Documentation DDTBench Version 1.2

Robert Gerstenberger
Chemnitz University of Technology

Timo Schneider
ETH Zurich

Torsten Hoefler
ETH Zurich

January 27, 2014

Contents

1 Using DDTBench

This benchmark implements data access patterns from a wide range of applications (WRF, MILC, NAS LU/MG, FFT, SPECFEM3D_GLOBE and LAMMPS). Each of the micro-apps performs several tests whose parameters are derived from real application runs and input files. Please refer to Schneider, Gerstenberger, Hoefler: "Micro-Applications for Communication Data Access Patterns and MPI Datatypes" [2] for details!

1.1 Build Instructions

Currently this benchmark is implemented in Fortran as well as in C.

1. edit `Makefile.inc` to set `FC`, `CC`, `FCFLAGS`, etc.
   - choose the right parameter for `HRT_ARCH` according to your hardware to use a high resolution timer instead of MPI_Wtime
   - choose your measurement mode (time, papi, time+papi)
2. build: `make`
3. run: `mpiexec -n 32 ./ddtbench_{c,f90,c_onesided,f90_onesided}`

The DDTBench binaries come with a small command line interface to influence the number of measurements, which in the Fortran case is implemented by using Fortran 2003 features (`command_argument_count`, `get_command_argument`). If one of those break your build, please comment out the lines 74 to 110 in the file `src_f90/ddtbench.F90` .

The Makefile offers a test option (`make test NPROCS=n OUTER=n INNER=n`): this fetches the sources of the stable and unstable version of OpenMPI and MPICH, builds them and runs tests with each of the MPI versions for all four DDTBench versions. The benchmark will be executed with `NPROCS` processes with `OUTER` number of outer loops and `INNER` number of inner loops. If you don't specify one or all of the parameters, the default settings mentioned in `test.inc` will be used.

1.2 Running DDTBench

- command line interface:
  `ddtbench_{c,f90,c_onesided,f90_onesided} <oloop> <iloop>`
  If no or not all parameter are provided the following defaults are used:
  - outer loop (number of creations of MPI derived datatypes (MPI DDTs) for each micro-application): `oloop` = 10
  - inner loop (number of communication operations per MPI DDT): `iloop` = 20

- The DDTBench binaries run several different micro-applications (16 all together), where each micro-app has several runs for different input sizes.
- Nearly all the micro-applications are running as a ping-pong, except for the FFT micro-app, which performs an Alltoall. The one sided versions of the FFT micro-app substitutes the Alltoall with NPROC put operations per process. In all cases only rank 0 measures the time.
- One should run the benchmark with at least 2 processes.
- If you like to run measurements with PAPI, the PAPI events are chosen over the environmental variables `PAPI_EVT1` and `PAPI_EVT2`. One can run the benchmark with counting one or two events.

1.3 Output File

The output file is named `ddtbench.out` and has 9 columns. The first column is the testname. The second column specifies the type of benchmark (the "method', see below), the third column states the number of transferred bytes for that particular test configuration (i.e., the MPI_Type_size of the used datatype). The fourth column identifies the step in the benchmark (the "epoch", see section 3) to which the time (in microseconds) in the fifth column corresponds. The sixth and the eight column specify the name of an PAPI event, while the seventh and ninth column report the respective PAPI event count.

The 16 tests have the following names: `WRF_{x,y}_{sa,vec}`, `MILC_su3_zd`, `NAS_MG_{x,y,z}`, `NAS_LU_{x,y}`, `FFT{NPROCS}`, `SPECFEM3D_{cm,mt,oc}` and `LAMMPS_{atomic,full}`. The four methods are:

- `mpi_ddt`: benchmark using MPI derived datatypes (MPI DDTs)
- `manual`: benchmark using the original pack routines from the applications
- `mpi_pack_ddt`: benchmark using MPI_Pack/MPI_Unpack with MPI DDTs
- `reference`: benchmark that does a traditional ping-pong of the same datasize

The five epoch names are: `ddt_create_overhead`, `pack`, `communication`, `unpack` and `ddt_free_overhead`. If one epoch isn't used in one test, this epoch will be printed at the end of this test with a time of zero nanoseconds, so that all epochs are present for each test.

Note that no statistical aggregation is done by the benchmark itself, the full information about each measured value is given to the user. A sample R analysis script is provided under `analysis_scripts/ddt_analysis.R`.

2 Input Parameter/Datatype Description

General: If data from more than one array has to be communicated, the different subarrays are packed together with MPI_Type_create_struct where the displacements of each array are relative to MPI_BOTTOM. The displacements are obtained by calls to MPI_Get_address.

