CERN-ACC-2014-0075
BNL C-A/AP/515
July 1, 2014
frank.schmidt@cern.ch

Report

Experience with OpenMP * for MADX-SC

Nicholas D'Imperio, Christoph Montag, Kwangmin Yu †, Valery Kapin ‡,
Eric McIntosh, Harry Renshall and Frank Schmidt §

Abstract

MADX-SC [1–3] allows the treatment of frozen space charge using beam-beam elements in a thin lattice, i.e. one can take advantage of the standard set-up of MAD-X [4] lattices without the need for specialized codes for the space-charge (SC) evaluation. The idea is to simulate over many turns without the problem of noise as in the PIC [1] SC codes. For the examples under study, like the PS and RHIC, it would be desirable to simulate up to 1 million turns or more. To this end one had to make an effort to optimize the scalar speed and, most importantly, get a speed-up of approximately a factor of 5 using OpenMP [5].

Keywords: Accelerator Physics, beam optics

*see page 9
†BNL
‡Fermilab
§CERN
[1]see page 9

Contents

1 Introduction

Kwangmin from BNL under the supervision of Nicholas has analyzed the tracking code in MADX-SC which is called "trrun.f90" to see which loops might be turned over to OpenMP for speed-up. A speed-up table will be provided and discussed. In parallel at CERN Frank & Harry have been investigating the scalar speed and some minor speed improvements have been achieved.

During a 2 week visit of Frank at BNL an attempt has been made to revisit all issues of speed-up of the code. It has been found quickly that special effort was needed to treat the loss of particles and keep bit-by-bit agreement with the scalar code.

Frank asked Eric for help to include into MADX-SC the speed optimized error function from SixTrack [6] which has been worked out a decade ago by the late Dr. G.A. Erskine.

Strict reproducibility, i.e. identical results for any number of cores on the same machine architecture, is essential for massive tracking studies and Frank & Harry had started this effort at CERN. At BNL Kwangmin could identify the loop of summation that has caused the lack or reproducibility of the results. Being a weak spot the OpenMP has been taken out for this loop (without major loss in speed) and it has been replaced by the Kahan summation algorithm for least loss of precision.

At the end of Frank's stay it was found that the OpenMP implementation produced a memory clash with the central part of MADX-SC which is written in C, which could be traced back to C CALLS to this central part within 2 OpenMP loops. Fortunately, these OpenMP loops could be taken out without too much loss in speed-up.

In an appendix we collect the relevant acronyms used through this paper.

2 Main Body

2.1 Why OpenMP and not MPI [2]?

Applications may be parallelized using many widely available API [3]'s, among them OpenMP, MPI, pthreads [4], and CILK [5]. Two API's, OpenMP and MPI were considered, however OpenMP was chosen for the parallelization of MADX-SC. OpenMP is typically used when an application can utilize shared memory. In contrast, MPI is typically used with a distributed memory model. There are also instances in which a hybrid implementation is useful using both MPI and OpenMP together.

The choice of API is problem dependent. MPI is efficient when the problem can utilize a large computational load locally with minimum data exchange between nodes. This is ideal, and even necessary, for problems with large computational domains that will not fit in the available memory of a single node. An example problem would be fluid dynamics using a finite difference method. The problem domain of MADX-SC is more limited and easily fits within the memory of a single node for typical problems. The main data exchange is done when the state data for each particle is updated. At that point, every particle requires the state data for every other particle. This is best accomplished using a global memory space. This, together with the modest memory requirements, make it an ideal problem for OpenMP parallelization. An MPI implementation would require a substantial rewrite of the existing code base as well as being less efficient from a performance perspective.

The primary disadvantage of OpenMP lies in the CPU [6] core count limit of a single node. At this time, most nodes have no more than 8 physical cores which, with linear speedup, allows for a single order of magnitude increase in performance. In practice, this limitation does not have significant impact when running typical examples with particle counts not exceeding ten thousand.

