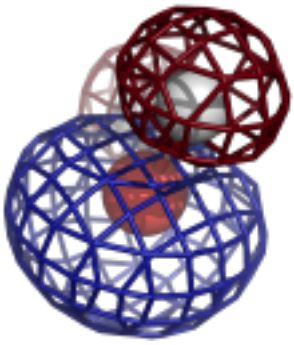
VOLUME NINE



COMPUTATIONAL CRYSTALLOGRAPHY NEWSLETTER

JANUARY MMXVIII

1.13, CRYO-EM, C BETA, GEMMI

Table of Contents

Editor

Nigel W. Moriarty, NWMoriarty@LBL.Gov

Phenix News

Announcements

Phenix 1.13 release

The Phenix developers are pleased to announce that version 1.13 of Phenix is now available (build 1.13-2998). Binary installers for Linux, Mac OSX, and Windows platforms are available at the download site.[1] Highlights for this version include new tools and feature enhancements:

- phenix.map_to_model - better symmetry support and improved runtime efficiency
- phenix.structure_search - structural library and internal support for mmCIF
- phenix.ligand_identification - limit ligand size for search
- Phaser 2.8.1 - various bug fixes
- Structure Comparison - more/improved validation information (ligands, waters, cis/trans peptides, HIS protonation)
- GUI - automatic validation after phenix.real_space_refine; visual improvements in validation
- Internal bug fixes and performance improvements

New video tutorials have been added to the Phenix Tutorial YouTube channel.[2] There include and overview of the cryo-EM tools in Phenix, a step-by-step guide on how to run MolProbity (web interface and Phenix GUI) and MolProbity: All atom contacts tutorial.

[1] http://phenix-online.org/download/

[2] www.youtube.com/c/phenixtutorials

Crystallographic meetings and workshops

The Astbury Conversation, Understanding Life in molecular detail, April 16–17, 2018

Location: University of Leeds. See website http://www.astburyconversation.leeds.ac.uk for details.

Expert advice

Fitting Tip #15 – New help to make your 2.5–4Å cryoEM structure even better

Christopher Williams, Lizbeth Videau, and Jane Richardson
Duke University

We join cryoEM structural biologists in being very excited and interested in the new, unprecedentedly higher-resolution structures now possible, and are working to develop new model-building, assessment and improvement tools suitably tuned for these new needs. We acted as assessors for the EMDB's CryoEM Model Challenge to try out the relevance and usefulness of such tools. This tip briefly summarizes progress so far, both within Phenix and MolProbity and our favorites from elsewhere.

In the 2.5 to 4Å resolution range the model-to-map fit can be quite convincing, but it often has enough leeway to prevent a unique answer. We find that refinement is therefore able to satisfy the traditional validation criteria of geometry, map fit, Ramachandran, and rotamers and still be stuck in the wrong local-minimum conformation in many places. Therefore, new criteria are needed that are not yet used in refinement and that integrate measures somewhat more broadly. We feel the two most generally useful such new tools for cryoEM models of protein are the CaBLAM backbone analysis from our lab (see below) and the EMRinger tool from the Fraser lab (Barad 2015). EMRinger looks for satisfaction of the expected chi1 angles at the Cβ atom – not to analyze rotamers, but to check for problems in the backbone that turn the Cα-Cβ in the wrong direction. You can get this analysis on your own structure at emringer.com. Tools for nucleic acid models are discussed below.

CaBLAM

The CaBLAM software from the Duke Phenix developer team (Williams 2013; 2018) uses Cα virtual dihedrals to follow the backbone intent of the relatively low-resolution model and density, and uses the virtual dihedral between adjacent peptide CO orientations to diagnose problems with the detailed conformation that has been fitted. A cryoEM example from KiNG 3D graphics

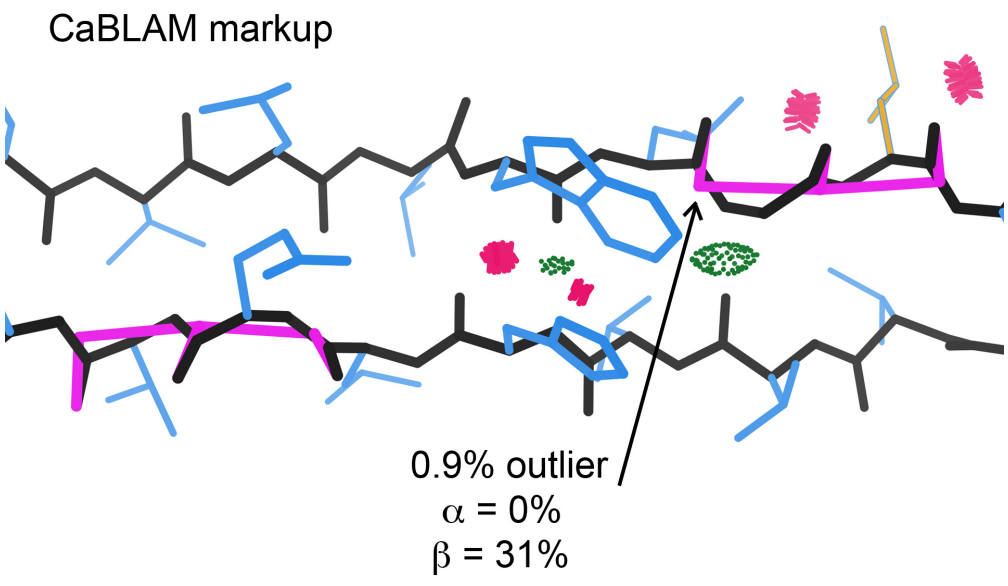


Figure 1: CaBLAM markup for incorrect peptide orientation on a CryoEM Model Challenge β-hairpin.

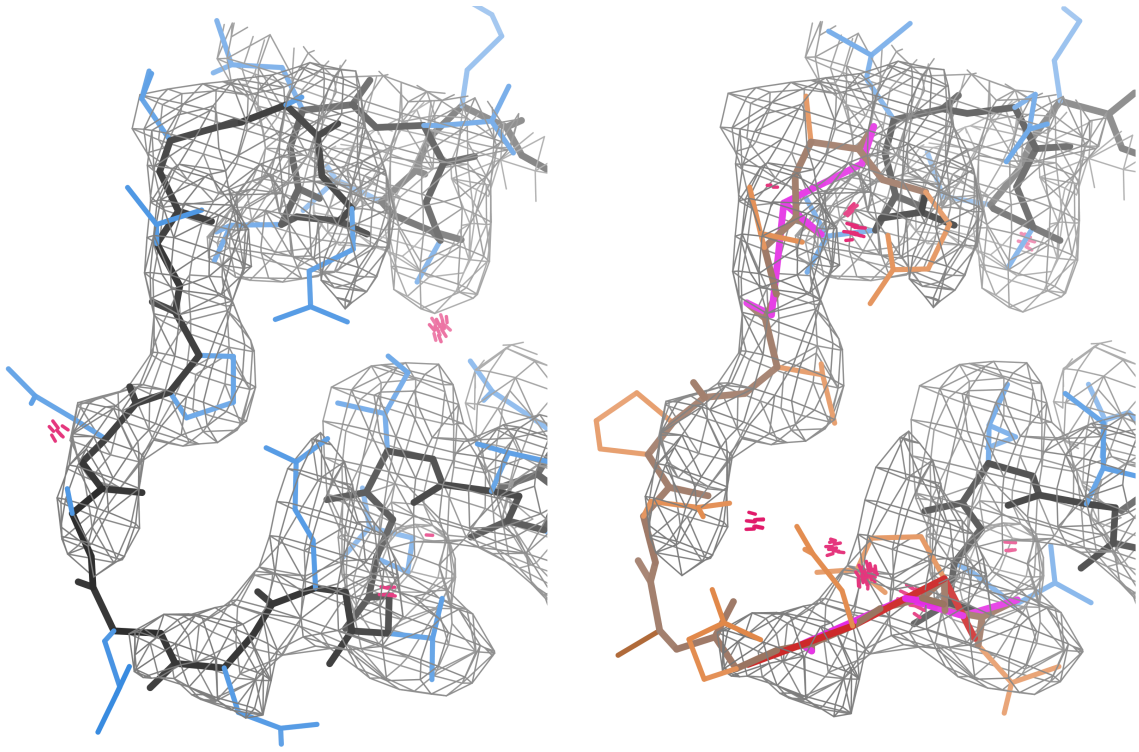


Figure 2: CaBLAM often flags one or both ends of a local sequence misalignment, even when no other validation metrics do. This figure compares the CryoEM Model Challenge target 1 cryoEM structure 4udv at 3.3Å (left) with the submitted model T0001EM188_1 (right, with misaligned region in brown & peach).

in MolProbity is shown in figure 1, where CaBLAM 1% outliers flag two places where 3 successive CO groups have been pointed in the same direction rather than alternating. This is never seen in good reference data and prevents some of the β-type H-bonds. As shown, when clicked on, the CaBLAM markup also reports on probability that the local structure is helix (0% here) or β-strand (31% here). This information can also be accessed in Phenix from the command-line.

The CaBLAM algorithm can diagnose many different sorts of problems, from wildly improbable peptide orientations within helices or beta strands, to analysis of Cα-only models, to sidechain-mainchain switches, to local sequence misalignments such as the example shown in figure 2. The central region of a local sequence misalignment often shows poor map fit, rotamers, and sterics at higher resolutions, but is not clearly diagnostic in the 2.5–4Å range. Even the ends can often be fit to avoid Ramachandran outliers, but are most reliably flagged by clashes and CaBLAM outliers. Note, though, that such outliers mean there is some fairly severe problem, but it is only sometimes a sequence misalignment.

Cis-nonPro and twisted peptides

CaBLAM also often flags *trans* residues that should have been fit as *cis* or vice versa. *Cis* prolines are fairly common at 5%, but *cis*-nonPro peptides are extremely rare at ~1 in 3000 or 0.03% (Williams 2015). Over the last decade, *cis*-nonPro were hugely overused without anyone noticing until recently; they are in the libraries and prior probabilities are not considered yet in model-building (the Phenix teams are working on that). There use is even more frequent than random, because a *cis* peptide seems to fit better into a shortened, curled-in loop. Now that MolProbity, Phenix, and Coot all flag *cis*-nonPro (Williams 2018), their overuse has decreased dramatically in crystal structures. CryoEM structures need also to watch out for this problem. Figure 3 shows that at 2.2Å the map fit can actually tell the difference of *cis* vs *trans* if you look for it. At ≥3Å the map

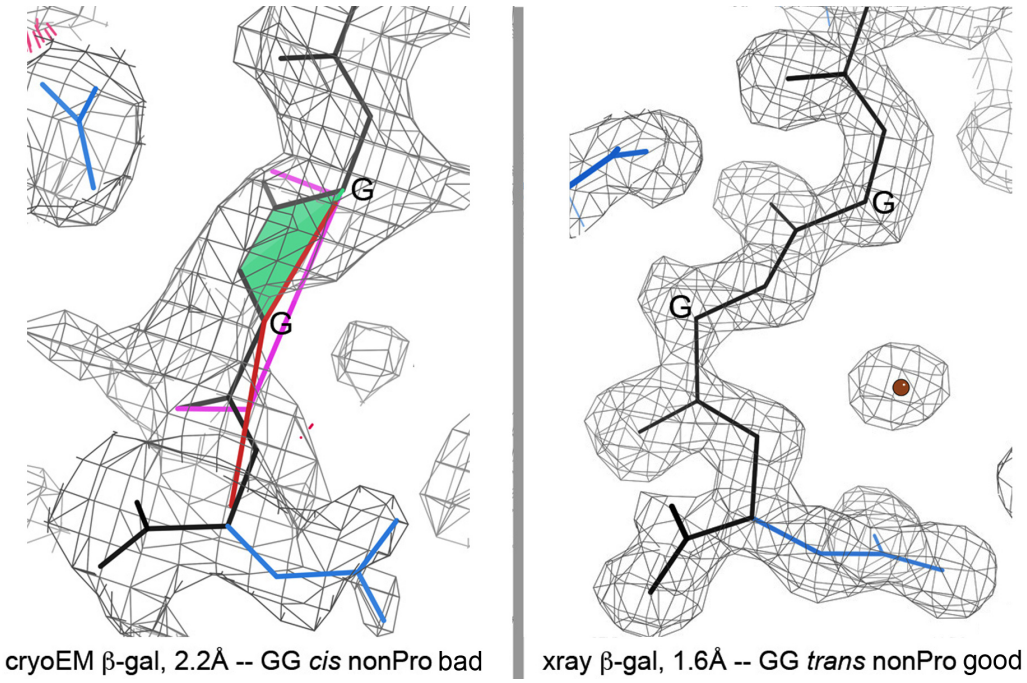


Figure 3: CaBLAM graphical flagging of an incorrect Gly-Gly *cis*-nonPro in the cryoEM 2.2A 5a1a β-galactosidase (left), compared with a clear fit as *trans* from the 1.6Å xray 4ttg (right; Wheatley 2015). At 2.2Å, the 5a1a cryoEM map, as well as CaBLAM, actually suggests that this local fit is incorrect.

cannot possibly distinguish, so you should only fit *cis* if it is known to occur for your protein at higher resolution (for instance, for 3 *cis*-nonPro at functional sites in the carbohydrate-active β-galactosidase).

RNA: ribose puckers and backbone suite conformers

The phosphates in nucleic acid structures are negatively charged and have lowered density in cryoEM maps, just as protein sidechain carboxyl groups. As shown in figure 4 below, the more positively-charged bases, in contrast, are stronger in cryoEM than in x-ray maps. Base-pairs are thus a really excellent way to locate double helices in both RNA and DNA.

RNA backbone is a tube between ribose lumps, very hard to fit in detail for either x-ray or cryoEM. Fortunately, MolProbity has developed validation that can infer ribose pucker (3'-endo or 2'-endo) with high reliability just from the well-seen and interpreted PO_4 and base positions (Richardson 2008; Jain 2014). That "P-perp" criterion enables pucker-specific targets in Phenix refinement, which can correct some of the problems, and it is easily seen by eye, to aid manual rebuilding for harder cases.

RNA backbone has distinct conformers when analyzed as sugar-to-sugar "suites", of which 54 have been identified and named by community consensus (Richardson 2008). These conformers can be used either for model-building or validation, and are analyzed in MolProbity. [Note that the wwPDB uses a criterion of "suiteness" (analogous to "rotamericity"), which is much less powerful or useful than puckers and conformers, since it is greatly affected by the percentage of A-form double helix in the structure.]

DNA conformation is simpler overall than RNA, since it is nearly always close to B-form double helix. However, it is actually more difficult than RNA to validate, because it is locally more flexible, so less restrained and therefore also more subject to error. Standard programs such as 3DNA (x3dna.org; Colisanti 2013) are helpful, but refinement may be using them by now. A new tool for DNA backbone conformers, to which no one is

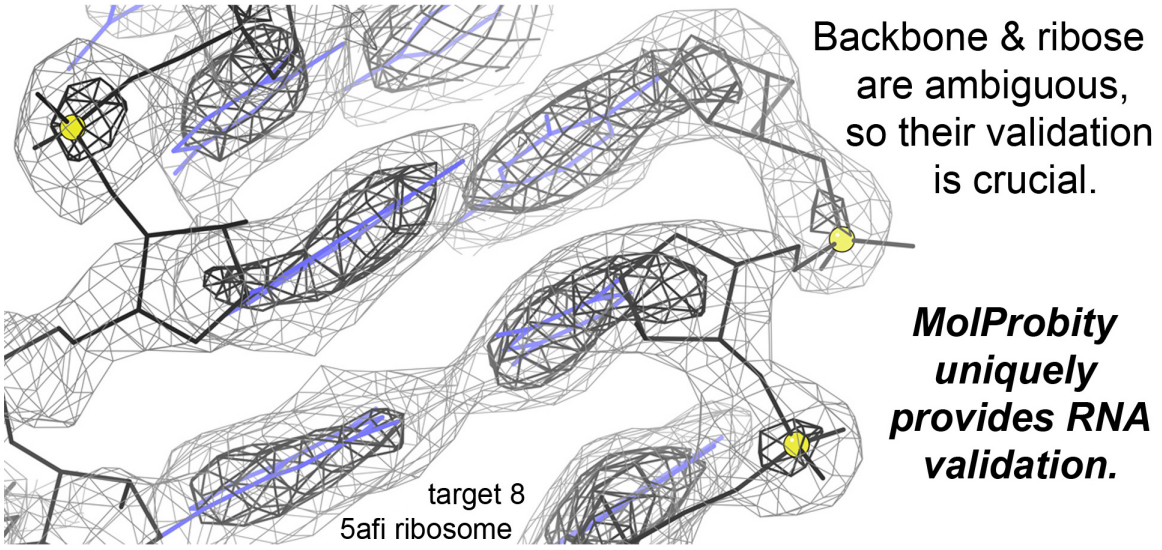


Figure 4: RNA structure seen in the cryoEM map of a ribosome at 2.9Å (5afi; Fischer 2015). Positive bases are very strong, while negative phosphates are weaker than in crystallographic density.

yet refining, is available at dnatco.org (Schneider 2018). Their data is not quality-filtered, so some of the conformers are not actually possible, but most are good and its use could help improve accuracy considerably.

The bottom line

Even if your cryoEM model has been fit and refined so as to get rid of nearly all outliers on the traditional validation metrics (geometry, Ramachandran, rotamer, and sterics), at 2.5 to 4Å resolution there are very likely to still be local regions stuck in the wrong local minimum. However, new tools are being developed which are both independent and also sensitive enough to diagnose locally misfit regions and guide their correction.

