Auto-vectorized MPI Reduction Operations: or GCC is Better than Supposed, Part 1

Dave Love
dave.love@manchester.ac.uk

2021-10-13

Vectorizing code for MPI reduction operations was recently shown to be useful. Hand-written intrinsics were used for several x86_64 micro-architectures on the basis on the basis that a compiler failed to auto-vectorize an example in the original code, although recent GCC succeeds with around half the loops. Simple changes to the original code allow GCC to vectorize most operations — more than done by hand for x86_64. Further simple changes allow recent GCC and GNU libc to provide run-time dispatch on micro-architecture without the previous manual programming. The result carries over to implementations for ARM and POWER without needing to write more target-specific code with potential bugs. This may be a useful tutorial example as well as potentially improving Open MPI.

1 Motivation and Previous Work

Zhong et al [1] discussed speeding up MPI reductions by vectorizing the operations, which was shown to have a significant effect. They discussed at length the use of SIMD code in linear algebra implementations (surprisingly without reference to BLIS [2, 3]), and stressed the use of AVX512 compiler intrinsics.[1] The implementation is x86_64-specific (unless it works sufficiently well with something like SIMD Everywhere on other architectures), and there seems not to be proper consistency between compilers' intrinsics implementations. While compiler auto-vectorization was rejected in [1], this work presents an alternative view from experience, and exercises the compiler. It was done initially to support a POWER9 system easily, but also provides better support for x86_64.

The prior work in the paper's discussion is mostly concerned with GEMM in Level 3 BLAS. In contrast to Level 1 (like reductions) that requires careful structuring of the code, matching the caches on the target prior to vectorization to achieve close to peak performance, and probably manual pre-fetching. It is well-known then that performance is roughly proportional to SIMD width, and doubled by fuse-multiply-add instructions.

Hand-crafted vectorization may not be necessary, however. BLIS' generic C ('reference') GEMM kernel compiled with reasonably recent GCC achieves ∼75% the performance of the hand-optimized Haswell AVX2 DGEMM implementation; it is likely that the gap is down to prefetching. The paper downplays the ability of compilers to auto-vectorize and mentions problems making that work in terms of flags etc. At least for the loops under consideration, that isn't an issue, compared with admittedly time-consuming and error-prone hand coding. The default flags that Open MPI configures already include vectorization for at least GCC, Intel, XL, Clang, and PGI.

[1]That requires care in linear algebra implementations like this, and elsewhere, due to the effect of possible clocking down of the CPU using full-width AVX512 instructions. That means it isn't a win for BLAS on Skylake CPUs with only one FMA unit. (There may also be an effect from AVX(2), but smaller, and normally ignored.) Since reductions don't have high computational intensity, a priori one might expect at least full-width AVX512 to be a loss, and the case for it is unclear in an Open MPI issue.

In contrast to the suggestion in [1], every current compiler to hand on GNU/Linux (GCC, Intel, IBM XL, PGI, and LLVM variants) can vectorize at least the sort of simple loops under consideration. BLIS, for instance, now does with the C implementation of Level 1 operations like AXPY, despite initial claims that GCC didn't work.[2] The statement in [1] that a compiler was unable to vectorize the Open MPI code would be expected for GCC 4.8 generally, and with later GCC for the two-buffer implementations. However, that is fixed simply with the help of GCC's diagnostics.

Despite myths about its poor optimization, the code GCC generates in this case appears good. Its competitive performance on a computational benchmark set is left to subsequent note.

2 Implementation

The implementation of [1] is now in Open MPI 4.1. It lives in directory `ompi/mca/op/avx`, with the original C implementation in the sibling `base`. This work is based on the 4.1.1 release, but `op` was even with the master branch at the time. With only `base` and `avx`, POWER (the target of interest) and ARM systems would appear not to benefit from vectorized reductions in the released code. In fact, recent GCC, e.g. 8, vectorizes a good number of the operations as-is, but maybe not the most relevant ones.