2.2 OpenMP - Speed-up Table

In this section, we describe the performance test of the OpenMP implementation on MADX-SC. Since the OpenMP parallelization is implemented primarily in the TRRUN() function, we measured only the

[2] see page 9
[3] see page 9
[4] see page 9
[5] see page 9
[6] see page 9

running time of TRRUN(). In this test, we used 100 turns and varied the number of particles. The hardware consisted of an Intel(R) Xeon(R) CPU E5-2670 @ 2.60GHz CPU with 128 GBytes of RAM [7]. We compare two FORTRAN compilers the proprietary IFORT [8] compiler and the GNU[9] compiler GFORTRAN [10].

Number of Cores	1	2	4	8	16
1k particles	930	559	357	287	264
2k particles	1830	1080	683	516	463
4k particles	3680	2130	1340	980	855
8k particles	7950	4530	2800	1990	1647
16k particles	16028	9093	5577	3902	3187

Table 1: Running time (sec) of subroutine TRRUN using the IFORT compiler

Number of Cores	1	2	4	8	16
1k particles	1130	662	433	339	318
2k particles	2230	1300	832	624	569
4k particles	4470	2590	1650	1220	1083
8k particles	9600	5480	3420	2460	2109
16k particles	19393	11055	6884	4917	4119

Table 2: Running time (sec) of subroutine TRRUN using the GFORTRAN compiler

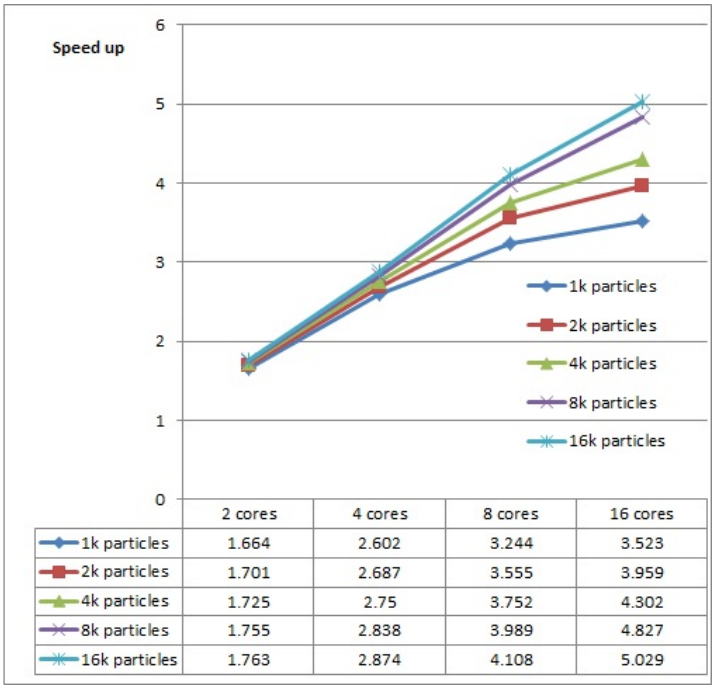
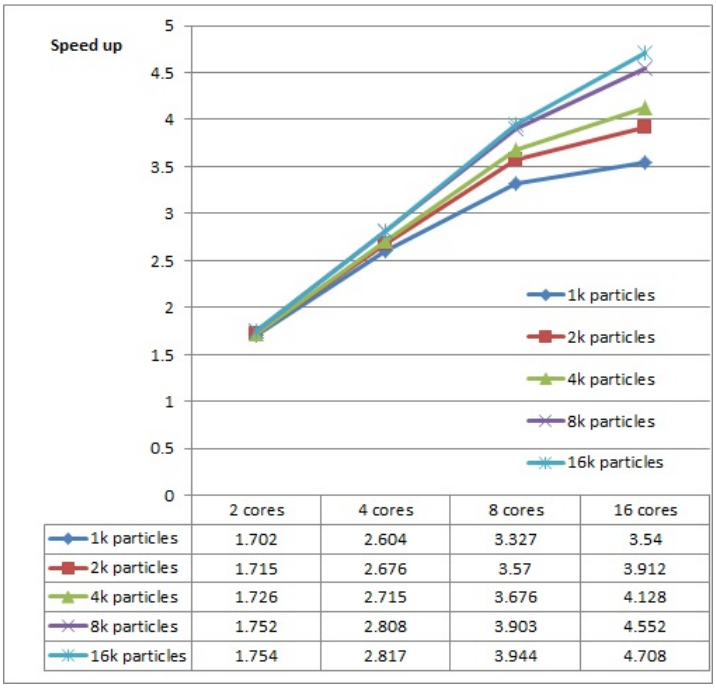