References

Barad BA, Echols N, Wang R Y-R, Cheng YC, DiMaio F, Adams PD, Fraser JS (2015) "EMRinger: Sidechain-directed model and map validation for 3D electron cryomicroscopy", *Nature Methods* **12**: 943-946

Bartesaghi A, Merk A, Bannerjee S, Matthies S, Wu X, Milne J, Subramaniam S (2015)"2.2 A resolution cryo-EM structure of beta-galactosidase in complex with a cell-permeant inhibitor" Science **348**: 1147-1151 [5a1a]

Colasanti A, Lu X-J, Olson WK (2013) "Analyzing and building nucleic acid structures with 3DNA", *JoVE*, **74**, e4401

Fischer N, Neumann P, Konevega AL, Bock LV, Ficner R, Rodnina MV, Stark H (2015) "2.9Å structure of *E coli* ribosome-EF-Tu complex by cs-corrected cryo-EM", *Nature* **520**: 567-570

Fromm SA, Bharat TAM, Jakobi AJ, Hagen WJH, Sachse C (2015) "Seeing tobacco mosaic virus through direct electron detectors" *J.Struct Biol* **189**: 87-97 [4udv]

Jain S, Kapral G, Richardson D, Richardson J (2014) "Fitting Tips #7: Getting the pucker correct in RNA structures", *Comput Crystallogr Newsletter* **5**: 4-7

Li N, Zhai Y, Zhang Y, Li W, Yang M, Lei J, Tye BK, Gao N (2015) "Structure of the eukaryotic MCM complex at 3.8Å", Nature 524: 186-191 [3ja8]

Richardson JS, Schneider B, Murray LW, Kapral GJ, Immormino RM, Headd JJ, Richardson DC, Ham D, Hershkovits E, Williams LD, Keating KS, Pyle AM, Micallef D, Westbrook J, Berman HM (2008) "RNA

Backbone: Consensus all-angle conformers and modular string nomenclature." *RNA* **14**: 465-481
Schneider B, Bozikova P, Necasova I, Cech P, Svozil D, Cerny J (2018) "A DNA structural alphabet provides new insight into DNA flexibility", *Acta Crystallographica* **D 74**:52-64
Wheatley RW, Juers DH, Lev BB, Huber RE, Noskov SY (2015) "Beta-galactosidase (E. coli) in the presence of potassium chloride" *Phys Chem Chem Phys* **17**: 10899-10909 [4ttg]
Williams CJ, Hintze BJ, Richardson JS, Richardson DC (2013) "CaBLAM: Identification and scoring of disguised secondary structure at low resolution", *Computational Crystallography Newsletter* **4**: 33-35
Williams CJ, Richardson JS (2015) "Fitting Tips #9: Avoid excess *cis* peptides at low resolution or high B", *Comp Cryst Newsletter* **6**: 2-6
Williams CJ, Hintze BJ, Headd JJ, Moriarty NW, Chen VB, Jain S, Prisant MG Lewis SM, Videau LL, Keedy DA, Deis LN, Arendall WB III, Verma V, Snoeyink JS, Adams PD, Lovell SC, Richardson JS, Richardson DC (2018) MolProbity: More and better reference data for improved all-atom structure validation, *Protein Science* **27**: 293–315

FAQ

What happens when the Phenix GUI prompts me to send an error report?

If the Phenix GUI pops up a window to send an error report, it is an opportunity to work with a Phenix developer and solve the current situation that caused the problem. There are a few things to do that can make this a useful and efficient experience for all.

1. Type in the correct email address. A Phenix developer answers all emails that contain a unique issue and user combination. Without a functioning email address, the process is cut short.
2. Consider typing in a description. This will help us diagnose the problem and may also help you with the standard questions from a developer that usually include:
   a. What OS are you using?
   b. What programs and options were you using when the problem occurred?
   c. Can you provide the inputs?
3. If you run across the same problem, you can rest assured that the problem has been registered and there is no need to resend the same issue.
4. Consider checking the website, phenix-online.org, for later versions of Phenix. Many issues are resolved daily by the developers and installing the latest Phenix may solve your problem. You can have multiple installations and choice to use any of those installed on your computer.

These tips can help with solving the problems which a high priority of the developers. Providing the inputs and a description of the events that led up to the crash is critical to reproduction of the crash by a developer and the eventual resolution.

Tools for model-building with cryo-EM maps

Tom Terwilliger

Los Alamos National Laboratory, Los Alamos NM 87545

New Mexico Consortium, 100 Entrada Dr, Los Alamos, NM 87544

Introduction

There are new tools available to you in *Phenix* for interpreting cryo-EM maps. You can automatically sharpen (or blur) a map with `phenix.auto_sharpen` and you can segment a map with `phenix.segment_and_split_map`. If you have overlapping partial models for a map, you can merge them with `phenix.combine_models`. If you have a protein-RNA complex and protein chains have been accidentally built in the RNA region, you can try to remove them with `phenix.remove_poor_fragments`. You can put these together and automatically sharpen, segment and build a map with `phenix.map_to_model`.

Sharpening a map with phenix.auto_sharpen

The `phenix.auto_sharpen` tool is designed to find the optimal sharpening for a cryo-EM (or crystallographic) map based on maximizing the detail in the map as well as the connectivity. Sharpening of the map is carried out by applying an overall sharpening B-factor to the amplitudes of the Fourier transform of the map up to the nominal resolution of the map. Beyond that resolution the amplitudes are damped. In this method, the key parameter is the value of the sharpening B-factor.

The detail in the map is represented by the surface area of contours at a fixed contour level. The (lack of) connectivity is represented by the number of regions in the map at that contour level; the contour level is chosen to enclose about 20% of the volume of the molecule. The target for optimization is the adjusted surface area calculated as the surface area minus a scale factor times the number of regions in the map. The scale factor is set automatically by requiring the adjusted surface area to be equal at the lowest and highest values of the sharpening B-value tested. The sharpening B-value is then adjusted to maximize the adjusted surface area. The auto-sharpen procedure creates a new map that is in the same position and has the same grid as the original map.

You can apply the auto-sharpening procedure globally (the same sharpening B-factor is applied to the entire map) or locally (a different sharpening B-factor is applied to each part of the map). In practice, typically the global sharpening is just as good as local sharpening. The local sharpening procedure uses the segmentation procedure in `phenix.segment_and_split_map` to find the regions in the map where the molecule is located. For each such region, a box of density is cut out surrounding the region and a local sharpening B-value is identified and applied to the grid points in the box. The density for points that are in more than one box is their average, weighted based on the distance of each point from the center of corresponding boxes. Points outside any of these boxes are sharpened with the global sharpening B-value.

You can also refine not just the overall sharpening B-factor, but also the high-resolution cutoff where the sharpening turns into blurring and the sharpness of the transition from sharpening

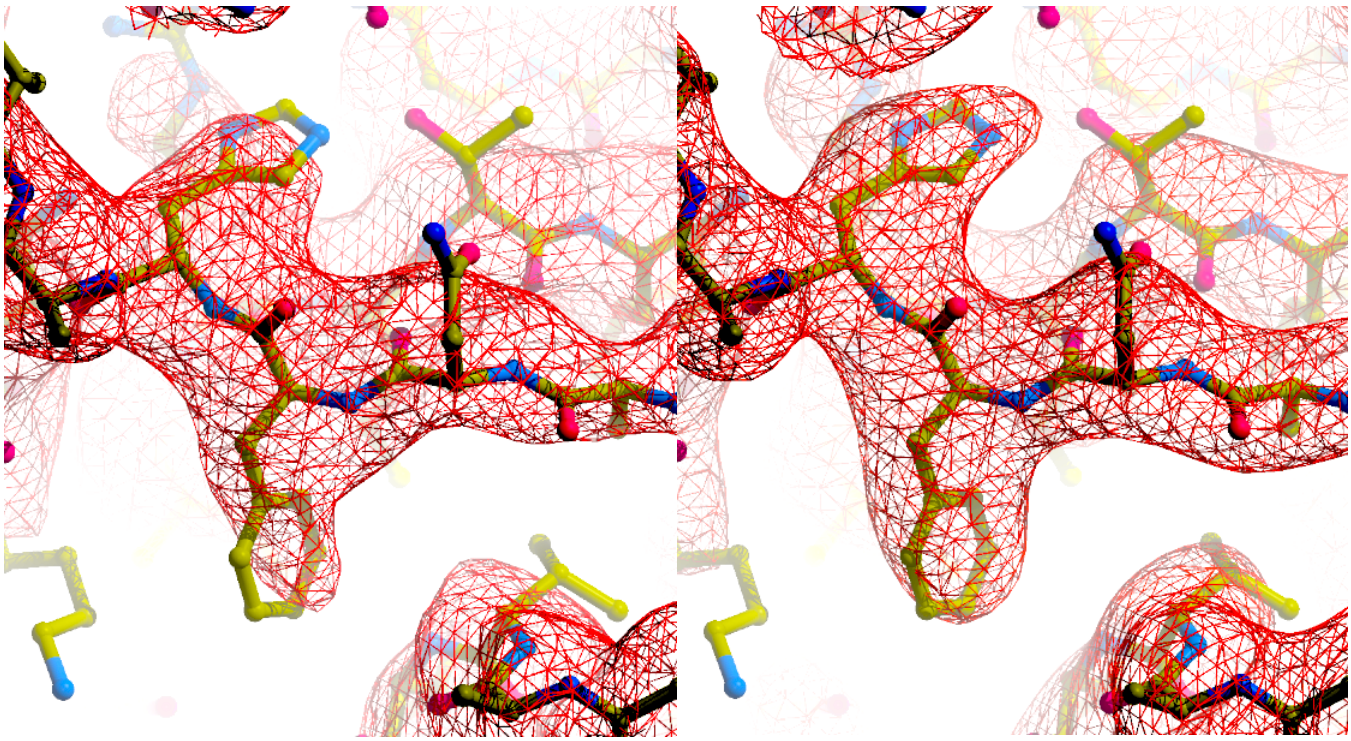


Figure 1: A small section through a map for EMDB entry 6344 along with the corresponding PDB entry (3jad) (left), and the results of automatic map sharpening (right).

to blurring. In our test cases, this procedure had little effect on the outcome so it is not the default.

Using the auto-sharpen procedure is easy. You just supply the map and the resolution. Figure 1 shows a small section through a map for EMDB entry 6344 along with the corresponding PDB entry (3jad) (left), and the results of automatic map sharpening (right). For local sharpening, you can optionally supply as well solvent content or a sequence file or the molecular mass of the molecule so that the regions of the map corresponding to the macromolecule can be identified. The solvent content will be guessed if not supplied.

You can see the methods in `phenix.auto_sharpen` in detail at (Terwilliger *et al.*, 2018).

Automatic segmentation of a map with phenix.segment_and_split_map

The next step in interpreting a cryo-EM map after sharpening is to segment the map. The purpose of segmentation is to break the map up into smaller pieces that are suitable for model-building. If the map has symmetry, segmentation can identify the basic unit of the map so that the same part of the map does not have to be interpreted many times. If the map has regions that are different chain types, segmentation can separate them and work with them individually.

There are three steps in segmenting a map with `phenix.segment_and_split_map`. The first is to contour the map and find all the regions defined by these contours. The next is to find the symmetry in the map. The third is to find a unique and compact set of regions that represents

the basic unit of the map. Applying symmetry to this basic unit of the map then can regenerate the entire map.

The `phenix.segment_and_split_map` tool contours the map using the same approach as was used above in auto-sharpening. The contour level is chosen to enclose about 20% of the volume representing the macromolecule but is adjusted further to enclose regions that typically correspond to about 50 residues in length. As symmetry-related parts of the map have the same densities, this procedure yields a set of regions that can be grouped in the next step based on the symmetry in the map.

The `phenix.segment_and_split_map` tool can identify symmetry in a map by comparing density at symmetry-related points. You can specify for example that the map has icosahedral symmetry, and the tool will test various common settings of this symmetry to see if they are present. Alternatively the tool can look through all common symmetries and settings to find the one with the highest symmetry that fits the map. You can also supply symmetry with a *Phenix* `ncs_spec` file or in BIOMTR records.

Using the symmetry in the map, the `phenix.segment_and_split_map` tool identifies the basic unit of the map. This is done by grouping the contoured regions in the map into symmetry-related sets and choosing a set of regions that forms a compact group. You can optionally surround this basic unit of the map with neighboring groups so that you have a better chance of enclosing a complete molecule in the basic unit of the map.

The tool writes out several map files. One is just the input map, shifted so that the (0,0,0) grid point of the map is at the coordinates (0. ,0. ,0.). Another is a map representing the basic unit of the original map, typically written to the file `box_map_au.ccp4`. This map is normally much smaller than the original map, particularly if there is a high degree of symmetry in the map. This map is shifted (again) so that its origin is again at (0,0,0). You can use this map for model-building if you want. You can even automatically shift the resulting model back to the position of the original map. Another set of maps are region maps, one corresponding to each unique region identified by `phenix.segment_and_split_map`. These maps all superimpose on the shifted input map, so their origins are not normally at (0,0,0).

Combining partial models with phenix.combine_models

The `phenix.combine_models` tool can combine the best parts of two or more models. You can use it in either of two different ways. One approach keeps segments in the models intact, and the second applies crossovers within segments.

In the first approach (`merge_by_segment_correlation=True`), each fragment in each model is scored based on correlation to the map. The scoring also includes a weight based on the square root of the length of the segment. It also includes weights based on whether a segment has been assigned to the sequence and the number of secondary structure hydrogen bonds in that segment. This approach can combine chains of different types as well. A final increment is added to the score for a segment if it is considered very likely to be correct with different thresholds for chains of different types.

Then the segments are picked in rank order to create a composite model. Any part of a segment that does not overlap an existing part of the composite model is kept. If symmetry is present, overlaps include that symmetry.

In the second approach, (`merge_by_segment_correlation=False`), a working model is created by taking all of the first model supplied and filling in any empty regions with fragments from other models. Then one by one, the segments in the working model are recombined with all other segments. To carry out recombination between two chains, residues that match in the two chains are identified. Then for each segment between matching pairs of residues, whichever chain has the higher correlation to the map is kept to create a composite model.

Removing poor parts of a protein-RNA model with phenix.remove_poor_fragments

When carrying out model-building for complexes between protein and RNA it sometimes happens that protein can be built accidentally into regions that are really RNA. The `phenix.remove_poor_fragments` tool is used to try and identify such incorrectly-built regions. The key idea in this approach is that these incorrectly-built regions tend to be isolated (not in the middle of a protein domain) and to have lower map-model correlation than the correct parts of the protein chains. The `phenix.remove_poor_fragments` tool ranks all protein segments on these criteria and removes the worst ones. The threshold that is chosen based on the number of residues of RNA expected in the molecule that were not built, the fraction of protein that was built, and the quality of the model, using the formula:

$$\texttt{residues\_to\_remove} = A * \texttt{protein\_built} * \texttt{rna\_not\_built}/(\texttt{cc}*\texttt{protein\_present}) \quad (1)$$

where `cc` is the map-model correlation for the protein part of the model, `protein_present` is the number of protein residues in the sequence file, `protein_built` are residues of protein built, and `rna_not_built` are residues of RNA in the sequence file minus the number built.

The logic of this formula is that there is a certain volume, proportional to `rna_not_built`, that is really RNA that might be accidentally built as protein. Furthermore the logic is that the number of residues of protein that might be built in that volume is higher if more residues of protein have been built and higher if the model-map correlation for the protein model is low. These relationships were seen in an analysis of ribosome structures. A scale factor A of 0.53 relating the optimal number of residues to remove to the other factors was found empirically.

Build a model with phenix.map_to_model

You can carry out fully automatic map interpretation and model-building with `phenix.map_to_model`. This tool first sharpens the map and carries out segmentation of the map to produce a map with the unique part of the original map and maps for each unique region in the map. The tool then carries out model-building both in the entire unique part of the map and in each individual region. Then `phenix.map_to_model` combines the best parts of all these models, applies symmetry, and refines the full model (see figure 2 for an example)

The `phenix.map_to_model` tool uses three different methods for model-building. One is standard resolve model-building, as in `phenix.autobuild`. This is applied to the entire unique

Figure 2: Example of a combined model.

part of the map. A second is finding regular secondary structure with `phenix.find_helices_strands`, also carried out on the entire unique portion of the map. The third is chain-tracing followed by refinement using secondary-structure restraints. This third method uses the trace_chain algorithm in `phenix.find_helices_strands`, which traces tubes of density by finding positions in the density that are separated by about 3.8 A. These C-alpha positions are converted to a main-chain model with Pulchra(Rotkiewicz & Skolnick, 2008). Then it uses the phenix tool `phenix.iterative_ss_refine`, which identifies secondary structure in a poor model, generates restraints based on an idealized version of that secondary structure, and refines with `phenix.real_space_refine` including those restraints. Each of these three approaches generates one preliminary model which is refined with `phenix.real_space_refine`. (These models are typically called `model_standard_PROTEIN.pdb`, `model_init_PROTEIN.pdb`, and model `_helices_strands_only_PROTEIN.pdb` respectively).

If there is RNA or DNA present in the structure (as guessed from the sequence file or specified by the user), RNA or DNA model-building is carried out in much the same way as was done for protein, except that there is no equivalent of the chain tracing algorithm used.

The preliminary models are then combined with the `phenix.combine_models` tool using the `merge_by_segment_correlation=True` approach. If there is symmetry present, the symmetry is included when combining models to ensure that there is no overlap. The resulting model is refined with `phenix.real_space_refine` to yield `model_PROTEIN.pdb`. If there is RNA or DNA present, those models will be in `model_RNA.pdb` or `model_DNA.pdb`.

To combine models with RNA, protein and DNA, the tool `phenix.combine_models` is used once again with the `merge_by_segment_correlation=True` approach. Each segment (of any chain type) is scored by length and map correlation and by presence of hydrogen bonds, and the best-scoring segment is picked for each part of the map.

In cases where RNA is a substantial part of the structure (at least 1/6 of the total), a final step is carried out in which poorly-fitting parts of the protein model are removed. This step is helpful for structures such as ribosomes where much of the structure is RNA, but where protein can sometimes be built accidentally into regions that are RNA but were not built as RNA. The tool `phenix.remove_poor_fragments` guesses how much of the protein part of a model might be built incorrectly and removes the appropriate amount of the worst-fitting and isolated parts of the protein part of the model.

Finally, any symmetry is applied to the model, parts that overlap by symmetry are removed, and the final model is refined with `phenix.real_space_refine`. The entire model is shifted to the original position of the original map and written out as `map_to_model.pdb` and the unique part of the model is also written out if symmetry is present (`map_to_model_unique.pdb`).

References:

Rotkiewicz, P. & Skolnick, J. (2008). *J. Comput. Chem.* **29**, 1460–1465.

Terwilliger, T. C., Sobolev, O., Afonine, P. V. & Adams, P. D. (2018). *BioRxiv*. 247049. https://www.biorxiv.org/content/early/2018/01/11/247049

Gemmi – new MX library

Marcin Wojdyr
Global Phasing Ltd, Cambridge, UK
Correspondence email: wojdyr@gmail.com

Introduction

At the beginning of 2017 CCP4 and Global Phasing Ltd. started a new software project – an open-source library called Gemmi[1] – funding one person (me) to work on it full-time for three years. The library is written in C++, with bindings for Python and Fortran. The project, agreed upon a few years ago, was largely motivated by the need to improve mmCIF handling in Refmac (Murshudov *et al*, 2011) and BUSTER (Bricogne *et al*, 2017) – refinement programs from the involved organizations. But the scope of the library is not limited to this. The scope, still flexible and driven by the needs of involved parties, can be split into handling of:

- macromolecular models (coordinate and restraint files),
- data on a 3D grid (CCP4 file format for maps and masks),
- and reflections (MTZ and SF mmCIF formats).

To implement the above points the library also needs to handle:

- the CIF syntax,
- and crystallographic symmetry.

All the listed areas include groundwork, such as reading and writing file formats, and higher-level functionality added on request from application developers. As an example of the latter, recently the 3D grid gained functions needed to determine the bulk solvent area.

The library is far from being finished and here I write only about a couple things that I have learnt during the first year of development.