2.1 Fixes for Auto-vectorization

There are two things that typically need to be fixed to enable vectorization of simple loops without dependencies or conditionals. One is to eschew IEEE-754 semantics for floating point operations, e.g. using GCC's `-funsafe-math-optimizations`. Although not strictly correct, that doesn't cause test suite failures in BLIS, for instance, and MPI doesn't require the strictness anyway. That is necessary for reductions, but the functions of interest here aren't actually reductions, even if they support `MPI_Reduce...` operations. However, ARM needs it to vectorize at all with Neon, per the GCC documentation, so it is used generally.

The other, most relevant here, is to arrange Fortran-style 'storage association' semantics (basically avoid possible aliasing though pointers), which in C amounts to using `restrict` appropriately.[3] That's obviously reasonable for Fortran-callable C code, and is assumed by the `avx` implementation.

It is obviously also necessary to use an appropriate optimization level that includes vectorization, e.g. GCC's `-O3` (or `-Ofast`, which includes also `-funsafe-math-optimizations`), and specify an appropriate target.

2.2 Analysis and Modifications for Vectorization

The steps to improve the generic C implementation in `base/op_base_functions.c` are as follows. They were initially done with GCC 10 on POWER. (GCC 10 provides cleaner diagnostics than version 8, which is normally used on that RHEL7 POWER9 system, but doesn't produce significantly different optimization.)

First we need to ensure that it is compiled with suitable options. Although Open MPI defaults to `-O3`, it seems safest to specify it to ensure vectorization in case non-default flags are configured. Thus, in a preprocessor conditional for GCC 4.8 and above, add it along with simple unrolling (which isn't included at any optimization level), and lax floating point maths for Neon:

```
#pragma GCC optimize ("O3", "unsafe-math-optimizations", "unroll-loops")
```

To get information on GCC's optimization, in the previously-built source, re-compile just `op_base_functions` with additional diagnostics, e.g. `-fopt-info-vec-missed`. Delete `ompi/mca/base/op_base_functions.lo` and run something like

```
make -C ompi/mca/op V=1 | grep op_base_functions.lo
```

[2]The `omp simd` pragmas BLIS uses are not actually necessary.

[3]However, the compiler may 'version' a loop to account for aliasing. That can be seen in optimization reports with recent GCC for functions in the original op code.

to see the compilation command line used by `libtool`. Run that from the `op` directory with additional option `-fopt-info-vec-missed`. To find loops which failed, and why,[4] log the stderr, or pipe it into `less` and look for

```
missed: couldn't vectorize loop
```

The failures are mostly from the aliasing analysis. To fix that, add `restrict` to the pointer arguments of the functions in the template macros like `OP_FUNC`. (`configure` requires C99, so `restrict` is available.) As well as, for instance, `in` and `out`, do `count` to avoid a potentially puzzling diagnostic that the number of iterations cannot be computed. (That diagnostic appears many times elsewhere in Open MPI for loops that might benefit similarly if they're time-consuming.) While it's not strictly necessary to fix the `3buff` versions for GCC due to loop versioning, it's worth doing for consistency and possibly to help other compilers.

Just that gets the bulk of the operations vectorized. An exception is the logical operations, with diagnostic "`control flow in loop`". It arises from the 'short-circuiting' semantics of C's boolean operators, i.e. (assuming the variable values are either 0 or 1):

`a && b` $\equiv$ `a ? a : b`

Short-circuiting isn't useful when we're running through the whole buffer so, after checking the equivalence modulo short-circuiting in the C standard, modify the relevant operations by replacing boolean operators with bit-wise equivalents: `&` instead of `&&`, etc. The result is vectorizable. (An obvious refinement would be to introduce amortised short-circuiting tests, assuming that it usually isn't necessary to process the whole buffer. A test for breaking the main loop could go after processing a block in a vectorizable sub-loop after unrolling the loop to an appropriate depth. That extra work hasn't been done.)

With that out of the way, most operations are vectorized, with loop versioning avoided. At least on ppc64le, the 128-bit datatypes are missed ("`no vectype for stmt`"), perhaps surprisingly with hardware 128-bit support. The other omission is `minloc` and `maxloc`, which may or may not be amenable to vectorization; that hasn't been investigated.

Notes Successful optimization reports from `-fopt-info` typically have output for multiple loops at the same source line, probably with one then removed by unrolling. The original loops are peeled to take care of remainder iterations from the runtime loop count modulo the SIMD vector size. Typically the peeled code uses a shorter SIMD vector size than the main loop, if available. The result isn't obviously slower than the Duff's device implementation of [1]. Various functions are recognized as being the same, and aliased rather than generating code for each, notably plain and Fortran-callable versions.

