Sparse Non-Blocking Collectives in Quantum Mechanical Calculations

Torsten Hoefler[1], Florian Lorenzen[2], and Andrew Lumsdaine[1]

[1] Open Systems Lab, Indiana University, Bloomington IN 47405, USA,
{htor,lums}@cs.indiana.edu,
[2] Institut für Theoretische Physik, FU Berlin, Arnimalle 14, 14195 Berlin, Germany
lorenzen@physik.fu-berlin.de

Abstract. For generality, MPI collective operations support arbitrary dense communication patterns. However, in many applications where collective operations would be beneficial, only sparse communication patterns are required. This paper presents one such application: Octopus, a production-quality quantum mechanical simulation. We introduce new sparse collective operations defined on graph communicators and compare their performance to MPI_Alltoallv. Besides the scalability improvements to the collective operations due to sparsity, communication overhead in the application was reduced by overlapping communication and computation. We also discuss the significant improvement to programmability offered by sparse collectives.

1 Introduction

Ab-initio quantum mechanical simulations play an important role in nano and material sciences as well as many other scientific areas, e. g., the understanding of biological or chemical processes. Solving the underlying Schrödinger equation for systems of hundreds or thousands of atoms requires a tremendous computational effort that can only be mastered by highly parallel systems and algorithms.

Density functional theory (DFT) [10, 11] is a computationally feasible method to calculate properties of quantum mechanical systems like molecules, clusters, or solids. The basic equations of DFT are the static and time-dependent Kohn-Sham equations:[1]

$$\boldsymbol{H}\boldsymbol{\varphi}_j = \varepsilon_j \boldsymbol{\varphi}_j \qquad\qquad i\frac{\partial}{\partial t}\boldsymbol{\varphi}_j(t) = \boldsymbol{H}(t)\boldsymbol{\varphi}_j(t) \tag{1}$$

The electronic system is described by the Hamiltonian operator

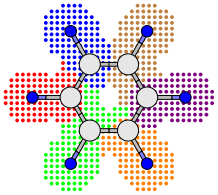
$$\boldsymbol{H} = -\frac{1}{2}\nabla^2 + \boldsymbol{V}, \tag{2}$$

where the derivative accounts for kinetic energy and $\boldsymbol{V}$ for the atomic potentials and electron-electron interaction. The vectors $\boldsymbol{\varphi}_j$, $j = 1, \ldots, N$, are the Kohn-Sham orbitals each describing one of N electrons.

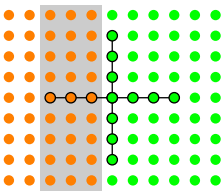
[1] i denotes the imaginary unit $i = \sqrt{-1}$ and t is the time parameter.

The scientific application `octopus` [3] solves the eigenvalue problem of Eq. (1, left) by iterative diagonalization for the lowest N eigenpairs $(\varepsilon_j, \boldsymbol{\varphi}_j)$ and Eq. (1, right) by explicitly evolving the Kohn-Sham orbitals $\boldsymbol{\varphi}_j(t)$ in time. The essential ingredient of iterative eigensolvers as well as of most real-time propagators [2] is the multiplication of the Hamiltonian with an orbital $\boldsymbol{H}\boldsymbol{\varphi}_j$. Since `octopus` relies on finite-difference grids to represent the orbitals, this operation can be parallelized by dividing the real-space mesh and assigning a certain partition (domain) to each node as shown in Fig. 1(a).

The potential $\boldsymbol{V}$ is a diagonal matrix, so the product $\boldsymbol{V}\boldsymbol{\varphi}_j$ can be calculated locally on each node. The Laplacian operator of (2) is implemented by a finite-difference stencil as shown in Fig. 1(b). This technique requires to send values close to the boundary (gray shading in Fig. 1(b)) from one partition (orange) to a neighboring one (green).



(a) A benzene ring distributed on six nodes.



(b) Boundary values for a third order stencil.

Fig. 1. Partitions of `octopus`' real-space finite-difference mesh.

