First publ. in: Journal of Applied Crystallography 33 (2000), pp. 1154-1161

# Computing in macromolecular crystallography using a parallel architecture

**Kay Diederichs**

Naturwissenschaftliche Sektion, Fachbereich Biologie, Universität Konstanz, M656, 78457 Konstanz, Germany. Correspondence e-mail: kay.diederichs@uni-konstanz.de

Despite advances in computer technology, computing in macromolecular crystallography keeps pace in its demand for CPU power. Improvements in CPU speed, together with advances in computing methods that depend on it, often translate into the possibility to solve structures that would otherwise require additional experiments. Programs for data reduction, molecular-replacement programs employing multidimensional searches on a grid in real, Patterson or reciprocal space, and phasing and refinement programs, currently have, among others, the highest requirements for CPU power. For these and other programs, speed-up of calculations as a result of parallel execution on multiprocessor computers is possible. This paper outlines the use of the OpenMP programming interface and reports its successful application for parallelization of *ESSENS* [Kleywegt & Jones (1997). *Acta Cryst.* D**53**, 179–185] and *SHELXL* [Schneider & Sheldrick (1997). *Methods Enzymol.* **277**, 319–343]. Parallel computing, which is possible as a result of the inherent parallelism of crystallographic algorithms, extends the range of problems in macromolecular crystallography that programs can be applied to and can significantly reduce the time required for progressing from a data set to a refined model.

## 1. Introduction

Computer speed has increased by more than a factor of ten over the past decade. On the other hand, in recent years, the methods for crystallization of biologically interesting macromolecules or their complexes, with thousands of atoms, have also improved. This leads to a growing number of projects in structural biology and a tendency towards larger structures can be seen. Increases in the total number of beamlines at synchrotron sites as well as their improved photon flux and brilliance have made large high-resolution data sets available for a growing proportion of projects. Furthermore, new CPU-demanding solutions to crystallographic problems have been programmed and algorithms from other or related fields of science are being applied to macromolecular data. For these reasons, crystallographic computing has kept pace in its demand for ever increasing CPU power.

Among the programs that are quite demanding in terms of raw CPU power are data reduction programs like *XDS* (Kabsch, 1988), *MOSFLM* (Leslie, 1992) and *DENZO* (Otwinowski & Minor, 1997), which, at third-generation synchrotron sites, can hardly reduce the data as fast as they are measured. The heavy-atom phasing package *SOLVE* (Terwilliger & Berendzen, 1999) and the maximum-entropy phasing program *SHARP* (De la Fortelle & Bricogne, 1997), genetic algorithms for molecular replacement, such as implemented in *EPMR* (Kissinger *et al.*, 1999), real-space molecular-replacement programs like *ESSENS* (Kleywegt & Jones, 1997), and refinement programs like *X-PLOR* (Brünger & Rice, 1997), *CNS* (Brünger *et al.*, 1998) and *SHELXL* (Schneider & Sheldrick, 1997), depend on ample computational resources. Practical use of some of these programs has only been made possible by improvements in CPU speed over the past decade and it can be expected that future improvements in CPU speed will allow further methodological advances.

Crystallographic computing deals with large numbers of items, such as atoms, reflections, model orientations or detector pixels, which differ in their individual attributes or values, but not in the way they are treated by the crystallographic algorithm. This means that, inherently, crystallographic algorithms have a high degree of parallelism.

The purpose of this work is twofold: to demonstrate, taking *ESSENS* and *SHELXL* as examples, that it is worthwhile to exploit this parallelism, and to outline the use of the programming interface used for this purpose.

## 2. Parallelization

### 2.1. Methods for parallel computing

The basic idea in exploiting the inherent parallelism of an algorithm is to divide its most time-consuming part into parallel processes running concurrently on a number of CPUs.

These processes should interact with one another as little as possible, as any interaction requires them to be synchronized. Obviously, to result in wall-clock time savings, the overhead associated with setting up the processes, distributing the work between them and synchronizing them must be significantly less than the time saved by dividing the work.

An algorithm with frequent need for synchronization of parallel processes is termed 'fine-grained'. As a rule, a specific part of an algorithm can be considered fine-grained if the time spent for useful calculation between synchronization events is no more than about one order of magnitude longer than the overhead of synchronization. In practice, parallelization at the level of an outer loop often produces coarse-grain parallelism if parallelization of its inner loop would produce fine-gain parallelism. If an algorithm is too fine-grained, parallelization achieves no reduction in overall computing time. Therefore, to minimize the time wasted as an overhead, a goal of parallel programming is to produce coarse-grained algorithms, which, by definition, can be efficiently executed in parallel.