WRF: WRF exchanges, in so called halo/period functions (about 30 to 40), faces of several arrays. Those arrays and their subarrays can be two, three or four dimensional. The faces are exemplary implemented with different MPI datatypes. The `WRF_{x,y}_sa` case uses MPI_Type_create_subarray, while the `WRF_{x,y}_vec` case uses nested MPI vectors in the following way:

- 2D: MPI_Type_vector
- 3D: MPI_Type_hvector of MPI_Type_vector
- 4D: MPI_Type_hvector of MPI_Type_hvector of MPI_Type_vector

The faces of different arrays are packed together as described above. The x and y denotes the communication in x and y direction of the processor grid. The datatypes are built in the same way, but with different input parameters.

The input parameter for the array boundaries were extracted from runs of the `em_b_wave` ideal case with different numbers of processes (4, 9, 16, 25, 64). The number of arrays is static (4x 2Ds, 3x 3Ds and 2x 4Ds) and is exemplary for the halo functions in WRF.

MILC: MILC represents space time with a four dimensional grid (Lx * Ly * Lz * Lt), decomposed with checkerboarding. It exchanges gluons from various positions. The datatype for the ZDOWN direction is build in the following way:

1. The base type is a contiguous type of six floats (in C) or 3 complex types (in Fortran).
2. The next datatype is a vector type of the above mentioned contiguous type with Lt blocks, each block contains Lx * Ly/2 elements and has a stride of Lx * Ly * Lz/2 elements.
3. The final datatype is a hvector type with 2 blocks of the above mentioned vector type. The stride depends from the model used, which defines among other things the distance between odd and even elements. For the test the hypercubic model was chosen, in which the stride is Lx * Ly * Lz * Lt/2 elements.

The input parameter were obtained by running MILC with 1024 processes with different grid sizes (from 64x64x32x32 to 128x128x64x64).

NAS MG: The NAS MG benchmark performs a face exchange in each direction of a cube (3D array). Each communication direction (x,y,z) was modeled since each surface needs a different MPI DDT and performs differently.

- x (yz surface): a hvector of vector with stride between every element of the surface

- y (xz surface): a vector with a stride after each line in the x dimension
- z (xy surface): a vector with a stride after each line in the x dimension, the stride is much smaller than in the y direction

The input parameter were obtained by running the classes S,W,A,C with 4 processes.

NAS LU: The NAS LU benchmark performs a 1D face exchange in x and y direction from a 2D array. Each surface needs a different datatype and performs differently. Each grid point consists of 5 doubles.

- x: a contiguous type
- y: A vector type with a stride between each grid point, which is modeled as a contiguous type.

The grid sizes of the classes S,W,A-E were directly taken as input parameters for the array size on each node.

FFT: This test performs with an Alltoall a transpose of a distributed matrix with interleaved vector types (different on sender and receiver side). For further details see [2]. The one sided version issues `NPROCS` puts per process instead of an MPI_Alltoall.

SPECFEM3D_mt: The SPECFEM3D_GLOBE performs the assembly of a distributed matrix in conjunction with a transpose of that matrix. To go along with the ping-pong scheme the sender assembles a xz plane using MPI DDTs (a vector type), and the receiver stores it as a xy plane and doesn't use MPI derived datatypes since the data is already contiguous.

The input parameter were obtained by running SPECFEM3D_GLOBE with 600 processes (10x10x6) with a constant c of 1,2,3 and 4, which is equal to `nproc_eta` = `nproc_xi` = 80, 160, 240 and 320.

LAMMPS: LAMMPS exchanges particles with different properties. There is a different array for each property, where the information for every particle is stored. A list stores the position of all the particles that will be transfered. This list is modeled as a MPI indexed type and since all the blocks of one property have the same size, MPI_Type_create_indexed_block is used. Since the blocklengths of different properties aren't the same, several MPI indexed types are used. To assemble the different arrays a struct type is used. On the receiver side the properties are stored at the end of each array with a different MPI DDT.

- sender side: struct of several MPI_Type_create_indexed_block
- receiver side: struct of several contiguous types

In the full case all of the six properties are present, while in the atomic case only four are used.

The input parameter for the full case were extracted from running the peptide example with 2, 4 and 8 processes. For the atomic case the crack example with 2, 4 and 8 processes was used.

SPECFEM3D: SPECFEM3D_GLOBE exchanges the acceleration data of grid points. The different layers of the earth have different relevant directions:

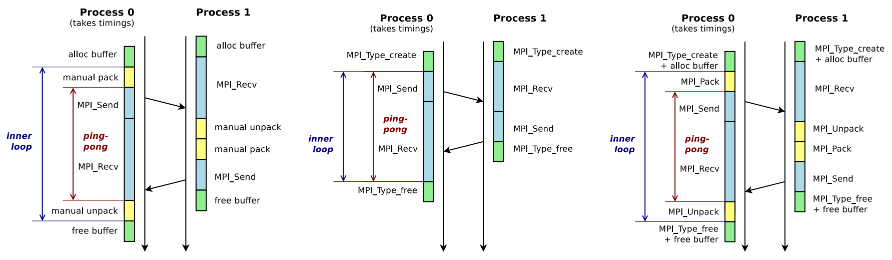
- oc (outer core): described by one float
- cm (crust mantle/inner core): described by three floats

Only some grid points will be exchanged (which is statically determinated by the mesher). The list of grid points is modeled as an MPI_Type_create_indexed_block. Since in the cm case the acceleration data of two different layers (crust mantle and inner core) is exchanged, we use a struct type on top of those MPI indexed types. The lists for the crust mantle and the inner core are different and so are the MPI indexed types.

