Adapting crystallographic programs to multiple-CPU computers

ECACOMSIG Computing School 2016

George M. Sheldrick

http://shelx.uni-ac.gwdg.de/SHELX/

Parallel processing and SHELX

Most modern computers have multiple CPUs, so it seems appropriate to use these to speed up crystallographic calculations.

I am very grateful to Kay Diederichs, who succeeded in making my crystal structure refinement program SHELXL parallel and so appreciably faster. Although I have made many changes to SHELXL since then, the method of parallelization devised by Kay has been retained.

Diederichs, K. *J. Appl. Cryst.* (2000) **33**, 1154-1161. *Computing in macromolecular crystallography using a parallel architecture.*

Based on this experience, I was also able to adapt the programs SHELXD, SHELXT and ANODE to parallel operation. A parallel version of SHELXE is under development.

Threads and Hyperthreading

Parallel processing involves ***threads*** that operate simultaneously. In the simplest case the number of threads is equal to the number of CPUs. One may start more threads than there are CPUs but there is no advantage.

The ***hyperthreading*** feature of many Intel CPUs effectively doubles the number of available CPUs, but unfortunately not for floating-point arithmetic. In some memory-intensive cases it may be better not to use it.

Complications

The performance of an OpenMP program is influenced by a number of factors, some of which can make benchmarking difficult.

1. Amdahl's law. As the number of threads increases, the non-parallel part of the code has a larger influence and sets an asymptotic limit to the possible speedup.

2. Some processors automatically increase their clock rates when not many threads are active. The limiting factor is the chip temperature.

3. Hyperthreading.

4. Memory access.

Amdahl's law (1967)

Amdahl's law describes the influence of the part of the program that does not operate in parallel. This includes various overheads necessary to set up the parallel processing as well as procedures that are slow but inherently difficult to make parallel, e.g. matrix inversion in a full-matrix crystal structure refinement.

If the scalar part takes ***s*** seconds and the parallel part ***p/n*** seconds, where ***n*** is the number of CPUs, the speedup factor is:

F = (time on a single CPU computer) / (time on a computer with ***n*** CPUs)
= ***(s + p) / (s + p/n)***

This approaches ***1+(p/s)*** asymptotically as the number of CPUs increases. Thus a program that is 75% parallel can never be as much as four times as fast, however many CPUs are employed!

Cache memory

Each CPU has its own cache. Accessing data in cache is much faster than RAM memory access. By dividing the data between the threads it may be possible to avoid ‘cache misses’ and so achieve ***super-scalar*** performance where the speedup relative to single-CPU operation is greater than the number of CPUs. This violates Amdahl’s law.

The Intel Sandy bridge and Ivy bridge I7 processors (introduced in 2012) have the following cache structure:

Level 1: 32KiB for data and 32 KiB for instructions per core

Level 2: 256 KiB per core

Level 3: 6 to 8 MiB

(1MiB = 2^{20} = 1048576 bytes; 1MB = 1000000 bytes)

Crystallographic examples

The programs SHELXD and SHELXL will be used to illustrate different approaches to achieving good parallel performance.

SHELXD makes many attempts to solve the phase problem starting each from random phases or atom positions, usually filtered for consistency with the Patterson function. This requires a ***critical*** section at the beginning of each attempt so that each starts from different phases. After a given number of cycles of iterative phase refinement, the program checks whether the solution is the best so far. If it is, a ***critical*** section is used to save the phases.

SHELXL divides the reflection data into blocks that are processed in parallel. Each block is designed to fit into cache. Special action is needed to combine the matrices generated by the different threads.

OpenMP – very simplified introduction

OpenMP provides many useful facilities to generate parallel code, but here we will stick to essentials.

The first question is usually to find out how many CPUs are available. In FORTRAN this is performed as follows:

```
C$    INTEGER omp_get_max_threads

      N=1
C$    N=omp_get_max_threads()
```

Note that if the OpenMP compiler flag is not set, the lines beginning with C$ are treated as comments and so this sets the number of threads to 1.

The value returned can be changed by setting the environment variable OMP_NUM_THREADS The default includes hyperthreads.

OpenMP – very simplified introduction

Often the parallel part takes the form of a subroutine (called PSUB here). We need to specify which data are SHARED and which are PRIVATE to a thread. SHARED date may be used by all copies of the subroutine but may only be changed when that subroutine has exclusive access (see ***critical*** later). Variables defined locally in a subroutine are PRIVATE.

```
C$ omp parallel do default (SHARED)
      DO 1 I=1,N
      CALL PSUB(I,A,B,C)
  1   CONTINUE
C$ omp end parallel do
```

In this case PSUB probably uses a ***critical*** section to save the best results in the shared arrays.

OpenMP – very simplified introduction

```
C$ omp parallel do default (SHARED)
      DO 1 I=1,N
      CALL PSUB(I,A,B,C)
  1   CONTINUE
C$ omp end parallel do
```

This construction has one serious disadvantage: it starts all N threads at the same time. If subroutine PSUB uses a lot of (private) memory for arrays, this could lead to cache misses. It is better to restrict N to the number of real CPUs and repeat the loop as required.