[1] https://project-gemmi.github.io/

mmCIF and mmJSON

mmCIF format, the primary format used by the wwPDB, is not well suited for web applications. At least that is what RCSB, PDBe and PDBj think, as each of them developed own file format (MMTF, Binary CIF and mmJSON, respectively) for use in web-based molecular viewers. MMTF is a binary format with a complex compression scheme that reduces the data transferred from the server by a factor of four. Binary CIF is similar to MMTF, but preserves all the metadata from mmCIF at the expense of slightly larger size.

mmJSON is quite different and I would argue it is the most practical coordinate format for most of applications. It is a simple translation of the mmCIF content into JSON. Tables as stored as columns (`"tagA": ["value1", "value2"], "tagB": [1, 2]`). As a side-effect of writing data columns-wise, the transported (gzip-compressed) data is reduced by about 40%. More importantly, the JSON format can be easily read and written in any programming language. Of course CIF precedes JSON by a decade, and it was historically a good choice. But today we can also see the cost (in man-years across multiple projects) of a format with a niche syntax. So even now, 25 years after the CIF syntax was invented, adding mmJSON to the formats in the wwPDB archive (currently: PDB, mmCIF and PDBML) would make many future projects easier.

That being said, I wrote yet another parser of the CIF 1.1 syntax, basing on a library called `PEGTL`. To my best knowledge, so far the `cif_api` library had the fastest CIF parser (Bollinger, 2016), reportedly several times faster than `iotbx`. The Gemmi parser is 3x faster than `cif_api` (at least in the hands of the author). At the same time the fastest JSON parsers, hand-coded and carefully optimized, can handle hundreds of MB/s, several

times more than my CIF parser. So there is still plenty of room for optimization.

One way to utilize the speed of a CIF parser is to do a PDB-wide analysis on mmCIF files. For this I made a small utility (`gemmi-grep`) that prints values of selected tags in CIF files, reading input at the rate of about 100MB/s. It can go through a local copy of the wwPDB archive in half an hour, using a single processor core and uncompressing files on the fly. The tool was used to gather various statistics from the PDB, such as MX software statistics[2].

Space group settings

Mapping space group notation to a set of operations has been coded many times before, but I still pondered:

1) What data should be tabulated and what should be recalculated?
2) What space group settings should be included?

Question 1 is approached differently by different programs. Each space group settings must come with either the full list of operations, or with a minimal set of generators for the group, or with something between (for example centering vectors can be listed separately to save space). Then the operations can be encoded in various ways. After considering all the possibilities I went with the approach from SgInfo and sgtbx (and clipper in CCP4) – generating operations from the overly implicit Hall notation.

Regarding question 2, `SgInfo/sgtbx` (Grosse-Kunstleve *et al*, 2002) and International Tables for Crystallography (ITfC) vol. B[3] have a list of 530 settings (527 distinct settings; three settings in the group 68 are given with two different names). This list is used by many programs, notably by `Spglib`, which seems to be more popular than sgtbx among physicists.

CCP4 uses a file called `syminfo.lib` that is based on an older file called `symop.lib` and the data from `sgtbx`. It includes the 530 settings and a few more. Similarly, the OpenBabel library also has a file listing symmetry settings and operations, with 530 + 14 settings. The extra 14 came from the COD database. Some of them are described in "*A Hypertext Book of Crystallographic Space Group Diagrams and Tables*"[4] (for example C-centered tetragonal space groups) and mentioned in the latest edition of ITfC vol. A[5] (which, unlike the vol. B, uses new naming, e.g. "C -4 2 g1"). All the differences can be confusing, but have little practical importance in MX software. So for now Gemmi just includes the settings from `syminfo.lib`.

Adding new space group settings requires coming up with a Hall symbol. ITfC vol. B writes about an unambiguous method to select the Hall symbol. The method involves sorting operations into "a strictly prescribed order based on the shape of their Seitz matrices", but the details seem to be gone from the Internet. In this situation I manually picked Hall symbols for settings absent in ITfC vol. B.

Technicalities

Gemmi is written in C++ (no Python, although most of the functions will have Python bindings). For the sake of portability it uses C++11, ignoring new features of C++14 and 17.

Many C++ libraries are header-only, which makes them trivial to install. So far Gemmi is also a header-only library, but it may change as the library grows. Still, some parts (CIF parser, symmetry library) will stay header-only. This makes integration much easier. If a program only

[2] https://project-gemmi.github.io/pdb-stats/

[3](International Tables) Volume B home page. *urn:isbn:978-1-4020-8205-4* doi:10.1107/9780955360206000108

[4] http://img.chem.ucl.ac.uk/sgp/mainmenu.htm

[5](International Tables) Volume A home page. *urn:isbn:978-0-470-97423-0* doi:10.1107/9780955360206000114

needs to know space group operations, one can just copy a single file (gemmi/symmetry.hpp) to their project, and that is all.

Python bindings are generated with pybind11, which is inspired by Boost.Python but is smaller, faster to compile and easier to use. Python bindings work with Python 2.7 and 3.x, with CPython and PyPy. The latest development version can be downloaded, compiled and installed with a single pip command:

```
pip install git+https://github.com/project-gemmi/gemmi.git
```

We also need bindings for Fortran 2003; the work on it has just started.

Finally, as we all observe more and more software moving to web browsers, it would be nice to port Gemmi, or parts of it, to either JavaScript or WebAssembly. This would benefit some of the existing JavaScript programs, in particular UglyMol. The relatively small size of the C++ code base should help, but the details are yet to be investigated.

Acknowledgements

The project is funded by Global Phasing Ltd. and CCP4, overseen by Eugene Krissinel, Garib Murshudov and Gerard Bricogne, and was helpfully discussed with many other members of CCP4 and GPhL (in particular with Andrey Lebedev, Claus Flensburg, Clemens Vonrhein).

References

Murshudov GN, Skubák P, Lebedev AA, Pannu NS, Steiner RA, Nicholls RA, Winn MD, Long F, Vagin AA (2011). *Acta Cryst. D*: **67**, 355

Bricogne G, Blanc E, Brandl M, Flensburg C, Keller P, Paciorek W, Roversi P, Sharff A, Smart OS, Vonrhein C, Womack TO (2017). Cambridge, United Kingdom: Global Phasing Ltd.

Bollinger JC (2016), *J. Appl. Cryst.* (2016). **49**, 28

Grosse-Kunstleve RW, Sauter NK, Moriarty NW, Adams PD (2010). *J. Appl. Cryst.* **35**, 126

Overfitting to Ramachandran and geometry criteria in the cryoEM Model Challenge

Christopher J Williams, David C. Richardson, and Jane S Richardson
Duke University

We are interested in the extent and effects of overfitting to validation criteria, and the EM Model Challenge[1] provided a useful dataset to analyze overfitting. The dataset was suitable because, for each modeler group, it contains multiple depositions of very different structures using very similar methods. This allows us to construct a better sense of each group's overall behavior. However, many modelers are willing – quite reasonably – to skip over loops and other difficult regions. This prevents us from constructing a truly complete picture of how their modeling tools behave.

Overfitting to Ramchandran

To assess overfitting to Ramachandran criteria, for each modeler group, we accumulated all the Ramachandran points from their first model for each target into a single distribution. The resulting Ramachandran distributions are strikingly varied. Appropriately to a modeling challenge, the modelers seem to have been very aggressive in using diverse methods. However, almost all of these methods have resulted in some degree of overfitting to Ramachandran criteria.

Figure 1 shows typical patterns of Ramachandran overfitting within the cryoEM Challenge dataset. Severals groups restrained their Ramachandran distribution to a different – and much stricter – set of contours than our Top8000-derived set (Richardson 2013). Group EM119 (1a) shows this alternate distribution most strongly. Group EM120 (1b) also shows restraint to contours

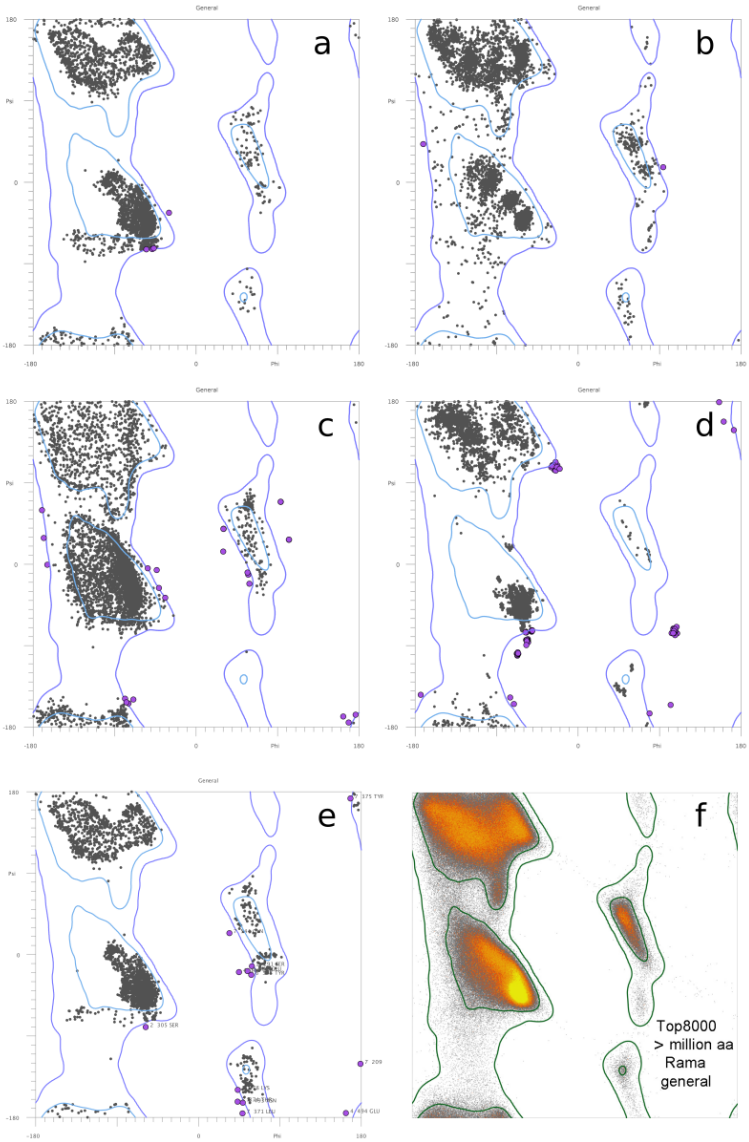


Figure 1: Ramachandran diagrams showing unusual distributions. Ramachandran distributions of all general-case residues from cryoEM Challenge groups EM119 (a), EM120 (b), EM130 (c), and EM189 (d) show various types and degrees of overfitting to Ramachandran criteria. The Ramachandran distribution from a deposited PDB structure, 3ja8, shows a similar pattern of overfitting (e). Our standard Top8000 Ramachandran distribution, with heat-mapped population density, is provided for comparison (f).

[1](http://challenges.emdatabank.org/?q=model_challenge)

within the *Favored* bounds, with the addition of alternating patches of highly-populated and empty regions. Group EM130 (1c) shows an edge-tracing effect, where the edge of the *Favored* region is unusually highly populated, possibly due to Ramachandran restraints moving *Allowed* residues until they cross just into the *Favored* region, or possibly due to an energy that particularly favors certain atomic contact distances. Group EM189 (1d) shows an ultra-restrained alpha helix region, which might be an extreme version of the patchy regions in 1b. Our standard Ramachandran distribution, with heat-mapped population density, is provided for comparison (1f).

We had expected the edge-tracing phenomenon in 1c to be relatively common, but in fact only a few groups showed significant tracing of our *Favored* contours. It should be noted, however, that EM119 (1a) does show an edge tracing effect at the bounds of their chosen contours. We also observed that many distributions (e.g. 1b and 1c) showed a fairly hard ϕ cutoff at about -60 degrees. This leaves the rightmost areas of both the alpha and the beta regions unnaturally empty.

These peculiar distributions are by no means limited to the sometimes-speculative software of the cryoEM Challenge, however. Figure 1e shows the Ramachandran distribution of 3ja8, a PDB-deposited EM structure from 2015. This distribution shares many characteristics with 1a, including tightened contours within the *Favored* region, an additional horizontal band below alpha, and a strong edge-tracing effect in beta. The overfitting that occurred during the cryoEM Challenge is also seen in final deposited cryoEM structures.

Overfitting of covalent geometry

We also assessed overfitting to covalent geometry criteria. For each group, we performed covalent geometry validation with mmtbx.mp_geo. For each mainchain bond or angle (those involving N, CA, C, O, and CB), we determined the percentage of the total population in each 1-standard-deviation bin from the ideal value, making no distinction between positive and negative deviations from ideal. If the covalent geometry deviations followed a normal distribution, the populations of the first three bins would be roughly 68, 27 and 4%. A deviation of 4σ or more is considered an outlier.

Figure 2 shows some representative distributions for covalent bond lengths on the left and angles on the right. Bond lengths were typically very highly restrained to within 1 sigma of ideal, as in EM119 and EM130, though some groups like EM120 clearly followed some other pattern. Bond angles were generally less tightly restrained and slightly closer to a normal distribution, see again EM119 and EM130, but real approximations of a normal distribution were very rare.

Effects of overfitting

Validation tools that modelers were not optimizing against, such as CaBLAM (Williams, 2018), give us suggestive evidence of structural harm caused by overfitting. Figure 3 shows one such example. Target T0002 (PDB 3j9i, a proteasome) contains a cluster of covalent geometry outliers (3a) around Ala107 of chain 1. In EM130's model, these geometry outliers have been resolved, but at the cost of introducing a Cα geometry outlier identified by CaBLAM (3b). The Cα geometry outlier is not very far from an allowed conformation, so permitting slightly more flexibility in the surrounding covalent geometry might prevent this distortion.

A more dramatic example of structural harm from overfitting is the placement or refinement of Ramachandran points into the wrong favored regions, which we have observed in both the cryoEM Challenge models and in the PDB. Figure 4 shows an alpha helix from PDB 3ja8 with a clear and severe modeling error at Leu234 (a). This modeling error placed the Ramachandran point for Leu234 in or near the beta region (b), and refinement moved it into the cluster at the edge of

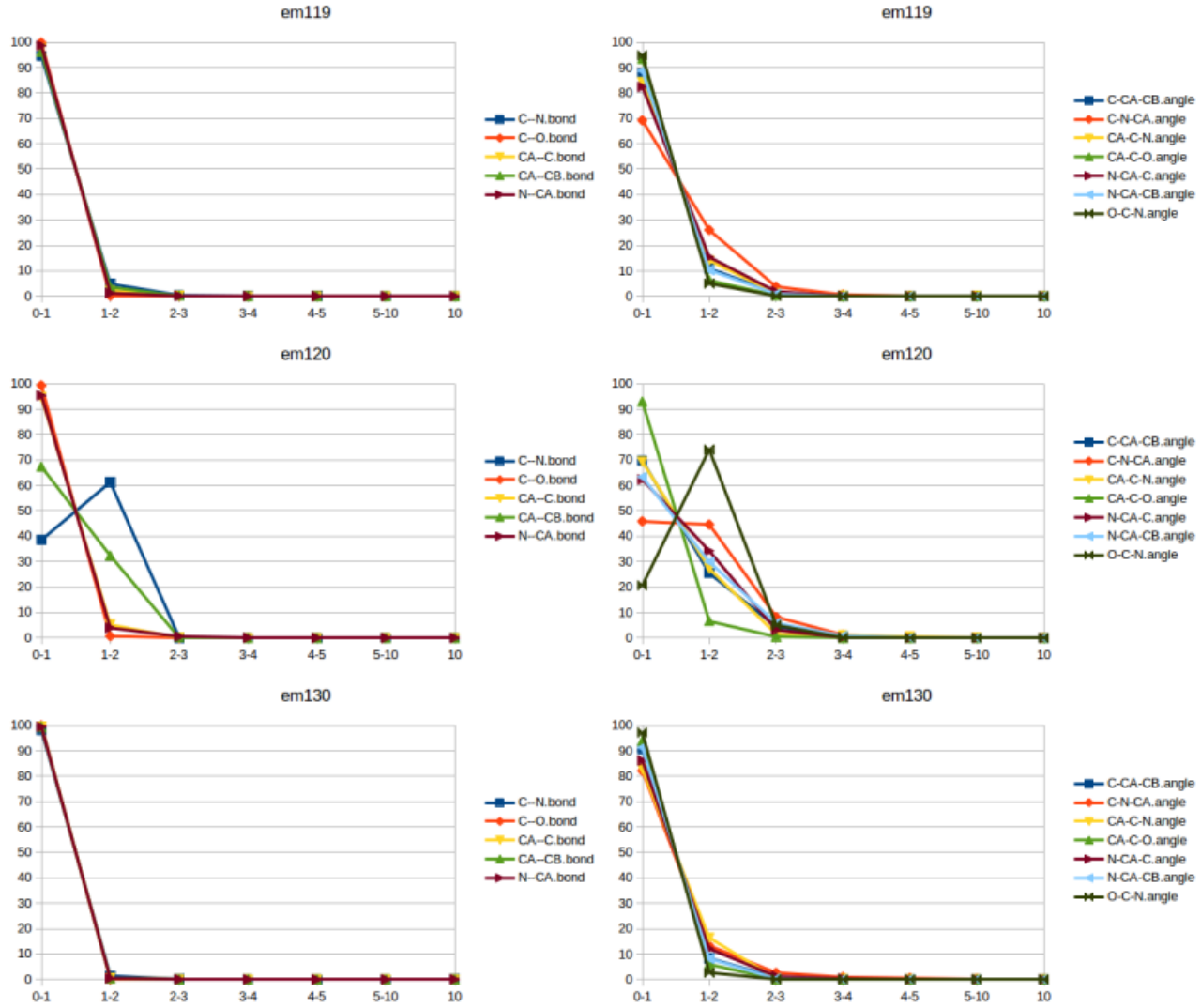


Figure 2: Population distributions for covalent bond geometry of cryoEM Challenge models. X-axis is number of standard deviations from ideal (4 is the outlier cutoff), y-axis is percent of population. Groups EM119 and EM130 show highly restrained distributions typical of the challenge models. Group EM120 shows an unusual distribution suggestive of either modeling errors or a mismatch between geometry assumptions.

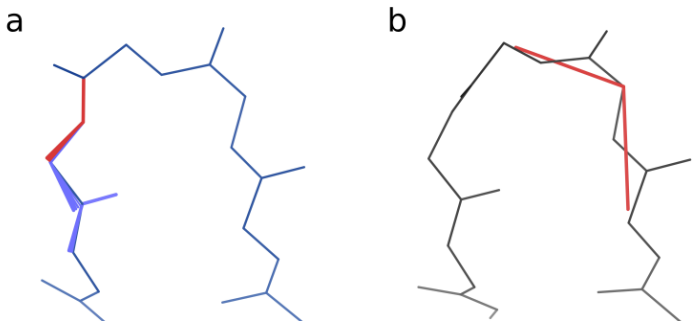


Figure 3: Resolution and introduction of geometry outliers in cryoEM Challenge target T0002 by group EM130. The target (a) contains severe bond angle outliers around chain 1 Ala107. The model (b) resolves these outliers but introduces a Cα geometry outlier – probably an unrealistic Cα virtual angle – in the process. The arrangement of the outliers suggests but cannot prove that over-idealization of covalent geometry lead to distorted Cα geometry in the model.