Figure 1: Speed-up comparison for the IFORT compiler case (Tab. 1)

[7] see page 9
[8] see page 9
[9] see page 9
[10] see page 10



	2 cores	4 cores	8 cores	16 cores
1k particles	1.702	2.604	3.327	3.54
2k particles	1.715	2.676	3.57	3.912
4k particles	1.726	2.715	3.676	4.128
8k particles	1.752	2.808	3.903	4.552
16k particles	1.754	2.817	3.944	4.708

Figure 2: Speed-up comparison for the GFORTRAN compiler case (Tab. 2)

According to Figure 1 and 2, both compilers show linear speed-up when we apply OpenMP parallelization and speed up improves with the number of particles. This is due to good weak scaling as the parallelization is applied to loops over the number of particles.

2.3 OpenMP - Lost Particles

During Frank's visit to BNL, Christoph found an immediate loss of all particles during his RHIC run. Kwangmin checked and tested the code using his input file and was able to determine the section of code causing the particle loss error.

The error was traced to subroutines calls, TMALI1 and TMALI2, from subroutine TRRUN. The root cause of the problem was in array arguments passed to these subroutines.

An input parameter "zz" is passed to TMALI1 and TMALI2. "zz" stores temporarily values copied from the two dimensional array "z". Algorithmically, "zz" should not be shared between threads. In FORTRAN, the array is passed by reference and when "zz" is passed to TMALI1 and TMALI2, a reference to the array is passed thus making the array itself commonly shared by all the threads. This caused variable contamination due to concurrent memory access by different threads. Similarly, the input parameter "re" also caused the same shared memory problem.

The solution was to make both arrays, "zz" and "re", private to each thread. This was accomplished by making a distinct copy of both arrays for each thread.

2.4 OpenMP - Reproducibility (Kahan summation algorithm)

The lack of bit-by-bit reproducibility found in summation of the step-function for Ex/Ey evaluation in subroutine IXY_FITTING in trrun.f90 file has been analyzed. The reason of the lack of bit-by-bit reproducibility was due to the order of summation. Kwangmin compared forward summation and backward summation and the results showed a difference of about 10^{-14}. With the Kahan summation algorithm applied by Frank (a nice explanation of this algorithm can be found at [7]), forward and backward summations shows bit-by-bit consistency. But when Kwangmin tested zigzag summation, summing numbers in odd indexes first and after that summing those in even indexes, difference of some 10^{-14} was found. In fact, this difference originates in round-off error of floating number arithmetic in computers. Since OpenMP applied on the summation loop would change the summation order according to the number of thread, it is not possible to achieve bit-by-bit consistency after applying OpenMP on

this summation loop although the reproducibility is on the level of round-off error. Since Frank insisted on a general bit-by-bit reproducibility of the MADX-SC results, OpenMP is taken out of this summation loop.

2.5 OpenMP - Interference with the MADX-SC central C part

This problem was also found by Christoph. This bug was difficult to diagnose due to its occurrence in test runs being sporadic. It occurred 4 times during 14 test runs. The bug caused a sudden crash of the code and it was determined that the crash position was in the function polish_value in mad_eval.c This error occurred when a global variable "polish_cnt" exceeds its maximum limit.

Although the exact cause of the bug was not determined due to time constraints, it did not occur when OpenMP parallelization was removed from the code responsible for the lost particle problem in section 2.3.

