  MPI_Isend(buf1,bufsize,MPI_DOUBLE,CS_recv_rank,0,MPI_COMM_WORLD,req);
  MPI_Isend(buf2,var_buf,MPI_DOUBLE,CN_recv_rank,0,MPI_COMM_WORLD,req+1);
  MPI_Waitall(2,req,status);
}

// cross-node receive
else if(rank == CN_recv_rank)
{
  stime = MPI_Wtime();
  MPI_Recv(buf2,var_buf,MPI_DOUBLE,CN_send_rank,
           0,MPI_COMM_WORLD,status+1);
  etime = MPI_Wtime();
  CN_time = etime-stime;
}

MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();
if(rank == output_rank)
  cout << "X= " << var_buf*8 << " Y= " << end-start << endl;
```

This code showed the same results as the 2:2 variant for the Cray system. The situation is more complex on Woody, however. Besides the circumstances concerning rank 0 which remain present, further conditions for overlapping communications emerged. It turned out that the message transfer protocol in use is of relevance.

Intel's MPI implementation (just as most other MPI libraries) supports two point-to-point protocols, *eager* and *rendezvous*. In eager mode "data is transferred to the receiver before a matching receive is posted" ([17] section 3.8, page 69). Therefore the receiver has to prepare an *eager buffer* at MPI initialization in order to support possible eager transfers. This makes eager communication fast but fairly memory intensive for large buffers. Rendezvous mode on the other hand implies preliminary negotiation of parameters (e.g. buffer size) between communication partners, and asynchronous end-to-end communication of data. With this synchronization overhead rendezvous mode is predestined for the transfer of larger buffers. The default point for eager/rendezvous crossover is at a buffer size of 256 kB [25].

Intel MPI provides a set of environment variables to configure the crossover point for various communication paths, e.g.

- `I_MPI_EAGER_THRESHOLD`
- `I_MPI_INTRANODE_EAGER_THRESHOLD`.

The first variable influences all communication paths, unless overwritten by a more specific like the latter.

These communication protocols have to be adjusted properly in order to achieve overlapping MPI communications with a single sender process. Overlap could be established solely for the situation where cross-node communication is in eager mode and cross-socket communication is in rendezvous mode for all buffer sizes. The latter is the default for these experiments, as a cross-socket buffer of 80 MB was used. But the requirement to transfer cross-node messages of up to several hundred megabytes in eager mode is anything but memory-efficient, since eager buffers in this scale have to be held for every participating process. Falling back to the default of 256 kB as maximum message size transferred in eager mode is not sufficient for real world applications. For the Jacobi solver benchmark this would restrict the size of one subdomain face to about 180 *double* elements per direction, leading to a maximum subdomain size of 45 MB.

3.4 Conclusions

A number measures have to be taken into account when optimizing pure MPI code for multi-core systems. Depending on the environment and the MPI implementation manual intervention might be inevitable. For instance, topology aware mapping of subdomains to nodes could not be determined for any of the MPI libraries probed. However, this issue can be corrected by replacing mechanisms for coordinate administration (e.g. `MPI_Cart_create()`).

Among all systems used for benchmarking, except for MVAPICH2 and the Cray MPI library non-blocking communication does not overlap with computation. This is contradictory to anticipated and common understanding, and a remedy can not be provided, except in the form of task mode hybrid programming (see section 4.2.2).

Finally, concurrent inter- and intra-node communication with Intel MPI is possible for limited situations. Constraints include avoiding rank 0 for cross-socket communication and forcing cross-node communication to use eager transportation mode. Again, cross-node and intra-node communication overlap for all probed situations on the Cray system.

4 MPI/OpenMP hybrid programming

The following chapter provides an insight into MPI/OpenMP hybrid programming. This is an approach towards developing applications tailored to modern clusters made up of shared-memory nodes connected with a high speed network (see figure 4.1). In order to fully utilize potentials of these hybrid machines, well-established techniques for both, distributed and shared memory computing (i.e. MPI and OpenMP), are combined.

Hybrid programming can be further subdivided according to the communication pattern implemented. *Vector mode* hybrid applications use MPI communication solely in serial regions, whereas this restriction is abolished for *task mode* hybrid codes. The first two sections give a theoretical background which is applied to a Jacobi solver benchmark afterwards.

4.1 Vector mode vs. task mode

4.1.1 Vector mode

The most straightforward approach towards hybrid programming is to start one MPI process per node which in turn starts a number of OpenMP threads. The number of threads is usually determined by the number of cores present within the node. OpenMP parallel regions are used for computation, whereas cross-node MPI communication is limited to serial regions. The resulting alternation of parallel and serial regions "resembles parallel programming on distributed-memory parallel vector machines" [28], explaining the denotation vector mode.

