Quantum Graph Partitioning

HPCSE II Report

Matthias Untergassmair
Project partner: Pascal Iselin

Swiss Federal Institute of Technology (ETH) Zürich

August 14, 2015

This project report presents the theoretical background and implementational ideas of our HPCSE II final project about quantum graph partitioning and gives a short analysis and discussion of the benchmark results.
A tailored Hamiltonian is used to characterize an Ising spin system whose lowest energy solution is equivalent to the optimal solution of the abstract graph partitioning problem.
The performance analysis was carried out on up to 32 compute nodes of the Piz Daint supercomputer at *Swiss National Supercomputing Centre* (CSCS) in Lugano, but the code should be portable on all platforms that support MPI, OpenMP and AVX Intel SIMD instructions.

1 Theoretical background

1.1 Graph partitioning problem

The graph partitioning problem is given as follows:

> *"Let us consider an undirected graph $G = (V, E)$. with an even number $N = |V|$ of vertices. We ask: what is a partition of the set V into two subsets of equal size $N/2$ such that the number of edges connecting the two subsets is minimized?"*

As shown in [1], the graph partitioning problem (which is $\mathcal{NP}$ hard) can be directly mapped to the physical problem of the Ising spin glass:

> We will place an Ising spin $s_v = \pm 1$ on each vertex $v \in V$ on the graph, and we will let $+1$ and -1 denote the vertex being in either the $+$ set or the - set.

In order to find a solution to the graph partitioning problem, we can canstruct a Hamiltonian H_P consisting of two components:

$$H_P = H_A + H_B$$

$$H_A := A\left(\sum_{i=1}^{N} s_i\right)^2 \qquad \text{(enforces equal partitioning of } + \text{ and } - \text{ spins)}$$

$$H_B := B \sum_{(u,v)\in E} \frac{1 - s_u s_v}{2} \qquad \text{(punishment for bonds between different sets)}$$

with the condition

$$\frac{A}{B} \geq \frac{\min\{2\Delta, N\}}{8}$$

ensuring that the minimization of the number of bonds between the two subsets does not conflict with the condition of having two equally large subsets.

1.2 Quantum Adiabatic Optimization

For two non-commuting Hamiltonians H_0, H_P where the ground state of H_0 is "easy" and known, we can construct a mixed Hamiltonian H

$$H(t) = \left(1 - \frac{t}{T}\right) H_0 + \frac{t}{T} H_P \tag{1}$$

By the *adiabatic theorem of quantum mechanics*, for large enough[1] T the following holds:

[1] Large enough T means that the time steps must be choosen small enough to avoid the Landau-Zener transition from the ground state to the first excited state

If we prepare a quantum system to be in the ground state of H_0, and then propagate it with the hybrid hamiltonian $H(t)$, the system will remain in the ground state of $H(t)$ and finally end up in the ground state of H_P.

In particular, we can choose H_0 to consist of transverse magnetic fields, i.e.

$$H_0 = -h_0 \sum_{i=1}^{N} \sigma_i^x$$

Then, the ground state is an equal superposition of all possible states in the eigenbasis of $H_P(\sigma_1^z, \ldots, \sigma_N^z)$[2].

Note that s_i denotes the spin at site i and is measured using the operator σ_i^z H_B can also be expressed as

$$H_B = B \sum_{i,j=1}^{N} J_{ij} \frac{1 - s_i s_j}{2} \qquad \text{where} \quad J_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

J_{ij} is a general coupling constant that can connect any two nodes in the system (Ising spin glass). Thus, we can rewrite the Hamiltonian 1 as

$$H(t) = \left(1 - \frac{t}{T}\right)(-h_0) \sum_{i=1}^{N} \sigma_i^x + \frac{t}{T} \left(A \left(\sum_{i=1}^{N} \sigma_i^z \right)^2 + B \sum_{i,j=1}^{N} J_{ij} \frac{1 - \sigma_i^z \sigma_j^z}{2} \right) \tag{2}$$

Time evolution We have seen that the graph partitioning problem can be solved by finding the equilibrium state of the quantum Ising model. To do this, we need to time-evolve the system until it reaches an equilibrium state.

We know that the time evolution of a wave function $|\psi\rangle$ is given by

$$|\psi(T)\rangle = \hat{U}(T, t_0) \, |\psi(t_0)\rangle \tag{3}$$

[2]Measurement in x (σ^x) leads to maximal uncertainty in z (σ^z)

where $\hat{U}$ is the evolution operator given as

$$\hat{U}(T, t_0 = 0) := \exp\left\{-\frac{\imath}{\hbar}\int_{t_0=0}^{T} H(\tau)\,\mathrm{d}\tau\right\} \approx \exp\left\{-\frac{\imath}{\hbar}\Delta t\sum_{k=1}^{M} H(t_k = k\Delta t)\right\} \tag{4}$$

$$\approx \prod_{k=1}^{M}\exp\left\{-\frac{\imath}{\hbar}\Delta t H(k\Delta t)\right\} \tag{5}$$

$$= \prod_{k=1}^{M}\exp\left\{-\frac{\imath}{\hbar}\Delta t\left[\left(1-\frac{k\Delta t}{T}\right)(-h_0)\sum_{i=1}^{N}\sigma_i^x + \frac{k\Delta t}{T}\left(A\left(\sum_{i=1}^{N}\sigma_i^z\right)^2 + B\sum_{i,j=1}^{N}J_{ij}\frac{1-\sigma_i^z\sigma_j^z}{2}\right)\right]\right\} \tag{6}$$

$$= \prod_{k=1}^{M}\exp\left\{-\frac{\imath}{\hbar}\Delta t\left[h_0\left(\frac{k}{M}-1\right)\sum_{i=1}^{N}\sigma_i^x + \frac{k}{M}\left(A\left(\sum_{i=1}^{N}\sigma_i^z\sum_{j=1}^{N}\sigma_j^z\right) + B\sum_{i,j=1}^{N}J_{ij}\frac{1-\sigma_i^z\sigma_j^z}{2}\right)\right]\right\} \tag{7}$$

$$= \prod_{k=1}^{M}\exp\left\{-\frac{\imath}{\hbar}\Delta t\left[h_0\left(\frac{k}{M}-1\right)\sum_{i=1}^{N}\sigma_i^x + \frac{k}{M}\sum_{i,j=1}^{N}\left(A\sigma_i^z\sigma_j^z + BJ_{ij}\frac{1-\sigma_i^z\sigma_j^z}{2}\right)\right]\right\} \tag{8}$$

$$\approx \prod_{k=1}^{M}\left[\prod_{i=1}^{N}E_0^{(i)}\cdot\prod_{i,j=1}^{N}E_P^{(i,j)}\right] \tag{9}$$

where $t_k := k\cdot\Delta t$ are the discretized time points and $\Delta t := \frac{T}{M}$ and

$$E_0^{(i)} = \exp\left\{-\frac{\imath}{\hbar}\Delta t\left(\frac{k}{M}-1\right)h_0\,\sigma_i^x\right\}$$
$$E_P^{(i,j)} = \exp\left\{-\frac{\imath}{\hbar}\Delta t\frac{k}{M}\left(A\sigma_i^z\sigma_j^z + BJ_{ij}\frac{1-\sigma_i^z\sigma_j^z}{2}\right)\right\}$$

Finally, to compute $E_0^{(i)}$ and $E_P^{(i,j)}$ we first choose the basis

$$\{|\uparrow_i\rangle, |\downarrow_i\rangle\} \qquad \text{for } E_0^{(i)}$$
$$\{|\uparrow_i\uparrow_j\rangle, |\uparrow_i\downarrow_j\rangle, |\downarrow_i\uparrow_j\rangle, |\downarrow_i\downarrow_j\rangle\} \qquad \text{for } E_P^{(i,j)}$$

leading to the matrix representation

$$\sigma_i^x = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \qquad\qquad \sigma_i^z = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

$$\sigma_i^z \sigma_j^z = \sigma_i^z \otimes \sigma_j^z = \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & +1 \end{pmatrix}$$

Note that $(\sigma_i^x)^2 = \mathbb{1}$. We can use the definition $\exp\{x\} := \sum_{k=0}^{\infty} \frac{x^k}{k!}$ to get

$$\exp\{i\xi \cdot \sigma_i^x\} = \sum_{k=0}^{\infty} \frac{(\imath\xi \cdot \sigma_i^x)^k}{k!} \tag{10}$$

$$= \sum_{k=0}^{\infty} \frac{(\imath\xi \cdot \sigma_i^x)^{2k}}{(2k)!} + \sum_{k=0}^{\infty} \frac{(\imath\xi \cdot \sigma_i^x)^{2k+1}}{(2k+1)!} \tag{11}$$

$$= \sum_{k=0}^{\infty} \frac{(-1)^k \xi^{2k} (\sigma_i^x)^{2k}}{(2k)!} + \imath \sum_{k=0}^{\infty} \frac{(-1)^k \xi^{2k+1} (\sigma_i^x)^{2k+1}}{(2k+1)!} \tag{12}$$

$$= \mathbb{1} \sum_{k=0}^{\infty} \frac{(-1)^k \xi^{2k}}{(2k)!} + \imath\sigma_i^x \sum_{k=0}^{\infty} \frac{(-1)^k \xi^{2k+1}}{(2k+1)!} \tag{13}$$

$$= \mathbb{1} \cos\xi + \imath\sigma_i^x \sin\xi \tag{14}$$

Since $\sigma_i^z \sigma_j^z$ is already diagonal, it can be exponentiated by exponentiating each diagonal element individually.