It is possible that this bug is related to that described in section 2.3 where "zz" and "re" arrays are passed as input parameters to the subroutines TMALI1 and TMALI2 called from TRRUN. Despite making the arrays private to each thread to avoid shared memory issues, they still seem to contaminate other areas of memory, specifically, it seems that the global variable "polish_cnt" is modified to a garbage value.

This problem is resolved by removing the OpenMP optimized code from the loops.

2.6 OpenMP - Difficulty and Risk of OpenMP parallelization for MADX-SC

Section 2.3 and 2.5 are examples of the difficulties and risks of OpenMP parallelization for MADX-SC. Common to both cases are memory conflicts. The memory conflict is aggravated by recursive calls in OpenMP parallelization and by passing parameters by reference. In particular, TMALI1 and TMALI2 have several recursive subroutines or function calls and manipulation of variables passed by reference leading to unintentional modification of variables in the calling section of code. Also, recursive subroutines or function calls increase the possibility of out of bound array errors which also can lead to crashes. Thus applying OpenMP parallelization must be done carefully on the code sections having potential memory conflicts.

2.7 Profiling Scalar Speed and In-lining

The execution of two test cases have been profiled using GPROF [11] and the scalar version of MADX-SC. The source was taken from the SVN [12] repository $//svn.cern.ch/reps/madx/trunk/madX$ and built with

```
make FC=ifort SHOW=yes PROFILE=yes
```

on the CERN machine $lxplus0087$ running the SLC6 system. The IFORT version was $14.0.2$ and the GCC version was:

```
4.4.7 20120313 (Red Hat 4.4.7-4) (GCC).
```

The two cases were $main\_deb\_mad\_6c.madx$, a CERN PS model, and $da\_tracking.madx$, a RHIC model. After a first profiling showing that 10% of the CERN PS case was spent in calling the functions DISNAN and SISNAN (to test for not a number) these two functions were forced inline by adding:

```
!DEC$ ATTRIBUTES FORCEINLINE :: disnan
```

and

[11] see page 10
[12] see page 10

```
!DEC$ ATTRIBUTES FORCEINLINE :: sisnan
```

in their code and adding the $-finline$ option to the IFORT compilation.

In each case a 500 turn run was made on a CERN lxbatch machine [13] of architecture $IntelXeon$ of 2 GHz frequency creating a $gmon.out$ file for analysis by GPROF.

The CERN PS case had the flag set: $bborbit = true$, started with 1000 particles and the turns command was:

```
RUN, turns=500, maxaper={ap_max,0,0,0,0,0}, n_part_gain=1,
sigma_z=9.59489078280046, track_harmon=2, deltap_rms=deltap_rms,
deltap_max=deltap_max; stop;
```

Flat profile showing the top ten entries: Each sample counts as 0.01 seconds.

```
 % cumulative self self total time seconds seconds calls Ks/call
Ks/call name 63.22 981.14 981.14 1150804738 0.00 0.00 ccperrf_ 8.51
1113.19 132.05 4227000 0.00 0.00 trcoll_ 5.62 1200.34 87.15 __powr8i4
4.81 1274.95 74.61 3421000 0.00 0.00 ttmult_ 3.90 1335.46 60.51
4802500 0.00 0.00 ttdrf_ 3.63 1391.84 56.38 576000 0.00 0.00
ttbb_gauss_ 2.98 1438.05 46.21 exp.A 1.75 1465.20 27.15 4164808312
0.00 0.00 inf_nan_detection_mp_disnan_ 0.68 1475.77 10.57 108413236
0.00 0.00 name_list_pos 0.63 1485.48 9.71 43284777 0.00 0.00
node_value_
```

The call graph analysis showed that CCPERF called no children and nearly all the calls to it in this example were from $ttbb\_gauss$. These calls are made in a loop over all tracks with each track being independent so was a clear candidate for parallel execution.