2.3 Dispatch by Micro-architecture

Although most operations are now vectorized, that's only with code for a single target (defaulting to SSE2 on x86_64). We may want to generate code to use multiple ISAs/micro-architectures and benefit from the best level of support on a given system. POWER only has VSX SIMD to worry about (assuming other differences between POWER8 and 9, currently, don't matter), but ARMv8 may have SVE(2) as well as Neon SIMD. x86_64, of course, isn't that simple. . . . Presumably nothing else matters for HPC currently.

The `avx` implementation builds separate objects for AVX, AVX2, and AVX512, and dispatches on the basis of run-time detected support, possibly overridden with MCA parameters. If only the operations supplied in `avx` are or interest there's no use for the optimized `base`, but we can consider what to do with the simple approach which turns out to provide more anyway.

One solution would be to re-use the `avx` framework and build a version of the base functions for each target of interest, with macros to mangle the function names appropriately in each case. POWER and ARMv8 can do something similar to `cpuid`-based dispatch in the `avx` framework.[5]

With GCC 6+ on recent-enough GNU/Linux, like Debian 11 (since it depends on glibc 2.23+ for support), the easy solution is to use the `target_clones` function attribute. That generates an

[4]Consult the GCC Internals Manual if you see the obscure term 'latch'.

[5]An example for Linux including ppc64le and aarch64 is `https://github.com/loveshack/blis/blob/power/frame/base/bli_cpuid.c`.

	SSE2	AVX	AVX2	AVX512	VSX
GCC 4.8.5	172	192	192		
GCC 10.2.1	184	208	214	214	210
intrinsics		117	117	129	
GCC 8, original	78	86	97	105	86

Table 1: Number of vectorized loops for the final code by x86_64 micro-architecture and for POWER9 VSX. 'intrinsics' is the `avx` version. The last line is for the original version 4.1.1 code.

optimized version for each of the specified list of targets, with runtime dispatch. It doesn't allow selecting specific targets at run time, but that shouldn't be needed outside testing. That is good enough for this demonstration.

So, add a macro, say `TARGETS`, to each operation template macro like `OP_FUNC`, defined appropriately for the architecture and compiler. `TARGETS` should default to empty but, under conditionals for GCC, glibc, and `__x86_64__` it can be defined as[6]

```
#define TARGETS __attribute__ ((target_clones \
  ("avx,arch=haswell,arch=knl,arch=skylake-avx512,default")))
```

Without `prefer-vector-width=512` flag added to a `target` pragma, say, half width (256-bit) AVX512 is used to avoid the potential down-clocking effect of using the full width. That isn't an obvious performance problem, as below.

For POWER, we can distinguish `power9` from the `power8` default similarly, but it isn't clear if that is worthwhile, and clones aren't supported by glibc on the system of interest anyway. Both use VSX SIMD. POWER9's 64-bit integer additions don't get any more loops vectorized. GCC 11 supports POWER10, given a recent gas, which vectorizes and extra six loops (64-bit integer products).

On ARMv8, the SIMD choices are Neon (always present) and SVE(2). An `a64fx` target clone was added to cover SVE, but the implementation hasn't been run on ARM.

`target_clones`, or an equivalent, appears not to be available in Clang or any other compiler, although Intel's has an option to do something similar by compilation unit rather than function. If one must use another C compiler than GCC, where `target_clones` would otherwise work, the modified code will at least vectorize for its default target. However, there's no obvious reason to avoid GCC to compile the basic MPI library, even if a different compiler is used for the non-portable Fortran interface; the ABI is well-defined, and GCC generally produces good code.

Availability The final implementation discussed is currently available from SourceHut.