4.1.2 Task mode

Task mode retains the basic hybrid principle of MPI processes spawning OpenMP threads, but MPI communication is no longer restricted to serial regions. With every thread being able to communicate, sub-teams of threads can be defined and assigned to particular tasks like computation or communication. Unlike pure MPI implementations this permits an explicit overlap of communication and computation. Functional decomposition of threads introduces an MPI-like programming style. Workloads and loop boundaries have to be determined manually for each thread, as OpenMP work sharing directives would apply to

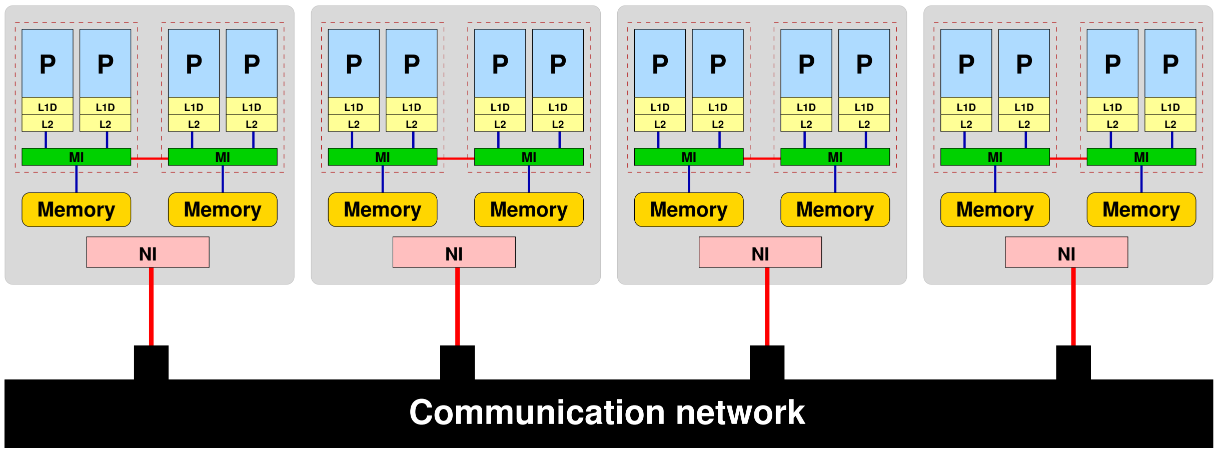

Figure 4.1: Typical hybrid cluster [11]. A high-speed network connects multi-socket nodes with multi-core processors. In this example each dual-core processor has its own physical memory, which is logically shared within the node (ccNUMA).

the entire thread team and are therefore not applicable within sub-teams. This could be bypassed with a nested parallel region, e.g. for all computing threads. Although nested parallelism conforms to the OpenMP standard and might work in specific environments, it is by no means portable and should therefore be avoided.

An elegant and more general solution is provided by OpenMP tasking (do not mix up task mode hybrid and OpenMP tasking). Instead of distributing e.g. loop iterations among threads as done with an OpenMP loop construct, a *OpenMP task* is created for the work done in each loop iteration. In OpenMP tasking terminology a task is a "specific instance of executable code and its data environment" [13]. After a task is created, it can be executed by any thread of the current parallel team. If no thread is available the task is queued until a thread reaches a *scheduling point*, e.g. a barrier or a `taskwait` region.

See the following code fragment for an example.

```
#pragma omp parallel
{
  int threadID = omp_get_thread_num();

  if(threadID == 0)
  {
    for(int i = 0; i < count; ++i)
    {
      #pragma omp task
      do_work(i);
    }
  }
  if(threadID == 1 || threadID == 2)
  {
    communicate(threadID);
  }
}  // implicit barrier
```

Here *count* tasks are created, each with a different input parameter. Note that task creation is performed by a single thread. Two threads (IDs 1 and 2) are dedicated to communication. How to decompose the thread team to functions (here: computation and communication) depends on application specific workload. A single communication thread was sufficient throughout all tests. In above example threads with IDs larger than 2 wait in the implicit barrier for tasks to be scheduled. Upon completion the threads of the communication sub-team join the pool of working tasks. Given a reasonable number of tasks, this alleviates the risk for load imbalance.

Since a mapping of tasks to executing threads is not defined due to dynamic task scheduling, NUMA aware data placement is not possible. It can therefore pay off to put some effort in manually assigning tasks to locality domains (LDs) of the NUMA system. The necessary steps are summarized below.

- The data set a task operates on has to be tagged with an ID representing the locality domain it resides in.