Call graph for CCPERRF:

```
 index % time self children called name 00547 0.02 0.00
 27648/1150804738 tmbb_gauss_ [78] 00548 981.12 0.00
 1150777090/1150804738 ttbb_gauss_ [8] 00549 [9] 63.2 981.14 0.00
 1150804738 ccperrf_ [9]
```

The RHIC case had the flag set: $bborbit = true$, started with 350 particles and the turns command was:

```
RUN, turns=500, maxaper={ap_max,0,0,0,0,0}, n_part_gain=1, sigma_z =
3.0, track_harmon=120, deltap_rms=deltap_rms, deltap_max=deltap_max;
stop;
```

Flat profile down to 1.5% of time: Each sample counts as 0.01 seconds.

```
 % cumulative self self total time seconds seconds calls Ks/call
Ks/call name 19.59 277.00 277.00 418321982 0.00 0.00 ccperrf_ 11.71
442.54 165.54 1084458390 0.00 0.00 polish_value 11.50 605.17 162.63
3175965511 0.00 0.00 name_list_pos 11.01 760.81 155.65 1001796201
0.00 0.00 node_value_ 2.81 800.51 39.70 388205734 0.00 0.00
get_node_vector_ 2.76 839.57 39.06 928398982 0.00 0.00 m66byv_ 2.65
876.99 37.42 464199481 0.00 0.00 tmali2_ 2.38 910.71 33.72 cos.N 2.09
940.25 29.54 464199501 0.00 0.00 tmali1_ 2.06 969.37 29.12 3173500
0.00 0.00 trcoll_ 2.02 997.95 28.58 2536000 0.00 0.00 ttmult_ 1.90
```

[13] see page 10

```
1024.78 26.83 1492737537 0.00 0.00 mycpy 1.65 1048.05 23.27 121267865
0.00 0.00 el_par_value 1.58 1070.40 22.35 __intel_memset 1.57 1092.54
22.14 464199481 0.00 0.00 sutran_
```

As in the CERN PS case CCPERRF is the largest single time user. The call graph shows that $name\_list\_pos$, which is a binary search to find the parameters of a character mad name in a structure, calls no children. Most of the calls to it are from the chain of $find\_variable$ called from $polish\_value$ called from $update\_vector$ called from get_node_vector called by SUELEM called by TMALI2 which is described as the transport map for orbit displacement at exit of an element. It is hard to see how this chain could be parallelized but perhaps it could be simplified for example to avoid the repeated binary search on character mad names which, given the calls count, seems to be called for each element in the ring for each turn in this case. The CERN PS case has no calls to TMALI2.

```
     index % time self children called name

              17.87 606.02 464199481/464199481 tmali2_ [6] [9] 44.1
17.87 606.02 464199481 suelem_ [9] 37.97 348.20 371359587/388205734
get_node_vector_ [10]


             128.24 0.00 2504365179/3175965511 find_variable [20]
14.07 316.34 387573627/388212900 get_node_vector_ [10] [12] 23.4
14.09 316.86 388212900 update_vector [12] 177.94 138.92
1074181301/1077934994 polish_value <cycle 2> [14]

[14] 21.6 165.54 139.41 1084458390 polish_value <cycle 2> [14] 11.38
 128.02 2500048627/2504365179 find_variable [20]

             128.24 0.00 2504365179/3175965511 find_variable [20]
 [19] 11.5 162.63 0.00 3175965511 name_list_pos [19]
```

2.8 Attempt to maximize speed of TWISS

Kwangmin has analyzed the twiss.f90 file and found several candidate loops for additional OpenMP parallelization. The candidate loops are in subroutine TMCLOR, TMTHRD, TMMULT, CCPERRF, and TMRFMULT. Analysis of running time showed the most time consuming part of TWISS is calling TMCLOR followed by TMFRST. Unfortunately, the main loop in TMCLOR contains GOTO statements and output to I/O channels rendering OpenMP parallelization impractical without extensive modification of the existing code base.