We finally define

$$\alpha_k = -\frac{1}{\hbar}\Delta t \frac{k}{M} A$$
$$\beta_k^{(i,j)} = -\frac{1}{\hbar}\Delta t \frac{k}{M} (BJ_{ij} - A)$$
$$\gamma_k = -\frac{1}{\hbar}\Delta t \left(\frac{k}{M} - 1\right) h_0$$

Then the matrix representation of the operators $E_0^{(i)}$ and $E_P^{(i,j)}$ is

$$\Rightarrow \quad E_0^{(i)} = \exp\{\imath\gamma_k \sigma_i^x\} = \begin{pmatrix} \cos\gamma_k & \imath\sin\gamma_k \\ \imath\sin\gamma_k & \cos\gamma_k \end{pmatrix} \tag{15}$$

$$E_P^{(i,j)} = \exp\left\{-\frac{\imath}{\hbar}\Delta t \frac{k}{M}\left(A\sigma_i^z\sigma_j^z + BJ_{ij}\frac{1-\sigma_i^z\sigma_j^z}{2}\right)\right\}$$

$$= \begin{pmatrix} \exp\{\imath\alpha_k\} & 0 & 0 & 0 \\ 0 & \exp\left\{\imath\beta_k^{(i,j)}\right\} & 0 & 0 \\ 0 & 0 & \exp\left\{\imath\beta_k^{(i,j)}\right\} & 0 \\ 0 & 0 & 0 & \exp\{\imath\alpha_k\} \end{pmatrix} \tag{16}$$

1.3 Initial state

The ground state of H_0 is given by

$$\bigotimes_{i=1}^{N} [|\uparrow_i\rangle - |\downarrow_i\rangle] \overset{N=2}{=} |\uparrow\uparrow\rangle - |\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle + |\downarrow\downarrow\rangle$$

We notice that we have a negative sign for an odd number of down spins. We represent up $= |\uparrow\rangle = 1$ and down $= |\downarrow\rangle = 0$ in our code.

2 Implementation

In the following, we will denote the number of nodes in the graph by N and the number of basis elements (i.e. the dimension) of the resulting quantum space by dim.
In our implementation, we interpreted each Bit of a 4Byte (32Bit) unsigned integer as an *up* $\uparrow$ or *down* $\downarrow$ quantum Ising spin (corresponding to 1 and 0 respectively).[3] In particular, each of the $dim = 2^N$ basis states has a unique `unsigned int` index associated with it.
Due to the exponential growth of the basis set, runtime and memory bottlenecks will occur even for moderately small N. Therefore, the aim of our parallel implementation was twofold: Firstly, we needed to extend the available memory by distributing our program over several nodes (MPI parallelism, 2.3), and secondly we tried to reduce computation time in order to achieve a feasible time to solution (MPI 2.3, OMP 2.4, SIMD 2.5).

Considering our implementation of the algorithm, with three vectors `psi_` (two for calculation results and one for sending and receiving data from other nodes), we can make the following simple consideration in order to determine the memory bottleneck of the program on a single node:
Denote by X the available memory on a single node. We know that a complex float, which is needed to store the complex coefficients of our wave vector, consists of two single floats with 4 Bytes each.

$$X \geq 3 \cdot 2 \cdot 4B \cdot 2^N = 24 \cdot 2^N \tag{17}$$

For a machine with 32GB of memory per node, like the Piz Daint (see subsection 2.1), memory limitations would occur at

$$\begin{aligned} 32 \cdot 10^9 &\geq 3 \cdot 2 \cdot 4 \cdot 2^N = 24 \cdot 2^N \\ \Rightarrow \quad N &\leq \frac{\log(\frac{32}{24} \cdot 10^9)}{\log 2} \approx 30.3 \end{aligned}$$

so the memory limit is at 30 nodes

It should also be noted here that the presented algorithm and code cannot compete with other graph algorithms and should not be used for partitioning graphs in practice. Rather, it can be viewed as a numeric quantum physics experiment that can also be easily extended to different quantum systems (by simply changing the Hamiltonian that characterizes the system).

[3] In fact, our current implementation is limited to a maximum of 32 graph nodes and both the MPI and AVX implementations heavily rely on the fact that $N \leq 32$.

2.1 The Platform - Piz Daint

Our benchmarks and plots were created by running the code on the *Piz Daint*, a Cray XC30 supercomputer at the CSCS in Lugano, Switzerland.
The system has a total of 5272 compute nodes each of which consisting of a 8-core 64-bit Intel SandyBridge CPU and a NVIDIA Tesla K20X GPU and equipped with 32GB of host memory with memory bandwidth 51.2GB/s [2].
This system can reach a theoretical peak performance of 7.787Petaflops and currently ranks 6th in the ranking of the worlds fastest supercomputers [3].

However, the presented C++ program should be portable on most modern processors and each level of parallelism (MPI, OMP, SIMD) can be individually turned on or off. In particular it was also tested on a dual core laptop with Intel i5 processors.

For the performance analysis in the following sections we only considered the maximum computing capacity of 32 nodes and did not take into account the GPU compute power.
For all runs we used the GCC compiler (version 4.8.2) with the flags `-std=c++11 -Wall -O3 -fopenmp -funroll-loops -fprefetch-loop-arrays -mavx -march=native`.

2.2 Simple serial code

The time evolution can be implemented as the multiplication with the evolution operator as follows:

```
// E_0
for(ri=0; ri<N_; ++ri) {

  // diagonal
  for(s=0; s<dim_; ++s) {
    psi_tmp_[s] = cos_gamma * psi_[s];
  }

  // off-diagonal
  for(s=0; s<dim_; ++s) {
    s_flip = s^(1<<ri);

    psi_tmp_[s_flip] += isin_gamma * psi_[s];
  }
  std::swap(psi_,psi_tmp_);

}

// E_P
for(ri=0; ri<N_; ++ri) {
  for(rj=0; rj<N_; ++rj) {

    expbeta = ( graph_.Jij(ri,rj) == 1 ? expbeta_connected :
        expbeta_disconnected );

    for(s=0; s<dim_; ++s) {
      parallel = (((s>>ri)^(s>>rj)) & 1) == 1;
      psi_[s] *= ( parallel ? expalpha : expbeta );
    }

  }
}
```

2.3 MPI

While the communication process could potentially be very expensive and require complicated communication strategies, the intrinsic nature of the quantum mechanic problem and the bitwise spin representation encoded in the state indices allowed a very efficient and simple communication pattern.
We know that the size of the basis is a power of 2, say it is $\texttt{dim} = 2^N$. The idea is that we divide the coefficient vector `psi_` in chunks of the same size $\texttt{dim\_local\_MPI\_} = \text{dim\_/size\_MPI\_} = 2^K$ (this is not a big restriction since the problem grows exponentially anyways) and distribute them over 2^{N-K} MPI nodes. We can then use the first $N-K$ bits of the state index for enumerating the different MPI nodes. Table 1 represents this principle.
There are two scenarios how a spinflip ($\Leftrightarrow$ bitflip, caused by the off-diagonal elements in $E_0^{(i)}$, see equation 15) could map a state index:

local mapping of single indices: if one of the least significant (rightmost) bits is flipped, the flipped index still remains in the range covered by `dim_local_MPI_` and no communication with other MPI nodes is required.

global mapping of entire chunk: if one of the most significant (leftmost) bits is flipped, all the rightmost indices remain unchanged. As noted above, the leftmost indices correspond to the enumeration of the MPI nodes, and therefore by flipping one of those bits we now have to move the data to a different MPI node. However, since the flip affects all states equally, we have to send all the `psi_` data to the same MPI node and since the rightmost indices are not affected, we can leave the ordering untouched.
This is a very important property of the underlying quantum mechanics and will be essential in explaining the MPI speedup graphs, in particular figure 6.

In order to allow communication between MPI nodes, we introduced an additional coefficient vector that receives the data from other processes.

See also the codesnippet 2 for the MPI communication process.
Additionally, we use some simple reduce and gather statements to synchronize the MPI processes among each other.

	loc	glob	N
R1	0	0	0000
	1	1	0001
	2	2	0010
	3	3	0011
R2	0	4	0100
	1	5	0101
	2	6	0110
	3	7	0111
R3	0	8	1000
	1	9	1001
	2	10	1010
	3	11	1110
R4	0	12	1100
	1	13	1101
	2	14	1110
	3	15	1111

Table 1: ψ Vector for a system size with $N = 4$ spins $\implies$ 16 different states in the Hilbert space. Most significant bit first.

2.4 OpenMP

For the OpenMP thread parallelization we parallelized the for loop over all the basis states (size dim) and paid attention to shared variables and thread private variables and arrays. It was not possible to further parallelize the nested loops over N since race conditions would occur.
For time reasons we did not investigate on the effect of using different scheduling strategies, but since the workload is very evenly balanced among all threads the automatic scheduler is probably a reasonable choice.
See codesnippet 1

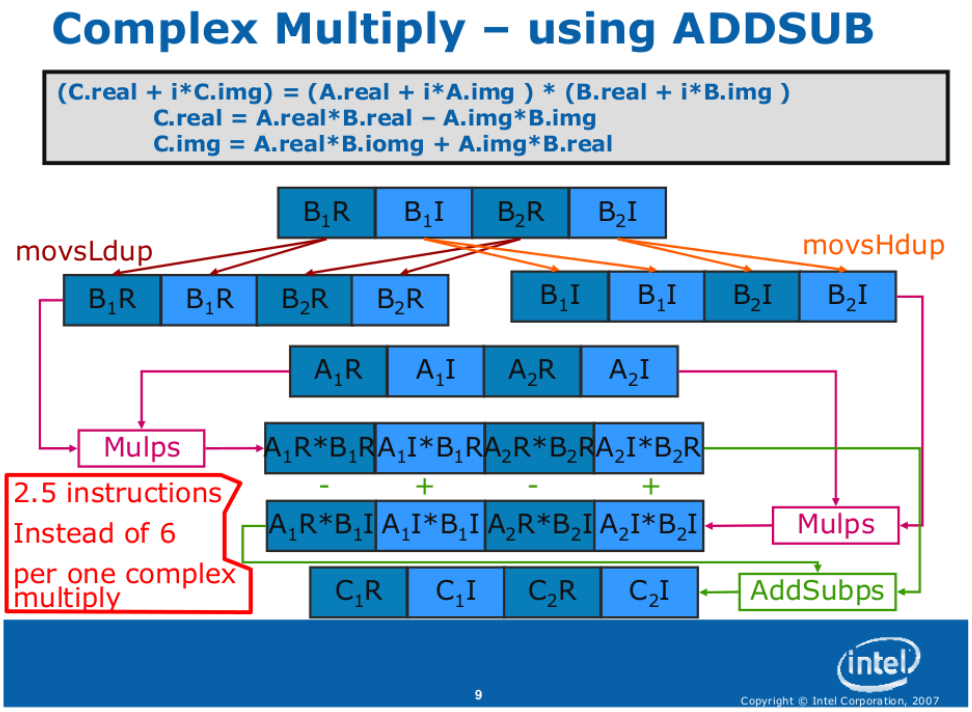
2.5 SIMD

We implemented *Single Instruction Multiple Data* parallelism by using a combination of `SSE2` and `AVX` instructions from the Intel Intrinsics instruction set [4]. While `AVX` uses 256Bit SIMD registers and is therefore twice as efficient as `SSE2`, it does not support integer operations[4] which were required for performing bitshift operations and bitwise logical operations in the `EP` loop (see codesnippet 1).