- For each locality domain a queue structure has to be defined and populated with the according data sets.
- A thread has to be able to determine the locality domain it is running in by its thread ID. This can be achieved by pinning threads to cores.
- Threads processing tasks dequeue data sets from the queue assigned to their locality domain until it is empty. Unless other queues are as well empty, processing non-local tasks (task stealing) might be sensible.

The topic is covered in detail by a paper on locality queues [18].

4.1.3 MPI thread compatibility

A fundamental prerequisite for hybrid programming as described in the previous sections is that the MPI library in use offers the required level of thread support. The current MPI standard (MPI version 2.2) [17] describes the following four levels.

- `MPI_THREAD_SINGLE`: Only one thread will execute.
- `MPI_THREAD_FUNNELED`: The process may be multi-threaded, but only the main thread will make MPI calls.
- `MPI_THREAD_SERIALIZED`: The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads.
- `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI, with no restrictions.

MPI functions `MPI_Init_thread()` and `MPI_Query_thread()` provide information about the multithreading capabilities of the running MPI library by returning one of the above levels.

In order to run a vector mode hybrid code, an MPI library that provides at least level `MPI_THREAD_FUNNELED` is sufficient, as communication is done in serial regions by the main thread only. Full threading support (`MPI_THREAD_MULTIPLE`) is necessary for task mode hybrid codes like the last example. Both threads of the communication sub-team must be able to use MPI functions simultaneously. As MPI communication functions address their counterpart solely by its rank, additional tagging of messages is required to identify the associated thread within the node. This is due to the application developer.

4.1.4 Benefits and drawbacks of MPI/OpenMP hybrid programming

This section gives a brief summary of possible benefits and drawbacks of hybrid programming compared to pure MPI programming on clusters of shared-memory nodes.

Benefits

Where a typical MPI application would start one process per available core, its hybrid correspondent would use one per node with the following positive effects. Less fragmentation of the domain is the result for the hybrid case, leading to a smaller number of larger subdomains. One direct advantage is the consolidation of communication. This is true for the case where a non-hybrid application transfers buffers so small that communication is bound to latency. With larger domain faces a hybrid code "rides the PingPong curve", i.e. bandwidth grows as the influence of communication overhead shrinks.

A further advantage is that intra-node communication becomes unnecessary in favor of shared memory access. As shown by the PingPong and PingPing tests in section 1.2.2, intra-node communication is by no means infinitely fast or free of cost. Moreover, overlap of intra-node and cross-node communication can not be safely depended on (see section 3.3).

The last advantage of coarser domain decomposition is that the mapping of ranks to subdomains is trivial, as only a single subdomain resides on a node.

Maybe the largest positive impact of hybrid programming is the possibility of an explicit overlap of communication and computation. It is no longer necessary to interrupt computation to synchronize with neighbors if communication and computation can be executed concurrently. Dedicating threads to communication is getting less influential with growing numbers of cores per node. This load imbalance can moreover be totally cleared by using techniques like OpenMP tasking.

Drawbacks

One obvious disadvantage of hybrid programming is coding complexity. No major modifications of existing MPI code might be required when introducing OpenMP parallel regions, but in the case of task mode hybrid programming without using OpenMP tasks, work has to be scheduled to threads manually, since OpenMP work-sharing directives are not applicable. If OpenMP tasks are in use, however, NUMA-aware memory placement causes additional administration effort due to a non-deterministic processing pattern.

A technical downside of OpenMP is the overhead resulting from thread startup. Negative influence is only perceptible for relatively small workloads. Another technical obstacle might be an MPI library with insufficient thread compatibility. However, Intel MPI showed no lack of thread support during our tests. In addition, the number of MPI implementations offering full thread support (`MPI_THREAD_MULTIPLE`) is growing [26].

4.2 Application to the Jacobi algorithm

4.2.1 Vector mode

Given an MPI parallel Jacobi solver and an MPI library with sufficient threading support, creating a vector mode hybrid code is a straightforward undertaking. In a nutshell, the loop nest iterating over the sub-domain of each process has to be parallelized with an OpenMP work-sharing construct. The number of threads participating in the parallel region can be specified via the environment variable `OMP_NUM_THREADS` or the library function `omp_set_num_threads()`. (For a brief introduction to OpenMP programming see section 1.3.1.) The following example code shows the OpenMP parallel kernel of a 3D Jacobi solver. Note that the loop nest does not iterate over single grid cells but groups of cells called blocks. (See section 1.3 for a short introduction to spatial blocking.) As one block is scheduled to a single core, optimal block dimensions assure maximum re-use of data from local cache.