2.9 SixTrack Error Function of a Complex Number

The MAD-X module TRRUN and SixTrack use CCPERRF and ERRF respectively to compute the function, both being basically the CERNLIB routine WWERF which computes the result to almost full double precision. These routines cannot be easily vectorized or pipe-lined over a set of arguments as they use an iterative loop. The late Dr. G. A. Erskine wrote the routines WZSET and WZSUB which are much faster (around ten times faster than CCPERRF), but much less precise with a maximum absolute error of 1E-8. WZSUB uses two methods to calculate the desired value. The routine WZSET computes values at regular intervals lying inside the rectangle (0,0) (7.77,7.46) using WWERF, and WZSUB then uses third order divided-difference interpolation for arguments inside this box. For arguments outside the rectangle a two-term rational approximation is used. Note that both arguments x,y must be greater than zero (not checked) and that values of x or y greater than 1d120 are set to that value.

Eric then developed WZSUBV(n,x,y,u,v) which accepts two vectors x and y of length n and produces the corresponding two result vectors. The pipe-lining of the computation is effective only when

both x and y lie either inside or outside the rectangle. A new version has been developed which accumulates pointers to argument values inside or outside the box in two small temporary vectors which then provide a further speedup of 40 to 50% using the Intel IFORT compiler with input vectors of length 50 to 10,000.

Frank has been testing for the PS case the latest version of WZSUBV and it turns out that a sizable reduction of computing time has been achieved compared to CCPERF as seen in Fig. 3.

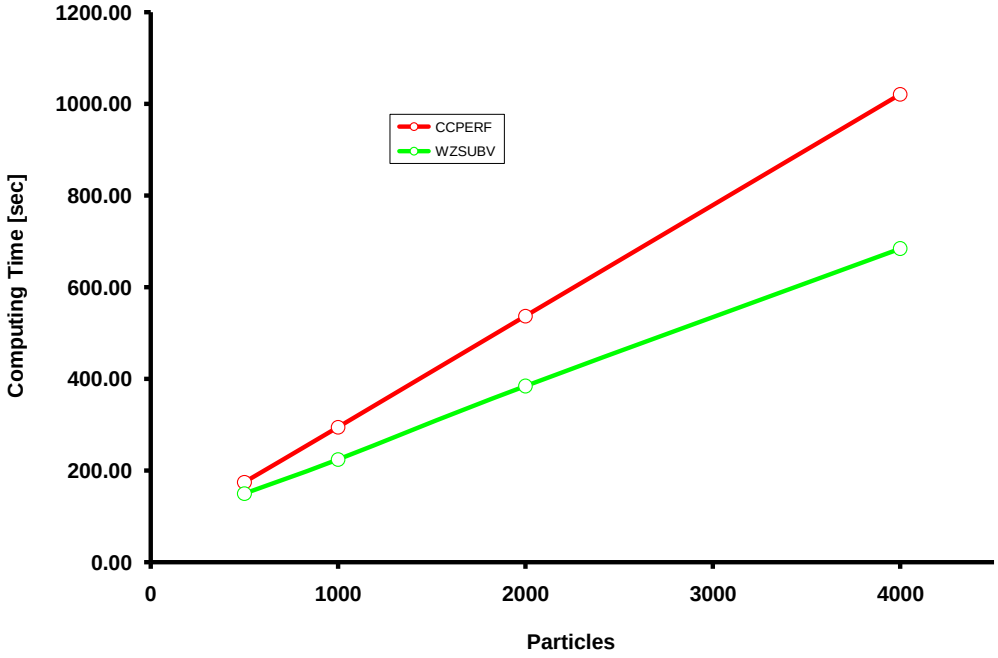


Figure 3: Comparison of computing time for 500 turns in the PS using the classical CCPERF routine of MAD-X and Eric's newest WZSUBV routine.

What counts is to evaluate the reduction of CPU time (see Fig. 4) which is expressed as the ratio of the CPU time needed for tracking with WZSUBV and CCPERF respectively. Between 500 and 2000 particles the reduction goes down from 75% to 68% pretty steeply. Thereafter, the reductions is less steep and goes to 65% for 4000 particles. In fact, this fits well with the simulation requirements to deal with a few thousand turns.