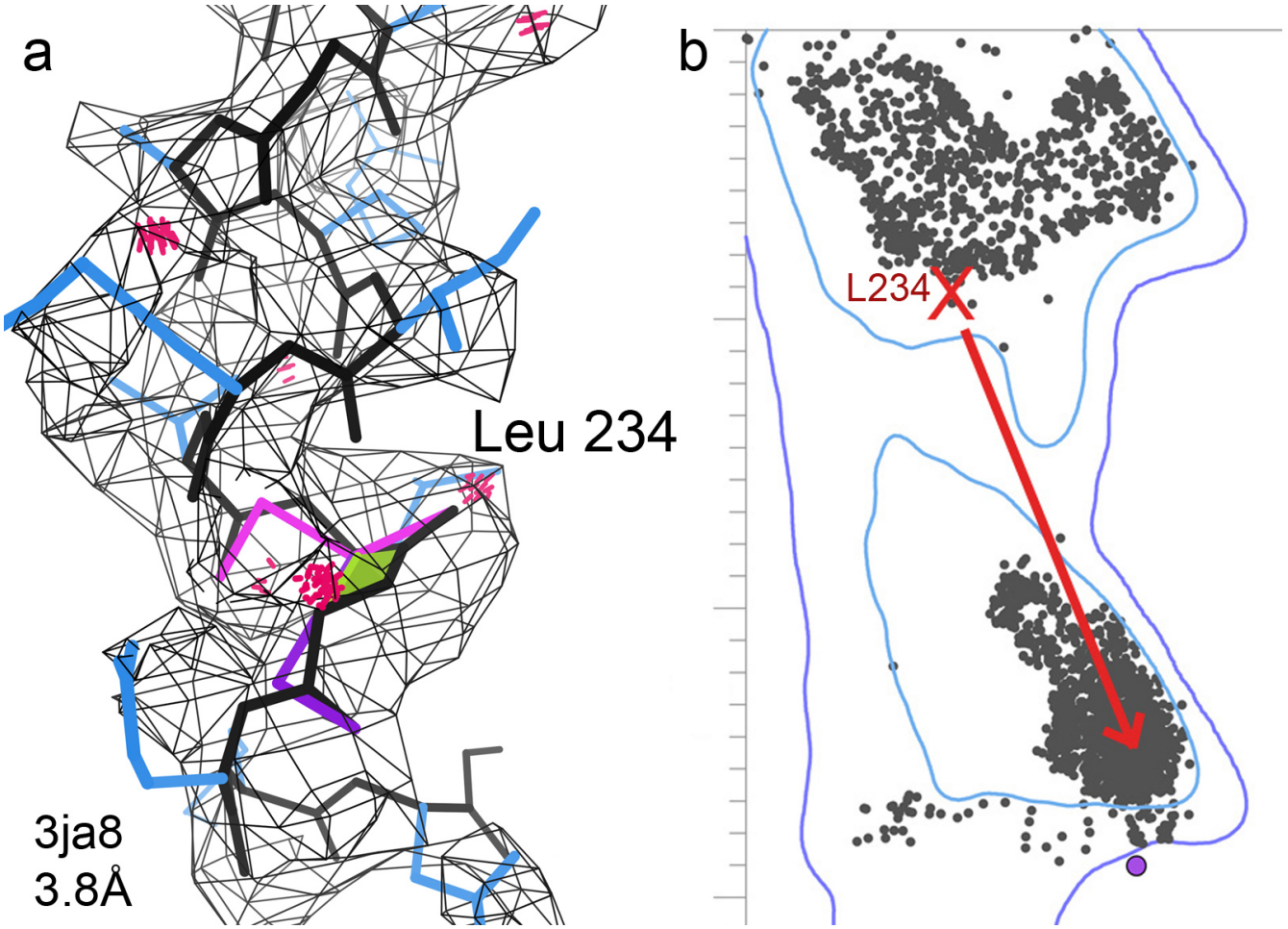


Figure 4: Compounding and obfuscation of a secondary structure error in 3ja8.pdb. An incorrect cis-peptide and distorted alpha helix (a) placed the Ramachandran point for Leu234 near the beta region (b). During refinement, overfitting to Ramachandran criteria moved this point solidly into beta, rather than correcting the underlying modeling error.

the beta contours. Overfitting to Ramachandran criteria has thus compounded an existing error and hidden it from casual validation.

Discussion

Given the overfitting to Ramachandran criteria in the challenge models, the simultaneous heavy restraining of covalent geometry criteria came as a surprise. It should not be possible to distort the Ramachandran distribution in the ways seen in Figure 1 without consequences, and bond geometry seemed a likely place to dump the consequences. And indeed, the geometry distributions of group EM120 show the sort of off-ideal peaks we might expect if small geometry deviations were taking up the strain of an artificially tight Ramachandran distribution. However, EM120 is the exception rather than the rule, and bond geometries, especially bond lengths, are highly restrained for almost all of the challenge models. Both Ramachandran and geometric criteria are being overfit, and this overfitting affects the implicit or explicit balance of fit-to-map and fit-to-geometry in the refinement target function. As we work as a community to assess and address overfitting, one of the questions we will have to explore is: where else do the consequences go?

References

Richardson, JS, Keedy, DA, Richardson, DC (2013). "The Plot" Thickens: More Data, More Dimensions, More Uses. In *Biomolecular Forms and Functions: A Celebration of 50 Years of the Ramachandran Map* (pp. 46-61).

Williams, CJ, Headd, JJ, Moriarty, NW, Prisant, MG, Videau, LL, Deis, LN, Verma, V, Keedy, DA, Hintze, BJ, Chen, VB, Jain, S, Lewis, SM, Arendall, WB, Snoeyink, J, Adams, PD, Lovell, SC, Richardson JS, Richardsdon DC. (2018) "MolProbity: More and better reference data for improved all-atom structure validation", *Protein Science*, **27**(1), 293-315.

Cβ deviations and other aspects in Amber vs CDL refinements

Jane Richardson and David Richardson, Duke University

Background

The Amber force-field (Case 2016) has been implemented for use in Phenix refinement. Nigel Moriarty and David Case have been tuning and extensively testing this new feature. In collaboration with that effort, the Duke Phenix team is looking at individual results from paired refinements that use Amber vs. the current default refinement that includes the CDL (Conformation-Dependent Library; Moriarty 2014). This article describes our most interesting results so far.

Analysis of parallel refinement results for 1xgo

The 1xgo example at 3.5Å resolution is an ideal test case for our purposes because there is an essentially identical structure available at high resolution to define the correct answers for local conformations. 1xgo ended up scoring somewhat better on most measures after Amber refinement than after CDL refinement, as seen in the MolProbity-chart stoplight comparisons in Figure 1, which also shows the original deposited 1xgo (Tahirov 1998) and the 1.75Å 1xgs structure done at the same time of the same molecule in a different space group (Tahirov 1998). The CDL refinement completely idealizes all the covalent geometry with tight restraints, rotamers are about equal and Amber does better on both Ramachandran and clashscore.

However, there was concern that six residues had Cβ-deviation outliers (≥0.25Å) after Amber refinement and whether those reflected any issues with the force-field terms. Figure 2 below shows the "bullseye" Cβ deviation plots for all four coordinate sets, centered at the ideal Cβ and with outliers emphasized. The CDL-refine plot is tight to an extreme degree, perhaps justifiable at low resolution but debatable. Even without an artificial Cβdev restraint, if Tau (N-Cα-C), N-Cα-Cβ and C-Cα-Cβ angles are all tightly restrained there will be no Cβdev outliers. The Amber plot is a bit on the loose side, but shows a good scatter with no evidence of target anomalies and extends farthest out in the direction that is indeed most vulnerable. The crucial issue is whether those outliers are just being allowed by too-lenient restraints or are diagnosing real

MOL PROBITY

Duke Biochemistry
Duke University School of Medicine

Summary statistics

		1xgo 3.5Å			CDL-refine			Amber-refine			1xgs 1.75Å		
All-Atom Contacts	Clashscore, all atoms:	78.83		16th	7.42		100th	0.42		100th	6.78		90th percentile
	Clashscore is the number of serious steric overlaps (> 0.4 Å) per 1000 atoms.												
Protein Geometry	Poor rotamers	65	26.75%		46	18.93%		50	20.58%		15	3.09%	
	Favored rotamers	133	54.73%		158	65.02%		156	64.20%		445	91.56%	
	Ramachandran outliers	24	8.19%		12	4.10%		11	3.75%		0	0.00%	
	Ramachandran favored	204	69.62%		239	81.57%		256	87.37%		573	97.78%	
	MolProbity score^	4.29		13th	3.09		85th	2.26		99th	1.79		77th percentile
	Cβ deviations >0.25Å	2	0.73%		0	0.00%		6	2.20%		2	0.37%	
	Bad bonds:	0 / 2353	0.00%		0 / 2353	0.00%		0 / 2353	0.00%		0 / 4706	0.00%	
	Bad angles:	11 / 3178	0.35%		0 / 3178	0.00%		29 / 3178	0.91%		3 / 6356	0.05%	
Peptide Omegas	Cis Prolines:	1 / 14	7.14%		1 / 14	7.14%		1 / 14	7.14%		2 / 28	7.14%	
	Twisted Peptides:							1 / 294	0.34%				
Low-resolution Criteria	CaBLAM outliers	15	5.14%		17	5.82%		11	3.77%		5	0.86%	
	CA Geometry outliers	6	2.05%		5	1.71%		7	2.40%		2	0.34%	

Figure 1: Comparison of MolProbity summary validation for 1xgo, CDL refinement, Amber refinement and 1xgs.

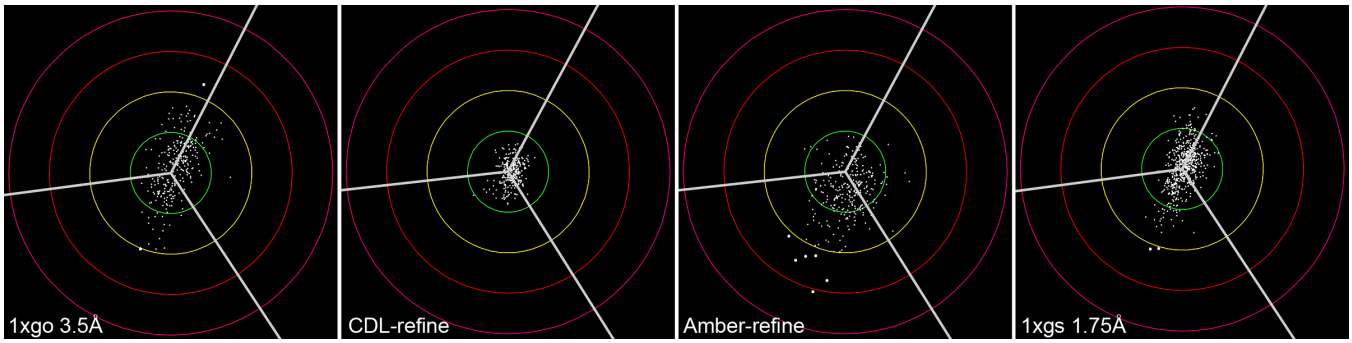


Figure 2: Comparison of "bullseye" plots of Cβ deviation for 1xgo, CDL refinement, Amber refinement and 1xgs.

problems – so we looked at them all and compared each with our expectations, with electron density and especially with the 1xgs 1.75Å structure.

Of course we will need to analyze more cases, but it is remarkable for 6 arbitrary examples of some effect to all point consistently and convincingly to the same conclusion. Each one of the outlier residues has a real problem uncorrectable by downhill refinement, with either the sidechain or the backbone or both in the wrong local-minimum conformation. This flagging of outliers is the most important purpose of validation: to support either people or software in testing distinct alternatives to correct conformational misfittings that need not just a nudge but a big change.

The 4-part comparison in Figure 3 illustrates Leu 204, the simplest of the 6 Amber Cβdevs, where the rotamer is wrong and pure refinement cannot correct it. The original model-building for 1xgo tried too hard to push all atoms into the density, resulting in a curled-up sidechain with a near-zero chi1 angle. The high-resolution 1xgs structure has completely unambiguous density showing Leu 204 with an **mt** conformation, the overwhelmingly most common Leu rotamer (Hintze 2016). Both refinements shifted enough to lose the clash that was in 1xgo, but the Cβdev from Amber, as well as the rotamer outlier, signal even at 3.5A that this sidechain needs to be rebuilt.

Alanine is seldom a Cβdev outlier, but the well-ordered 3.5Å electron density for the Ile-Gly-Ala194-Gly semi-extended sequence just does not have enough information content to specify its conformation correctly. As seen in Figure 4, both refinements resolve the clashes in 1xgo, but do not improve the backbone conformation, as reported by CaBLAM outliers (Williams 2018) and by Amber's Ala Cβdev. With all backbone COs clear in the density at 1.75Å, 1xgs identifies a favorable, unambiguously correct conformation. To achieve that, ϕ,ψ angles change by at least 30°

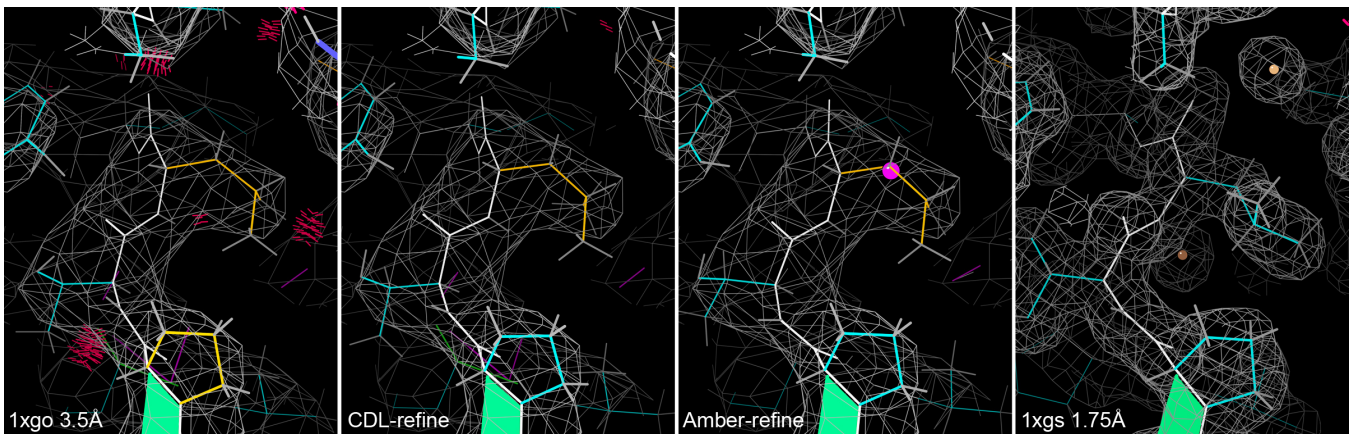


Figure 3: Comparison of Leu 204 rotamer outlier for 1xgo, CDL refinement, Amber refinement and 1xgs.

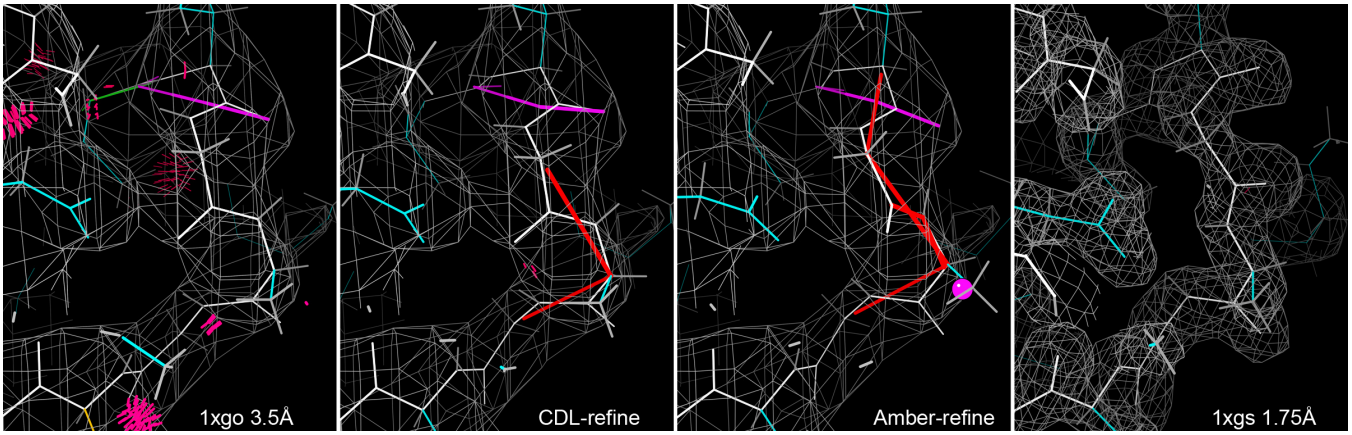


Figure 4: Comparison of Ile-Gly-Ala194-Gly backbone for 1xgo, CDL refinement, Amber refinement and 1xgs.

in each of the 4 residuesand ψ of Gly193 and ϕ of Ala194 change more than 120°.

For Leu 253 on a helix, both sidechain rotamer and backbone shifts are involved (see Figure 5). In the original 1xgo model, Leu 253 is in the very low-population **pt** rotamer, which unsurprisingly is maintained in both refinements although slightly shifted. However, the 1xgs structure shows very clear density for the highest-population **mt** rotamer and that conformation looks as though it would have fit the low-resolution density just as (moderately) well as **pt** did. The helix conformation is very non-ideal in 1xgo, with a distorted overall shape and a Ramachandran outlier. Both refinements resolve the numerous clashes while rotamer outliers stay similar. CDL refinement does not move the backbone; Amber improves it toward ideality, although still not as regular as shown by the high-resolution 1xgs.

In order to understand better what each of these refinement protocols can and cannot do, our group will need to examine details where rotamers or clashes have improved with pure refinement. The questions are whether distinct conformations can ever be interconverted and in which circumstances the changes from outlier to allowed rotamer (or Ramachandran) have only shifted across a close borderline versus when they have indeed discovered the correct rotamer.