The SHELX programs do this where necessary, e.g. for the P1 structure solution stage in SHELXT, where N defaults to 4 but can be changed by using the –t command line switch. Often 4 tries solve the structure and the program can go onto the next stage (using the phases to determine the space group, which involves a parallel loop over all possible space groups).

OpenMP – very simplified introduction

A ***critical*** section looks like this:

```
C$omp critical
      some FORTRAN code
C$omp end critical
```

This forces all the other threads to wait if they encounter a ***critical*** section that is already being executed by a thread. ***Critical*** sections involve appreciable delays and overheads and should be used sparingly.

For example SHELXD can test whether it has found the best solution so far without going ***critical***, but if it is the best, then it uses a ***critical*** section to save it. A nice side-effect is that the program accelerates as it is running, because the best solution so far is found less often!

OpenMP – very simplified introduction

SHELXL uses an alternative, possibly FORTRAN-specific, way to pass data between the scalar and parallel parts of the code. A number of SHARED arrays are defined as two-dimensional in the scalar part and as one-dimensional in the parallel subroutines, similar to the following:

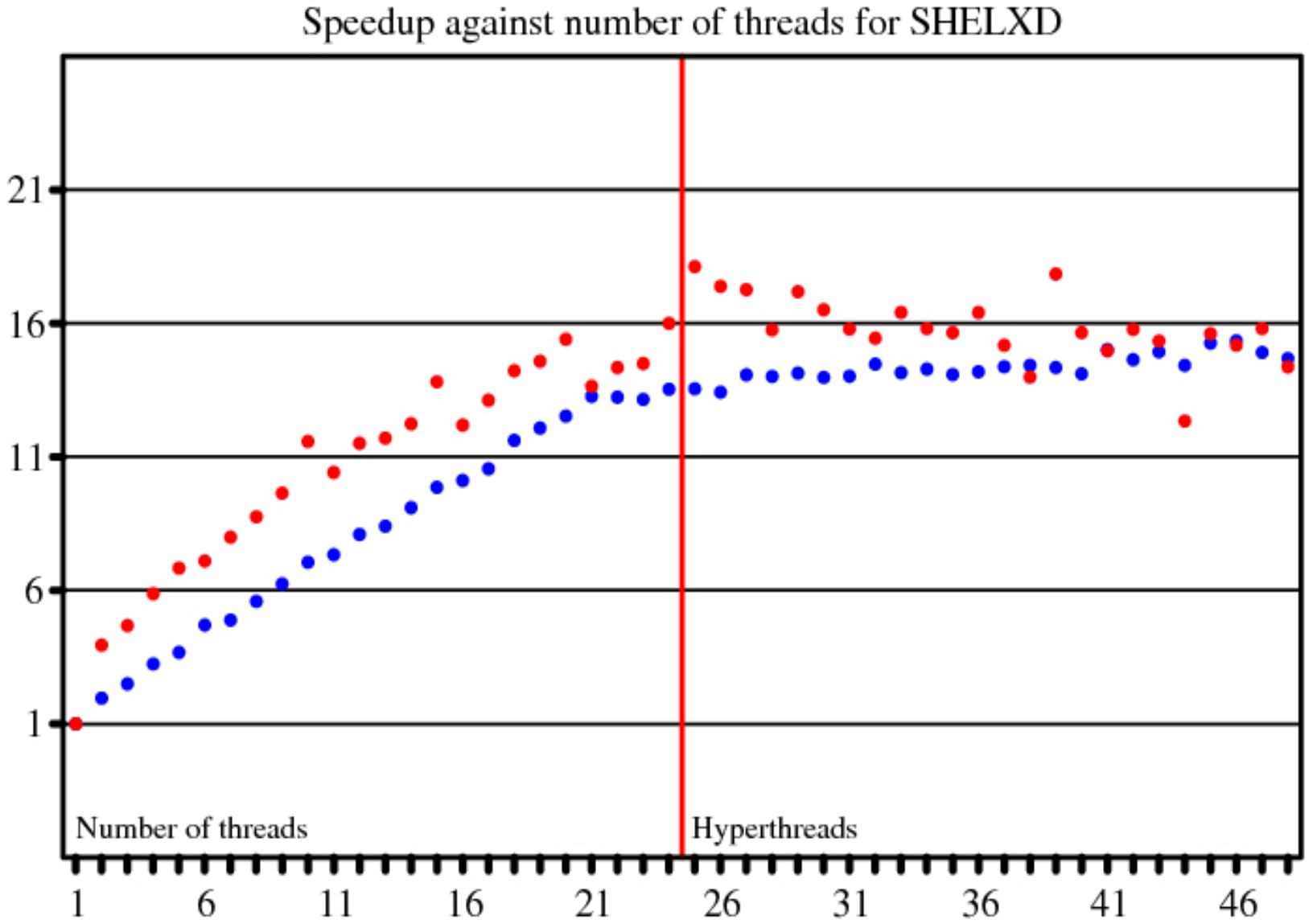
```
      REAL A(100,N)

C$ omp parallel do default (SHARED)
      DO 1 I=1,N
      CALL PSUB(A(1,I))
  1   CONTINUE
C$ omp end parallel do

      SUBROUTINE PSUB(A)
      REAL A(100)
```