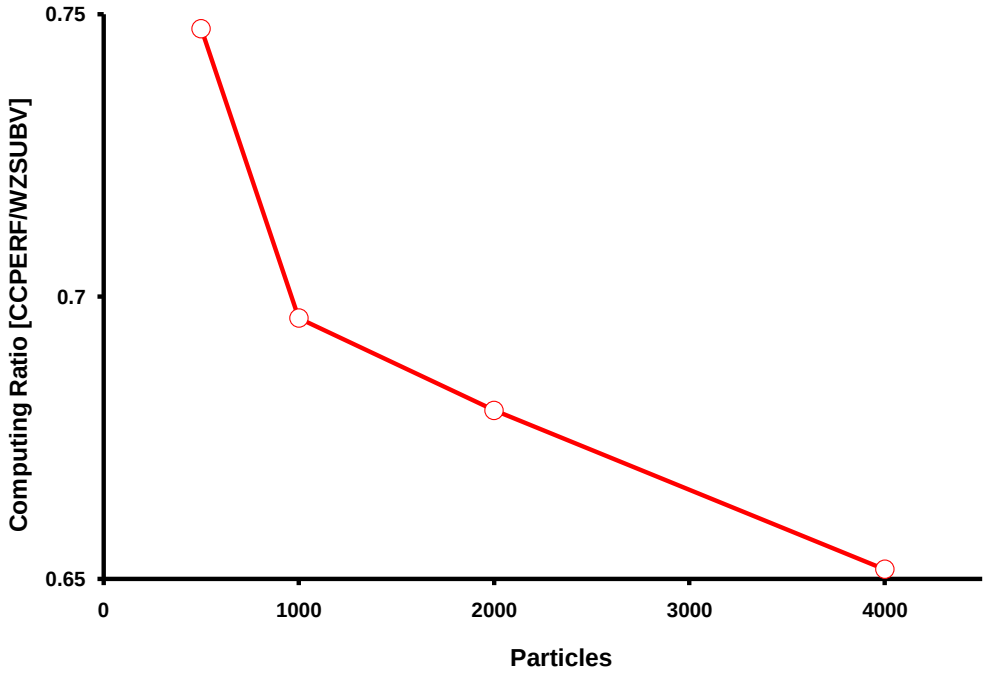


Figure 4: Reduction of CPU time expressed as the ratio [WZSUBV/CCPERF].

Bibliography

[1] V. Kapin and Yuri. Alexahin, “Space Charge Simulation Using MADX with Account of Synchrotron Oscillations”, Proc. XXII Russian Particle Accelerator Conference RuPAC-2010, Protvino, Moscow region, Oct 27, 2010, pp. 204-206.

[2] V. Kapin, “Space Charge Simulation Using MADX with Account of Longitudinal Motion”, FNAL Beamsdoc- 3582 v2, Apr. 2011.

[3] V. Kapin and F. Schmidt “Frozen space charge model in MAD-X with adaptive intensity and sigma calculation", Workshop "Space Charge 2013”, CERN, April, 2013.

[4] L. Deniau et al., “MAD - Methodical Accelerator Design”, CERN web page: http://mad.web.cern.ch/mad/.

[5] www.openmp.org

[6] R. de Maria, “SixTrack web page”, http://sixtrack.web.cern.ch/SixTrack/.

[7] http://en.wikipedia.org/wiki/Kahan_summation_algorithm.

3 Appendix: Used Acronyms

The explanations for the following acronyms have been taken from Wikipedia and other publicly available Internet resources.

OpenMP: **Open M**ulti- **P**rocessing is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems, including Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, and Windows platforms.