```
#pragma omp parallel for schedule(static, 1)
for (int bi = 0; bi < numberOfLines; ++bi)
{
  for (int bj = 0; bj < numberOfColumns; ++bj)
  {
    for (int bk = 0; bk < numberOfDepths; ++bk)
    {
      ComputeBlock(grids[0]->Blocks()[bi][bj][bk]);
    }
  }
}
```

In combination with a static OpenMP schedule, above loop construct can be adapted for parallel page placement required on ccNUMA systems.

4.2.2 Task mode

Manual scheduling

In order to overlap communication and computation phases of a Jacobi solver, both tasks are assigned to thread groups within the current team. It turned out for the benchmark scenario that it is sufficient to dedicate a single thread to communication, as additional threads would step up load imbalance. A further improvement in terms of load balance and concurrency can be achieved by causing the communicating thread to update the very boundary cells it is supposed to send. The remaining threads, identified by their thread ID, update the bulk area of the domain. Scheduling blocks to threads has to be done manually, as a nested parallel region is ineligible (see previous section). The following code chunk shows an example for manual scheduling, which is executed by all but the communication thread. Boundary checks are omitted for brevity.

```
#pragma omp parallel
{
  int threadNum = omp_get_thread_num();
  int workers = omp_get_num_threads() - 1;
  if (threadNum < workers)
  {
    // resChunks is used for balanced distribution
    int resChunks = numberOfLines % workers;
    int chunkSize = numberOfLines / workers;
    int firstLine = 0;
    int lastLine = numberOfLines - 1;
    if (threadNum < resChunks)
    {
      ++chunkSize;
      firstLine = (threadNum) * chunkSize;
      lastLine = (threadNum + 1) * chunkSize - 1;
    }
    else
    {
      firstLine = (threadNum ) * chunkSize + resChunks;
      lastLine = (threadNum + 1) * chunkSize + resChunks - 1;
    }
    for (int bi = firstLine; bi <= lastLine; ++bi)
    {
      for (int bj = 0; bj < numberOfColumns; ++bj)
      {
        for (int bk = 0; bk < numberOfDepths; ++bk)
        {
          ComputeBlock(grids[0]->Blocks()[bi][bj][bk]);
        }
      }
    }
  }
  else  // calculate and communicate halo
}
```

Due to the reduced number of worker threads it can be advantageous to re-adjust block sizes in order to make overall block count an integral multiple of worker thread count. Again, this static scheduling construct can be re-used for NUMA-aware page placement.

OpenMP task scheduling

Manual distribution of work chunks to threads as described in the previous section is quite inflexible and bears the risk of load imbalance. In spite of larger overhead per iteration compared to a worksharing for loop, a task loop is the choice for situations that can not be expressed efficiently using a `parallel for` construct.

The following code fragment shows how OpenMP tasking can be applied to the Jacobi kernel from the previous examples.

```
#pragma omp parallel
{
  int threadnum = omp_get_thread_num();
  int workers = omp_get_num_threads() - 1;

  if (threadnum < workers)
  {
    if (threadnum==0)  // only a single thread creates tasks
    {
      for (int bi = 0; bi < numberOfLines; ++bi)
      {
        for (int bj = 0; bj < numberOfColumns; ++bj)
        {
          for (int bk = 0; bk < numberOfDepths; ++bk)
          {
            #pragma omp task
            ComputeBlock(grids[0]->Blocks()[bi][bj][bk]);
          }
        }
      }
    }
  }
  else  // calculate and communicate halo, join task processing afterwards
}
```

Task creation is done by a single thread. The most basic form of a task could be a single stencil update, but due to task start overhead a task should update a whole block of stencils. Spatial blocking is therefore essential for OpenMP tasking.

As assignment of tasks to threads is not predictable, placing memory pages to cores in a round-robin fashion is reasonable. However, the implementation of NUMA-aware task queuing [18] should be considered for the reasons mentioned above.

If all threads start at the same time, they all might communicate first (to update their halo creating high amounts of traffic on the network) and compute afterwards (leaving the network unused). This model offers the possibility to skew communication and computation phases of the threads against each other which might lead to a more balanced and therefore efficient usage of resources.

Performance

In this section we investigate whether the theoretical benefits of MPI/OpenMP hybrid programming can be verified with real world applications. One typical application is sparse matrix-vector multiplication (MVM). Due to its algorithmic complexity it is not appropriate as an introductory example in this context. For an in-depth study of MPI/OpenMP hybrid programming based on MVM we refer to [27]. To keep matters simple we stick to the Jacobi solver benchmark used throughout this work. However, this requires particular precautions.