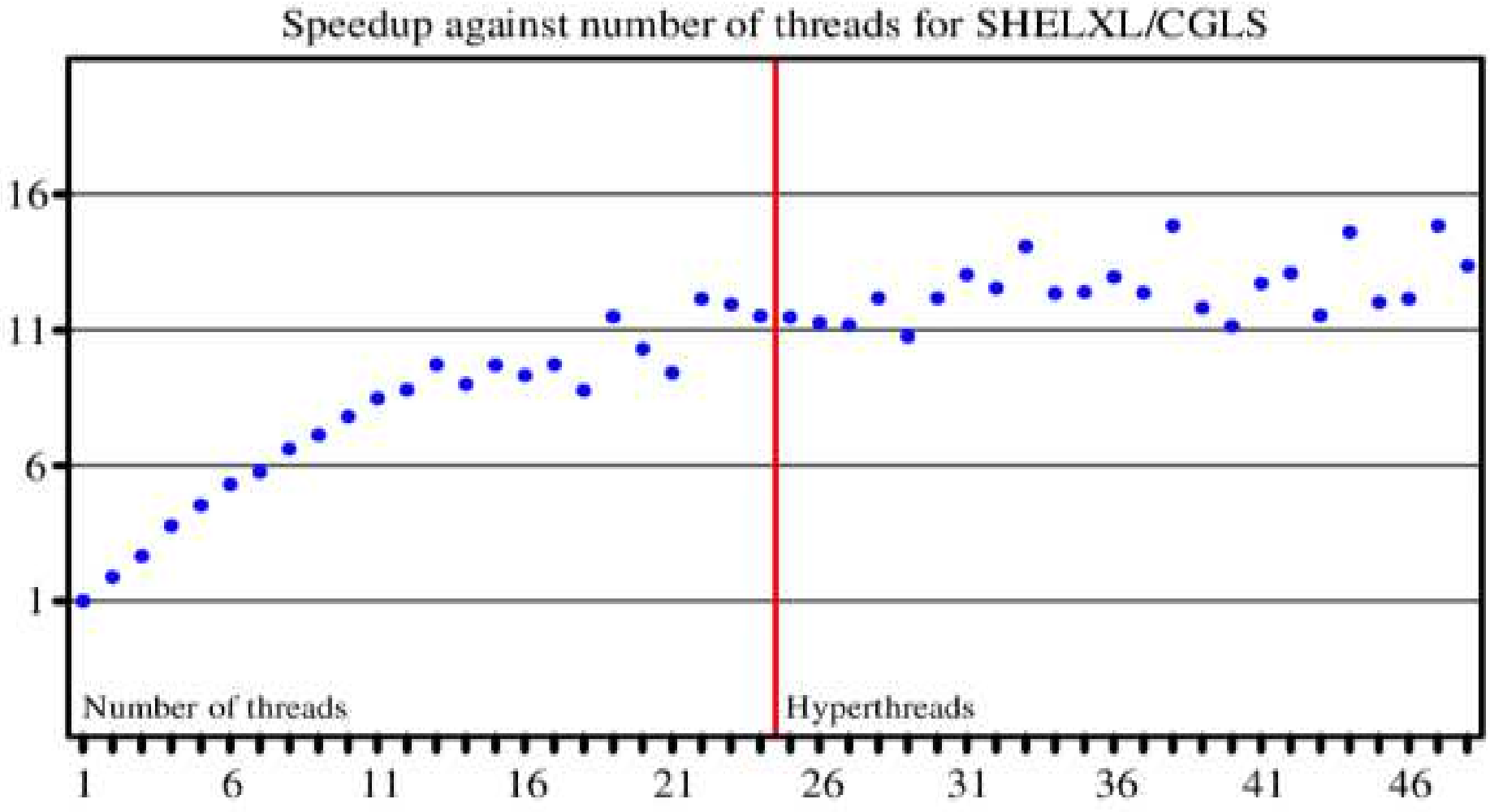
No critical section is required because each thread uses a different part of array A.

Typical SHELXD (sub)structure solution



Speedup = (time for single thread) / (time for N threads)

Typical SHELXL CGLS refinement



Speedup = (time for single thread) / (time for N threads).

Conclusions

The actual speedup achieved is very problem dependent. Both algorithms presented here perform reasonably well, but the hyperthreading adds little.

In particular full-matrix crystal structure refinement (L.S.) tends to give smaller speedups (of the order of two to six) than conjugate gradient (CGLS) refinement. This is because of the appreciable scalar part (matrix inversion) and the high memory requirements that result in frequent cache misses. Every element of the large (triangular) least-squares matrix needs to be added to for each reflection.

Acknowledgements

SHELX is available free for academic use and may be obtained via the SHELX homepage that has moved to ***shelx.uni-goettingen.de***

I am particularly grateful to Kay Diederichs for introducing me to parallel programming and for his parallel version of SHELXL.