Clearly, the overhead itself depends on the mechanisms used for synchronization. Two basic ways of parallel execution of processes have been developed over the years. First, there are (among others) the PVM (Parallel Virtual Machine, see http://www.epm.ornl.gov/pvm/pvm_home.html) and MPI (Message Passing Interface, see http://www.mcs.anl.gov/mpi) concepts, which are designed to operate between computers in a heterogeneous network with possibly different operating systems and CPU speeds. Both packages have mechanisms to distribute, set up and synchronize processes on different computers. However, start-up and synchronization of processes across a network is a costly operation, and is only going to result in wall-clock time savings if the bulk of independent work in each process is substantial.

The second method can efficiently deal with algorithms requiring smaller chunks of independent work as its overhead is significantly lower. It requires a multiprocessor (MP) shared-memory computer equipped with a compiler that accepts the OpenMP extensions to the C/C++, Fortran 77 and Fortran 90/95 computer languages. The OpenMP application programming interface (API) has been developed in the past few years and is the *de facto* standard for MP programming. It is the successor of an ANSI effort for standardization of vendor-specific extensions that were primarily developed for the Fortran programming language. The OpenMP set of extensions has been adopted by almost all major suppliers of compilers and OpenMP compilers are available for most hardware platforms, such as those produced by SGI/Cray, HP, Compaq/Digital, Sun, IBM and Intel (for a complete list, see http://www.openmp.org).

Fortran is, for historical and efficiency reasons, the programming language used for most crystallographic programs. In its latest incarnations, Fortran 90 and Fortran 95, most shortcomings of older Fortran dialects have been overcome without sacrificing compatibility with existing code. Especially, those new language elements that allow definition and handling of matrices lend themselves to efficient parallelization, and vendors of MP computers often supply optimized parallel libraries and routines. However, existing programs are mostly written in older Fortran dialects and array handling is only one aspect of crystallographic computing. It is therefore important to realise the potential of OpenMP for parallelization of crystallographic code.

### 2.2. The OpenMP application programming interface

OpenMP is, by design, a programming interface with little overhead. It offers a small but efficient set of language constructs (see Table 1) that supports both fine- and coarse-grained parallelism tasks. Whenever two or more tasks of a program that are normally processed serially are logically independent from each other, an OpenMP construct allows these sections to be distributed to concurrent processes (called 'threads').

While OpenMP compilers are often supplied with the standard compilers of the operating system of MP computers, automatic parallelizers that generate OpenMP from standard C or Fortran are not as widely available. Automatic parallelizers can analyse and insert OpenMP extensions into existing code, thereby hiding the OpenMP API from the programmer. In simple cases, automatic parallelization might be the method of choice, as it does not require any parallelization effort or expertise on the part of the programmer. On the other hand, an automatic analysis of the source code will often only be able to parallelize small sections of code, as it is difficult to prove independence of code sections by automatic reasoning. Thus, automatic parallelization generally produces a parallelism that is more finely grained compared to what may be achieved from insight into an algorithm.

Manual parallelization of existing source code with OpenMP directives requires a basic understanding of its programming language to reach fine-grained parallelism, for example, at the level of an inner loop. In order to achieve coarse-grained parallelism, normally in addition a basic understanding of the underlying algorithm is required, because parallelization at a higher level requires an insight into the dependencies between calculations. Sometimes the original code needs to be slightly changed to uncouple those sections that are potentially parallelizable, *e.g.* by introducing additional temporary variables. This type of change usually incurs very little or no overhead; therefore, the OpenMP-adapted code can still be efficiently executed on a uniprocessor machine.

In most cases, OpenMP directives (Table 1) simply can be inserted into the code. For a compiler not aware of OpenMP directives (or with their recognition switched off), the code appears unchanged, as all OpenMP directives start in column 1 with the pseudo-comment '`C$OMP`' (or '`!$OMP`' or '`*$OMP`'), and those lines that should only be executed by a parallel program (conditional compilation) start with the pseudo-comment '`C$` (or '`!$`' or '`*$`').

The most basic target of parallelization is the `DO` loop. As the most simple example,

**Table 1**
Important OpenMP constructs.

For complete definitions, explanations and examples see http://www.openmp.org. Copyright notice: OpenMP is a trademark of the OpenMP Architecture Review Board. This table has been derived from the OpenMP Language Application Program Interface Specification.