The bottom line

Our overall conclusion is that refinement cannot be expected to correct local conformations in the wrong local minimum, especially at low resolution. Ideally, as many as possible of those problems should be corrected early-on, either by

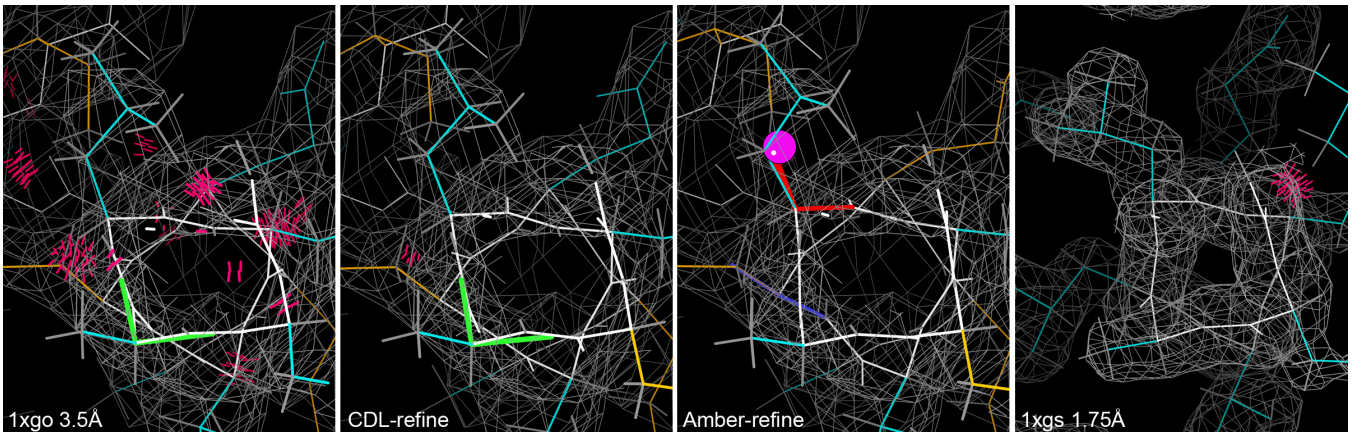


Figure 5: Comparison of Leu 253 on helix for 1xgo, CDL refinement, Amber refinement and 1xgs.

manual rebuilding or by an automated procedure that evaluates distinct alternatives. It is very counterproductive to sweep such problems under the rug by artificially adding terms that disallow validation outliers. The CDL refinement corrects all geometry outliers, preventing them from showing problems, but does leave backbone and some steric flags. The Amber refinement not only corrects clashes but improves van der Waals contacts (even among incorrect rotamers or backbone!), but allows geometry to report problems. A strategy partway in between might be even better. However, we feel very positive about the Amber target after looking at these details: if it were presented with a model where most big misfittings had been corrected, it would do an excellent job of tuning the details correctly. Refinement should work honestly on all independent single terms, while the whole point of combined, non-refined validation criteria (Rfree, rotamers, Ramachandran, Cbdev, CaBLAM, RNA backbone conformers and ribose puckers, …) is to flag places that probably need large changes between barrier-separated conformations. We should not try to hide those flags artificially, but should develop better procedures for using what they tell us.

References:

Case DA, Betz RM, Cerutti DS, Cheatham TE III, Darden TA, Duke RE, Giese TJ, Gohlke H, Goetz AW, Homeyer N, Izadi S, Janowski P, Kaus J, Kovalenko A, Lee TS, LeGrand S, Li P, Lin C, Luchko T, Luo R, Madej B, Mermelstein D, Merz KM, Monard G, Nguyen H, Nguyen HT, Omelyan I, Onufriev A, Roe DR, Roitberg A, Sagui C, Simmerling CL, Botello-Smith WM, Swails J, Walker RC, Wang J, Wolf RM, Wu X, Xiao L, Kollman PA (2016), AMBER 2016, University of California, San Francisco

Hintze BJ, Lewis SM, Richardson JS, Richardson DC (2016) "MolProbity's ultimate rotamer-library distributions for model validation", *Proteins: Struc Func Bioinf* **84**: 1177-1189

Moriarty NW, Adams PD, Karplus PA (2014) Details of the conformation-dependent library, *Comput Crystallogr Newsletter* **5**: 42-49

Tahirov TH, Oki H, Tsukihara T, Ogasahara K, Yutani K, Ogata K, Izu Y, Tsunasawa S, Kato I (1998) Methionine aminopeptidase from hyperthermophile *Pyrococcus furiosus, J Mol Biol* **284**: 101-124

Williams CJ, Hintze BJ, Headd JJ, Moriarty NW, Chen VB, Jain S, Prisant MG Lewis SM, Videau LL, Keedy DA, Deis LN, Arendall WB III, Verma V, Snoeyink JS, Adams PD, Lovell SC, Richardson JS, Richardson DC (2018) MolProbity: More and better reference data for improved all-atom structure validation, *Protein Science* **27**: 293–315

A few benchmark tests of various compilers on Linux and Windows.

Robert D Oeffner

Cambridge Institute for Medical Research, University of Cambridge, Cambridge Biomedical Campus
Wellcome Trust/MRC Building, Hills Road, Cambridge CB2 0XY

Correspondence email: rdo20@cam.ac.uk

Introduction

The Phenix crystallographic software suite is deployed on three different platforms; MacOS, Windows and Linux. In this article we present the results of a few benchmark tests of the Phaser executable produced by compilers from GNU, Intel and Microsoft. The platforms we test on are Ubuntu 16 and Windows 10 as well as Windows 10 with Windows Subsystem for Linux (WSL) running Ubuntu 16.

Motivation

We were interested to determine which platform and compiler provides the fastest performance of the Phenix software. Moreover, benchmark tests on the WSL platform are of interest given it allows running Linux executables directly on Windows without the overhead of starting up a virtual machine, but it might incur a performance cost.

We are interested in the overall time taken to run executables, i.e. the wall time. Phenix is a software suite consisting of python scripts and C++ code compiled into python modules. It is the choice of C++ compilers we test here. We focus on Phaser which, like the rest of Phenix, depends on the CCTBX library. During the time of performing the benchmark tests BIOS and OS patches for the "Meltdown" bug for Intel CPUs became available (see Patches in the appendix). Additional benchmark tests for the patched platforms have therefore been included.

Compilers and platforms

We tested the platforms and compilers listed in Table 1.

The specific versions of the operating systems are:

- Ubuntu 16.04.3 kernel 4.10.0-28-generic
- Windows 10, version 1709, build 16299.125
- Ubuntu 16.04.3 kernel 4.4.0-109-generic
- Windows 10, version 1709, build 16299.192

The first two operating systems in the above list were not patched for the Intel Meltdown bug whereas the last two have been patched for the Intel Meltdown bug as has the BIOS on the PC.

The Visual Studio 2008 compiler is used for Phenix Windows builds due to it using the same C-runtime as the python executable that is deployed with the python interpreter distributed from python.org.

The Visual Studio 2015 compiler is used for building Python 3 on Windows. As Phaser distributed with Phenix eventually will be migrated to Python 3 this compiler or newer ones are of interest.

The MinGW-W64 Gnu 5.3.0 compiler is used for building Phaser distributed with CCP4 version 7 on Windows. This is a Gnu compiler ported to Windows, which is of interest as it comes with the Gnu implementation of the C++ Standard Template Library and OpenMP library.

The Intel Parallel Studio XE 2018 for Linux compiler, unlike the others, is not free, but is reputed to produce very fast executables.

Table 1: List of compilers and platforms used for benchmarking. Abbreviations in bold are used in subsequent figures and tables denoting the executables produced by the respective compilers.

C++ Compiler	Platforms (64 bit)	Comment
Visual Studio 2008 (**VS2008**) by Microsoft	Windows 10	used for current Phenix build with python 2 on Windows
Visual Studio 2015 (**VS2015**) by Microsoft	Windows 10	to be used in future for Phenix build with python 3 on Windows
MinGW-W64 g++ version 5.3.0 (**MinGW 5.3.0**)	Windows 10	used for CCP4 Windows build of Phaser
Intel Parallel Studio XE 2018 for Linux (**Intel**)	Native Ubuntu 16.04, Ubuntu 16.04 on WSL on Windows 10	uses the C++ Standard Template Library from the Gnu compiler
g++ version 5.4.0 for Linux (**Gnu 5.4.0**)	Native Ubuntu 16.04 Ubuntu 16.04 on WSL on Windows 10	default on Ubuntu 16.04
g++ version 4.4.7 for Linux (**Gnu 4.4.7**)	Native Ubuntu 16.04 Ubuntu 16.04 on WSL on Windows 10	default on Centos 6, build of Phaser in Phenix

The Gnu 5.4.0 compiler suite for Linux is the default on Ubuntu 16.04. It can therefore be anticipated that a user of Phenix will use this compiler if rebuilding Phenix from sources on Ubuntu. Given the popularity of Ubuntu amongst other flavours of Linux this compiler version is therefore relevant to benchmark.

The g++ 4.4.7 compiler for Linux is what is currently used for building Phenix for Linux on the Centos6 platform. The combination of this compiler and platform is convenient as it permits Phenix to be installed on a variety of Linux platforms.

In total there are six different executables three of which will run both on WSL as well as native Linux. In the subsequent text executables produced by the Visual Studio or the MinGW compilers will be referred to as Windows executables or win32 executables. The Intel, the Gnu 5.4.0 and the Gnu 4.4.7 executables tested on native Linux will be tagged with "native". If they are tested on WSL they will be tagged with "WSL".

Hardware

The PC used for the test is an 8 core Intel Xeon CPU E5-1660 v3 @ 3.0GHz. Hyperthreading is enabled meaning that the operating systems see it as having 16 logical cores. The available memory is 32 Gb with L1, L2 and L3 cache memory of 512Kb, 2Mb and 20Mb respectively.

To benchmark on Windows 10 or on WSL the PC was booted with the installed Windows 10. To benchmark on native Ubuntu 16 it was booted with an Ubuntu 16.04.3 ISO image from an external USB device. This ensured the hardware was the same during all tests.

Patching the PC for the meltdown bug entails doing a BIOS update, a Windows update as well as updating the kernel of an existing

Ubuntu 16.04.3 installation on a different PC. The Linux LiveKit software available from https://www.linux-live.org/ was then used to deploy this patched Ubuntu version as a bootable image to an external USB device. This USB device could then be used for booting the PC into the patched version of Ubuntu 16.04.3.

Software

The software used for the tests is written in C++. The times were recorded with the Linux "time" command. The CPU usage is recorded with the Linux "top" command or on Windows with a custom written C# script using Windows Management Information templates obtained from https://www.microsoft.com/en-us/download/details.aspx?id=8572. We are interested in how execution speeds scale with the number of threads. The threading technology tested here is OpenMP. Given the hardware exposes 16 logical cores the maximum number of OpenMP threads tested for is 16.

When a program run in serial mode, i.e. only uses one CPU, the usage is 100%. But if it is allowed 16 OpenMP threads a CPU usage of up to 1600% may occur whenever an OpenMP parallelized loop is executed. In all the figures depicting execution times against number of OpenMP threads the axes have been set to logarithmic scale as a way of verifying the scalability of parallelization.

Benchmark Tests

The following sections demonstrate the differences in execution time achieved with executables produced with the different compilers and on different platforms as listed above. Each test program has been executed on the above platforms and execution times are presented as graphs and tables. The wobbles in some of the graphs are due to the PC occasional executing other uninterruptable tasks such as virus scanning.

The π-program

We tested a small program that calculates the natural number **π** through looping over a slowly converging Fourier series expansion that was made even slower by redundant use of exp(), cos() and sqrt() operations. It can take advantage of OpenMP parallelization as illustrated below in figure 11 in the appendix.

Benchmarks of the π-program on un-patched platforms

As is seen in figure 1 the program scales well with the number of OpenMP threads being used; data points for each compiler follow almost straight lines sloping down albeit not quite achieving half the execution time as the number of threads doubles.

It is apparent that the execution times can be categorised into three groups:

- Intel executable
- Gnu5.4.0, MinGW5.3.0 and VS executables,
- Gnu4.4.7 executable

As seen in table 2 the Intel executable clearly outperforms all other executables. This is whether or not the executable is run on native Linux or on WSL. The speed of executables built with MinGW5.3.0, VS2015, VS2008, Gnu5.4.0 WSL or Gnu5.4.0 native Linux are roughly the same, about twice as slow as the Intel executable. The speed of an executable built with Gnu4.4.7 running either on native Linux or on WSL is between three or five times slower than the Intel executable.

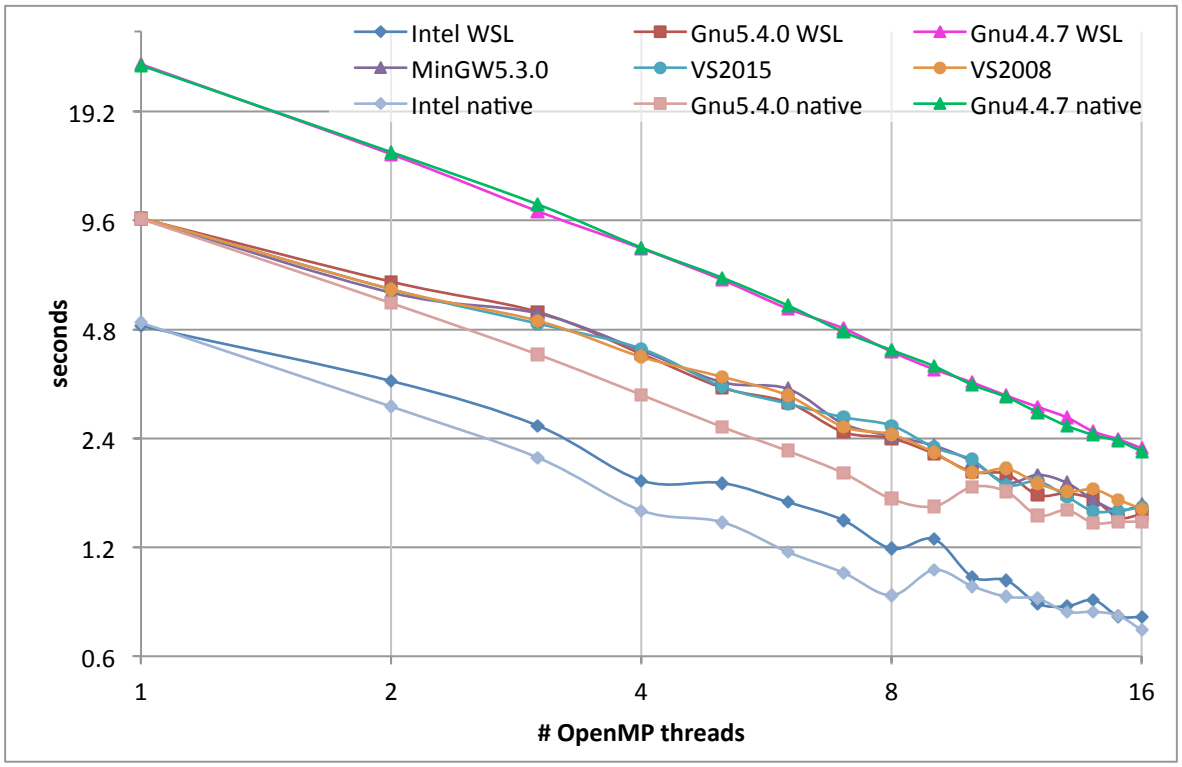


Figure 1: Graphs of execution times in seconds of the π calculation test job as a function of OpenMP threads for Phaser built with different compilers and running on different un-patched platforms.

Table 2: Execution times in seconds of the π calculation test job as a function of OpenMP threads for Phaser built with different compilers and running on different un-patched platforms.

OpenMP threads	Intel WSL	Gnu5.4.0 WSL	Gnu4.4.7 WSL	MinGW 5.3.0	VS2015	VS2008	Intel native	Gnu5.4.0 native	Gnu4.4.7 native
1	4.905	9.718	25.956	9.713	9.727	9.753	5.001	9.700	25.787
2	3.457	6.501	14.627	6.065	6.196	6.195	2.937	5.673	14.821
3	2.603	5.366	10.183	5.316	4.981	5.073	2.124	4.094	10.637
4	1.835	4.108	8.036	4.193	4.247	4.030	1.516	3.166	8.077
5	1.807	3.314	6.582	3.430	3.334	3.551	1.409	2.582	6.672
6	1.600	3.017	5.479	3.290	2.994	3.154	1.165	2.216	5.593
7	1.426	2.498	4.833	2.635	2.745	2.583	1.021	1.924	4.727
8	1.190	2.401	4.165	2.432	2.604	2.459	0.884	1.632	4.207
9	1.263	2.176	3.726	2.292	2.265	2.194	1.039	1.557	3.799
10	0.995	1.941	3.426	2.074	2.092	1.932	0.938	1.761	3.376
11	0.972	1.914	3.148	1.808	1.784	1.979	0.876	1.711	3.121
12	0.839	1.673	2.931	1.902	1.827	1.803	0.865	1.469	2.831
13	0.825	1.691	2.743	1.809	1.658	1.709	0.796	1.520	2.599
14	0.858	1.627	2.505	1.622	1.509	1.735	0.796	1.402	2.452
15	0.775	1.450	2.386	1.513	1.516	1.624	0.778	1.411	2.360
16	0.770	1.479	2.253	1.575	1.547	1.532	0.708	1.413	2.210

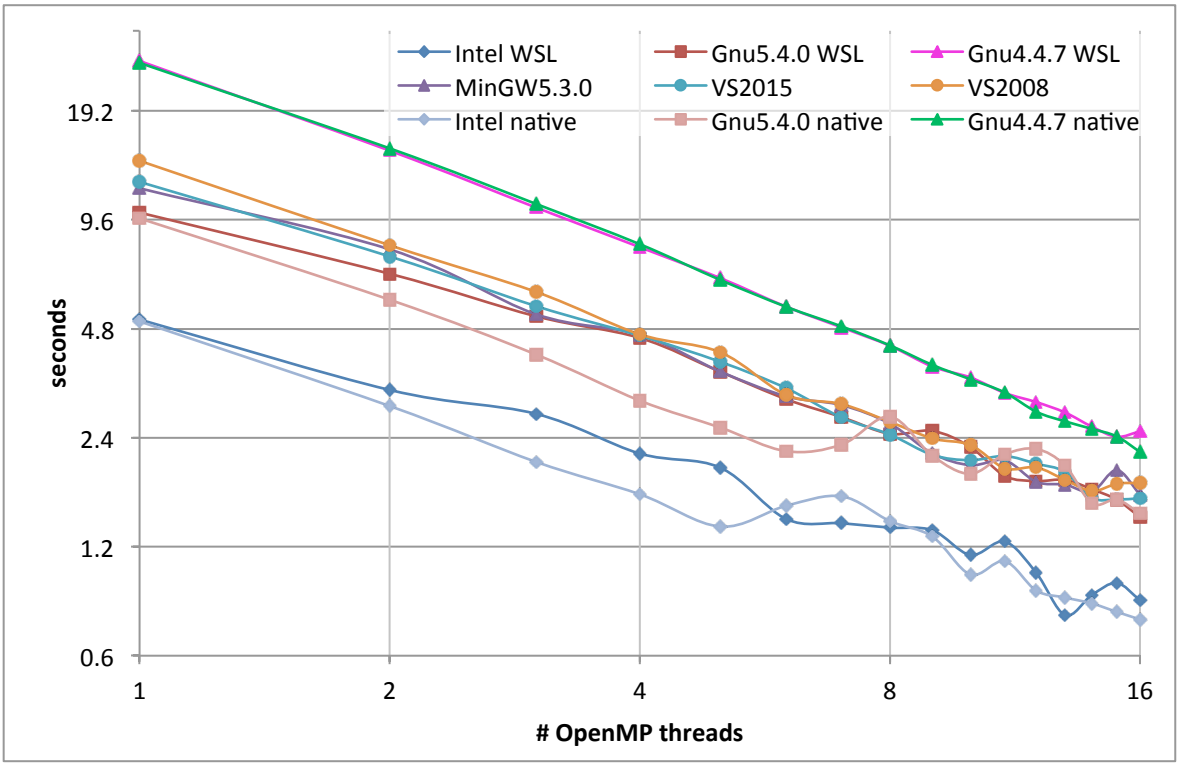


Figure 2: Graphs of execution times in seconds of the π calculation test job as a function of OpenMP threads for Phaser built with different compilers and running on different un-patched platforms.