As the dominance of communication over computation is not distinct enough for this type of application, optimizations regarding communication effort do not lead to significant performance improvements on modern high speed interconnects like InfiniBand. Of course, the ratio of communication to computation can be influenced by the size of the subdomains. But very small domains, which would emphasize communication against computation, bear the risk of inefficient computation due to OpenMP overhead. In order to actually demonstrate the influence of hybrid programming on the basis of a plain benchmark code, we throttle network speed by using Gigabit Ethernet instead of InfiniBand.

Figure 4.2 shows for the Townsend cluster that DDR InfiniBand offers a sustained Ping-Ping bandwidth of 1 GB/s and Gigabit Ethernet 90 MB/s. This factor of 10 is sufficient to establish a sensible ratio of communication to computation.

We opted for the Townsend cluster as a benchmark system, as it features the most steady Gigabit network among reasonable machine options. As Gigabit Ethernet is basically intended to serve as a management network in those systems, design and configuration turned out to be not appropriate for high throughput scenarios. Another advantage of this machine is its simple node topology. With only 2 cores per node, optimal mapping

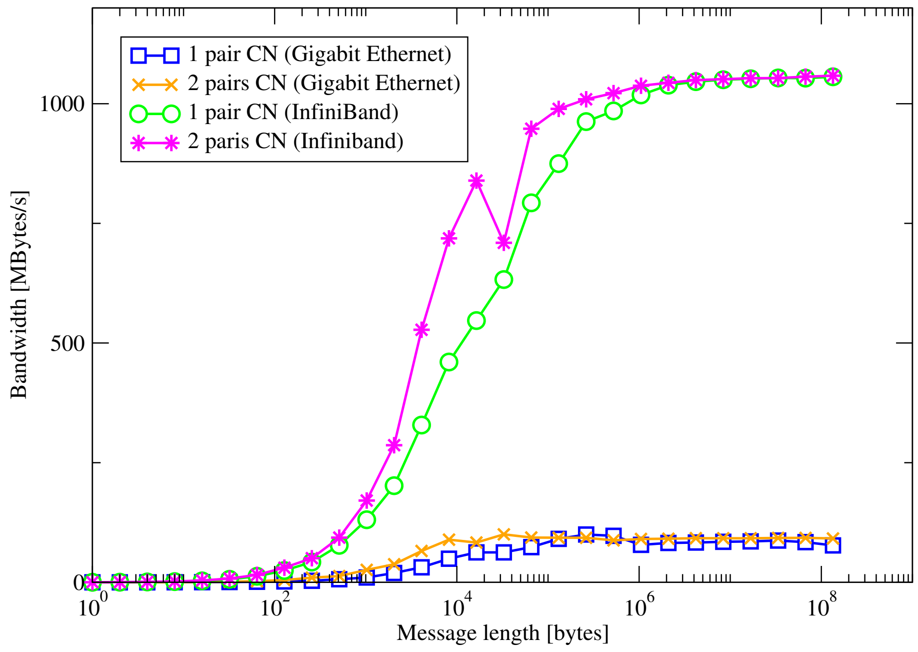

Figure 4.2: IMB PingPing on the RRZE Townsend Cluster. Double data rate (DDR) InfiniBand is compared to Gigabit Ethernet interconnect using 1 or 2 (multi-mode) PingPing pairs. Sustained bandwidth is 1 GB/s for InfiniBand and 90 MB/s for Gigabit Ethernet.

of ranks to subdomains is given for vanilla MPI codes using `MPI_Cart_create()` (see figure 3.3). Measures as described in section 3.1 are therefore not necessary.

Four implementations of a Jacobi solver with spatial blocking are investigated in a weak scaling study on 1 to 32 nodes. Subdomain size for one node is 120 *double* elements in each of the three dimensions. Benchmarks include the following versions:

1. `MPI`: Pure MPI implementation with two processes per node.
2. `hybrid vector mode`: A single process per node spawns two OpenMP threads for computation. After each sweep, halo communication is done in a serial region.
3. `hybrid task mode (manual)`: A single process per node spawns two OpenMP threads, one dedicated to halo computation and communication, the other dedicated to computation of bulk blocks. Tasks are therefore scheduled manually by thread ID.
4. `hybrid task mode (OpenMP tasks)`: Like the previous version, with the following modification: The bulk thread creates an OpenMP task from each loop iteration. After finishing communication, the halo thread joins task processing. Scheduling of computational work to threads is load dependent.