| Directive | Optional clauses† | Meaning |
|---|---|---|
| Parallel region construct | | |
| `C$OMP PARALLEL`<br>Fortran code with work-sharing constructs<br>`C$OMP END PARALLEL` | `PRIVATE` (*list*)<br>`SHARED` (*list*)<br>`DEFAULT` (`PRIVATE`\|`SHARED`\|`NONE`)<br>`FIRSTPRIVATE` (*list*)<br>`REDUCTION` (*operator*\|*intrinsic*:*list*)<br>`IF` (scalar logical expression)<br>`COPYIN` (*list*) | Start a parallel region. The work-sharing constructs may also be in subroutines called from within the parallel region |
| Work-sharing constructs within a parallel region | | |
| `C$OMP DO`<br>Fortran `DO` loop<br>`C$OMP END DO` | `PRIVATE` (*list*)<br>`FIRSTPRIVATE` (*list*)<br>`LASTPRIVATE` (*list*)<br>`REDUCTION` (*operator*\|*intrinsic*:*list*)<br>`SCHEDULE` (*type*[,*chunk*])<br>`ORDERED`<br>`NOWAIT` | The Fortran `DO` loop should be performed in parallel. The statement after `C$OMP DO` will only be executed once all threads have reached it (unless `NOWAIT` is specified) |
| `C$OMP SECTIONS`<br>`C$OMP SECTION`<br>Block of Fortran statements‡ (more `C$OMP SECTION` directives could follow)<br>`C$OMP END SECTIONS` | `PRIVATE` (*list*)<br>`FIRSTPRIVATE` (*list*)<br>`LASTPRIVATE` (*list*)<br>`REDUCTION` (*list*)<br>`NOWAIT` | The following sections of code should be performed in parallel. The meaning of `NOWAIT` is as for the `C$OMP DO` directive |
| `C$OMP SINGLE`<br>Block of Fortran statements‡<br>`C$OMP END SINGLE` | `PRIVATE` (*list*)<br>`FIRSTPRIVATE` (*list*)<br>`NOWAIT` | The following block is to be executed by one thread only. The other threads wait at the statement following the end single directive unless `NOWAIT` is specified |
| Combined work-sharing constructs | | |
| `C$OMP PARALLEL DO`<br>Fortran `DO` loop<br>`C$OMP END PARALLEL` | See the clauses of `C$OMP PARALLEL` and `C$OMP DO` | Shortcut for combining `C$OMP PARALLEL` with a single `C$OMP DO` |
| `C$OMP PARALLEL SECTIONS`<br>`C$OMP SECTION`<br>Block of Fortran statements‡ (more `C$OMP SECTION` directives could follow)<br>`C$OMP END PARALLEL SECTIONS` | See the clauses of `C$OMP PARALLEL` and `C$OMP SECTIONS` | Shortcut for combining `C$OMP PARALLEL` with a single `C$OMP SECTIONS` |
| Synchronization constructs | | |
| `C$OMP MASTER`<br>Block of Fortran statements‡<br>`C$OMP END MASTER` | | The block is only executed by the master of the team of threads |
| `C$OMP CRITICAL` (*name*)<br>Block of Fortran statements‡<br>`C$OMP END CRITICAL` (*name*) | | The block is only executed by one thread at a time. A thread waits for execution until no other threads execute a section with the same name. The optional *name* is a global entity of the program (this implies that distant sections of code can be synchronized) |
| `C$OMP BARRIER` | | Threads wait at this directive until all other threads have reached it, too |
| `C$OMP ATOMIC`<br>Fortran assignment statement | | The assignment statement which immediately follows is only executed by one thread at a time. Only certain types of assignment statements are allowed here |
| `C$OMP ORDERED`<br>Block of Fortran statements‡<br>`C$OMP END ORDERED` | | The block is executed in the same order as it would be executed during sequential processing. Important *e.g.* for I/O |

**Table 1 (continued)**

| Directive | Optional clauses† | Meaning |
|---|---|---|
| Data environment construct | | |
| C$OMP THREADPRIVATE (*list*) | | The *list* contains names of COMMON blocks containing thread-private copies of variables that are to be filled in by use of the COPYIN clause on the C$OMP PARALLEL statement |
| Run-time library routines and functions used to control and query the parallel execution environment | | |
| SUBROUTINE OMP_SET_NUM_THREADS() | | Set the number of threads |
| FUNCTION OMP_GET_MAX_THREADS() | | Get the maximum number of threads |
| FUNCTION OMP_GET_THREAD_NUM() | | Within a parallel region: get the number of the executing thread [between 0 and OMP_GET_MAX_THREADS() − 1] |
| FUNCTION OMP_IN_PARALLEL() | | Return TRUE if called from within a parallel region, FALSE otherwise |
| Low-level locking routines | | |
| SUBROUTINE OMP_INIT_LOCK()<br>SUBROUTINE OMP_DESTROY_LOCK()<br>SUBROUTINE OMP_SET_LOCK()<br>SUBROUTINE OMP_UNSET_LOCK()<br>FUNCTION OMP_TEST_LOCK() | | Subroutines to test, set and un-set specific locks |

† The purpose of the most important clauses is explained in the text. ‡ No GOTO into or out of the block is allowed.

```
C$OMP PARALLEL DO SHARED(a, b) PRIVATE(i)
      do 100 i = 1, n
        a(i) = a(i) + b(i)
  100 continue
C$OMP END PARALLEL DO
```

would distribute the work of adding array b to array a to all threads. The number of threads can be set at run time with the environment variable OMP_NUM_THREADS, which is usually chosen to be less than or equal to the number of processors and of which the value can be queried with the OpenMP function OMP_NUM_THREADS(). Distribution of work means that if nthread = OMP_NUM_THREADS() is the number of threads, about n/nthread additions would be executed by each thread.