Benchmarks of the π-program on patched platforms

Figure 2 and table 3 are the execution times for the same calculations as above but for the platforms patched for the Meltdown bug.

It is seen that the benchmark times on the patched platforms are roughly the same for all the Linux executables regardless of whether they are running on native Ubuntu or WSL. But all the Windows executables have increased their run time with up to 25% for single threaded use (VS2015: 12.223 seconds on the patched platform compared to 9.727 on the un-patched platform). On the other hand this discrepancy becomes smaller when the number of threads are increased.

Phaser

Phaser is a crystallographic software program making extensive use of the C++ Standard Template Library (STL) for manipulating dynamically allocated data objects that may often exceed hundreds of megabytes of memory. It also can take advantage of OpenMP parallelization. Much of the program runs sequentially however, as many functions cannot easily be parallelized. Given particular input data for a Phaser calculation which ultimately determines the size of dynamically allocated data objects for the calculation OpenMP parallelized loops may or may not significantly reduce the overall runtime of the calculation. Source details given in "Details of Phaser source versions" in the appendix.

The tests calculations discussed below were chosen with the criteria to be single component searches with clear unambiguous MR solutions to run for more than one but less than 20 minutes. As with the **π** program Phaser was compiled for maximum speed and with OpenMP for all compilers and as static executables, except for the Gnu 4.4.7 build

Table 3: Execution times in seconds of the π calculation test job as a function of OpenMP threads for Phaser built with different compilers and running on different patched platforms.

OpenMP threads	Intel WSL	Gnu5.4.0 WSL	Gnu4.4.7 WSL	MinGW 5.3.0	VS2015	VS2008	Intel native	Gnu5.4.0 native	Gnu4.4.7 native
1	5.093	10.062	26.355	11.719	12.223	13.973	5.034	9.697	26.016
2	3.253	6.811	14.904	7.954	7.604	8.164	2.936	5.774	15.122
3	2.794	5.198	10.379	5.266	5.545	6.070	2.059	4.076	10.622
4	2.171	4.528	8.066	4.625	4.595	4.634	1.675	3.039	8.220
5	1.984	3.651	6.625	3.656	3.890	4.134	1.364	2.561	6.549
6	1.432	3.068	5.524	3.125	3.290	3.153	1.557	2.204	5.521
7	1.394	2.737	4.840	2.953	2.728	2.972	1.653	2.298	4.872
8	1.355	2.461	4.308	2.625	2.449	2.656	1.413	2.745	4.319
9	1.332	2.507	3.774	2.172	2.152	2.389	1.280	2.140	3.814
10	1.139	2.263	3.521	2.016	2.081	2.292	1.006	1.908	3.481
11	1.239	1.883	3.191	2.078	2.136	1.976	1.095	2.160	3.200
12	1.017	1.822	3.011	1.813	2.036	1.988	0.908	2.235	2.832
13	0.777	1.836	2.818	1.781	1.931	1.835	0.869	2.012	2.663
14	0.882	1.730	2.576	1.703	1.638	1.710	0.837	1.581	2.540
15	0.953	1.619	2.415	1.953	1.620	1.791	0.793	1.619	2.420
16	0.854	1.449	2.499	1.672	1.633	1.801	0.756	1.484	2.192

where we used a Phenix installation of the Phenix-1.13-rc1-2965 version built on Centos6.

MR_AUTO on PDB entry 1ioM

This test consists of a data set with 86550 reflections. The keyword script is listed in figure 12 in the appendix. From figure 3 and the corresponding data in Table 4 it is apparent that for all executables and platforms there is no significant benefit from using OpenMP with more than about five or six threads.

Benchmarks of an MR_AUTO run with Phaser on 1ioM on un-patched platforms

In this calculation the executables built by the Microsoft compilers are the fastest if more than one OpenMP thread is used. Figure 4 shows the CPU usage as a function of wall time as recorded during the Phaser calculation with 16 OpenMP threads being allowed.

Considerable differences between the CPU loads of the executables can be seen. Although the calculation with the VS2015 executable is the fastest compared to the other executables it appears to spend more than 100 seconds within the first OpenMP parallelized loop whereas the Intel and the Gnu executables spend less than 50 seconds. On the other hand the VS2015 executable more than makes up for that sluggishness outside the parallelized loop since it finishes the calculation in less than 500 seconds.

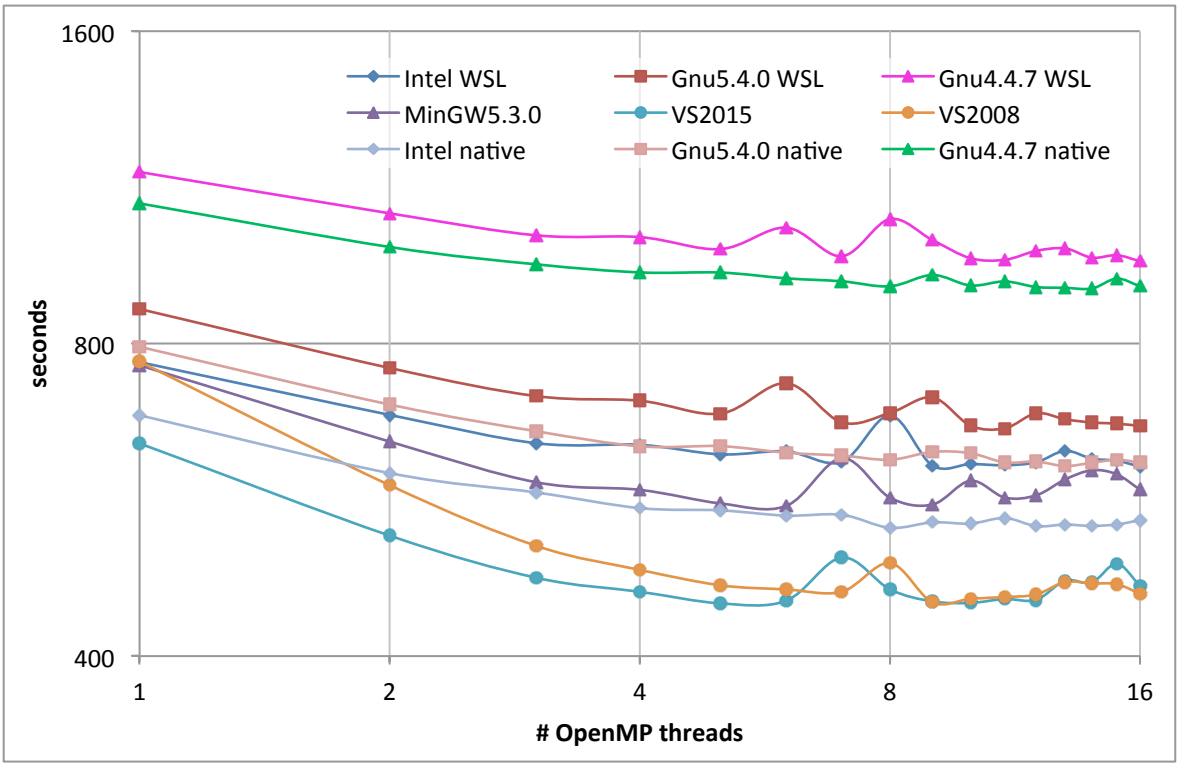


Figure 3: Graphs of execution times in seconds of the MR_AUTO on 1ioM test job as a function of OpenMP threads for Phaser built with different compilers and running on different un-patched platforms.

Table 4: Execution times in seconds of the MR_AUTO on 1ioM test job as a function of OpenMP threads for Phaser built with different compilers and running on different un-patched platforms.

OpenMP threads	Intel WSL	Gnu5.4.0 WSL	Gnu4.4.7 WSL	MinGW 5.3.0	VS2015	VS2008	Intel native	Gnu5.4.0 native	Gnu4.4.7 native
1	767.45	864.17	1170.32	762.39	641.45	768.3	682.07	793.82	1091.42
2	682.89	757.74	1067.72	643.62	522.5	584.24	600.16	698.72	990.89
3	641.36	712.01	1016.93	588.2	475.96	510.89	575.03	658.62	953.6
4	639.05	704.69	1012.92	578.32	461.35	484.34	555.38	636.97	936.61
5	625.68	684.83	986.27	561.17	449.6	468.04	552.64	637.17	936.31
6	629.86	732.58	1034.38	558.01	452.27	463.77	546.2	627.89	924.4
7	615.25	671.62	970.86	620.36	497.68	461.36	547.17	623.91	918.35
8	681.97	685.8	1054.11	568.42	463.83	491.66	531.66	618.24	908.26
9	610.35	709.89	1006.31	559.72	451.6	450.62	538.1	629.13	931.18
10	612.28	666.84	966.61	590.15	450.07	454.21	536.65	627.29	909.87
11	610.78	662.82	963.15	568.43	453.98	455.92	543.32	614.5	918.04
12	614.82	685.35	982.43	571.36	453.14	458.61	533.57	615.77	906.73
13	630.53	676.28	987.78	591.8	472.84	471.43	535.41	609.63	905.16
14	619.69	671.5	967.76	603.99	470.91	470.01	533.88	614.14	903.73
15	616.96	669.77	973.02	598.67	490.95	469.15	535.59	618.11	923.75
16	608.77	666.39	960.16	578.55	467.45	459.37	540.73	614.55	908.38

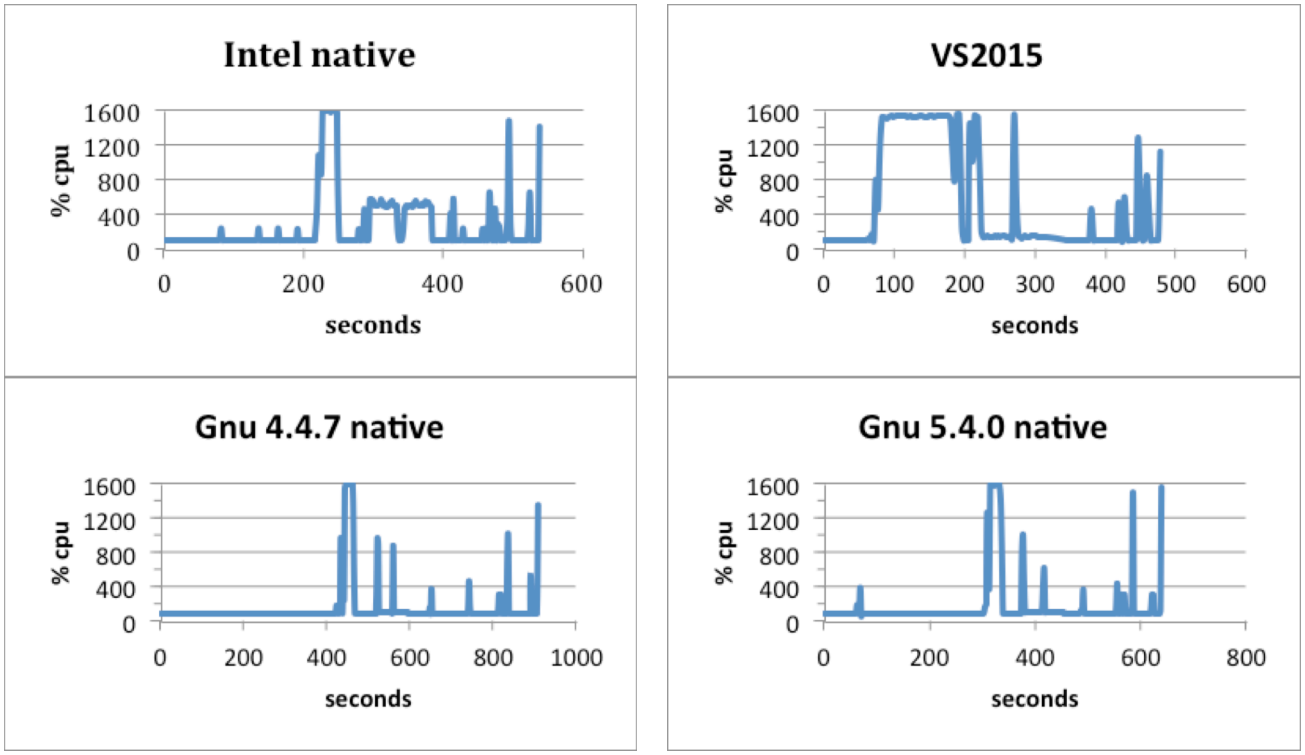


Figure 4: CPU usage graphs for four selected executables running on un-patched platforms with 16 OpenMP threads for the MR_AUTO on 1ioM test job.

Benchmarks of an MR_AUTO run with Phaser on 1ioM on patched platforms

Figure 5 and table 5 display the execution times for the same calculations but for the platforms patched for the Meltdown bug.

We note that the Windows executables suffer a run time increase of up to 37% (VS2015: one OpenMP thread 881.37 seconds on patched platform compared to 641.45 seconds on the un-patched platform). For the Linux executables there is virtually no change in speed when running them on patched or un-patched Ubuntu platforms. But running them on WSL they also suffer a performance hit of up to 25% increase in execution time.

Although not shown here, the graphs of the CPU usage as a function of wall time recorded during the Phaser calculation with 16 OpenMP threads being allowed are very similar to the graphs of executables running on un-patched platforms in figure 4.

MR_AUTO on 1HBZ

This test consists of a data set with 95355 reflections. The keyword script is listed in figure 13 in the appendix. In figure 6 it is apparent that for all executables and platforms there is no significant benefit from using OpenMP with more than about five or six threads. In fact on the WSL platform and for the win32 executables a slight overhead occurs if more than about 8 threads are used. This is not so for the Intel and Gnu executables running on native Ubuntu.

Benchmarks of an MR_AUTO run with Phaser on 1HBZ on un-patched platforms

For this test the Intel executable on native Ubuntu has the fastest execution time closely followed by the Gnu5.4.0 executable on native

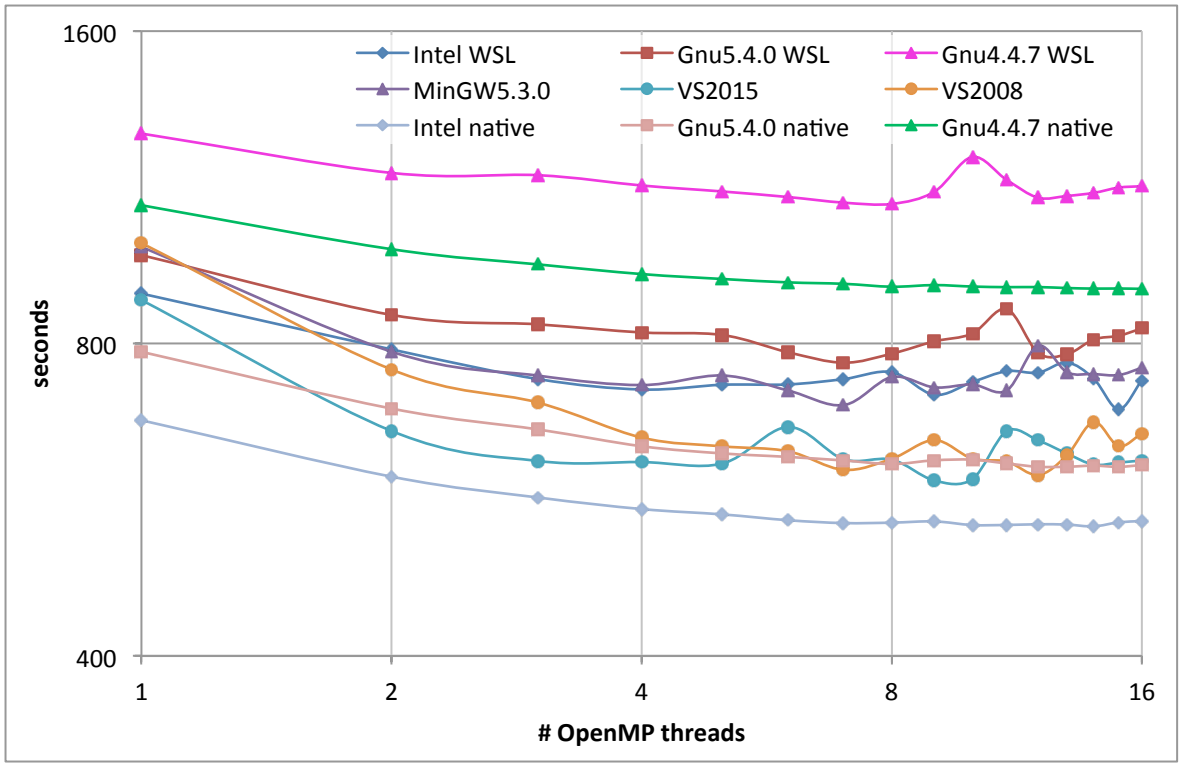


Figure 5: Graphs of execution times in seconds of the MR_AUTO on 1ioM test job as a function of OpenMP threads for Phaser built with different compilers and running on different patched platforms.

Table 5: Execution times in seconds of the MR_AUTO on 1ioM test job as a function of OpenMP threads for Phaser built with different compilers and running on different patched platforms.