PIC: The **P**article-**I**n-**C**ell method refers to a technique used to solve a certain class of partial differential equations. In this method, individual particles (or fluid elements) in a Lagrangian frame are tracked in continuous phase space, whereas moments of the distribution such as densities and currents are computed simultaneously on Eulerian (stationary) mesh points. PIC methods were already in use as early as 1955, even before the first Fortran compilers were available. The method gained popularity for plasma simulation in the late 1950s and early 1960s by Buneman, Dawson, Hockney, Birdsall, Morse and others. In plasma physics applications, the method amounts to following the trajectories of charged particles in self-consistent electromagnetic (or electrostatic) fields computed on a fixed mesh.

MPI: **M**essage **P**assing **I**nterface is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran or the C programming language. There are several well-tested and efficient implementations of MPI, including some that are free or in the public domain. These fostered the development of a parallel software industry, and there encouraged development of portable and scalable large-scale parallel applications.

API: **A**pplication **P**rogramming **I**nterface specifies how some software components should interact with each other.

Pthreads: Pthreads is a POSIX standard for describing a thread model, it specifies the API and the semantics of the calls. Model popular, nowadays practically all major thread libraries on Unix systems are Pthreads-compatible.

Cilk: Cilk is a general-purpose programming language designed for multithreaded parallel computing. The C++ incarnation is called Cilk Plus.

CPU: A central processing unit (formerly also referred to as a central processor unit) is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system. The term has been in use in the computer industry at least since the early 1960s. The form, design, and implementation of CPUs have changed over the course of their history, but their fundamental operation remains much the same.

RAM: **R**andom-**A**ccess **Me**mory is a form of computer data storage. A random-access memory device allows data items to be read and written in roughly the same amount of time regardless of the order in which data items are accessed. In contrast, with other direct-access data storage media such as hard disks, CD-RWs, DVD-RWs and the older drum memory, the time required to read and write data items varies significantly depending on their physical locations on the recording medium, due to mechanical limitations such as media rotation speeds and arm movement delays.

IFORT: Intel Fortran Compiler, also known as IFORT, is a group of Fortran compilers from Intel. The compilers generate code for IA-32 and Intel 64 processors and certain non-Intel but compatible processors, such as certain AMD processors. A specific release of the compiler (11.1) remains available for development of Linux-based applications for IA-64 (Itanium 2) processors. On Windows, it is known as Intel Visual Fortran. On Linux and OS X, it is known as Intel Fortran.

GNU: GNU is a Unix-like computer operating system developed by the GNU Project. It is composed wholly of free software. It is based on the GNU Hurd kernel and is intended to be a "complete Unix-compatible software system" GNU is a recursive acronym for "GNU's Not Unix!", chosen because GNU's design is Unix-like, but differs from Unix by being free software and containing no Unix code.

GFORTRAN: gfortran is the name of the GNU Fortran compiler, which is part of the GNU Compiler Collection (GCC). gfortran has replaced the g77 compiler, which stopped development before GCC version 4.0. It includes support for the Fortran 95 language and is compatible with most language extensions supported by g77, allowing it to serve as a drop-in replacement in many cases. Parts of Fortran 2003 and Fortran 2008 have also been implemented.

Gprof: Gprof is a performance analysis tool for Unix applications. It uses a hybrid of instrumentation and sampling and was created as extended version of the older "prof" tool. Unlike prof, gprof is capable of limited call graph collecting and printing.

SVN: Apache Subversion (often abbreviated SVN, after the command name svn) is a software versioning and revision control system distributed as free software under the Apache License. Developers use Subversion to maintain current and historical versions of files such as source code, web pages, and documentation. Its goal is to be a mostly compatible successor to the widely used Concurrent Versions System (CVS).

LXBATCH: Batch Service for batch computing jobs.The CERN batch computing service currently consists of around 30,000 CPU cores running Platform LSF® providing computing power to the CERN experiments for tasks such as physics event reconstruction, data analysis and physics simulations. The public batch service is open to all CERN users. It aims to share the resources fairly and as agreed between all the CERN experiments that make use of the system. The relative share you have on the service will depend on which experiment you belong to and the activity you are doing within the experiment.