SHARED(a,b) denotes that all threads use the same storage locations of arrays A and B. Whenever variables are only read and not modified, they can be declared as SHARED and can then be used concurrently by the threads.

Obviously, a problem arises if a thread writes to a storage location that a different thread reads or writes to, as the order of operations ('read before write' or 'write before read') is unpredictable and depends on the scheduling of threads by the operating system. In the above example, this problem only arises for the variable i, as the threads modify non-overlapping parts of array a. Also, many loops require scalar variables for temporary storage; then the key to successful parallelization is to assign variables that will be read and written to by individual threads in a unique way. These variables have to be declared as PRIVATE, which means that each thread is given an independent copy of the variable that it can modify. Variations of the PRIVATE clause address the problem of proper initialization (FIRSTPRIVATE) of each thread's temporary copy, or the use of its final value (LASTPRIVATE).

A similar effect could in principle be achieved by explicitly converting the scalar variable to an array, with each array element being assigned to one thread. Use of the PRIVATE clause delegates this task to the OpenMP compiler. A different way is to convert the body of a DO loop to a subroutine which is executed by each thread. Then, all local variables (scalars and arrays) that are modified by the subroutine will automatically be allocated by the operating system.

A different problem arises if all threads have to modify a common scalar variable (*e.g.* by addition) in each loop iteration, and the result is later used in the program (*i.e.* the variable is not only used for temporary storage). In this case, a REDUCTION clause that specifies the type of modification [allowed types are: +, −, *, /, min(), max() and several logical operators] can be used with the C$OMP PARALLEL directive. Effectively, this declares the variable as PRIVATE and automatically combines each thread's copies of the variable at the end of the DO loop according to the type of modification given.

Other clauses can appear with the C$OMP PARALLEL directive, *e.g.* the scheduling of loop indices to threads can be influenced by a SCHEDULE clause, the DEFAULT type of variables (SHARED, PRIVATE or NONE, which means that all variables must be declared) in the parallel region can be chosen, and serial or parallel execution can be chosen based on the

value of a logical expression (IF clause) evaluated at run time. Support for proper treatment of COMMON blocks is provided (the COPYIN clause works together with the C$OMP THREADPRIVATE directive).

A parallel program can be developed, compiled and tested on a single-processor workstation; upon execution, the threads are all started on the same CPU and perform the same work as on an MP computer, albeit normally (*i.e.* unless input/output can overlap computation) without savings in wall-clock time. If the user requests more threads than available processors, the total wall-clock time may rise considerably in the case of fine-grained parallelization constructs, as their synchronization requires frequent task switches.

Definition of the OpenMP parallelization API, a tutorial on parallelization techniques and other OpenMP-related links can be found at http://www.openmp.org. A Web-accessible program for automatic insertion of OpenMP directives into existing Fortran 77 code is available at http://punch.ecn.purdue.edu. It is based on the Polaris parallelizing compiler (Blume *et al.*, 1996), which represents the current state-of-the-art in automatic parallelization.

## 3. Parallelization of crystallographic code

### 3.1. A simple example program: *CRYSDEMO*

A small self-contained example (*CRYSDEMO*) that makes use of the OpenMP API for parallelization of a typical crystallographic calculation (direct summation of structure factors) is presented in Fig. 1. No effort was made to choose a particularly efficient algorithm or implementation for this task.

Normally, the effort exerted by an OpenMP programmer must be weighted against the speed-up reachable, which would suggest that in practice only the parallelization of the computational part would be employed for this program. However, for demonstration purposes, all possible aspects of input/output and calculation were parallelized. This also illustrates the possibility of incremental changes to a program, which is one design goal of OpenMP.

In *CRYSDEMO*, first a coordinate file [Protein Data Bank (PDB) format] and a reflection file (free format) are read in by two threads executing simultaneously. In many practical cases, the number of reflections will significantly exceed the number of coordinate records. However, during or after coordinate input, bookkeeping and analysis of coordinate records may be performed by the same thread, so that the computational workload appears to be approximately balanced between the two threads.