The input parameters for this benchmarks were obtained in the same way as for the `SPECFEM3D_mt` micro-application.

3 Evaluation Scheme

3.1 Message Passing Scheme



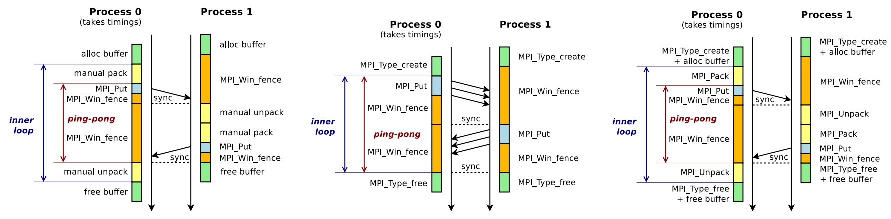
(a) Manual Pack Loop (b) Send/Recv with DDTs (c) MPI_Pack

Figure 1: Measurement loops for the micro-applications with the message passing scheme (Source: [2])

Each micro-application is implemented as a benchmark with a ping-pong scheme between two processes using MPI_Send and MPI_Recv. Three different methods are used to assemble and communicate the data. The manual method (Figure 1(a)) uses the pack and unpack loop, that were found in the original application, to serialize/deserialize the data into/from a contiguous buffer. With this method only contiguous data is communicated. The second method

(Figure 1(b)) specifies MPI derived datatypes directly in the MPI_Send and MPI_Recv calls, so optimizations of the pack/unpack process and the communication are entirely handled by the MPI library. The third method (Figure 1(c)) uses MPI_Pack and MPI_Unpack with MPI derived datatypes to serialize/deserialze the data into/from a contiguous user-specified buffer, so that only contiguous data is communicated. Additionally a traditional ping-pong with contiguous data is performed to serve as a reference measurement. Each method is divided into two nested loops and the number of iterations per loop can be configured via command line parameters (see 1.2). The outer loop is used to measure static overhead like MPI DDT creation/destruction or the (de)allocation of user buffer space while the inner loop measures the time of the packing and the communication. To avoid synchronized clocks, time is only measured on one process (Process 0 in Figure 1). The reported communication time of process 0 contains the complete ping-pong (including the pack and unpack epochs on process 1). This has to be considered, if you want to write your own analysis scripts.

3.2 One Sided Scheme



(a) Manual Pack Loop (b) Put with DDTs (c) MPI_Pack

Figure 2: Measurement loops for the micro-applications with the one sided scheme (Original source for 2(a) and 2(b): [1])

The one sided scheme was introduced in DDTBench Version 1.2. In MPI One Sided one process specifies the local and the remote parameters for a communication call, so the target isn't involved in the communication. In message passing synchronization is done implicitly, while in the one sided context it has to be done explicitely. The one sided scheme of DDTBench replaces the MPI_Send and MPI_Recv calls with MPI_Put and a subsequent call to MPI_Win_fence (a global synchronization) to ensure memory consistency and notify the target that the communication is finished. The use of MPI_Win_fence allows DDTBench to evaluate implementations that are only MPI-2 compliant. To be comparable with the message passing scheme, the reported time for the communication epoch is the combination of the communication and the synchronization time, so that the same analysis scripts can be used for both schemes. The time to create MPI windows is not reported and is done before the measurement loops.

4 Citation

Any published work which uses this software should include one of the following citations:

T. Schneider, R. Gerstenberger and T. Hoefler: Micro-Applications for Communication Data Access Patterns and MPI Datatypes. In Recent Advances in the Message Passing Interface (EuroMPI'12) [2]

---

or

T. Schneider, R. Gerstenberger and T. Hoefler: Application-Oriented Ping-Pong Benchmarking: How to Assess the Real Communication Overheads. In Journal of Computing, May 2013 [3]

If you refer to the one sided scheme, please include the following citation:

R. Gerstenberger and T. Hoefler: Handling Datatypes in MPI-3 One Sided. ACM Student Research Competition Poster at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13) [1]

References

[1] R. Gerstenberger. Handling datatypes in MPI-3 one sided. In *ACM Student Research Competion Poster at the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*, Nov. 2013.

[2] T. Schneider, R. Gerstenberger, and T. Hoefler. Micro-applications for communication data access patterns and MPI datatypes. In *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, volume 7490, pages 121–131. Springer, Sep. 2012.

[3] T. Schneider, R. Gerstenberger, and T. Hoefler. Application-oriented ping-pong benchmarking: How to assess the real communication overheads. *Journal of Computing*, May 2013. doi: 10.1007/s00607-013-0330-4.