OpenMP threads	Intel WSL	Gnu5.4.0 WSL	Gnu4.4.7 WSL	MinGW 5.3.0	VS2015	VS2008	Intel native	Gnu5.4.0 native	Gnu4.4.7 native
1	893.92	973.13	1274.63	991.73	881.37	999.68	674.85	785.34	1086.48
2	789.47	852.26	1167.63	785.51	658.87	755.02	595.36	692.44	985.95
3	739.27	834.61	1161.77	744.93	616.52	701.71	568.54	661.14	953.29
4	722.31	819.72	1135.5	729.48	614.98	649.24	554.11	636.93	933.04
5	729.94	815.17	1120.59	745.11	612.9	636.86	547.61	626.89	922.95
6	730.49	784.57	1106.87	720.79	664.67	629.89	540.58	622.05	915.99
7	739.17	766.55	1093.13	697.73	618.86	604.46	537.15	617.16	913.09
8	750.7	782.09	1090.04	742.59	618.31	619.45	537.67	612.31	907.21
9	713.89	803.67	1119.75	725.72	590.35	645.63	539.16	617.37	910.51
10	734.34	817.78	1209.7	730.37	592.19	619.68	534.82	618.25	907.56
11	752.3	863.43	1149.69	720.76	658.54	615.76	534.92	613.11	906.36
12	749.82	783.43	1105.23	795.82	645.31	597.56	535.74	608.59	906.52
13	766.81	781.07	1109.49	750.69	626.7	624.14	535.36	608.6	904.57
14	739.92	807.06	1117.08	747.56	611.49	672.13	533.23	610.11	903.43
15	691.24	813.66	1130.17	745.79	614.25	637.13	538.02	608.51	903.62
16	737.22	827.17	1133.72	757.98	617.17	655.36	539.52	610.9	902.94

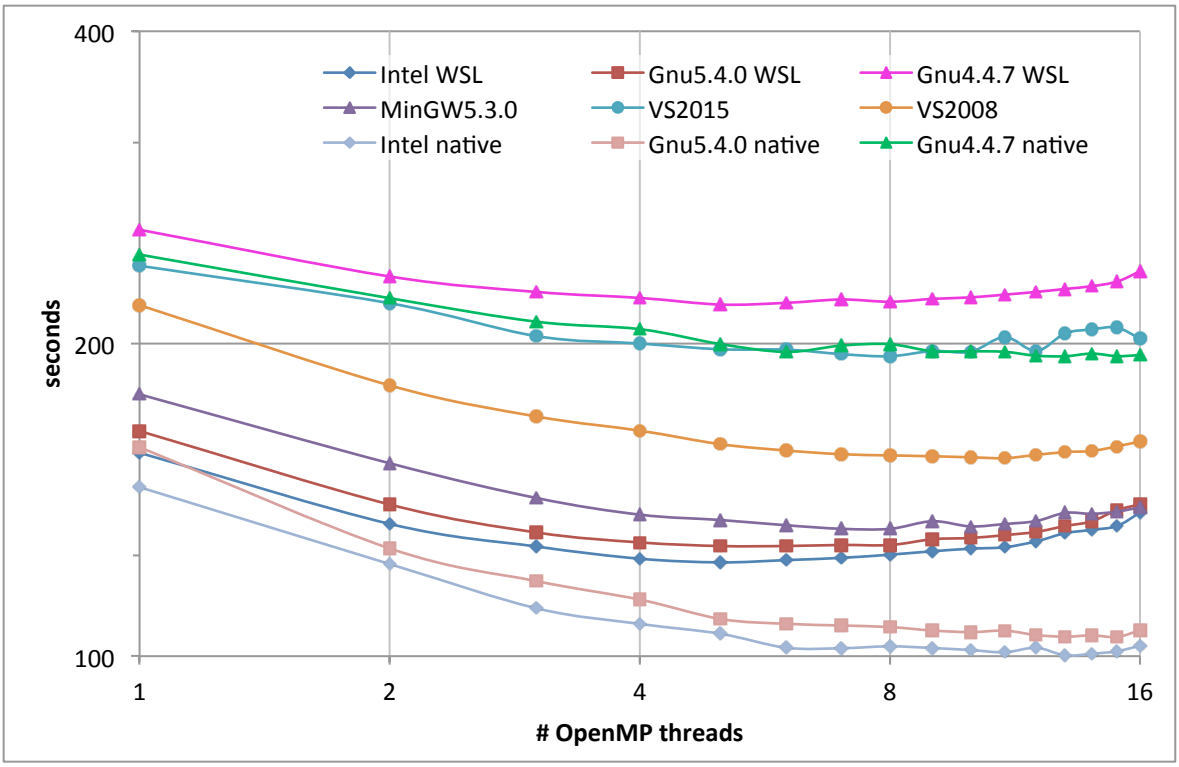


Figure 6: Graphs of execution times in seconds of the MR_AUTO on 1HBZ test job as a function of OpenMP threads for Phaser built with different compilers and running on different un-patched platforms.

Ubuntu as seen in table 6. Among the win32 executables it is interesting to note the ranking with the MingGW5.3.0 executable faster than the VS2008 executable that in turn is faster than the VS2015 executable. Using 16 OpenMP threads the Gnu4.4.7 executable runs almost twice as slowly as the Gnu 5.4.0 executable.

Benchmarks of an MR_AUTO run with Phaser on 1HBZ on patched platforms

For this test calculation the execution times on the patched platforms differ significantly from the un-patched runs. In figure 7 the most immediate observation is the significant overhead when running Linux executables on the WSL platform with more than 4 OpenMP threads. All three executables run slower with 16 threads than with just one thread. The execution times of the Windows executables plateau for more than five OpenMP threads.

The Linux executables running on native Ubuntu are practically unaffected by the patches for the Meltdown bug as seen in table 7. The increase in execution time is mostly under one percent regardless of number of OpenMP threads. This is contrary to the Windows executables where the fastest Windows executable, MinGW 5.40, suffer from an increase in execution time of between 14% and 24%.

MR_FRF on vz170x37_P1

This test consists of a data set with 156982 reflections. The keyword script is listed in figure 14 in the appendix. The fast rotation mode in Phaser predominantly exercises one of the OpenMP parallelized functions within Phaser. Avoiding serial single processing calculations should better inform us on how OpenMP execution times scales with the number of threads for Phaser.

Table 6: Execution times in seconds of the MR_AUTO on 1HBZ test job as a function of OpenMP threads for Phaser built with different compilers and running on different un-patched platforms.

OpenMP threads	Intel WSL	Gnu5.4.0 WSL	Gnu4.4.7 WSL	MinGW 5.3.0	VS2015	VS2008	Intel native	Gnu5.4.0 native	Gnu4.4.7 native
1	156.98	164.75	257.41	178.71	237.67	217.77	145.46	158.94	243.62
2	134.08	139.98	232.08	153.38	218.72	182.29	122.67	126.99	221.18
3	127.54	131.58	224.28	142.04	203.37	170.19	111.23	118.13	209.96
4	124.12	128.69	221.31	136.86	200.03	164.89	107.42	113.37	206.5
5	123.12	127.66	218.07	135.25	197.49	160.01	105.1	108.6	199.78
6	123.74	127.67	218.9	133.68	197.33	157.79	101.89	107.44	196.34
7	124.36	127.97	220.54	132.63	195.37	156.48	101.78	107.04	199.13
8	125.24	127.94	219.43	132.62	194.46	156.08	102.21	106.67	199.73
9	126.19	129.6	220.83	134.88	196.74	155.78	101.82	105.85	196.63
10	126.91	129.96	221.56	133.34	196.4	155.43	101.37	105.48	196.62
11	127.33	130.79	222.93	134.06	202.91	155.13	100.87	105.82	196.37
12	128.99	131.72	224.27	134.93	196.7	156.25	101.89	104.79	194.69
13	131.34	133.45	225.74	137.41	204.55	157.24	100.18	104.41	194.46
14	132.32	134.85	227.14	137.13	206.37	157.62	100.54	104.72	195.61
15	133.43	138.22	229.41	137.76	207.24	159.22	101.09	104.35	194.52
16	137.42	139.98	234.64	139.13	202.23	160.97	102.32	105.87	194.91

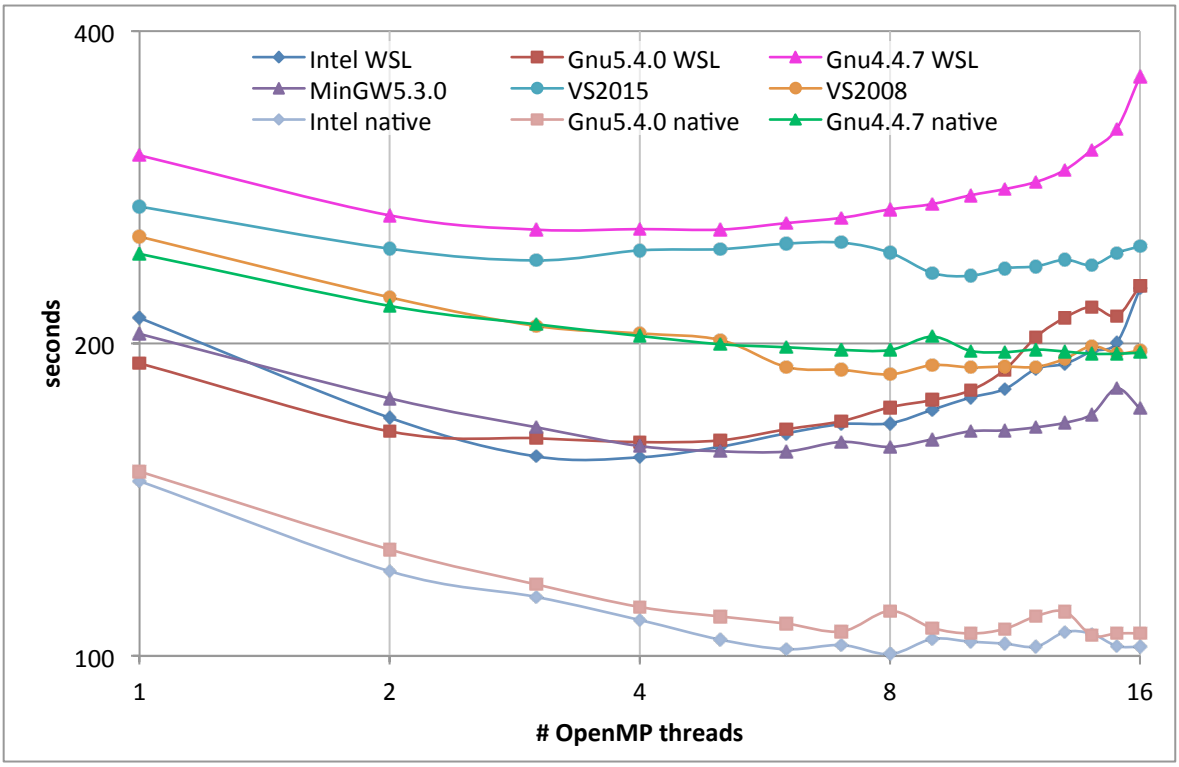


Figure 7: Graphs of execution times in seconds of the MR_AUTO on 1HBZ test job as a function of OpenMP threads for Phaser built with different compilers and running on different patched platforms.

Table 7: Execution times in seconds of the MR_AUTO on 1HBZ test job as a function of OpenMP threads for Phaser built with different compilers and running on different patched platforms.

OpenMP threads	Intel WSL	Gnu5.4.0 WSL	Gnu4.4.7 WSL	MinGW 5.3.0	VS2015	VS2008	Intel native	Gnu5.4.0 native	Gnu4.4.7 native
1	211.59	191.31	303.5	204.16	270.8	253.33	147.42	150.66	244.07
2	169.61	164.57	265.69	177.07	246.75	221.53	120.69	126.64	217.33
3	155.69	162.08	257.39	166.1	240.43	207.91	114.01	117.23	208.7
4	155.37	160.7	257.72	159.3	245.84	204.51	108.31	111.49	203.34
5	159.07	161.33	257.42	157.47	246.5	201.54	103.7	109.17	199.64
6	163.84	165.33	261.16	157.36	249.53	189.85	101.52	107.46	198.32
7	167.24	168.35	264.24	160.78	250.09	188.62	102.46	105.61	197.16
8	167.56	173.61	269.24	159	244.59	186.75	100.42	110.5	197.1
9	172.65	176.41	272.42	161.72	233.81	190.61	103.86	106.41	203.03
10	177.39	180.21	277.77	164.59	232.34	189.65	103.23	105.13	196.76
11	180.76	188.64	281.61	164.92	236.27	190	102.79	106.21	196.18
12	189.04	202.94	286.16	166.06	237.27	189.86	102.12	109.25	197.29
13	191.18	211.81	293.87	167.76	240.83	193.32	105.48	110.33	196.44
14	196.52	216.92	307.18	170.94	237.9	198.83	105.04	104.68	195.47
15	200.3	212.59	321.86	181	244.4	195.66	102.25	105.16	195.47
16	225.8	227.12	361.56	173.2	248.1	196.84	102.04	105.15	195.97

Benchmarks of an MR_FRF run with Phaser on vz170x37_P1 on un-patched platforms

Figure 8 shows the Intel executable on native Linux being the fastest closely followed by the Gnu5.4.0 executable on native Linux. With 13 OpenMP threads it executes the calculation in 29.81 seconds closely followed by the speed of the Gnu5.4.0 executable also on native Linux. This is about three times faster than the builds made with the Microsoft compilers. For a single threaded calculation the speed differences become less dramatic with the Intel or Gnu5.4.0 executable only being about 1.5 times faster than VS2015 executable. The slowest executable for this calculation is the VS2008 executable which only after using more than 8 OpenMP threads begin to catch up where the speed of other executables reach a plateau.

The CPU usage graphs on figure 9 of the executables with 16 OpenMP threads indicate a peculiar behavior. The Intel executable on native Ubuntu never quite attains full usage of the CPU but nevertheless finishes in 31 seconds. Both the Windows executables (MinGW 5.3.0 and VS2015) attain the full CPU usage of 1600% but apparently still run more than twice as slow as the Intel executable on native Ubuntu. On WSL however the Intel executable again never reaches 1600% usage but settles for about 500% and finishes the calculation in a similar time as the MinGW5.3.0 and the VS2015 executable. This points to inefficiencies in the WSL platform compared to native Linux.

Benchmarks of an MR_FRF run with Phaser on vz170x37_P1 on patched platforms

We note in figure 10 that the WSL platform is clearly affected for this calculation; the execution times plateau already above two OpenMP threads.

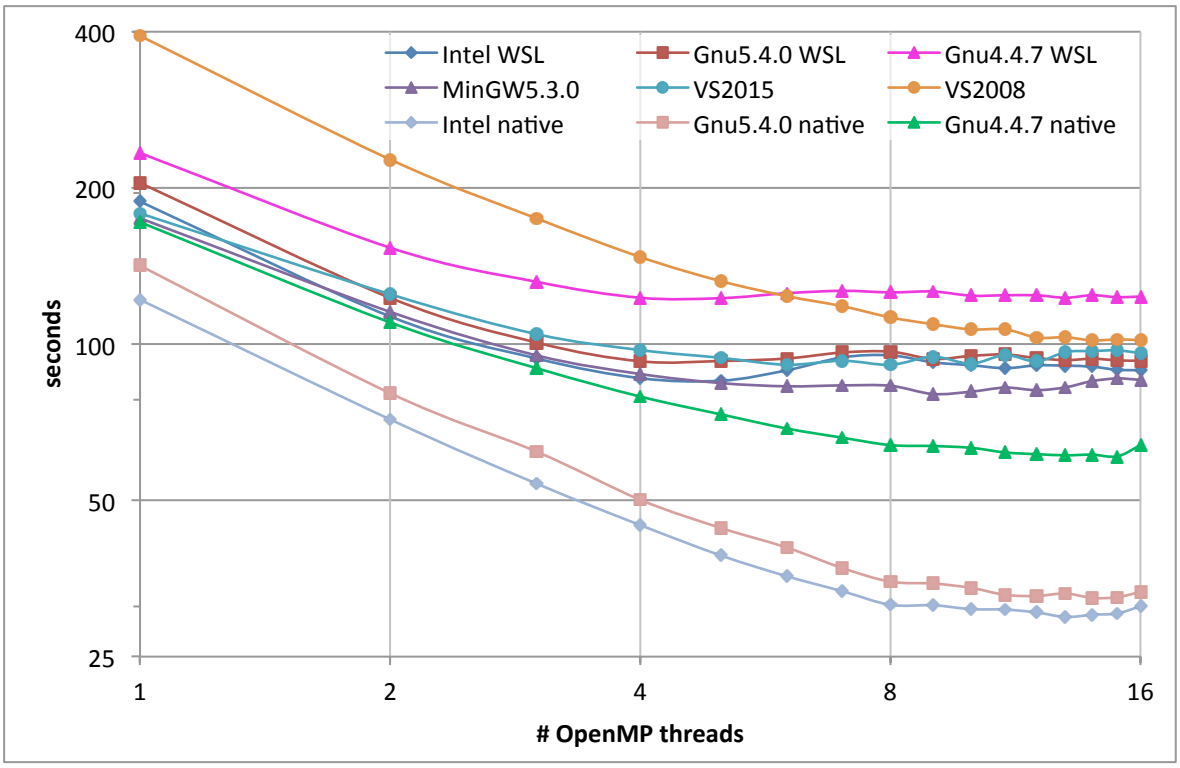


Figure 8: Graphs of execution times in seconds of the MR_FRF on vz170x37_P1 test job as a function of OpenMP threads for Phaser built with different compilers and running on different platforms.

Table 8: Execution times in seconds of the MR_FRF on vz170x37_P1 test job as a function of OpenMP threads for Phaser built with different compilers and running on different un-patched platforms.