The dominating floating point operations in our simulation are complex-complex addition (= 2 additions) and multiplication (= 4 multiplications + 2 × 2 addi-

[4] will be implemented in `AVX2`

tions). However, SIMD operations only operate on vectors of single floats (or doubles). While in the case of addition this difference could be ignored (real and imaginary part of a complex number are added individually), the implementation of the vectorized complex-complex multiplication required more work.
We found a pseudo-code of an ingenious way to implement complex-complex multiplication in AVX on the Intel Developer Zone [5]:



This implementation only requires two rather than four SIMD multiplications. Furthermore, we exploited the fact that in the `E0` loop we multiply with purely real and with purely imaginary coefficients and implemented two specific multiplication functions accordingly, see codesnippet 3.

3 Analysis

3.1 Correctness

In order to check the correctness of the algorithm, we applied it to specifically constructed, simple graphs. Throughout the parallelization process, we made sure that the results calculated by the parallel versions of our code were bitwise the same as the serial version when applied to the same problems.
The two test graphs (graph consisting of two maximally connected subclusters and cyclic graph) and a random graph are shown in figure 1. The coloring of the graph nodes corresponds to the partitioning found by our quantum graph partitioning program.

3.2 Benchmarks

For the performance benchmarks, we ran $M = 2$ timesteps for different graph sizes and parallelization parameters. We did not propagate the system until convergence.

All error bars are representing the standard deviation of our time measurements and in the case of derived quantities (e.g. speedup $= \frac{t_0}{t}$) the well known formula for propagation of error (see Appendix 6.1) was used.

3.2.1 MPI

Figures 4 and 5 show the relative time fraction that is consumed by each of the main parts of the program for the two graph sizes 12 and 28 respectively.
By comparing the two figures it immediately becomes clear that for reasonably large systems, the dominating and most expensive part is the E_P loop which is of size $N^2 \cdot 2^N$ where N denotes the number of nodes in the graph.

Figure 6 shows the strong scaling for a run with 1 and 8 OMP threads. It is easy to spot that in the case of 1 OMP thread, the strong scaling for graph size 28 is even better than optimal, which is strictly speaking not possible. Even table 2 shows that the data points lie clearly above the optimal scaling line, and even if we account for measurement errors and variance, this is not enough to explain this surprising behavior.
However, we found out that the nature of the quantum mechanical problem leads to some advantages in terms of cache effects: in fact, when chunks of the `psi_` vector are exchanged among MPI nodes, the spinflip operations become local operations which gives us a speedup with respect to the normal calculation where we access the memory location of the bit-flipped index. See lines 215-255 in the codesnippet 1.
Figure 7 helps to visualize this speedup effect due to better cache usage: It can be seen that the total runtime of the sum of `E0` and `COMM` loop decreases,

although the communication time itself increases.
This explains why the program shows something like "super-optimal"-scaling.

The nature of our problem does not really allow us to create a weak scaling plot in the conventional sense. The reason for this is that when we add one graph node ($N \to N+1$), the loops `E0` and `EP` grow by a factor of $N2^N$ and $N^2 2^N$ respectively. While we can easily account for the 2^N factor by doubling the number of MPI ranks (or OMP nodes), the increased workload due to the factors N and N^2 remains.
We tried to rescale the timings `E0` and `EP` accordingly to create a plot that compensates for this fact in figure 8. However, for performance analysis of the given algorithm we think that the strong scaling plot is a better indicator.

3.2.2 OpenMP

The OpenMP strong scaling plots are shown in figure 9 and 10. The scaling behavior looks very much as expected and we see that for reasonably large problem sizes OMP threading can achieve good speedups.

3.2.3 SIMD

Our vectorization gave a speedup factor of $2.7 - 2.8$, see table 4. It is worth noting that the gcc-compiler vectorization did not help us gain any speedup at all.
After some initial testing and benchmarking with and without AVX parallelism, we performed all the remaining benchmarks with vectorization turned on.

3.3 Energy

Figures 11 and 12 clearly show that using a higher level of parallelism can also mean a lower energy consumption. This is especially true for OpenMP parallelism, since it ensures that the resources on one active node are used as extensively as possible.

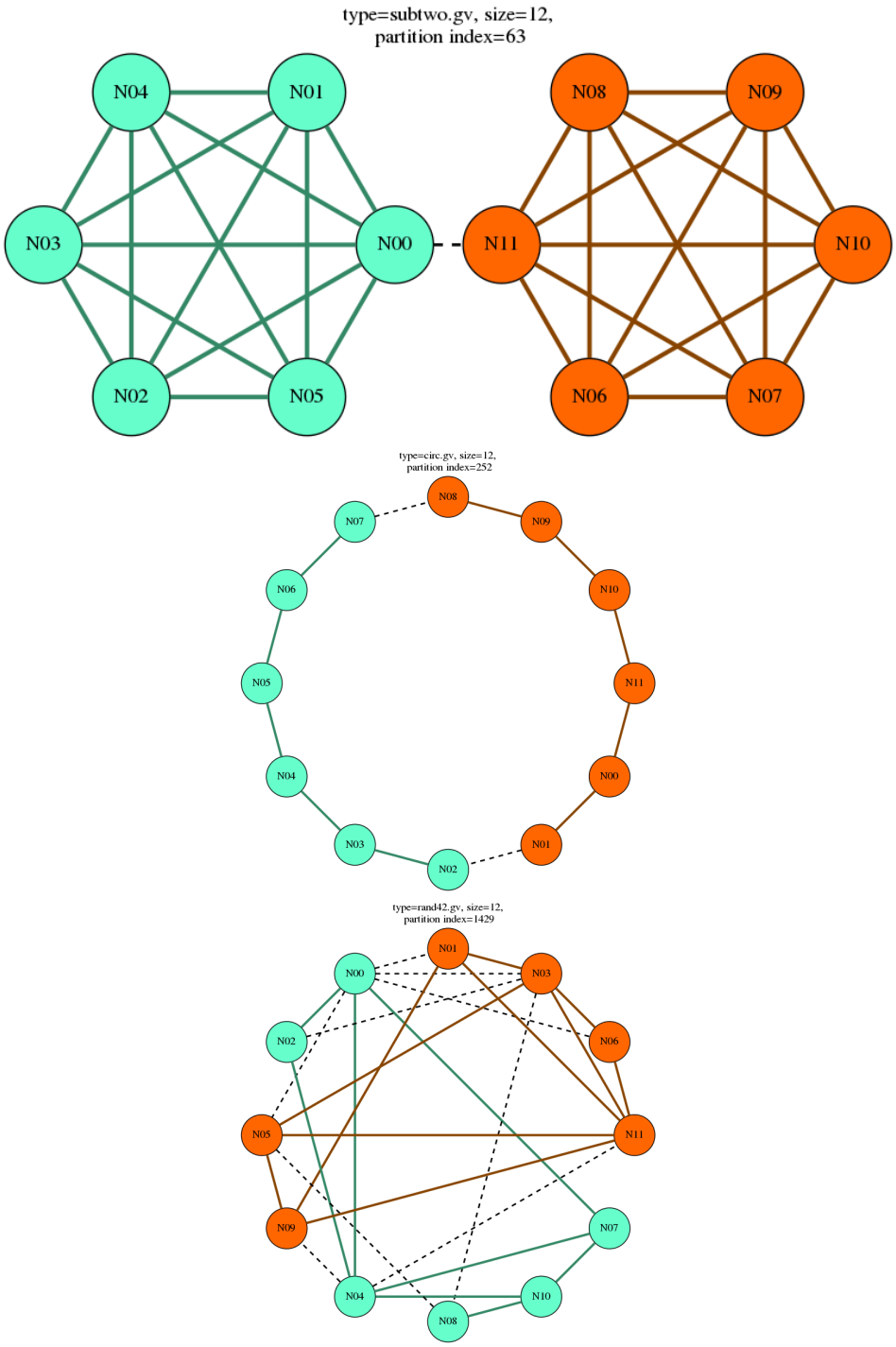


Figure 1: Test graphs and random graph: the coloring of the nodes corresponds to the partitioning of the cluster, the algorithm tries to minimize the connections between nodes of different color (dashed lines) while keeping the constraint that the number of nodes of each color must be the same.

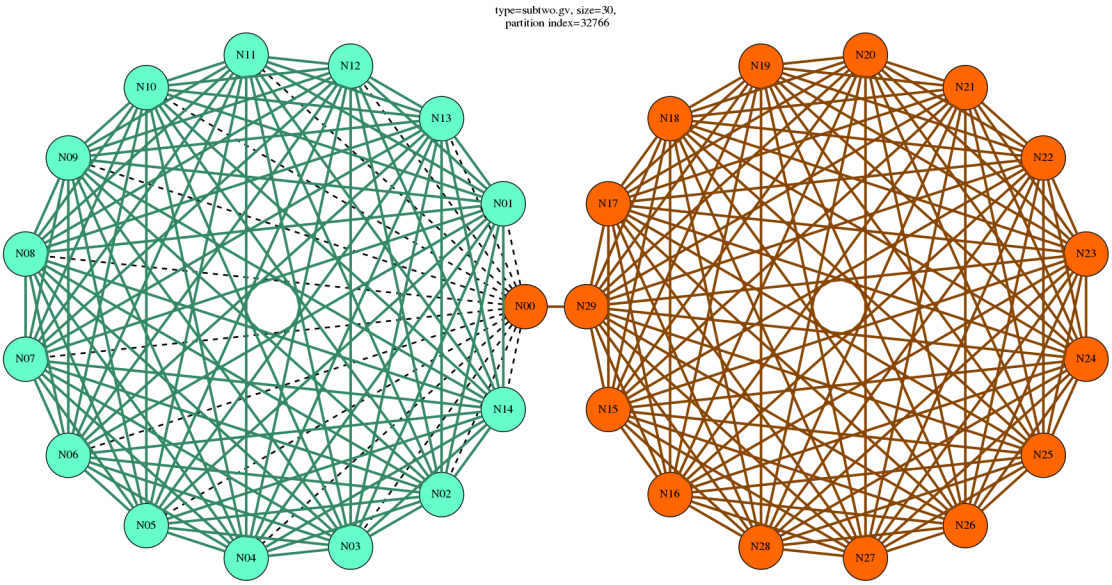


Figure 2: Result of a physical run with 1000 timesteps on a test graph of size 30: The result has not yet converged towards the optimal solution (note there is an orange node in the left subcluster), but it is very close to the optimal solution. More timesteps would probably allow convergence towards the correct partitioning.