The original implementation of $\boldsymbol{H}\boldsymbol{\varphi}_j$ is:

1. Exchange boundary values between partitions
2. $\boldsymbol{\varphi}_j \leftarrow -\frac{1}{2}\nabla^2\boldsymbol{\varphi}_j$ (apply kinetic energy operator)
3. $\boldsymbol{\varphi}_j \leftarrow \boldsymbol{\varphi}_j + \boldsymbol{V}\boldsymbol{\varphi}_j$ (apply potential)

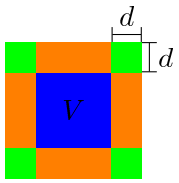
In this article, we describe a simplified and efficient way to implement and optimize the neighbor exchange with non-blocking collective operations that are defined on topology communicators.

2 Parallel Implementation

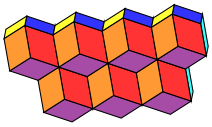
This section gives a detailed analysis of the communication and computation behavior of the domain parallelization and presents alternative implementations using non-blocking and topology-aware colletives that provide higher performance and better programmability.

2.1 Domain parallelization

The application of the Hamiltonian to an orbital $\boldsymbol{H}\boldsymbol{\varphi}_j$ can be parallelized by a partitioning of the real-space grid. The best decomposition depends on the distribution of grid-points in real-space which depends on the atomic geometry of the system under study. We use the library METIS [9] to obtain partitions that are well balanced in the number of points per node.



(a) An area V enlarged by d.



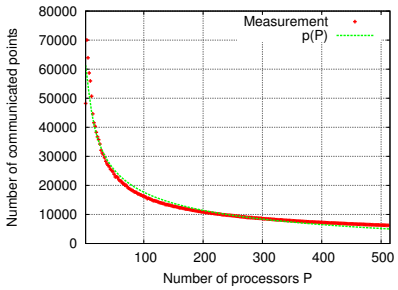
(b) Rhombic dodecahedrons, consisting of twelve rhombic faces, stacked to fill up space.

Fig. 2. Message sizes and number of neighbors.

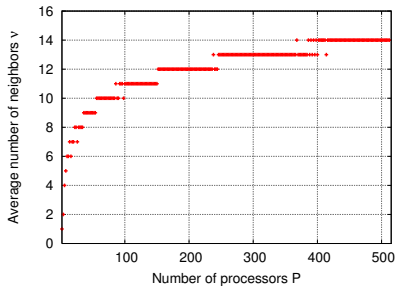
The communication overhead of calculating $\boldsymbol{H}\boldsymbol{\varphi}_j$ is dominated by the neighbor exchange operation on the grid. To determine a model to assess the scaling of the communication time which can be used to predict the application's running time and scalability, we need to assess the message-sizes, and the average number of neighbors of every processor. Both parameters are influenced by the discretization order d that affects how far the stencil leaks into neighbouring domains, and by the number of points in each partition. Assuming a nearly optimal domain decomposition, NP points in total, and P processors we can consider the ratio $V = NP/P$ as "volume" per node. The number of communicated points is $p(P) = V_d - V$ with V_d being the volume V increased by the discretization order d and reads

$$p(P) = \alpha d^3 + \beta d^2 \sqrt{V(P)} + \gamma d \sqrt[3]{V(P)^2} \tag{3}$$

with coefficients α, β, γ depending on the actual shape of the partitions. The derivation of (3) is sketched schematically in Fig. 2(a) for a 2D situation: an area V is increased by d in each direction. The enlargement is proportional to d^2 (green) and $d\sqrt{V}$ (red). In 3D, the additional dimension causes these terms to be multiplied by d and leads to one more term proportional to $d\sqrt[3]{V^2}$. Fig. 3(a) shows the number of exchanged points measured for a cylindrical grid of 1.2 million points and the analytical expression (3) fitted to the data-points.



(a) Exchanged points as a function of the number of processors for a large grid.



(b) Average and maximum number of neighbors ν per partition.

Fig. 3. Communicated points and neighbor-count for different numbers of processors.