3 Results

3.1 Compiler Vectorization Coverage v. Intrinsics

Though POWER was the target of interest, it's worth comparing with the `avx` version on x86_64. The vectorized `base` loops can be counted by adding `-fopt-info-vec-optimized` to the GCC flags and piping stderr into

```
grep -i 'loop vectorized' | sed 's/ using .*$//' | sort -u | wc -l
```

Examining the `avx` build, a way to count the operations it implements is

```
for a in '' 2 512; do nm avx/.libs/liblocal_ops_avx$a.a |
 grep -c ompi_op_avx_[23]buff; done
```

Results are listed in table 1. The modified `base` is more comprehensive even with the RHEL7 base GCC (4.8), especially as it treats Fortran operation versions as aliases, so it will supplement `avx` usefully. It is similarly comprehensive on POWER.

[6]KNL may not be worthwhile these days, but it needs a different AVX512 subset from SKX.

3.2 Vectorized Code Quality

A sample operation (`ompi_op_base_2buff_prod_double`) for the SKX target was examined with the MAQAO [4] code quality analyzer and compared with the equivalent `avx` version, `mpi_op_avx_2buff_mul_double_avx512`.[7] CQA showed the main loop to be fully vectorized, but complained about the default 256-bit width for SKX code. The full width is probably not worthwhile with low computational intensity, as above. That is supported by the benchmark results below. MAQAO spotted that loop unrolling was missed initially, which was included in the final version. (`-O3`'s unroll-and-jam is for loop nests.)

3.3 Benchmark

Some minimal speed tests were done on x86_64 and POWER9 with the new code. The source contains a benchmark in `test/datatype` (built with `make reduce_local`) which seems to be used by the developers for testing, so that was run.

3.3.1 AVX

For AVX512, GCC 8 was used and the result run under Debian 11 on a 3.6 GHz SKX system with (two AVX512 FMA units). The modified `base` code produced essentially the same times within variability for the compiler- and manually-vectorized implementations. (Note that the former was using half-width AVX512 and the latter full-width.) Many repetitions and large arrays are needed for sufficiently long runs to time them reliably, and it isn't clear how meaningful that is. Longer runs do provide a reasonable number of samples for `perf`, which is more useful.

So `reduce_local` was run under `perf` with short and long double precision buffers (considering the 32 KB L1 and 8 MB L2 caches). Other operations and data types haven't been checked. The CPU was clocked down by the same amount in each case according to `perf` (to 3.5 GHz for long buffers and 3.4 GHz for short ones), so the `avx` and non-`avx` cycle count percentages are directly comparable.

The results are in figures 1 and 2. The difference in functions of the two implementations, like `prod_fortran_real8` v. `mul_double`, is due to the choice of alias for ones with identical code noted above. Note the substantial cost paid to `memmove` (for alignment effects?).

3.3.2 POWER9

Similar measurements were made on the RHEL 7 POWER9 system (with 32 KB L1 and 10 MB L3 caches) that was the original interest, using GCC 8.4. In this case the original and modified `base` were compared. The results in figure 3 and figure 4 only show benefit from vectorization with short buffers. (The `2buff` loops are not auto-vectorized in the original code, unlike the `3buff` ones.)

References

[1] Dong Zhong et al, 2020. Using Advanced Vector Extensions AVX-512 for MPI Reductions. In *EuroMPI/USA '20 27th European MPI Users' Group Meeting*, 1–10. `https://doi.org/10.1145/3416315.3416316`, `https://www.icl.utk.edu/files/publications/2020/icl-utk-1416-2020.pdf`.

[2] Field G. Van Zee and Robert A. van de Geijn, 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. ACM Trans. MathṠoft. 41, 3, 14-33. `https://doi.acm.org/10.1145/2764454`, `https://www.cs.utexas.edu/users/flame/pubs/blis1_toms_rev3.pdf`.

[3] BLAS-like Library Instantiation Software Framework. `https://github.com/flame/blis`.

[4] MAQAO (Modular Assembly Quality Analyzer and Optimizer)`http://www.maqao.org/`.