In the second half of the program, structure factors are calculated from coordinates assuming, for simplicity, equal point-scatterers in space group $P1$ ($a = 70$, $b = 80$, $c = 90$ Å, $\alpha = \beta = \gamma = 90^\circ$). For direct summation of scattering factors, all available threads are used. The number of threads is requested by a statement that is subject to conditional compilation. As the outer loop, which runs over the atoms, can be parallelized, the parallelism reached is coarse-grained.

Finally, the crystallographic $R$ factor between observed and calculated structure-factor amplitudes is calculated, again using the REDUCTION clause.

```
      program crysdemo
C demonstrate some parallelization constructs in a crystallographic program
C the program assumes equal point scatterers in space group P1 !
c SGI IRIX 6.5: compile with f77 -mp -mpio -O3
c Compaq Alpha OSF1: compile with f90 -omp -O3
c Portland Group compiler on Intel/Linux: compile with pgf77 -mp -fast
c
      implicit none
      integer maxatm,maxref,natom,nref,i,j
      parameter (maxatm=10000, maxref=200000)
c
      integer hkl(3,maxref)
c$    integer omp_get_max_threads,nthread
      real coord(3,maxatm),fo(maxref),fc(maxref)
      real x,y,z,fosum,fcsum,rfnum
      character*80 pdblin,hkllin
      complex fcmplx
c
  101 format(a80)
  102 format(t31,3f8.3)
  103 format(1x,3i4,f12.2,f10.2)
c
      print*,'enter name of PDB file'
      read(5,'(a80)') pdblin
      print*,'enter name of structure factor file (h,k,l,Fobs)'
      read(5,'(a80)') hkllin
c query and save the environment variable OMP_NUM_THREADS:
c$    nthread = omp_get_max_threads()
c ... and adjust the number of threads for the I/O section:
c$    if (nthread.gt.1) call omp_set_num_threads(2)
c$    print*,'using',min(nthread,2),' threads for I/O'
c
c$omp parallel shared(pdblin,hkllin,coord,natom,nref,rfnum,x,y,z,hkl,
c$omp& fo,nthread) default(none)
c
c$omp sections
c$omp section
      open(1,file=pdblin,status='old')
      natom=0
  100 read(1,101,end=220) pdblin
      if (pdblin(1:5).eq.'ATOM '.or.pdblin(1:7).eq.'HETATM ') then
        natom=natom+1
        if (natom.gt.maxatm) stop 'maxatm'
c the next statement does internal file I/O (see restriction below):
        read(pdblin,102) x,y,z
        coord(1,natom)=x/70.
        coord(2,natom)=y/80.
        coord(3,natom)=z/90.
      endif
      goto 100
  220 close(1)
c$omp section
      open(2,file=hkllin,status='old')
      nref=0
  300 nref=nref+1
      if (nref.gt.maxref) stop 'maxref'
c we can't read from unit 1 because that's taken by the other thread.
c at least on SGI, only one thread can do internal file I/O at a time.
      read(2,103,end=400)(hkl(i,nref),i=1,3),fo(nref)
      goto 300
  400 nref=nref-1
      close(2)
c$omp end sections
c$omp end parallel
c
      print*,natom,' atoms found'
      print*,nref,' reflections found'
c$    call omp_set_num_threads(nthread)
c$    print*,'using',nthread,' threads for computation'
      fosum=0.
      fcsum=0.
      rfnum=0.
c
c$omp parallel shared(i,j,fcmplx,nref,natom,coord,hkl,fc,fo,
c$omp& fosum,fcsum,rfnum) default(none)
c$omp do reduction(+:fosum,fcsum) private(i,j,fcmplx)
      do 600 i=1,nref
        fcmplx=(0.,0.)
        do 500 j=1,natom
          fcmplx=fcmplx+exp(2.*3.14159*(0.,1.)*(coord(1,j)*hkl(1,i)+
     &                    coord(2,j)*hkl(2,i)+coord(3,j)*hkl(3,i)))
  500   continue
        fosum=fosum+fo(i)
        fc(i)=abs(fcmplx)
        fcsum=fcsum+fc(i)
  600 continue
c$omp end do
c$omp do reduction(+:rfnum) private(i)
      do 700 i=1,nref
        rfnum=rfnum+abs(fo(i)-fosum/fcsum*fc(i))
  700 continue
c$omp end do
c$omp end parallel
c
      print*,'the scale factor is',fosum/fcsum
      print*,'the R-factor is    ',rfnum/fcsum
      stop
      end
```

**Figure 1**
A simple crystallographic program (available from http://strucbio.biologie.uni-konstanz.de/~kay).

### 3.2. Parallelizing *ESSENS*

*ESSENS* (about 1300 lines of Fortran 77 code) is a program for real-space molecular replacement when estimates of phases are available. It rotates and translates a template PDB file through an experimental electron density and calculates a score for the match of the template and the electron density for each rotation/translation combination. The score can also be used to obtain a map that visualizes the match of, say, an alpha helix template to the electron density, and can therefore make map interpretation easier. A separate program, *SOLEX*, extracts the best rotation/translation combinations from the *ESSENS* output.