Since the average number of neighbors (ν) depends on the structure of the input system, we cannot derive a generic formula for this quantity but instead

give the following estimate: METIS minimizes edge-cut which is equivalent to minimization of surfaces. This can be seen in Fig. 1(a) where the partition borders are almost between the gray Carbon atoms, the optimum in this case. In general, the minimal surface a volume can take on is spherical. Assuming the partitions to be stacked rhombic dodecahedrons as approximation to spheres, shown in Fig. 2(b), we conclude that, for larger P, ν is clearly below P because each dodecahedron has at maximum twelve neighbors. This consideration, of course, assumes truly minimum surfaces that METIS can only approximate. In practice, we observe an increasing number of neighbors for larger P, see Fig. 3(b). Nevertheless, the number of neighbors is an order of magnitude lower than the number of processors.

Applying the well-know LogGP model [4] to our estimations of the scaling of the message sizes and the number of neighbors ν, we can derive the following model of the communication overhead (each point is represented by an 8 byte double value):

$$t_{comm} = L + o\nu + g(\nu - 1) + G(\nu \cdot 8\, p(P)) \tag{4}$$

We assume a constant number of neighbors ν at large scale. Thus, the communication overhead scales with $O\left(\sqrt{NP/P}\right)$ in P. The computational cost of steps 2 and 3 that determines the potential to overlap computation and communication scales with NP/P for the potential term and $\alpha d^3 + \beta d^2\sqrt{NP/P} + \gamma d\sqrt[3]{(NP/P)^2} + \delta NP/P$ for the kinetic term.[2] We observe that our computation has a similar scaling behaviour as the communication overhead, cf. Eq. (4). We therefore conclude that overlapping the neighbor exchange communication with steps 2 and 3 should show a reasonable performance benefit at any scale.

Overlapping this kind of communication has been successfully demonstrated on a regular grid in [1]. We expect the irregular grid to achieve similar performance improvements which could result in a reduction of the communication overhead.

Practical benchmarks show that there are two calls that dominate the communication overhead of `octopus`. On 16 processors, about 13% of the application time is spent in many 1 real or complex value MPI_Allreduce calls caused by dot-products and the calculation of the atomic potentials. This communication can not be optimized or overlapped easily and is thus out of the scope of this article. The second biggest source of communication overhead is the neighbor communication which causes about 8.2% of the communication overhead. Our work aims at efficiently implementing the neighbor exchange and reducing its communication overhead with new non-blocking collective operations that act on a process topology.

2.2 Optimization with Non-blocking Collective Operations

Non-blocking collective operations that would support efficient overlap for this kind of communication are not available in the current MPI standard. We used

[2] The derivation of this expression is similar to (3) except that we shrink the volume by the discretzation order d.

the open-source implementation LibNBC [6] that offers a non-blocking interface for all MPI-defined collective operations.

Implementation with NBC_Ialltoallv The original implementation used MPI_Alltoallv for the neighbor exchange. The transition to the use of non-blocking collective operations is a simple replacing of MPI_Alltoall with NBC_Ialltoallv and the addition of a handle. Furthermore, the operation has to be finished with a call to NBC_Wait before the communicated data is accessed.

However, to achieve the best performance improvement, several additional steps have to be performed. The first step is to maximize the time to overlap, i. e., to move the NBC_Wait as far behind the respective NBC_Ialltoallv as possible in order to give the communication more time to proceed in the background. Thus, to overlap communication and computation we change the original algorithm to:

1. Initiate neighbor exchange (NBC_Ialltoallv)
2. $\varphi_j \leftarrow \boldsymbol{v}\varphi_j$ (apply potential)
3. $\varphi_j \leftarrow \varphi_j - \frac{1}{2}\nabla^2\varphi_j^{inner}$ (apply kinetic energy operator to inner points)
4. Wait for the neighbor exchange to finish (NBC_Wait)
5. $\varphi_j \leftarrow \varphi_j - \frac{1}{2}\nabla^2\varphi_j^{edge}$ (apply kinetic energy operator to edge points)

We initiate the exchange of neighboring points (step 1) and overlap it with the calculation of the potential term (step 2) and the inner part of the kinetic energy, which is the derivative of all points that can be calculated solely by local points (step 3). The last step is waiting for the neighbor-exchange to finish (step 4) and calculation of the derivatives for the edge points (step 5).