4 Plots & Data

4.1 Roofline Model

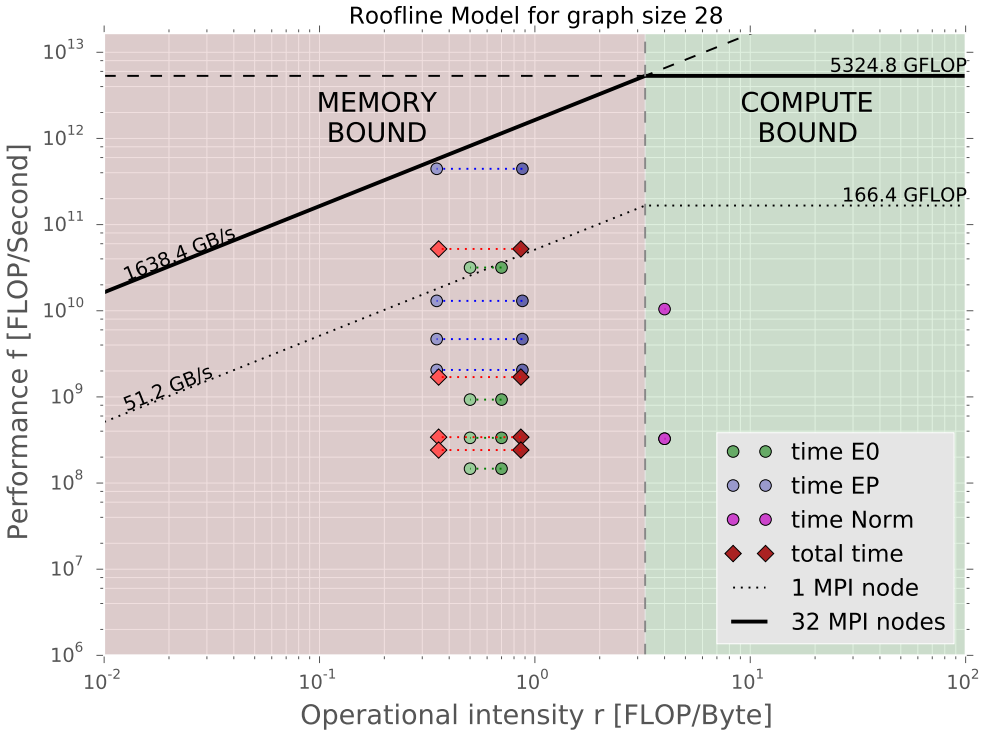


Figure 3: Roofline plot for a maximum of 32 MPI compute nodes, graph size 28. Data points from bottom to top: Serial, Vectorization on, 8 OMP threads, 32 MPI ranks

FLOP and Load count (best/worst case cache scencario)
M (multiplication), **A** (addition), **I** (integer operations), **L** (load), **W** (write)

	FLOP			MemAccess	
	M	**A**	**I**	**L**	**W**
E_0: $N2^N\times$	4+4	2+4	0	1+2/2+3	1+1
E_P: $N^22^N\times$	4	2	1	1/4	1
norm: $2^N\times$	2	2	0	1	0

Note that a complex multiplication

$$(a_r + a_i\imath)(b_r + b_i\imath) = (a_r b_r - a_i b_i) + (a_r b_i + a_i b_r)\imath$$

consists of 4 real multiplications and 4 real additions (2 for the real and 2 for the imaginary part).

Notes about the following table: the first digit in the sum in the E_0 row corresponds to the diagonal part of E_0 and the second to the off-diagonal part. Further, for E_0 and E_P only the operations in the innermost for loops were counted (others are negligible due to large scaling factors) and the scaling factors are given in the leftmost column.
In the E_P loop, we accounted the sum of two bitshifts, one logical *OR* and one logical *AND* as 1**I** which can be considered to be roughly as expensive as **A** and **M**.

Finally, assuming that all FLOP are equally expensive and that also load and write operations come at the same cost, we get the following table:

	FLOP	**MemAccess**
E_0	$14 \cdot N2^N$	$5/7 \cdot N2^N$
E_P	$7 \cdot N^2 2^N$	$2/5 \cdot N^2 2^N$
norm	$4 \cdot 2^N$	2^N
tot	$2^N(4 + N(14 + N \cdot 7))$	$2^N(1 + N(5/7 + N \cdot 2/5))$

where the numbers divided by a / denote the best/worst case scenario.
For calculating the operational intensities, it is important to note that each Float consists of 4Bytes, therefore an operation on one float operates on 4Bytes.
The operational intensities are thus given by

	best	**worst**
E_0	$\frac{14}{4\cdot 5} = 0.70$	$\frac{14}{4\cdot 7} = 0.5$
E_P	$\frac{7}{4\cdot 2} = 0.87$	$\frac{7}{4\cdot 5} = 0.35$
norm	4	4
tot	$\frac{4+14N+7N^2}{4(1+5N+2N^2)}$	$\frac{4+14N+7N^2}{4(1+7N+5N^2)}$

4.2 MPI

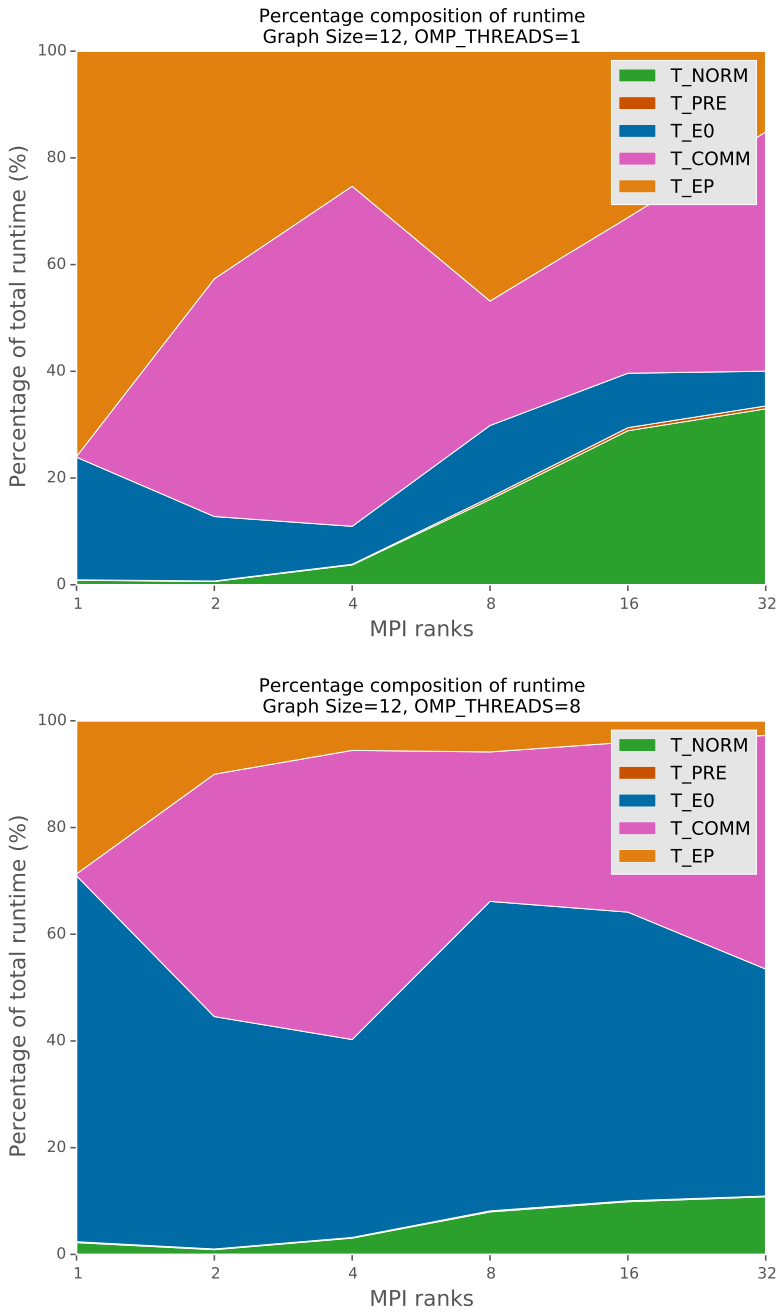


Figure 4: Percentage composition of runtime as a function of MPI ranks for 1 OpenMP thread (top) and 8 OpenMP threads (bottom), graphsize 12: Clearly, for the given graph size the overhead of communication time and the calculation of expensive sine, cosine, and exponential functions dominates the runtime. For graphs of such moderate size it's not beneficial to choose a high level of MPI parallelism.