*ESSENS* is an attractive option for a crystallographer who cannot solve the molecular-replacement problem for a protein, but has obtained an experimental map from single/multiple isomorphous replacement (SIR/MIR) or multiple anomalous scattering (MAD) data. Even if the map cannot be interpreted right away, it is often possible with *ESSENS* to find those rotation/translation combinations that position a similar molecule into the map in the optimal fashion. As the program uses all available amplitude and phase information, it produces a significantly higher signal-to-noise ratio than molecular-replacement programs that only use the amplitude information. *ESSENS* can be expected to be of importance in the context of structural genomics, where procedures for automatic structure solution are required.

Because of the six-dimensional nature of the search, the computer time required to position a model is substantial; it can be in the region of several days. On the other hand, the calculation with its six nested `DO` loops (three rotation and three translation variables) lends itself naturally to efficient parallelization. This most CPU-demanding part of the program was put into a separate subroutine, which is called once by each thread. Each thread then treats all rotation possibilities but only its particular fraction of translation space. Only one parallelization directive was used. No synchronization between the threads has to occur; therefore, the program would also be well suited for distribution across different computers with the help of the PVM or MPI interface.

To test the potential for automatic parallelization, the program was also compiled using the `-apo` option of the SGI f 77 compiler. However, no substantial speed-ups were obtained and inspection of the generated code showed that only fine-grained parallelism had been detected by the compiler.

The parallelized version of *ESSENS* is distributed in binary form for the SGI and Compaq Alpha platforms by the author of the original code (e-mail: gerard@xray.bmc.uu.se).

### 3.3. Parallelizing *SHELXL*

*SHELXL* (Schneider & Sheldrick, 1997) is a program (about 18800 lines of Fortran 77 code) originally developed for small-molecule refinement, but also available in a version that has been dimensioned for protein refinement. It uses direct summation for the evaluation of structure factors and their derivatives, and is therefore slow when compared with macromolecular-refinement programs, such as *X-PLOR* (Brünger & Rice, 1997), *CNS* (Brünger *et al.*, 1998), *TNT* (Tronrud, 1997) and *REFMAC* (Murshudov *et al.*, 1997), which employ the fast Fourier transform. However, the *SHELXL* options, such as anisotropic refinement of displacement parameters, automatic assignment and refinement of hydrogen positions, handling of disorder and multiple conformations, estimation of positional uncertainties and proper treatment of twinning, make *SHELXL* ideal for high-resolution refinement of macromolecules.

*SHELXL* is highly optimized for speed in the uniprocessor case and the program has undergone a long evolution. As a result, it does not employ 'structured programming' in the sense of using `IF...THEN...ELSE...ENDIF` statements and avoiding `GOTO` statements. To comply with earlier versions of Fortran, calculations with complex quantities are emulated using `REAL` variables and only one-dimensional arrays are used for storage.

Profiling the program with the tools (`ssrun`, `prof`) provided by the Irix operating system showed that about 85% of the total CPU time is spent in subroutine *SX3H* and the low-level subroutines for matrix and trigonometric operations called by it. *SX3H* is the subroutine that calculates the structure factors and their derivatives during refinement cycles, populates the normal matrix, solves it for atomic parameter shifts and applies these, and is therefore the central part of the program. As the calculation in principle resembles that of *CRYSDEMO*, it was clear that efficient parallelization could be achieved by distributing the reflections to parallel threads.

For historical reasons, *SHELXL* applies a 'blocking' technique by dividing the reflection array into blocks of a few hundred reflections, performing all relevant calculations on these, and later combining the results of those calculations. This technique ensures that the program runs efficiently on uniprocessor computers with very little memory, but it also prevents coarse-grain parallelism without considerable changes of the source code.

Finding a way to parallelize this program efficiently therefore presented a considerable challenge. Only after elucidating the correspondence of code sections of *SX3H* to the tasks required for crystallographic refinement was it possible to find the appropriate coarse-grain parallel constructs and thus to avoid unnecessary synchronization delays. In particular, it turned out to be inefficient to parallelize all the low-level subroutines called by *SX3H* as these are called once or multiple times for each atom and block of reflections. Rather, a parallel construct was designed (requiring minor changes to the source code) that assigns a sub-block of reflections to each thread. The thread then loops over all atoms, calling the relevant low-level subroutines in turn, and thus avoids frequent start-ups of new threads and costly synchronization periods.