Looking at single node performance (left of figure 4.3), the penalty for statically splitting off one thread for communication in the manual task mode case becomes apparent. Performance trade-off is not 50 %, as one might suspect seeing utilization of only one of

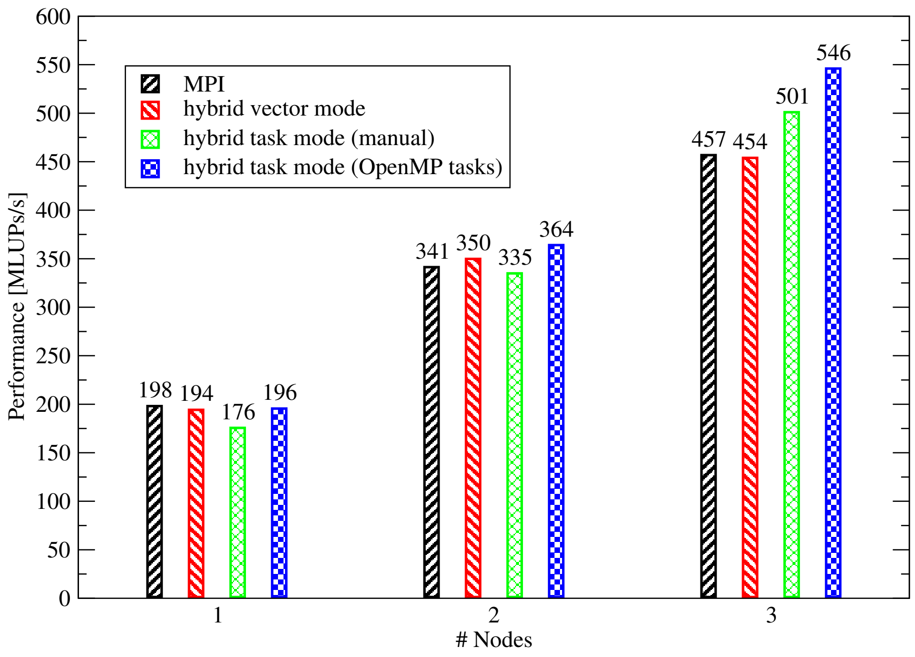

Figure 4.3: 3D Jacobi with spatial blocking on 1 to 3 nodes of the Townsend Cluster (2 cores per node) and Gigabit Ethernet interconnect. Domain size for each node is 120^3 `double` elements (weak scaling). A pure MPI solver with two processes per node is compared to three hybrid implementations with one process starting two threads per node. The manual task mode version dedicates one thread to halo computation and communication, and one to bulk computation. Computation of bulk blocks is scheduled to threads with OpenMP tasking in the second task mode case.

two cores available, but mere 10 %. This is due to the fact that memory bandwidth is almost charged to capacity from a single core in this system. Further concurrent streams as used in the three other versions gain only little additional bandwidth. When communication becomes involved, the effect alleviates (see middle of figure 4.3), and finally vanishes (right). Furthermore, explicit overlap of communication and computation (as for the hybrid case) starts to pay off.

The gap between task mode hybrid and both other versions keeps on widening during transition to larger node counts (see figure 4.4), resulting in up to 75 % higher performance compared to the pure MPI implementation. Overlap of communication and computation is the reason, no matter if OpenMP tasking or manual thread scheduling is utilized. The similarity of both task mode measurements for more than three nodes has its source in the machine's configuration with two cores per node. For two and three nodes, communication effort is quite small, allowing the communicator thread to compute bulk tasks as well, but in the static scheduling scenario it is idle after communicating. With growing overall communication volume, the task mode variant using OpenMP tasking roughly equals "manual" task mode in work distribution and performance.

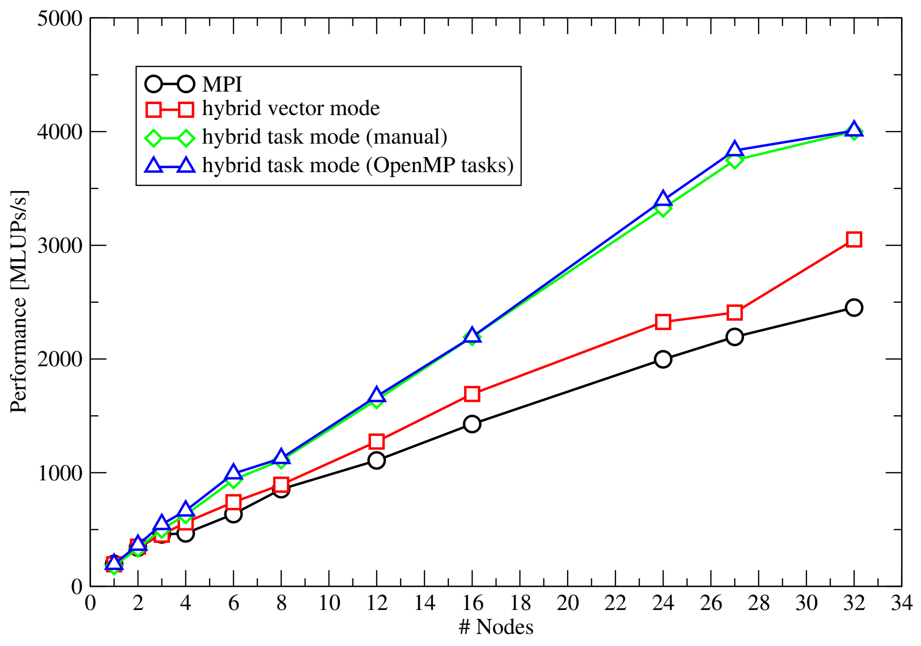

Figure 4.4: 3D Jacobi with spatial blocking on up to 32 nodes of the Townsend Cluster (2 cores per node) and Gigabit Ethernet interconnect. Caption of figure 4.3 applies.