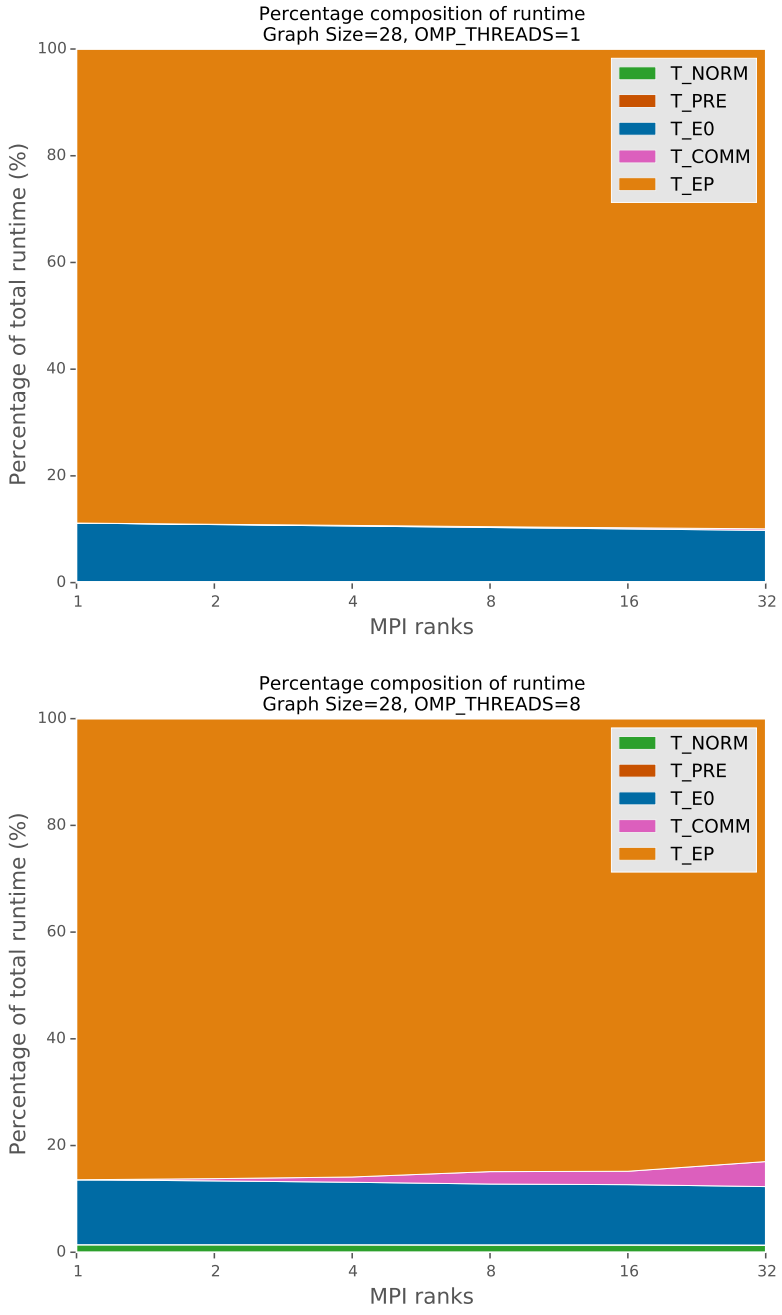


Figure 5: Percentage composition of runtime as a function of MPI ranks for 1 OpenMP thread (top) and 8 OpenMP threads (bottom), graphsize 28: It can be clearly seen that the percentage used by the EP loop of the program is dominating the runtime taking up a percentage of 80-90%. There is almost no overhead due to calculation of expensive functions and normalization. Even the communication time takes up a reasonably small fraction of time.

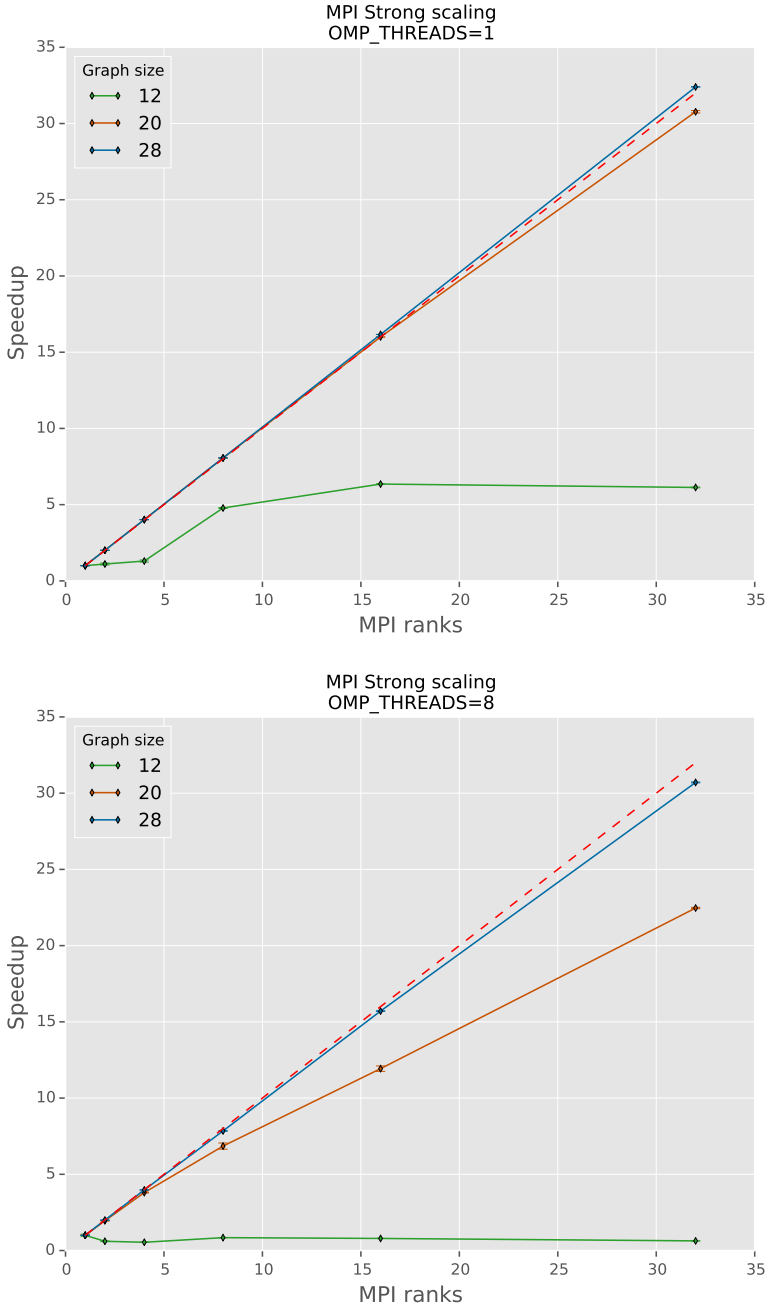


Figure 6: MPI strong scaling plots for 1 OpenMP thread (top) and 8 OpenMP threads (bottom) The scaling behavior of our program becomes better with increasing graph size and is almost optimal for graph size 28 or larger. The reason for this is that the largest loop `EP` (size $N^2 2^N$) does not rely on communication at all and therefore outweighs most of the other parts of the program.
The top-plot even shows super-optimal scaling which is usually not possible. However, a careful analysis of the code leads to the conclusion that using more MPI nodes leads to better cache-use in the `E0` loop. More explanations are given in the text.

OMP_THREADS=1

Graphsize 12					
Ranks	2	4	8	16	32
Speedup	1.11 ± 0.06	1.30 ± 0.07	4.78 ± 0.03	6.35 ± 0.02	6.13 ± 0.03

Graphsize 20					
Ranks	2	4	8	16	32
Speedup	2.01 ± 0.00	4.01 ± 0.01	8.06 ± 0.00	16.00 ± 0.01	30.77 ± 0.08

Graphsize 28					
Ranks	2	4	8	16	32
Speedup	2.01 ± 0.00	4.02 ± 0.00	8.06 ± 0.00	16.16 ± 0.00	32.40 ± 0.00

Table 2: Speedup for MPI strong scaling. ($M = 2$ timesteps, `OMP_THREADS=1`)

OMP_THREADS=8

Graphsize 12					
Ranks	2	4	8	16	32
Speedup	0.61 ± 0.02	0.54 ± 0.01	0.85 ± 0.01	0.79 ± 0.00	0.63 ± 0.01

Graphsize 20					
Ranks	2	4	8	16	32
Speedup	1.95 ± 0.02	3.80 ± 0.02	6.85 ± 0.20	11.93 ± 0.18	22.47 ± 0.04

Graphsize 28					
Ranks	2	4	8	16	32
Speedup	2.00 ± 0.01	3.97 ± 0.01	7.85 ± 0.02	15.71 ± 0.02	30.70 ± 0.03

Table 3: Speedup for MPI strong scaling. ($M = 2$ timesteps, `OMP_THREADS=8`)

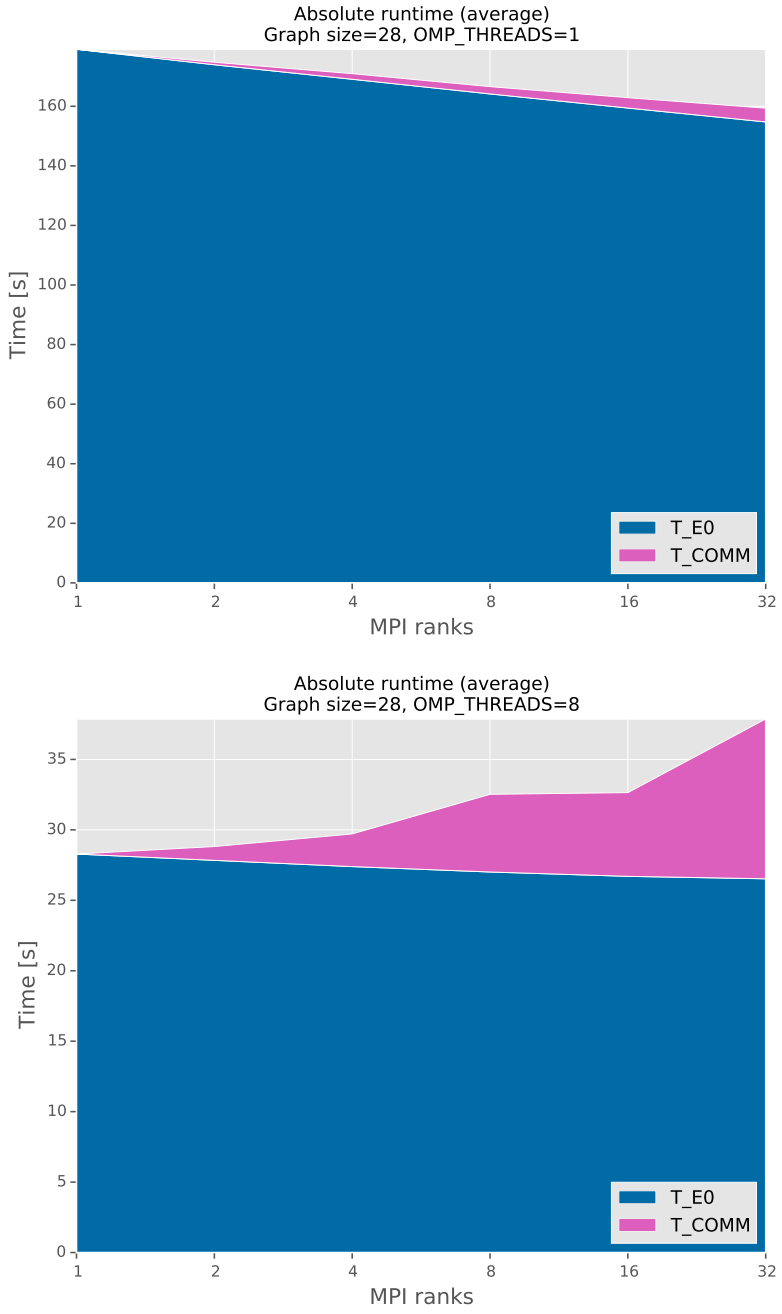


Figure 7: Comparison of compute time for the E_0 and MPI communication time for 1 OpenMP thread (top) and 8 OpenMP threads (bottom). The times are summed up over all MPI ranks.
In the case of 1 OMP thread, the sum of both runtimes `E0` and `COMM` decreases. The reason for this is a more efficient cache reuse during the multiplication process. However, for 8 OMP threads the computation part of `E0` is significantly faster and the longer communication time outweighs the difference due to the faster `E0` loop. It can also be noted that communication is very efficient in the case of 16 MPI nodes