[7]MAQAO was unhappy with `target_clones` code, so it was run on the base version without clones, with explicit `-m`.

```
$ mpirun --bind-to core -n 1  perf record ./reduce_local \
  -l 1000 -u 1000 -s 64 -t d -r 100000 -o prod,sum,min
MPI_PROD MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000013 0.00000016 0.00000016 0.00000016
MPI_SUM  MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000012 0.00000016 0.00000016 0.00000016
MPI_MIN  MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000012 0.00000016 0.00000016 0.00000016

# Event count (approx.): 5002050711
# Overhead  Shared Object         Symbol
# ........  ...................  ................................................
    78.40%  reduce_local         [.] main
     5.61%  libc-2.31.so         [.] __memmove_avx_unaligned_erms
     3.39%  libmpi.so.40.30.1    [.] PMPI_Reduce_local
     3.14%  mca_op_avx.so        [.] ompi_op_avx_2buff_mul_double_avx512
     2.40%  mca_op_avx.so        [.] ompi_op_avx_2buff_min_double_avx512
     2.35%  mca_op_avx.so        [.] ompi_op_avx_2buff_add_double_avx512


$ mpirun --mca op ^avx --bind-to core -n 1  perf record ./reduce_local \
  -l 1000 -u 1000 -s 64 -t d -r 100000 -o prod,sum,min
MPI_PROD MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000011 0.00000016 0.00000016 0.00000016
MPI_SUM  MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000011 0.00000016 0.00000016 0.00000016
MPI_MIN  MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000011 0.00000016 0.00000016 0.00000016

# Event count (approx.): 4985446738
# Overhead  Shared Object      Symbol
# ........  ................. ..............................................................
    80.41%  reduce_local      [.] main
     4.91%  libc-2.31.so      [.] __memmove_avx_unaligned_erms
     2.54%  libmpi.so.40.30.1 [.] PMPI_Reduce_local
     2.48%  libmpi.so.40.30.1 [.] ompi_op_base_2buff_sum_fortran_real8.arch_skylake_avx512.3
     2.34%  libmpi.so.40.30.1 [.] ompi_op_base_2buff_min_fortran_real8.arch_skylake_avx512.3
     2.14%  libmpi.so.40.30.1 [.] ompi_op_base_2buff_prod_fortran_real8.arch_skylake_avx512.3
```

Figure 1: Output and perf results for reduce_local on x86_64 with short buffers (reformatted for presentation).

```
$ mpirun -n 1 perf record ./reduce_local -l 1048576 -u 1048576 \
  -s 64 -t d -r 100 -o prod,sum,min
MPI_PROD MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00116271 0.00115893 0.00117254 0.00116126
MPI_SUM  MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00116873 0.00116338 0.00116366 0.00116891
MPI_MIN  MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00120915 0.00120909 0.00121301 0.00121441

# Event count (approx.): 13807767124
# Overhead  Shared Object     Symbol
# ........  ................. ...........................................
    40.25%  reduce_local      [.] main
    25.72%  libc-2.31.so      [.] __memmove_avx_unaligned_erms
    11.52%  libmpi.so.40.30.1 [.] ompi_op_avx_2buff_min_double_avx512
    11.13%  libmpi.so.40.30.1 [.] ompi_op_avx_2buff_add_double_avx512
    11.03%  libmpi.so.40.30.1 [.] ompi_op_avx_2buff_mul_double_avx512


$ mpirun --mca op ^avx -n 1 perf record ./reduce_local -l 1048576 -u 1048576 \
  -s 64 -t d -r 100 -o prod,sum,min
MPI_PROD MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00110719 0.00111214 0.00112716 0.00110833
MPI_SUM  MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00110870 0.00111356 0.00111279 0.00111901
MPI_MIN  MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00114017 0.00115219 0.00115465 0.00115664

# Event count (approx.): 13315039004
# Overhead  Shared Object      Symbol
# ........  ................  ............................................................
    40.28%  reduce_local      [.] main
    26.01%  libc-2.31.so      [.] __memmove_avx_unaligned_erms
    11.37%  libmpi.so.40.30.1 [.] ompi_op_base_2buff_min_fortran_real8.arch_skylake_avx512.3
    10.99%  libmpi.so.40.30.1 [.] ompi_op_base_2buff_prod_fortran_real8.arch_skylake_avx512.3
    10.98%  libmpi.so.40.30.1 [.] ompi_op_base_2buff_sum_fortran_real8.arch_skylake_avx512.3
```