A usual second step to optimize for overlap is to introduce NBC_Test() calls that give LibNBC the chance to progress outstanding requests. This is not necessary if the threaded version of LibNBC is running on the system. We have shown in [5] that the a naively threaded version performs worse, due to the loss of a computational core. However, for this work, we use the InfiniBand optimized version of LibNBC [7] which does not need explicit progression with NBC_Test() if there is only a single communication round (which is true for all non-blocking operations used in `octopus`).

As shown in Sec. 2.1, the maximum number of neighbors is limited. Thus, the resulting communication pattern for large-scale runs is sparse. The MPI_Alltoallv function, however, is not suitable for large-scale sparse communication patterns because it is not scalable due to the four index arrays which have to be filled for every process in the communicator regardless of the communication pattern. This results in arrays mostly filled with zeros that still have to be generated, stored and processed in the MPI call and is thus a performance bottleneck at large-scale. Filling those arrays correctly is also complicated for the programmer and a source of common programming errors. To tackle the scalability and implementation problems, we propose new collective operations [8] that are defined on the well known MPI process topologies. The following section describes the application of one of the proposed collective operations to the problem described above.

Topological Collective Operations We define a new class of collective operations defined on topology communicators. The new collective operation defines a neighbor exchange where the neighbors are defined by the topology. MPI offers a regular (cartesian) topology as well as a graph topology that can be used to reflect arbitrary neighbor relations. We use the graph communicator to represent the neighborship of partitions generated by METIS for the particular input system. MPI_Graph_create is used to create the graph communicator. We implemented our proposal in LibNBC, the functions NBC_Get_neighbors_count and NBC_Comm_neighbors return the neighbor count and the order of ranks for the send/receive buffers respectively. The operation NBC_Ineighbor_xchg performs a non-blocking neighbor exchange in a single step.

Programmability It seems more natural to the programmer to map the output of a graph partitioner (e. g., an adjacency list that represents topological neighbors) to the creation of a graph communicator and simply perform collective communication on this communicator rather than performing the Alltoallv communication. To emphasize this, we demonstrate pseudocodes that perform a similar communication operation to all graph neighbors indicated in an undirected graph (`list[i][0]` represents the source and `list[i][1]` the destination vertex of edge `i` and is sorted by source node).

```
1  rdpls = malloc(p*sizeof(int)); sdpls = malloc(p*sizeof(int));
   rcnts = malloc(p*sizeof(int)); scnts = malloc(p*sizeof(int));
   for(i=0; i<p; i++) { scnts[i] = rcnts[i] = 0; }
   for(i=0; i<len(list); i++) if(list[i][0] == rank)
       scnts[list[i][1]] = count; rcnts[list[i][1]] = count;
6  sdispls[0] = rdispls[0] = 0;
   for(i=1; i<p; i++) {
      sdpls[i] = sdpls[i-1] + scnts[i];
      rdpls[i] = rdpls[i-1] + rcnts[i]; }
   NBC_Ialltoallv(sbuf, scnts, sdpls, dt, rcnts, rdpls, dt, comm, req);
11 /* computation goes here */
   NBC_Wait(req, stat);
```

Listing 1.1. NBC_Ialltoall Implementation.

Listing 1.1 shows the NBC_Ialltoall implementation which uses four different arrays to store the adjacency information. The programmer is fully responsible for administering those arrays. Listing 1.2 shows the implementation with our newly proposed operations that acquire the same information from the MPI library (topology communicator layout). The processes mapping in the created graph communicator might be rearranged by the MPI library to place tightly coupled processes on close processors (e. g. on the same SMP system). The collective neighbor exchange operation allows other optimizations (e. g. starting off-node communication first to overlap local memory copies of on-node communication). Due to the potentially irregular grid (depending on the input system), the number of points communicated with each neighbor might vary. Thus, we