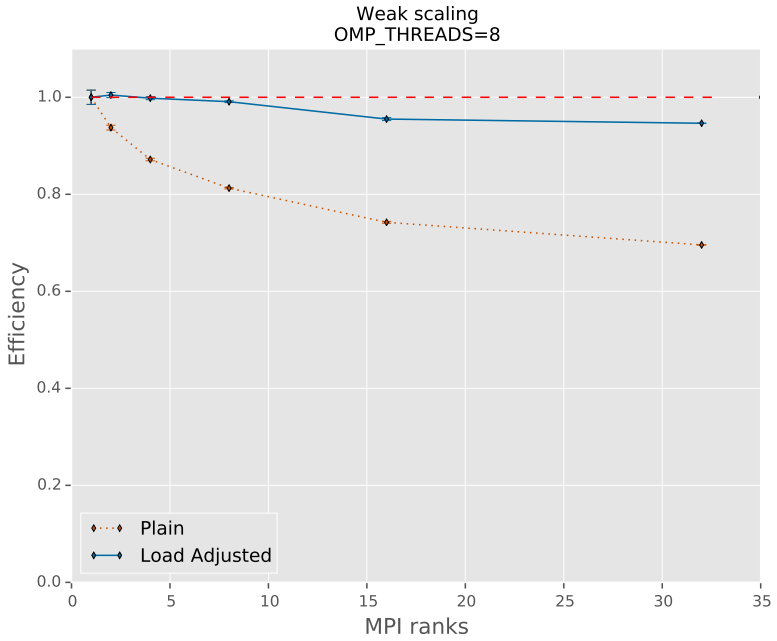


Figure 8: MPI weak scaling: the dashed line shows the weak scaling for the case where we added one graph node ($N \to N+1$) and at the same time doubled the number of MPI ranks. However, this is not a fair representation since the loop length of `E0` grows by an additional factor of N and the length of `EP` grows by an extra factor of even N^2.
The solid line shows the weak scaling plot where the times were normalized by using these scaling factors.

4.3 OMP

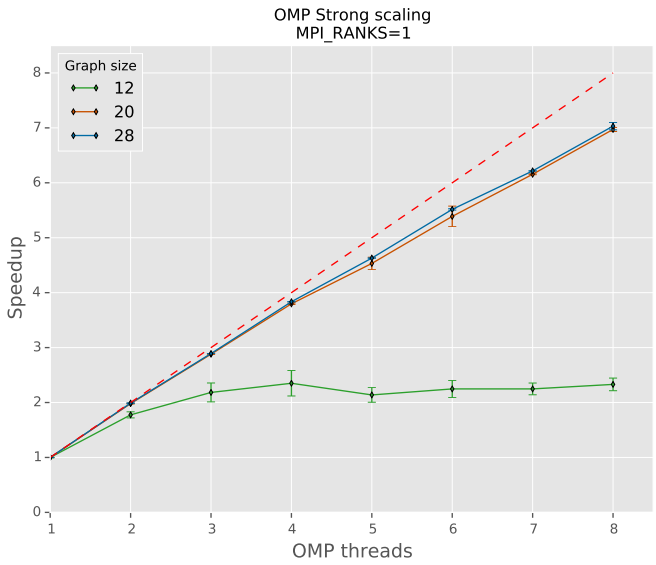


Figure 9: OMP strong scaling for 1 MPI rank: The scaling improves with increasing graph size and the speedup is almost identical for sizes 20 and 28. The remaining gap between measured and optimal speedup is caused by the overhead for thread management.

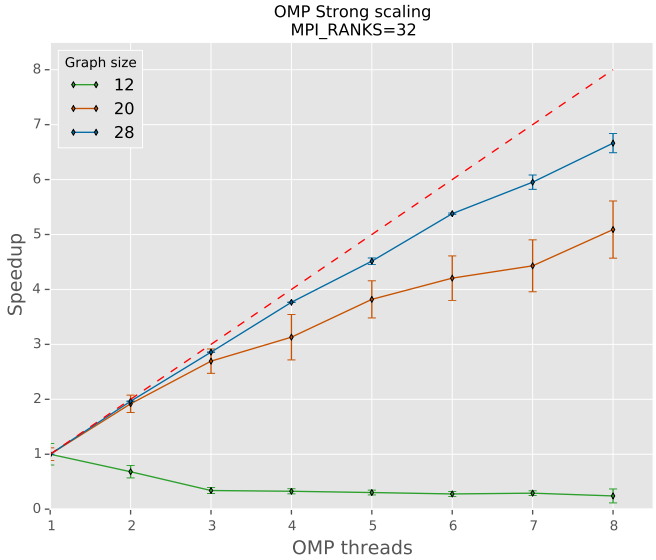


Figure 10: OMP strong scaling for 32 MPI ranks: The qualitative behavior is similar to the case of 1 MPI rank, but since less work needs to be distributed among the threads, the parallelization only pays off for larger systems. If the systems are small, the overhead for spawning and managing threads dominates the timing while only little computations need to be performed per thread.

4.4 SIMD

Graphsize 16

SIMD	time $\mathbf{E_0}$	time $\mathbf{E_P}$	time total
NO	43 ms	345 ms	389 ms
GCC	43 ms	345 ms	389 ms
Manual	25 ms	116 ms	143 ms
Speedup	1.72	2.97	**2.72**

Graphsize 18

SIMD	time $\mathbf{E_0}$	time $\mathbf{E_P}$	time total
NO	195 ms	1745 ms	1942 ms
GCC	195 ms	1745 ms	1942 ms
Manual	114 ms	588 ms	705 ms
Speedup	1.71	2.97	**2.75**

Graphsize 24

SIMD	time $\mathbf{E_0}$	time $\mathbf{E_P}$	time total
NO	16 836 ms	198 194 ms	215 235 ms
GCC	16 836 ms	198 201 ms	215 243 ms
Manual	9576 ms	66 724 ms	76 505 ms
Speedup	1.76	2.97	**2.81**

Table 4: Timings for analyzing vectorization speedup.
Setup: $M = 2$ timesteps, `MPI_RANKS=1`, `OMP_THREADS=1`

4.5 Energy

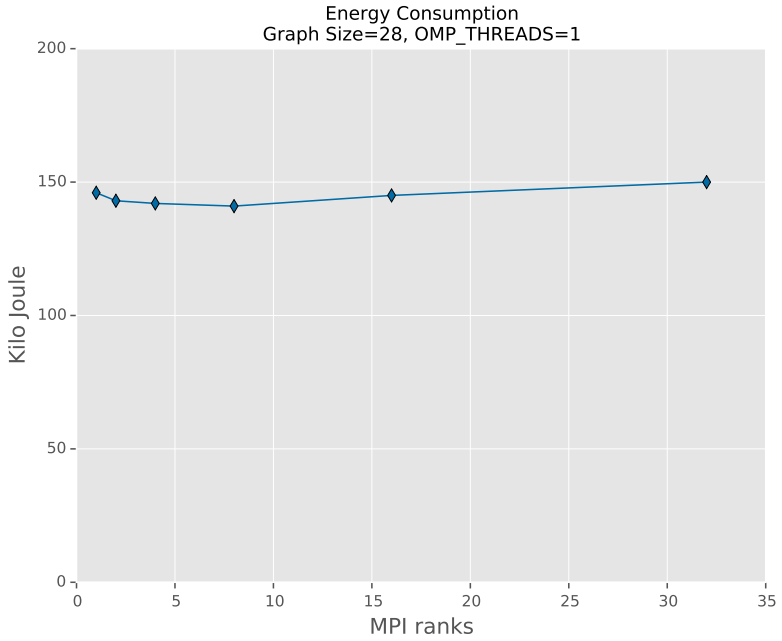


Figure 11: Energy consumption graph as a function of the number of MPI ranks (one sample only, not averaged!): The energy consumption remains roughly constant. While using a number X of MPI ranks will cost X times as much energy, the runtime of the simulation is also reduced by a factor of X and the overall energy consumption remains the same.