Figure 2: Output and perf results for reduce_local on x86_64 with long buffers (reformatted for presentation).

```
# New code
$ mpirun --bind-to core -n 1  perf record ./reduce_local -l 1000 -u 1000 \
  -s 64 -t d -r 100000 -o prod,sum,min
MPI_PROD MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000064 0.00000064 0.00000061 0.00000064
MPI_SUM  MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000061 0.00000064 0.00000061 0.00000064
MPI_MIN  MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000000 0.00000070 0.00000067 0.00000070

# Event count (approx.): 7888156140
# Overhead  Shared Object                 Symbol
# ........  .......................  ......................................................
    40.74%  lt-reduce_local          [.] main
    12.82%  libmpi.so.40.30.1        [.] ompi_op_base_2buff_sum_double
    11.16%  libmpi.so.40.30.1        [.] ompi_op_base_2buff_min_double
    11.07%  libmpi.so.40.30.1        [.] ompi_op_base_2buff_prod_double
    10.08%  libc-2.17.so             [.] __memcpy_power7
     4.17%  libmpi.so.40.30.1        [.] PMPI_Reduce_local
     2.17%  [vdso]                   [.] __do_get_tspec


# Original 4.1.1
$ mpirun --bind-to core -n 1  perf record ./reduce_local -l 1000 -u 1000 \
  -s 64 -t d -r 100000 -o prod,sum,min
MPI_PROD MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000121 0.00000121 0.00000121 0.00000121
MPI_SUM  MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000121 0.00000121 0.00000121 0.00000121
MPI_MIN  MPI_DOUBLE 64 count 1000 time (seconds/shifts) 0.00000000 0.00000130 0.00000131 0.00000130

# Event count (approx.): 10482521509
# Overhead  Shared Object             Symbol
# ........  .........................  .........................................................
    34.28%  lt-reduce_local            [.] main
    17.48%  libmpi.so.40.30.1          [.] ompi_op_base_2buff_min_double
    15.50%  libmpi.so.40.30.1          [.] ompi_op_base_2buff_prod_double
    15.19%  libmpi.so.40.30.1          [.] ompi_op_base_2buff_sum_double
     7.62%  libc-2.17.so               [.] __memcpy_power7
```

Figure 3: Output and perf results for reduce_local with short buffers on POWER9 (reformatted for presentation).

```
# New code
$ mpirun --bind-to core -n 1 perf record ./reduce_local -l 1048576 -u 1048576 \
  -s 64 -t d -r 100 -o prod,sum,min
MPI_PROD MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00062897 0.00063674 0.00061439 0.00062571
MPI_SUM  MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00063022 0.00063336 0.00062052 0.00062694
MPI_MIN  MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00064590 0.00065579 0.00063064 0.00065148

# Event count (approx.): 9234819475
#
# Overhead  Shared Object           Symbol
# ........  ......................  ..............................................................
#
    40.33%  lt-reduce_local         [.] main
    24.35%  libc-2.17.so            [.] __memcpy_power7
    10.74%  libmpi.so.40.30.1       [.] ompi_op_base_2buff_min_double
    10.18%  libmpi.so.40.30.1       [.] ompi_op_base_2buff_sum_double
    10.04%  libmpi.so.40.30.1       [.] ompi_op_base_2buff_prod_double


# Original 4.1.1
$ mpirun --bind-to core -n 1 perf record ./reduce_local -l 1048576 -u 1048576 \
  -s 64 -t d -r 100 -o prod,sum,min
MPI_PROD MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00065604 0.00063306 0.00061443 0.00062531
MPI_SUM  MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00063490 0.00063549 0.00061839 0.00062829
MPI_MIN  MPI_DOUBLE 64 count 1048576 time (seconds/shifts) 0.00065571 0.00067283 0.00064353 0.00066278

# Event count (approx.): 9225501608
# Overhead  Shared Object           Symbol
# ........  ......................  ..............................................................
    40.34%  lt-reduce_local         [.] main
    24.23%  libc-2.17.so            [.] __memcpy_power7
    10.81%  libmpi.so.40.30.1       [.] ompi_op_base_2buff_min_double
    10.14%  libmpi.so.40.30.1       [.] ompi_op_base_2buff_prod_double
    10.12%  libmpi.so.40.30.1       [.] ompi_op_base_2buff_sum_double
```

Figure 4: Output and perf results for `reduce_local` with long buffers on POWER9 (reformatted for presentation).