Apart from the noticeable performance impact achieved from explicitly overlapping communication and computation, improvements can as well be observed for the vector mode hybrid approach. A general benefit of 15 to 20 % compared to pure MPI is apparent for more than 3 nodes. It can be attributed to the lack of explicit intra-node communication in the hybrid case, due to utilization of shared memory. As seen in section 3.3, cross-socket and cross-node MPI communication do not overlap in usual situations. This extends communication time and therefore reduces overall performance. However, for 8 and 27 nodes vector mode is only 5 resp. 10 % better than pure MPI.

In both cases the communication pattern changes significantly compared to the next smaller node count. Eight is the smallest number of nodes where a typical domain decomposition yields more than one subdomain in each direction. This leads to communication in each of the three instead of merely two dimensions.

The maximum count of six possible neighbors is given for a subdomain located in the center of a domain decomposed into at least three elements per dimension. This is the case for 27 nodes, explaining the almost stagnating performance at this point. The graph for the pure MPI solver does not show corresponding alterations. A possible explanation for this is a different decomposition, as the number of subdomains has to be scaled up by the number of cores per node, which in turn affects the communication pattern.

5 Conclusion and outlook

5.1 Results

This thesis has analyzed four possible programming models for hybrid "commodity" hardware, i.e. clustered multi-socket multi-core compute nodes:

- *Pure MPI*
- *Hybrid vector mode*, where MPI communication occurs only outside OpenMP-parallel regions
- *Hybrid task mode with manual work distribution*, where communication can be overlapped with computation but most of the convenient OpenMP worksharing functionality is lost
- *Hybrid task mode with OpenMP tasking*, where the new (as of OpenMP 3.0) tasking constructs enable better utilization of resources, dynamic load balancing, and natural work sharing.

First, using results from low-level bandwidth and latency benchmarks, performance models for the three-dimensional Jacobi solver were developed. These models enabled a qualitative analysis of the expected gains of hybrid parallelization versus pure MPI. Since it does not make sense to compare a mediocre MPI implementation with highly optimized hybrid codes, an attempt was made to arrive at a "perfect" pure MPI Jacobi code. Along the way some typical myths about MPI programming were dispelled:

- Mapping of MPI ranks to subdomains is suboptimal with current MPI implementations. Not even the number of running processes per compute node is taken into consideration by the topology functions, leading to cross-node communication volumes that are larger than they would have to be. This could be corrected by substituting some MPI functionality with an optimized algorithm that performs a rank-subdomain mapping which minimizes cross-node traffic.
- Truly asynchronous MPI communication using non-blocking point-to-point functions is only available in very few MPI libraries. The common case is to handle non-blocking communication only when the program is inside the library.
- Overlap between intra-node and cross-node point-to-point communication is only possible under rare circumstances, which can usually not be established in stencil codes.

Especially the first two observations lead to the assumption that there may be large benefits from hybrid parallelization.

Turning to hybrid implementations of the 3D Jacobi solver, vector mode and task mode implementations were compared. It was shown that task mode, due to its capability of overlapping computation with communication even without direct MPI support, can significantly outperform vector mode and also pure MPI on slow networks. The additional gain from using OpenMP tasking constructs depends crucially on the bandwidth saturation properties of the processors under consideration.

5.2 Future work

The results presented in this thesis could be applied and extended in a variety of ways. First, a more thorough coverage of recent MPI implementations should give a coherent and timely overview on the capabilities of MPI libraries to overlap communication with computation and several communication processes with each other. While this thesis concentrated mainly on the Intel compiler, other popular compilers like, e.g., the GCC should be benchmarked with regard to performance of tasking constructs and general OpenMP overhead, which may impede hybrid performance.