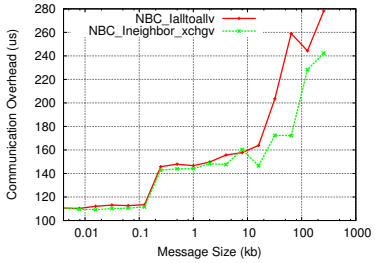
used the vector variant NBC_Ineighbor_xchgv to implement the neighbor exchange for `octopus`.

```
last = list[0][0]; counter = 0; // list is sorted by source
for(i=0; i<len(list); i++) {
 if(list[i][0] != last) index[list[i][0]] = counter;
 edges[counter++] = list[i][1];
}
MPI_Graph_create(comm, nnodes, index, edges, 1, topocomm);
NBC_Ineighbor_xchg(sbuf, count, dt, rbuf, count, dt, topocomm, req);
/* computation goes here */
NBC_Wait(req, stat);
```

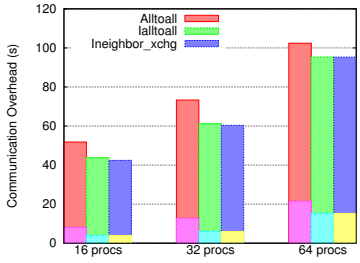
Listing 1.2. NBC_Ineighbor_xchg Implementation.

3 Performance Analysis

We benchmarked our implementation on the CHiC supercomputer system, a cluster computer consisting of nodes equipped with dual socket dual-core AMD 2218 2.6 GHz CPUs, connected with SDR InfiniBand and 4 GB memory per node. We use the InfiniBand-optimized version of LibNBC [7] to achieve highest performance and overlap. Each configuration was ran three times on all four cores per node (4-16 nodes were used) and the average values are reported.



(a) Overhead of LibNBC on 64 Processes on 16 Nodes.



(b) `Octopus` Communication Overhead on 4-16 Nodes.

Fig. 4. LibNBC and `octopus` communication overhead.

Fig. 4(a) shows the microbenchmark results for the overhead of NBC_Ialltoallv and NBC_Ineigbor_xchgv of *NBCBench* [6] with 10 neighbors under th assumption that the whole communication time can be overlapped. The overhead of the new neighbor exchange operation is slightly lower than the NBC_Ialltoallv overhead because the implementation does not evaluate arrays of size P. Fig. 4(b) shows the communication overhead of a fixed-size ground state calculation of a chain of Lithium and Hydrogene atoms. The overhead varies (depending on the technique used) between 22% and 25% on 16 processes. The bars in Fig. 4(b) show the total communication overhead and the tackled neighbor exchange overhead (lower part). We analyze only the overhead-reduction and easier implementation of the neighbor exchange in this work. The application of non-blocking neighbor collective operations efficiently halves the neighbor exchange overhead and thus

improves the performance of `octopus` by about 2%. The improvement is smaller on 64 processes because the time to overlap is due to the strong scaling problem much smaller than in the 32 or 16 process case. The gain of using the nearest neighbor exchange collective is marginal at this small scale. Memory restrictions prevented bigger strong-scaling runs.

4 Conclusions and Future Work

We proposed a new class of collective operations that enable collective communication on a processor topology defined by an MPI graph communicator and thus simplify the implementation significantly. We showed the application of the new operations to the quantum mechanical simulation program `octopus`. The communication overhead of the neighbor exchange operation was efficiently halved by overlapping of communication and computation improved the application performance.

Acknowledgements The authors want to thank Frank Mietke for support with the CHiC cluster system and Xavier Andrade for support regarding the implementation in `octopus`. This work was partially supported by a grant from the Lilly Endowment, National Science Foundation grant EIA-0202048 and a gift the Silicon Valley Community Foundation on behalf of the Cisco Collaborative Research Initiative.

References

1. T. Hoefler at al. Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. *Journal of Parallel Computing*, 33(9):624–633, 9 2007.
2. A. Castro, M. A. L. Marques, and A. Rubio. Propagators for the time-dependent kohn-sham equations. *The Journal of Chemical Physics*, 121(8):3425–3433, 2004.
3. A. Castro et al. `octopus`: a tool for the application of time-dependent density functional theory. *phys. stat. sol. (b)*, 243(11):2465–2488, 2006.
4. Albert Alexandrov et al. LogGP: Incorporating Long Messages into the LogP Model. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1995.
5. T. Hoefler et al. A Case for Standard Non-Blocking Collective Operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface 2007*, volume 4757, pages 125–134. Springer, 10 2007.
6. T. Hoefler et al. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, 11 2007.
7. T. Hoefler et al. Optimizing non-blocking Collective Operations for InfiniBand. In *22nd International Parallel & Distributed Processing Symposium*, 04 2008.
8. T. Hoefler, F. Lorenzen, D. Gregor, and A. Lumsdaine. Topological Collectives for MPI-2. Technical report, Open Systems Lab, Indiana University, 02 2008.
9. George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.
10. W. Kohn and L.J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev*, 140:A1133, 1965.
11. E. Runge and E. K. U. Gross. Density-functional theory for time-dependent systems. *Phys. Rev. Lett.*, 52(12):997, 1984.