OpenMP threads	Intel WSL	Gnu5.4.0 WSL	Gnu4.4.7 WSL	MinGW 5.3.0	VS2015	VS2008	Intel native	Gnu5.4.0 native	Gnu4.4.7 native
1	188.16	204.36	233.31	174.72	178.51	392.27	121.73	141.6	171.71
2	113.01	122.72	153.28	115.33	124.69	226.72	71.58	80.41	110.05
3	93.82	100.68	131.79	95.11	104.5	174.61	53.86	62.08	89.87
4	85.96	92.64	122.79	87.57	97.37	147.11	44.8	50.12	79.22
5	84.98	92.76	122.57	84.11	94.08	132.19	39.16	44.24	73.21
6	88.99	93.78	125.29	82.91	91.24	123.7	35.72	40.57	68.77
7	94.24	96.32	126.62	83.24	92.86	118.29	33.42	37.04	66.02
8	95.13	96.69	125.75	83.1	91.23	112.55	31.46	34.86	63.88
9	92.21	93.59	126.25	80.13	94.51	109.2	31.4	34.58	63.62
10	91.05	94.87	123.93	80.95	91.68	106.73	30.87	33.92	63.12
11	89.85	95.49	124.14	82.43	95.21	106.74	30.8	32.88	61.83
12	90.98	93.95	124.13	81.49	92.59	102.92	30.41	32.72	61.36
13	90.86	93.13	122.79	82.54	96.51	103.04	29.81	33.08	61.05
14	90.45	93.69	124.16	84.85	96.95	101.63	30.1	32.43	61.23
15	89.21	93.04	123.1	85.84	97.22	102.04	30.28	32.53	60.76
16	89.07	92.77	123.49	85.35	96.04	101.82	31.27	33.3	63.83

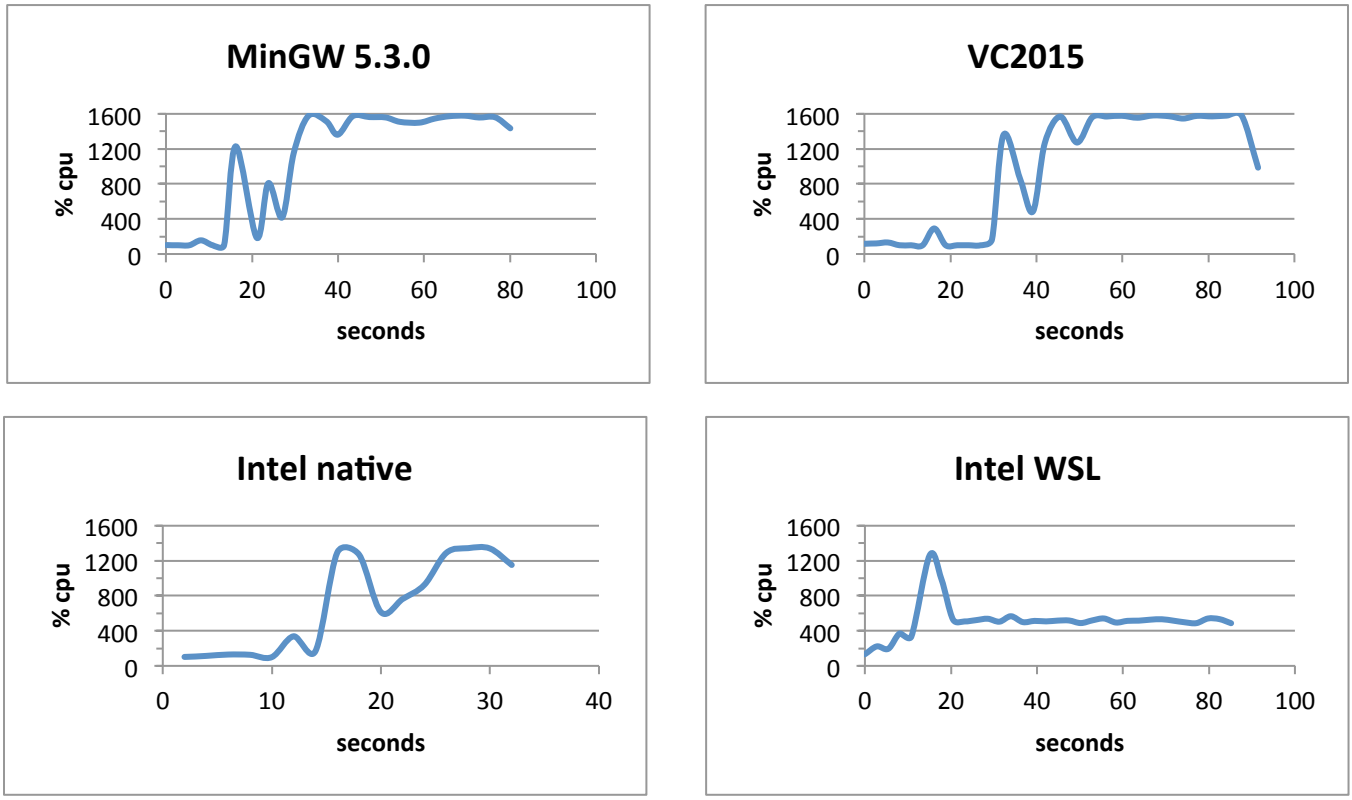


Figure 9: CPU usage graphs of four selected executables running on un-patched platforms with 16 OpenMP threads for the MR_FRF on vz170x37_P1 test job

From table 9 it is noted that the Linux executables running on native Ubuntu suffer from almost no performance hit after the platform has been patched for the Meltdown bug. The Intel executable time increases at most 3.5% when run with 16 threads and the Gnu 5.4.0 executable with at most 2.2%. The execution time of the Windows executables on the other hand increases dramatically with up to 60% in one instance for the VS2015 executable.

Although not shown, the graphs of the execution times for the tests on the patched platforms are quite similar to the graphs in figure 9.

Discussion

Drawing conclusions from the benchmark tests is greatly complicated by the unanticipated dependence on the type of calculations performed. It was thought the relative speed of an executable built with one compiler would remain the same when compared to an executable built with a different compiler regardless of the type of calculations. The test performed here demonstrates that this is not true. Nevertheless there are several points to note from the benchmark tests.

For small programs like the π-calculator the Intel compiler clearly comes out as a favourite. It produces executables that are three or four times as fast as executables built by the Gnu4.4.7 compiler and about twice as fast as executables built by the Gnu5.4.0 compiler. It is also noted that the execution speeds of all executables scale well with the

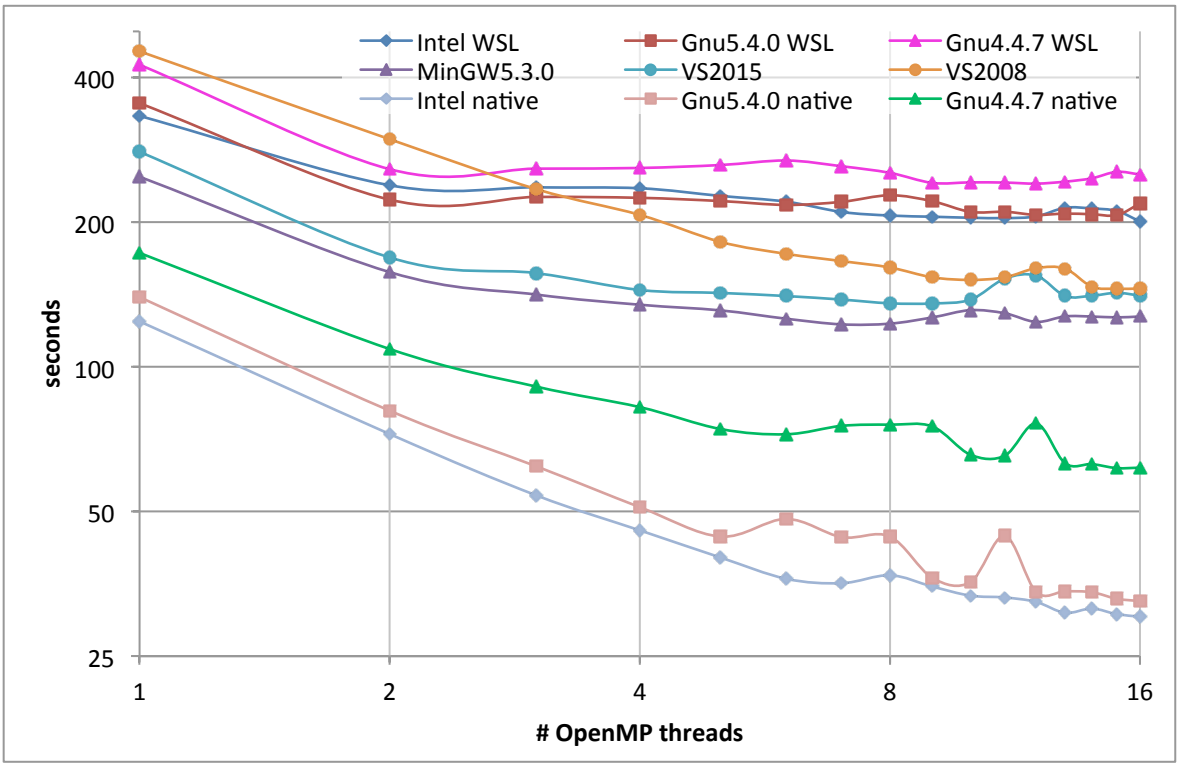


Figure 10: Graphs of execution times in seconds of the MR_FRF on vz170x37_P1 test job as a function of OpenMP threads for Phaser built with different compilers and running on different patched platforms.

Table 9: Execution times in seconds of the MR_FRF on vz170x37_P1 test job as a function of OpenMP threads for Phaser built with different compilers and running on different patched platforms.

OpenMP threads	Intel WSL	Gnu5.4.0 WSL	Gnu4.4.7 WSL	MinGW 5.3.0	VS2015	VS2008	Intel native	Gnu5.4.0 native	Gnu4.4.7 native
1	333.55	353.84	425.33	249.3	280.3	453.65	124.65	139.72	172.81
2	238.89	222.97	258.02	157.55	169.03	297.84	72.52	81.11	108.99
3	236.5	225.86	258.67	141.51	156.8	234.59	54.09	62.17	91.06
4	235.47	224.92	259.52	134.76	144.65	207.29	45.72	51.09	82.51
5	227.01	221.5	263.2	131.17	142.61	182.17	40.15	44.38	74.3
6	220.85	217.42	268.95	126.03	140.5	171.99	36.25	48.28	72.44
7	210.17	220.66	261.65	122.65	138.13	166.1	35.49	44.25	75.42
8	206.61	227.72	253.52	123.08	135.68	161.1	36.86	44.38	75.84
9	205.46	221.28	241.59	126.82	135.59	153.66	34.92	36.4	75.2
10	204.44	210.27	242.22	131.3	138.08	151.81	33.43	35.68	65.68
11	204.09	210.38	241.99	129.36	152.7	153.65	33.14	44.53	65.39
12	205.75	207.29	240.77	124.12	154.97	160.55	32.44	34.02	76.33
13	214.48	208.53	243.04	127.45	140.51	160.23	30.79	34.14	63.04
14	213.74	207.91	246.98	127.35	140.71	146.92	31.43	33.99	62.78
15	211.19	207.01	254.77	126.83	142.67	145.49	30.6	32.94	61.61
16	200.89	218.85	251.54	127.37	140.77	145.64	30.23	32.57	61.72

number of OpenMP threads. This suggests that the small size of the involved functions doing the calculations corresponds to an equally small number of assembler instructions that easily fit into cache memory on the CPU during execution.

For a large program like Phaser the picture is much more mixed; Phaser built with the Gnu 5.4.0 compiler for the calculations presented here, is almost as fast as Phaser built with the Intel compiler. As the Intel compiler uses the same C++ Standard Template Library as the Gnu compiler we speculate that because this library is used by both Phaser executables that could explain the lack of time differences.

Phaser built with the Gnu 4.4.7 compiler is generally the slowest in all tests except for the test of MR_FRF calculations where it remains faster or on par with Phaser executables for Windows regardless of the number of OpenMP threads.

Phaser built with the VS2015 compiler is generally faster than when built with VS2008, often more so when using just one OpenMP thread. The exception is the MR_AUTO on 1HBZ calculation where Phaser built with the VS2008 compiler happens to be somewhat faster than when built with the VS2015 compiler.

Back in 2008 benchmark tests using Phaser in MR_FRF mode were done on the vz170x37_P1 data set identical to the ones presented here. Although the tests back then were done on slower hardware they ran faster because the current Phaser version is doing more preliminary calculations before it begins the actual MR_FRF calculation. The conclusion back then was that a VS2005 executable on Windows XP was faster than g++ 4.1 executable on Fedora 7 when running Phaser with one OpenMP thread (329 seconds against 431 seconds). In light of our current results and assuming that the VS2008 compiler produces executables with the same speed or better than the VS2005 compiler this suggests that Gnu compilers have improved more quickly over the past ten years than have Microsoft compilers.

As for Phaser built with the MinGW 5.4.0 compiler its execution times are faster than when built with the Microsoft compilers except for the MR_AUTO on 1ioM calculation. In other words it is a competitive alternative to Microsoft compilers when building for the Windows platform.

Regarding OpenMP the tests carried out here indicates that OpenMP threading on native Linux scales better than on Windows 10. This is likely to be specific to the operating systems rather than differences between Windows compilers and Linux compilers in how they implement OpenMP threading.

Conclusion

From these tests one can conclude that it is worth using the more recent Gnu5.4.0 compiler when building for Linux. The Gnu4.4.7 compiler that is currently used produces executables often being half the speed of executables built with the Gnu5.4.0 compiler currently available in Ubuntu 16.04.

The striking advantage in the case of the Intel compiler for Linux builds of the small π-program does not hold for Phaser and presumable other large programs which use the C++ Standard Template Library. Phaser run with about the same speed when built with the Gnu5.4.0 compiler.

The Meltdown bug

Until the arrival of the patches for the meltdown bug deciding between platforms Linux and WSL the picture was clear. Running calculations on native Linux can be expected to run faster if the program is large and to run with the same speed on WSL if the program is small. Between Linux executables and win32 executables however there seemed to be no clear winner in terms of speed. Some calculations are faster on Windows and others are faster on Linux. Hence other priorities may determine the choice of platform.

However, the patches for the meltdown bug changes this conclusion in favour of native Linux where the execution times of Phaser is practically unaffected by the new patches. In contrast the Windows 10 platform is affected far more severely. It goes beyond the scope of this article to speculate what the exact source of these inefficiencies is. Moreover, it can be expected that the patches for the operating systems in time will improve and reduce the apparent inefficiencies currently experienced.

APPENDIX

Patches

Verification that the BIOS and the OS patches have been properly installed was done with the scripts available on https://github.com/speed47/spectre-meltdown-checker for Linux and https://support.microsoft.com/en-gb/help/4073119/protect-against-speculative-execution-side-channel-vulnerabilities-in for Windows.

π-program source

Program code is shown in figure 11.

Details of Phaser source versions

We used Phaser available from https://git.uis.cam.ac.uk/x/cimr-phaser/phaser.git (git hash: 731557) compiled with the CCTBX available from https://github.com/cctbx/cctbx_project (git hash: cb94e1b and 1f0b0593) as present on November 21, 2017. These versions are present in the Phenix-1.13-rc1-2965 build and the CCTBX-installer-dev-1230 build (no differences in C++ sources between revisions cb94e1b and 1f0b0593).

Keyword Scripts

In the keyword scripts below that were run from a Bash shell $ncpu is the parameter that specifies the number of OpenMP threads available throughout the execution. Figures 12–14 are mentioned in the main text.

```
// OpenMPtest.cpp an example of multithreading using OpenMP
// Compiler command line for
// Linux g++: "g++ -fopenmp -static -g -O3 OpenMPtest.cpp -oOpenMPtest.gnu.X"
// Linux Intel: "icpc -qopenmp -fast -static -debug all OpenMPtest.cpp -oOpenMPtest.intel.X"
// Windows Visual Studio: "cl OpenMPtest.cpp /Ox /fp:fast /EHsc /openmp /FeOpenMPtest.exe"

#define _USE_MATH_DEFINES
#include <omp.h>
#include <cmath>
#include <iostream>
#include <iomanip>

using namespace std;

class PiCalculator
{
  long lmax; // Gaussian integration slices
  long kmax; // terms in Pi series
  double gaussint;
  double D, b, elapsed_time;
  double GetSlowPi(long m);
public:
  PiCalculator(int nthreads);
  ~PiCalculator();
  void GaussIntegrate();
  double GetPiResult() { return gaussint; }
  double GetElapsedTime() { return elapsed_time; }
};

PiCalculator::PiCalculator(int nthreads)
{
  lmax = 20000; // Gaussian integration slices
  kmax = 30000; // terms in Pi series
  gaussint = 0.0;
  D = 20.0, b = 3.0;
  elapsed_time = 0.0;

  cout << "This program is the slow Pi calculator, a small OpenMP test for your system." << endl;
  const int nprocs = omp_get_num_procs();
  if (nprocs < nthreads)
  {
    cout << "\nWarning!\nRunning this calculation with more threads than processors makes no sense.\n"
      << "Unless you want to stress test your system don't use more than " << nprocs << " threads." << endl;
    nthreads = nprocs;
  }
  omp_set_num_threads(nthreads);
}

PiCalculator::~PiCalculator()
{
  cout << "\nThanks for using the slow Pi calculator." << endl;
}

double PiCalculator::GetSlowPi(long m)
{// Compute Pi = 3*sqrt(3)*Sum_{k=1}(-1)^k/(3k+1) - log(2)*sqrt(3)
 // hence converges fairly slowly
  double pisum = 0.0;
  for (long k = 0; k<(kmax + m); k++)
  {
    double frac, div = 3.0*k + 1.0;
    if (k % 2 == 0) // i.e. k is even
      frac = 1.0 / div;
    else
      frac = -1.0 / div;
    pisum += 3.0*sqrt(3.0)*frac;
  }
  pisum -= log(2.0)*sqrt(3.0);
  return pisum;
}
```

Figure 11: Source code for the π-program

```
void PiCalculator::GaussIntegrate()
{
  double start, finish;
  start = omp_get_wtime();
  double sum = 0.0;
  const double dx = 2.0*D / lmax;
#pragma omp parallel
  {
#pragma omp single
    cout << "Number of threads is: " << omp_get_num_threads() << endl;
#pragma omp for reduction(+ : sum) // add all openmp for-loop results into the variable gaussint
    for (long m = 0; m< lmax; m++)
    {
      // integrate a Gaussian multiplied with Pi
      double x = m*dx - D;
      sum += GetSlowPi(m)
        *1.0 / sqrt(2.0*b*GetSlowPi(m + 7))*exp((cos(GetSlowPi(m + 13))*x*x) / (2.0*b)) * dx;
    }
  }
  // end #pragma omp parallel
  finish = omp_get_wtime();
  gaussint = sum;
  elapsed_time = (finish - start);
}

int main()
{
  int nthreads = 1;
  cout << "Enter number of OpenMP threads to use for this calculation: ";
  cin >> nthreads;
  PiCalculator myPi(nthreads);
  cout << setprecision(18);
  myPi.GaussIntegrate();
  cout << "\rtime= " << myPi.GetElapsedTime() << " sec, gausssum = " << myPi.GetPiResult() << endl;
  return 0;
}
```

Figure 11: Source code for the π-program (cont.)

```
HKLIN ../1ioM.mtz
LABIN F = FOBS_X  SIGF = SIGFOBS_X
COMPOSITION PROTEIN SEQUENCE ../1ioM_ChainA.seq NUM 1
MODE MR_AUTO
ENSEMBLE MR_2R26_A PDB ../sculpt_2R26_A_singlechain.pdb  IDENTITY 34.218
SEARCH ENSEMBLE MR_2R26_A NUM 1
JOBS $ncpu
```

Figure 12: Keyword script for running Phaser with the data set with the PDB code 1ioM. The model file, `sculpt_2R26_A_singlechain.pdb`, is derived from chain A of the structure with PDB code 2R26 and has been trimmed with the program Sculptor.

```
HKLIN ../1hbz.mtz
LABIN F = FOBS_X  SIGF = SIGFOBS_X
COMPOSITION PROTEIN SEQUENCE ../1HBZ_ChainA.seq NUM 1
MODE MR_AUTO
ENSEMBLE MR_1A4E_A PDB ../sculpt_1A4E_A.pdb  IDENTITY 41.393
SEARCH ENSEMBLE MR_1A4E_A NUM 1
JOBS $ncpu
```

Figure 13: Keyword script for running Phaser with the data set with the PDB code. The model file, `sculpt_1A4E_A.pdb`, is derived from chain A of the structure with the PDB code 1HZB and has been trimmed with the program Sculptor.

```
HKLIN ../vz170x37_P1.mtz
MODE MR_FRF
LABIN F=F_vz SIGF=SIGF_vz
ENSEMBLE octamer PDBFILE ../octamer.pdb IDENT 1.0
COMPOSITION PROTEIN MW 18000 NUMBER 32
SEARCH ENSEMBLE octamer NUM 1
MACANO PROTOCOL OFF
MACTNCS PROTOCOL OFF
RESOLUTION HIGH 0.5
RESOLUTION AUTO HIGH 0.5
JOBS $ncpu
```

Figure 14: Keyword script for running Phaser with the VZ170x37_P1 data set. This is in-house data kindly provided by Andrea Mattevi. The model file, octamer.pdb, consists of 8 NCS copies of chain A of the hexameric structure with the PDB code 2W5E sited in the corners of a twisted cube.