Second, the hybrid task mode paradigm could be applied to other, more realistic stencil codes like the lattice Boltzmann algorithm (LBM) in order to show its advantages for real applications. Beyond fluid dynamics applications, sparse eigenvalue solvers could also be a worthwhile optimization target; although it was shown already in 2003 [27] that hybrid task mode is beneficial for this kind of algorithm, a new evaluation with regard to modern multi-core systems seems in order.

Finally, the implications of OpenMP tasking constructs when running parallel programs on ccNUMA hardware are far from fully understood. Along the lines of the basic analysis in Ref. [18], a NUMA-optimized hybrid task mode code could be developed that shows better performance on the ccNUMA compute nodes that are common in all commodity clusters today.

Bibliography

[1] G. E. Moore: *Cramming more components onto integrated circuits.* Electronics **38(8)**, (1965) 114-117.

[2] `http://www.openmp.org`

[3] `http://www.mpi-forum.org`

[4] `http://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/systeme/woodcrest-cluster.shtml`

[5] `http://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/systeme/tinyblue-cluster.shtml`

[6] `http://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/systeme/ia32-cluster.shtml`

[7] `http://www.nersc.gov/nusers/systems/franklin/about.php`

[8] Intel Corporation: *Intel MPI Benchmarks: User Guide and Methodology Description.* Version 3.2, August 2008.
`Intel Corporation Document Number: 320714-001`

[9] Georg Hager: *IMB multi-mode Ping-Pong demystified?* Retrieved October 11, 2009.
`http://www.blogs.uni-erlangen.de/hager/stories/1119/`

[10] G. Hager, H. Stengel, T. Zeiser, G. Wellein: *RZBENCH: Performance evaluation of current HPC architectures using low-level and application benchmarks.* In: S. Wagner et al. (Eds.), High Performance Computing in Science and Engineering, Garching/Munich 2007. Transactions of the Third Joint HLRB and KONWIHR Status and Result Workshop, Dec 3-4, 2007, LRZ Garching, Springer, ISBN 978-3-540-69181-5 (2009) 485-501.
`http://arxiv.org/abs/0712.3389`

[11] G. Hager, G. Wellein: *Introduction to High Performance Computing for Scientists and Engineers* (Chapman & Hall / CRC Computational Science), 2010.

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick: *Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures.* In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Austin, Texas, November 15 - 21, 2008). Conference on High Performance Networking and Computing. IEEE Press, Piscataway, NJ, 1-12.

[13] *OpenMP Application Program Interface.* Version 3.0, May 2008.
`http://www.openmp.org/mp-documents/spec30.pdf`

[14] `http://www.top500.org/connfam/6`

[15] J. D. McCalpin: *Memory bandwidth and machine balance in current high performance computers.* IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, (1995) 19–25.

[16] M. Wittmann, G. Hager and G. Wellein: *Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory.* Workshop on Large-Scale Parallel Processing (LSPP), IPDPS 2010, April 23rd, 2010, Atlanta, GA.
`http://arxiv.org/abs/0912.4506`

[17] *MPI: A Message-Passing Interface Standard.* Version 2.2 (September 4, 2009).
`http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`

[18] M. Wittmann, G. Hager: *A Proof of Concept for Optimizing Task Parallelism by Locality Queues.* (2009).
`http://arxiv.org/abs/0902.1884`

[19] `http://www.open-mpi.org/software/plpa/`

[20] `http://www.nersc.gov/nusers/systems/franklin/programming/`

[21] `http://http://www.mcs.anl.gov/research/projects/mpich2/`

[22] `http://www.open-mpi.org/`

[23] `http://mvapich.cse.ohio-state.edu/`

[24] `http://software.intel.com/en-us/intel-mpi-library/`

[25] Intel Corporation: *Intel MPI Library for Linux* OS Reference Manual.* Version 3.2, Update 1, April 2009.
`http://software.intel.com/file/15429`

[26] R. Thakur, W. Gropp: *Test Suite for Evaluating Performance of MPI Implementations That Support $MPI\_THREAD\_MULTIPLE$.* Proc. of the 14th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2007), September 2007, pp. 46-55.
`http://www.mcs.anl.gov/~thakur/papers/thread-tests.pdf`

[27] R. Rabenseifner, G. Wellein: *Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures.* International Journal of High Performance Computing Applications **17(1)**, (2003) 49-62.

[28] G. Hager, G. Jost, R. Rabenseifner: *Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes.* Proceedings of CUG09, May 4-7 2009, Atlanta, GA.
`http://www.cug.org/7-archives/previous_conferences/CUG2009/`
`bestpaper/9B-Rabenseifner/rabenseifner-paper.pdf`

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Nürnberg, 15. März 2010