A Formula for the message size

Given an arbitray surface S described by a function $r = r(\varphi, \theta)$ in sperical coordinates. The volume V of S is the integral

$$V = \int_0^{2\pi} d\varphi \int_0^{\pi} d\theta \sin\theta \int_0^{r(\varphi,\theta)} dr' r'^2(\varphi, \theta). \tag{5}$$

Enlarging this volume by a constant offset d in each direction, i. e. performing the transformation $r(\varphi, \theta) \to r(\varphi, \theta) + d$ yields the new volume integral

$$\begin{aligned} V_d &= \int_0^{2\pi} d\varphi \int_0^{\pi} d\theta \sin\theta \int_0^{r(\varphi,\theta)+d} dr' r'^2(\varphi, \theta) \\ &= \int_0^{2\pi} d\varphi \int_0^{\pi} d\theta \sin\theta \left[\frac{1}{3} r'^3(\varphi, \theta)\right]_0^{r(\varphi,\theta)+d} \\ &= \int_0^{2\pi} d\varphi \int_0^{\pi} d\theta \sin\theta \frac{1}{3} \left[r^3(\varphi, \theta) + 3r^2(\varphi, \theta)d + 3r(\varphi, \theta)d^2 + d^3\right]. \end{aligned} \tag{6}$$

Subtracting equation (5) from (6) we get the expression

$$V_d - V = \int_0^{2\pi} d\varphi \int_0^{\pi} d\theta \sin\theta \frac{1}{3} \left[3r^2(\varphi, \theta)d + 3r(\varphi, \theta)d^2 + d^3\right] \tag{7}$$

for the enlargement which we rewrite in terms of $V = V(r(\varphi, \theta)^3)$:

$$V_d - V = \int_0^{2\pi} d\varphi \int_0^{\pi} d\theta \sin\theta \frac{1}{3} \left[3\sqrt[3]{V}d + 3r\sqrt{V}d^2 + d^3\right] \tag{8}$$

Substituting $\frac{NP}{P}$ for V and introducing the shape-dependent constants α, β, γ, we obtain equation (3).

About the δ: $V = \int_0^{2\pi} d\varphi \int_0^{\pi} d\theta \sin\theta \int_0^{r(\varphi,\theta)} dr' r'^2(\varphi, \theta) = \int_0^{2\pi} d\varphi \int_0^{\pi} d\theta \sin\theta \left[\frac{1}{3} r'^3(\varphi, \theta)\right]_0^{r(\varphi,\theta)}$. Subtracting this expression from (6) yields the expression (8) for the enlargement. So, there is no δ. I admit, the final step is a bit handwavy. It just states that, for a given $r(\varphi, \theta)$ the expression for $V_d - V$ is composed of a term proportional to the surface of V, to the circumference of V and the the cube of the enlargement d. One can see it like this: Let's assume V is a cube with side length a, then the enlargement is $(a + d)^3 - a^3 = a^3 + 3a^2d + 3ad^2 + d^3 - a^3$. With $a = \sqrt[3]{V}$, $a^2 = \sqrt[3]{V^2}$, we get $\alpha\sqrt[3]{V^2}d + \beta\sqrt{V}d^2 + \gamma d^3$ and $\alpha = 3$, $\beta = 3$ and $\gamma = 1$. As every volume can be thought of as composed of infinitesimal cubes, one can generalize this statement, but I have not done the precise mathematical steps for this, i. e. doing the $\lim_{a\to 0}$ etc.