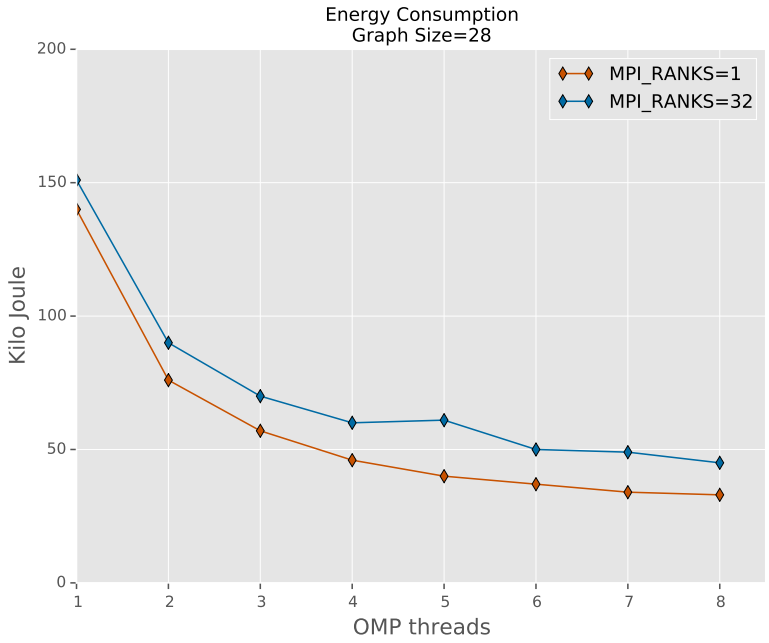


Figure 12: Energy consumption graph as a function of the number of OMP threads (one sample only, not averaged!): The energy consumption decreases by a large factor, meaning that using more OMP threads is generally more energy efficient than using fewer ones of them. By using all the 8 cores of each node we are making optimal use of the hardware and by reducing the runtime (computations get faster with more OMP threads) we are also saving energy.

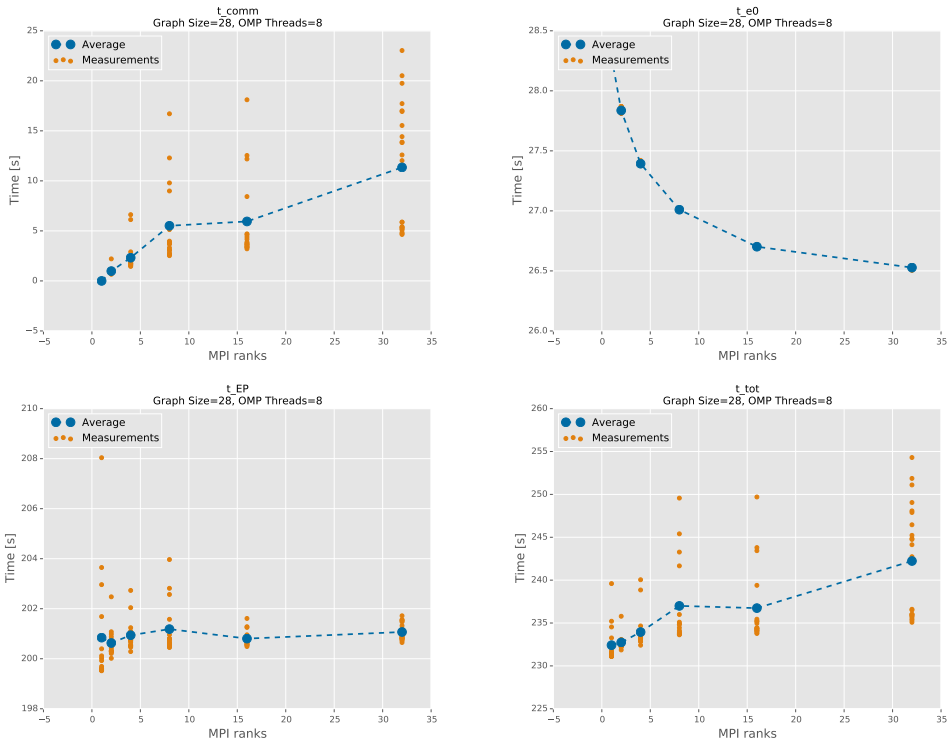


Figure 13: Cumulated execution times for `COMM` (top left), `E0` (top right), `EP` (bottom left) and total simulation time (bottom right) for graph size 28 and 8 OMP threads: The communication time increases, but the time for `E0` decreases for higher numbers of MPI ranks.

5 Code

Listing 1: equilibrate function from isingchain_local.cpp

```
void isingChain_local::equilibrate(unsigned M, type_scalar T) {

  switch_to_timecat_(PRE_);

  M_ = M;
  Tmax_ = T;

  B_ = h0_;
  A_ = inequalityFactor_ * B_/8. * std::min(type_bitcounter(2*graph_.
      maxDegree()),N_); // 2*short is still short

  type_scalar const tau = T/M;

  type_scalar alpha_k, beta_k_ij_connected, beta_k_ij_disconnected, gamma_k;
  type_complex cos_gamma, isin_gamma, expalpha, expbeta, expbeta_connected,
      expbeta_disconnected;
  type_state s_flip, start_flip;

  for(unsigned k=1; k<=M; ++k) {

    switch_to_timecat_(PRE_);

    ////////////////////////////////////////////////////
    // Apply hamiltonian H(t_k)
    ////////////////////////////////////////////////////

    // Precomputing constants, exponential & trigonometric functions
    alpha_k = -1./hbar_ * tau * type_scalar(k)/M * A_;
    beta_k_ij_connected  = -1./hbar_ * tau * type_scalar(k)/M * (B_-A_);
    beta_k_ij_disconnected = -1./hbar_ * tau * type_scalar(k)/M * ( -A_);
    gamma_k = -1./hbar_ * tau * (type_scalar(k)/M-1) * h0_;

    expalpha = std::exp(type_complex(0,alpha_k));
    expbeta_connected  = std::exp(type_complex(0,beta_k_ij_connected));
    expbeta_disconnected = std::exp(type_complex(0,beta_k_ij_disconnected));
    cos_gamma = type_complex(std::cos(gamma_k),0);
    isin_gamma = type_complex(0,std::sin(gamma_k));

#if VECTORIZE > 0
     type_scalar cos_gamma_realpart = cos_gamma.real();
     type_scalar isin_gamma_imagpart = isin_gamma.imag();
#endif // VECTORIZE

    ////////////////////////////////////////////////////
    // E_0
    switch_to_timecat_(E0_);
    for(type_bitcounter ri=0; ri<N_; ++ri) {

       // NOTE: psi_ = psi_MPI_out_
       // NOTE: psi_other_guy_ = psi_MPI_in_
```

```
        // diagonal
        // LOCAL OPERATIONS
        #pragma omp parallel for shared(cos_gamma)
        for(type_state s_local_vecindex=0; s_local_vecindex<dim_local_MPI_;
            s_local_vecindex+=4) {
          // IMPORTANT: current implementation of vectorization assumes that
              type_scalar = float
#if VECTORIZE > 0
          __m256 cg = _mm256_set1_ps(cos_gamma_realpart);
          // +=4 for complex float = 8 single float, reinterpret complex<
              float> as float[2]
          __m256 psi = _mm256_load_ps(reinterpret_cast<type_scalar*>(&psi_[
              s_local_vecindex])); // Loading from memory
          __m256 psitmp = hpcseavx::avx_multiply_float_real_(cg,psi);
          _mm256_stream_ps(reinterpret_cast<type_scalar*>(&psi_tmp_[
              s_local_vecindex]),psitmp); // Writing to memory, bypassing
              cache
#else
          // <==> psi_tmp_[s_local] = cos_gamma * psi_[s_local];
          psi_tmp_[s_local_vecindex] = cos_gamma * psi_[s_local_vecindex];
          psi_tmp_[s_local_vecindex+1] = cos_gamma * psi_[s_local_vecindex
              +1];
          psi_tmp_[s_local_vecindex+2] = cos_gamma * psi_[s_local_vecindex
              +2];
          psi_tmp_[s_local_vecindex+3] = cos_gamma * psi_[s_local_vecindex
              +3];
#endif // VECTORIZE
        }

        // off-diagonal
        // check if the flip is local to MPI-unit
        start_flip = range_begin_MPI_^(1<<ri); // flipping spin of start
            index of current MPI block
        bool localflip = (start_flip >= range_begin_MPI_) && (start_flip <
            range_end_MPI_);
        if(localflip) {
          // no transfer of data required
          #pragma omp parallel for private(s_flip) shared(isin_gamma)
          for(type_state s_local=0; s_local<dim_local_MPI_; ++s_local) {
            s_flip = s_local^(1<<ri);
            // NOTE: cannot be conveniently vectorized, since s_flip !=
                s_local
            psi_tmp_[s_flip] += isin_gamma * psi_[s_local]; // use local data
                in psi_
          }
        } else {
          // write state coefficients from psi_ to psi_other_guy_
          switch_to_timecat_(COMM_);
          communicate_flippedstates_(ri,(k*N_)+ri); // unique communication
              tag k*N + ri
          switch_to_timecat_(E0_);

          #pragma omp parallel for shared(isin_gamma)
```

```
        for(type_state s_local_vecindex=0; s_local_vecindex<dim_local_MPI_;
             s_local_vecindex+=4) {
          // use transferred data in psi_other_guy_
          // communication implicitly takes care of s_flip, therefore no
              flipping required
#if VECTORIZE > 0
            __m256 sg = _mm256_set1_ps(isin_gamma_imagpart);
            // +=4 for complex float = 8 single float, reinterpret complex<
                float> as float[2]
            __m256 psi = _mm256_load_ps(reinterpret_cast<type_scalar*>(&
                psi_other_guy_[s_local_vecindex])); // Loading from memory
            __m256 psitmp = _mm256_load_ps(reinterpret_cast<type_scalar*>(&
                psi_tmp_[s_local_vecindex])); // Loading from memory
            psi = hpcseavx::avx_multiply_float_imag_(sg,psi);
            psitmp = _mm256_add_ps(psitmp,psi);
            _mm256_stream_ps(reinterpret_cast<type_scalar*>(&psi_tmp_[
                s_local_vecindex]),psitmp); // Writing to memory, bypassing
                cache
#else
            // <==> psi_tmp_[s_local] += isin_gamma * psi_other_guy_[s_local
                ];
            psi_tmp_[s_local_vecindex] += isin_gamma * psi_other_guy_[
                s_local_vecindex];
            psi_tmp_[s_local_vecindex+1] += isin_gamma * psi_other_guy_[
                s_local_vecindex+1];
            psi_tmp_[s_local_vecindex+2] += isin_gamma * psi_other_guy_[
                s_local_vecindex+2];
            psi_tmp_[s_local_vecindex+3] += isin_gamma * psi_other_guy_[
                s_local_vecindex+3];
#endif // VECTORIZE

        }
       }

    std::swap(psi_,psi_tmp_);
   }



   ////////////////////////////////////////////////////
   // E_P
   switch_to_timecat_(EP_);
   #pragma omp parallel private(expbeta)
   {
    // Initialized after spawning threads - makes vectors thread-private
    // automatically private since declared within parallel region

    std::vector<type_state> s_global(4); // private

#if VECTORIZE > 0
    type_coefvector exp_factor(4); // private
    type_coefvector expalpha_vec(4); // private
    std::fill(expalpha_vec.begin(), expalpha_vec.end(), expalpha);
    type_coefvector expbeta_vec(4); // private
```

```
#else
275       type_complex exp_factor;
      bool parallel;
#endif // VECTORIZE

      #pragma omp for
280       for(type_state s_local_vecindex=0; s_local_vecindex<dim_local_MPI_;
          s_local_vecindex+=4) {

        s_global[0] = loc_to_glob_state_(s_local_vecindex);
        s_global[1] = loc_to_glob_state_(s_local_vecindex+1);
        s_global[2] = loc_to_glob_state_(s_local_vecindex+2);
285       s_global[3] = loc_to_glob_state_(s_local_vecindex+3);

#if VECTORIZE > 0
        __m256 psi_avx = _mm256_load_ps(reinterpret_cast<type_scalar*>(&psi_[
            s_local_vecindex]));
        __m256 expalpha_avx, expbeta_avx, expfac_avx;
290 #endif // VECTORIZE

        for(type_bitcounter ri=0; ri<N_; ++ri) {
          for(type_bitcounter rj=0; rj<N_; ++rj) {

295         // LOCAL OPERATIONS
            expbeta = ( graph_.Jij(ri,rj) == 1 ? expbeta_connected :
                expbeta_disconnected );
#if VECTORIZE > 0

            // Cannot use set1 since complex numbers
300         std::fill(expbeta_vec.begin(), expbeta_vec.end(), expbeta);

            // SSE2
            //  cast unsigned int to signed int and load to sse2 register
            __m128i s_sse = _mm_load_si128( (__m128i*)&s_global[0] );
305
            //  bitshift with ri, rj:
            //  shifting r-th bit to sign bit:
            //  - shift right by r to get to least significant bit
            //  - shift left by 31 to move least significant bit to sign bit
310         //  <==> shift left by 31-r = 0x1F-r
            __m128i shift_ri_sse = _mm_set1_epi32(0x1F-ri); // 0x1F = 31:
                shift relevant bit to sign bit
            __m128i shift_rj_sse = _mm_set1_epi32(0x1F-rj); // 0x1F = 31:
                shift relevant bit to sign bit
            shift_ri_sse = _mm_sll_epi32(s_sse,shift_ri_sse);
            shift_rj_sse = _mm_sll_epi32(s_sse,shift_rj_sse);
315
            // Check if bits (i.e. spins) are parallel
            s_sse = _mm_xor_si128(shift_ri_sse,shift_rj_sse);
            __m128 mask_float_sse = _mm_castsi128_ps(s_sse); // cast mask to
                float

320         // AVX
```

```
          // cast mask to avx and duplicate entries for real and complex
              floats
          __m256 mask_avx = _mm256_castps128_ps256(mask_float_sse);
          mask_avx = _mm256_insertf128_ps(mask_avx,mask_float_sse,0x01);
          __m256 mask_lo_avx = _mm256_unpacklo_ps(mask_avx,mask_avx);
          __m256 mask_hi_avx = _mm256_unpackhi_ps(mask_avx,mask_avx);
          mask_avx = _mm256_blend_ps(mask_lo_avx,mask_hi_avx,0xF0);

          expalpha_avx = _mm256_load_ps(reinterpret_cast<type_scalar*>(&
              expalpha_vec[0]));
          expbeta_avx = _mm256_load_ps(reinterpret_cast<type_scalar*>(&
              expbeta_vec[0]));

          expfac_avx = _mm256_blendv_ps(expbeta_avx,expalpha_avx,mask_avx);
          psi_avx = hpcseavx::avx_multiply_float_complex_(expfac_avx,psi_avx
              );
#else
          for(unsigned short k=0; k<4; ++k) {
            parallel = (((s_global[k]>>ri)^(s_global[k]>>rj)) & 1) == 1;
            exp_factor = ( parallel ? expalpha : expbeta );
            psi_[s_local_vecindex+k] *= exp_factor;
          }
#endif // VECTORIZE
        }
      }
#if VECTORIZE > 0
      _mm256_stream_ps(reinterpret_cast<type_scalar*>(&psi_[
          s_local_vecindex]),psi_avx);
      // NB: store is faster for small graphs, stream is faster for large
          graps
#endif // VECTORIZE
    } // end of parallel for
  } // end of parallel region
}

switch_to_timecat_(NORM_);
normalize_();
switch_to_timecat_(COMM_);
MPI_Barrier(MPI_COMM_WORLD);
switch_to_timecat_(NONE_);
}
```