After parallelizing this central part of *SHELXL*, further profiling runs were performed and additional targets for parallelization were identified in subroutines *SX3G*, *SXMM* and *SX3N*. In these subroutines, efficient parallelization could

also be achieved; however, the impact of these changes on the total CPU time used was minor compared to the savings made by parallelization of *SX3H*. In *SX3G*, the same parallel construct could be used as that applied in *SX3H*. The source code also had to be changed in both *SXMM* and *SX3N*; in *SX3N* a loop constructed using `GOTO` and `IF` was converted to a `DO` loop and a fine-grained parallelization directive inserted. Understanding the use of temporary variables in *SXMM* was possible after submitting the relevant part of the subroutine to the Polaris service (http://punch.ecn.purdue.edu); inspection of the returned Fortran code allowed identification of the suitable synchronization construct.

Again, the possibility of automatic parallelization of *SHELXL* was tested with the SGI f 77 compiler. As in the case of *ESSENS*, only fine-grained parallelism was detected.

The OpenMP changes are being included in the distributed version of *SHELXL* (release 97–2), which is available from its author (e-mail: gsheldr@shelx.uni-ac.gwdg.de).

## 4. Results

### 4.1. Efficiency of parallelization

The speed-up $S(p)$ obtained when executing a program on $p$ processors is defined as

$$S(p) = T(1)/T(p),$$

where $T(1)$ and $T(p)$ are the turnaround times with one processor and with $p$ processors, respectively. The efficiency $E(p)$ is then defined as

$$E(p) = S(p)/p.$$

For a perfectly parallelized program with $T(p) = T(1)/p$, we obtain

$$S(p) = p, \qquad E(p) = 1.$$

In theory, any program can be decomposed into a parallel part (of fraction $f$) that benefits from parallelization, and a serial part (fraction $1 - f$) that will not. Only if the serial part is non-existent can the program be perfectly parallelized, obtaining an efficiency of 1. However, all real programs have a significant serial part and it is this fraction of the program that limits performance on an MP computer and poses an upper limit on the efficiency. This is summarized in Amdahl's law, which states that

$$S(p) = 1/(f/p + 1 - f).$$

It follows that the maximum speed-up is $S(\infty) = 1/(1 - f)$ and that $f$ can be calculated from the observed speed-up as

$$f = p[S(p) - 1]/[S(p)(p - 1)].$$

### 4.2. Speed-up in *CRYSDEMO*, *ESSENS* and *SHELXL*

For the programs used for this study, the speed-up of parallel computation when compared with the uniprocessor case is shown in Table 2.

*CRYSDEMO* is only meant to demonstrate the principle. Clearly, the benefits of parallelization efforts are minor, as the total savings in wall-clock time are only of the order of seconds. The total time is influenced by the input of coordinates and structure factors; factoring out this component would show that the rest of the calculation is highly parallel. Other test programs developed by the author demonstrate that for simple trigonometric evaluations performed in a loop, parallelization becomes beneficial in terms of wall-clock time if the loop count is higher than a few hundred (SGI Origin 2000, SGI Octane) to a few thousand (Compaq ES40). Therefore, even the fine-grained $R$ factor loop of *CRYSDEMO* benefits considerably from parallelization if the number of reflections is high enough.

The very coarse-grained parallelization in *ESSENS* benefits most from execution on an MP computer, as its serial fraction $1 - f$ is quite low (about 0.03). As only a single OpenMP directive was used, *ESSENS* has a potential for further parallelization in its final stages of statistical calculations. For small problems, this fairly trivial optimization would decrease the wall-clock time even more.

*SHELXL* displays a more significant serial fraction $1 - f$ between about 0.12 (BIG case) and 0.35 (6RXN), which leads to diminishing improvements in

**Table 2**
Wall-clock times (s), speed-ups relative to uniprocessor calculation and calculated parallel fractions.

All times were obtained on an SGI Origin 2000 equipped with 250 MHz R10000 processors under Irix 6.5.5, Fortran 77 version 7.3. Calculations with one to four processors were also performed on a Compaq Alpha server ES40 with 500 MHz Alpha 21264 processors under OSF1 V4.0F, Fortran 90 version 5.2. Similar results were obtained for wall-clock times and speed-up.

| | | *CRYSDEMO*† | *ESSENS*‡ | *SHELXL* 6RXN§ | BIG ¶ |
|---|---|---|---|---|---|
| One processor | Time (s) | 29 | 15437 | 33 | 29346 |
| Two processors | Time (s) | 16 | 8034 | 25 | 16165 |
| | Speed-up | 1.81 | 1.92 | 1.32 | 1.82 |
| | Parallel fraction | 0.90 | 0.96 | 0.48 | 0.90 |
| Three processors | Time (s) | 12 | 5363 | 19 | 12120 |
| | Speed-up | 2.42 | 2.88 | 1.74 | 2.42 |
| | Parallel fraction | 0.88 | 0.98 | 0.64 | 0.88 |
| Four processors | Time (s) | 10 | 4162 | 16 | 10081 |
| | Speed-up | 2.9 | 3.71 | 2.06 | 2.91 |
| | Parallel fraction | 0.87 | 0.97 | 0.69 | 0.88 |
| Eight processors | Time (s) | 6 | 2180 | 14 | 7625 |
| | Speed-up | 4.83 | 7.08 | 2.36 | 3.85 |
| | Parallel fraction | 0.91 | 0.98 | 0.66 | 0.85 |

† 1151 atoms, 131819 reflections. ‡ 370 atoms, 1183 rotations (30° grid), 3455881 grid points in map. § 5031 reflections, 438 atoms, 10 CGLS cycles (*SHELXL* example: refinement of rubredoxin). ¶ 32281 reflections, 3480 atoms, 10 CGLS cycles (refinement of a 363 amino acid protein at 2 Å resolution).

turnaround times with more than four processors (Table 2). Still, the speed-up in *SHELXL* is between two and three when four processors are used, which represents significant savings in wall-clock time. It should be noted that often larger problems than BIG occur in high-resolution refinement of macromolecules. In these cases, a speed-up of three on a four-processor computer is highly desirable, and the inefficiency associated with $E(4) = 0.75$ appears to be tolerable.

It is interesting to note that the BIG problem in *SHELXL* displays a significantly larger speed-up than the 6RXN case, which shows that parallelization is comparatively more beneficial for large problems. This is due to the fact that an increase in the number of data that are processed by the parallel threads can turn fine-grained parallelism into coarse-grained.

## 5. Concluding remarks

Given existing plans for high-throughput structure determination in 'structural genomics' projects, based on the enormous rate of data collection at current third-generation synchrotrons, a decrease in computational turnaround times will continue to be mandatory. MP computers already exist in many crystallographic laboratories and at synchrotron sites, and, although they constitute considerable investments, their potential is often not fully realised. Rather, they are often utilized like clusters of uniprocessors with a common management.

The purpose of this paper is therefore twofold. First, it reports the availability of an extension to existing programming languages that could be used for the speed-up of crystallographic calculations if a multiprocessor computer is available. This has the potential of decreasing the time needed for progressing from a data set to a refined atomic model. Second, it demonstrates the successful application of this tool to two programs which are vastly different in terms of code complexity. In both cases, it was not required that the original author of the program ported the program into its parallelized form. As the adaptation of existing code to OpenMP can be performed in an incremental way, it appears that crystallographic programs that are well structured and documented, and are distributed in source code, will benefit most from the application of a *de facto* standard for parallelization.

The author wishes to thank R. Eigenmann, G. Kleywegt, G. Sheldrick and W. Welte for critical reading of the manuscript, and the computing centres of the universities of Freiburg and Konstanz for access to their SGI Origin 2000 and Compaq ES40 computers, respectively.

## References

Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L. & Tu, P. (1996). *IEEE Comput.* (December 1996), pp. 78–82.

Brünger, A. T., Adams, P. D., Clore, G. M., DeLano, W. L., Gros, P., Grosse-Kunstleve, R., Jiang, J.-S., Kuszewski, J., Nilges, M., Pannu, N. S., Read, R. J., Rice, L. M., Simonson, T. & Warren, G. L. (1998). *Acta Cryst.* D**54**, 905–921.

Brünger, A. T. & Rice, L. M. (1997). *Methods Enzymol.* **276**, 366–396.

De la Fortelle, E. & Bricogne, G. (1997). *Methods Enzymol.* **276**, 472–494.

Kabsch, W. (1988). *J. Appl. Cryst.* **21**, 916–924.

Kissinger, C. R., Gehlhaar, D. K. & Fogel, D. B. (1999). *Acta Cryst.* D**55**, 484–491.

Kleywegt, G. J. & Jones, T. A. (1997). *Acta Cryst.* D**53**, 179–185.

Leslie, A. G. W. (1992). *Joint CCP4/ESF-EACMB Newsl. Protein Crystallogr.* No. 26.

Murshudov, G. N., Vagin, A. A. & Dodson, E. J. (1997). *Acta Cryst.* D**53**, 240–253.

Otwinowski, Z. & Minor, W. (1997). *Methods Enzymol.* **276**, 307–326.

Schneider, T. R. & Sheldrick, G. M. (1997). *Methods Enzymol.* **277**, 319–343. New York: Academic Press.

Terwilliger, T. C. & Berendzen, J. (1999). *Acta Cryst.* D**55**, 849–861.

Tronrud, D. E. (1997). *Methods Enzymol.* **277**, 306–319.