Listing 2: `communicate_flippedstates_` function from `isingchain_local.cpp`

```
void isingChain_local::communicate_flippedstates_(type_state const r,
    unsigned const comm_tag) {

  //MPI_Barrier(MPI_COMM_WORLD);

  // calculate other guy
  type_state s_start_flipped = (range_begin_MPI_^(1<<r));
  assert(s_start_flipped%dim_local_MPI_ == 0);
  int other_guy = s_start_flipped/dim_local_MPI_;

  MPI_Status status;

  // Sendrecv uses a blocking send / receive
  int msg = MPI_Sendrecv(&psi_.front(), 2*dim_local_MPI_, MPI_FLOAT,
      other_guy, comm_tag,
      &psi_other_guy_.front(), 2*dim_local_MPI_, MPI_FLOAT,
      other_guy, comm_tag,
      MPI_COMM_WORLD, &status); // MPI_STATUS_IGNORE);

  assert(msg == 0); // check if there was a major communication problem

  //MPI_Barrier(MPI_COMM_WORLD);
}
```

Listing 3: `avx_functions.hpp`: AVX helper functions for multiplication with complex numbers

```
// Vectorization
#include "aligned_allocator.hpp"
#include <x86intrin.h>
#include <immintrin.h>

namespace hpcseavx {

  // PRE: all vectors aligned,
  //    real_c = [r1,r1,...,r4,r4]
  //    vec = [v1r,v1i,...,v4r,v4i]
  //    component-wise multiplication
  // POST: returns [r1*v1r,r1*v1i,...,r4*v4r,r4*v4i]
  inline __m256 avx_multiply_float_real_(const __m256& real_c, const __m256&
      vec) {
    return _mm256_mul_ps(real_c,vec);
  }

  // PRE: all vectors aligned,
  //    imag_c = [i1,i1,...,i4,i4]
  //    vec = [v1r,v1i,...,v4r,v4i]
  //    component-wise multiplication
  // POST: returns [-i1*v1i,i1*v1r,...,-i4*v4i,i4*v4r]
  inline __m256 avx_multiply_float_imag_(const __m256& imag_c, const __m256&
      vec) {
    static const __m256 zero = _mm256_setzero_ps();
    __m256 vec1 = _mm256_mul_ps(imag_c,vec);
    vec1 = _mm256_permute_ps(vec1,0xB1);
    return _mm256_addsub_ps(zero,vec1);
  }

  // PRE: all vectors aligned,
  //    vecA = [A1r,A1i,...,A4r,A4i]
  //    vecB = [B1r,B1i,...,B4r,B4i]
  //    full complex multiplication
  // POST: returns [A1r*B1r-A1i*B1i,A1r*B1i+A1i*B1r,...,A4r*B4r-A4i*B4i,A4r*
      B4i+A4i*B4r]
  // NOT NEEDED
  /*****************************************************************
   * This technique for efficient SIMD complex-complex multiplication was
       found at
   *     https://software.intel.com/file/1000
  *****************************************************************/
  inline __m256 avx_multiply_float_complex_(const __m256& vecA, const __m256&
       vecB) {
    __m256 vec1 = _mm256_moveldup_ps(vecB);
    __m256 vec2 = _mm256_movehdup_ps(vecB);
    vec1 = _mm256_mul_ps(vecA,vec1);
    vec2 = _mm256_mul_ps(vecA,vec2);
    vec2 = _mm256_permute_ps(vec2,0xB1);
    return _mm256_addsub_ps(vec1,vec2);
  }
```

```
48 }
```

6 Appendix

6.1 Propagation of Error

For uncorrelated, independent variables x_i, $1 \leq i \leq n$ and a function $f(x_1, \ldots, x_n)$ the variance of f is given by

$$Var(f) = \sum_{i=1}^{n} Var(x_i) \left(\frac{\partial f}{\partial x_i} \right)^2$$

In particular, in our calculation for the speedup $s = \frac{t_0}{t}$ we get that

$$\begin{aligned} Var(s) &= Var(t_0) \left(\partial_{t_0} \frac{t_0}{t} \right)^2 + Var(t) \left(\partial_t \frac{t_0}{t} \right)^2 \\ \Rightarrow \qquad \sigma_s &= \sqrt{Var(t_0) \left(\frac{1}{t} \right)^2 + Var(t) \left(-\frac{t_0}{t^2} \right)^2} \end{aligned}$$

References

[1] Andrew Lucas. "Ising formulations of many NP problems". In: *frontier in Physics* 0.0005 (Feb. 2014). URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/11MC12_Avoiding_2BAVX-SSE_2BTransition_2BPenalties_2Brh_2Bfinal.pdf.

[2] CSCS. *Piz Daint & Piz Dora*. 2015. URL: http://www.cscs.ch/computers/piz_daint/index.html.

[3] *Top500*. URL: http://www.top500.org/lists/2015/06/.

[4] Intel. *Intel Intrinsics Guide*. 2015. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#.

[5] Intel Mostafa Hagog. *1000 - Looking for 4x speedups? SSE to the rescue!* URL: https://software.intel.com/file/1000.

[6] Patrick Konsor. "Avoiding AVX - SSE Transition Penalties". In: *Intel Website* (). URL: https://software.intel.com/sites/default/files/m/d/4/1/d/8/11MC12_Avoiding_2BAVX-SSE_2BTransition_2BPenalties_2Brh_2Bfinal.pdf.

[7] Matthias Troyer. "Lecture Script - Computational Quantum Physics". In: *ETH Lecture Script* ().