Tuning Vector and Parallel Performance for Molecular Dynamics Codes on the Cray X1e

Konrad Wawruch, Lukasz Bolikowski, Wojciech Burakiewicz,
Maciej Cytowski, Maria Fronczak, Mariusz Kozakiewicz,
Michal Lopuszynski, Franciszek Rakowski, Joanna Slowinska
Interdisciplinary Center for Modeling, Warsaw University

May 18, 2005

Abstract

ICM development team has ported and optimized several molecular dynamics codes on Cray Vector Architecture. present Cray X1e performance comparison with the PC Opteron cluster on these codes, description of porting issues, vectorization and parallelism tuning. The applications we have worked on are CHARMM, DFTB, VASP, CPMD, Siesta, GROMOS.

KEYWORDS: Cray X1, CHARMM, DFTB, CPMD, VASP, Siesta, GROMOS, vectorization, molecular dynamics

1 Introduction

ICM is Interdisciplinary Center for Modeling, located at Warsaw University in Poland. To serve its purpose for the scientists using our supercomputers, ICM has worked on porting and optimizing selected Molecular Dynamics Chemistry applications on Cray X1.

The applications we have selected for development are:

- **CHARMM** – *Chemistry at Harvard Macromolecular Mechanics*, a program for macromolecular energy, minimization and dynamics calculation;
- **DFTB** – *Dense Functional – Tight Binding Method* code;
- **CPMD** – *Car-Parrinello Molecular Dynamics* code;
- **VASP** – *Vienna Ab-Initio Simulation Package*;
- **SIESTA** – *Spanish Initiative for Electronic Simulations with Thousands of Atoms*, ab-initio package.
- **GROMOS** – molecular dynamics code.

2 CHARMM

The CHARMM, *Chemistry at Harvard Macromolecular Mechanics*, a program for macromolecular energy, minimization and dynamics calculation, is one of the most popular software packages at ICM.

The CHARMM code size is about 17.5 MB and is grouped into 41 directories, which correspond to different calculation types provided by the package. The majority of them is not computationally demanding. In principle, the program is used by scientists to calculate energy and forces for a huge molecules in solvent environment. Thus, this calculations have several steps which are "bottle neck" for program flow. The main one are non-bonding interactions grouped in the code in directory `nbond`. Another approach to interaction calculations is to use Ewald summation technique that is very demanding either. Generally, 6% of the code is responsible for over 90% of computations time.

We have performed porting of two CHARMM releases to Cray X1/X1e. Both versions: c29b1 and c31b1 were successfully ported, and all types of calculations with CHARMM offers were tested. Several test were not passed and required additional work.

2.1 Initial Performance

The table below presents the initial performance for the selected types of calculations with CHARMM c31b1.

Test set	# CPU	X1 Time	PC Time
Molecular dynamics, 15000 steps GTP in water, Verlet Algorithm	1	3.24 h	55.4 min
Molecular dynamics 1000 steps Cytochrome in water, counterions, CPT algorithm	1	3.51 h	37.8 min
Molecular dynamics, 1000 steps Cytochrome in water, counterions, Ewald summation	1	1.24 h	39.52 min

The tests were conducted with SSP binaries. The PC reference time is the time of test execution on single Intel(R) Pentium(R) 4 CPU 2.66GHz computer.

As one can see all test clearly show that the CHARMM efficiency after straight forward porting without optimization is lower than on PC computer. Thus we focused on optimization of selected types of calculation with are of special interest of the majority of scientific groups.

Focusing on crucial for the molecular dynamics calculations of non-bonding interactions, the percentage time contribution of procedures is presented below. Those procedures are collected in `source/nbonds` directory.

Samp	**Cum.Samp**	**Function**	**Source**
90.1%	90.1%	enbfs8p_	enbfs8p.f
3.9%	93.9%	nbondma_	nbondm.f
2.5%	96.4%	nbonda_	nbonda.f
1.0%	97.4%	mkimat_	upimag.f

2.2 Development

The optimization effort was directed towards the computationally demanding procedures responsible for non-bonding interactions. Despite some work, optimizing has not resulted with decreasing the computation time for this part of calculations.

As far as Ewald summation technique is concerned, the contributions of procedures is as follows:

Samp	Cum.Samp	Function	Source
28.4%	28.4%	fill_ch_grid_	pme.f
18.3%	46.7%	rewald_	ewald.f
18.0%	64.7%	grad_sumrc_	pme.f
9.5%	74.2%	nbondma_	nbondm.f
8.9%	83.0%	mkimat_	upimag.f
8.4%	91.4%	nbonda_	nbonda.f
2.0%	93.4%	cfftb1_	pmeutil.f
2.0%	95.3%	cfftf1_	pmeutil.f

The vectorization which was performed has changed the summary report to the following form:

Samp	Cum.Samp	Function	Source
39.3%	39.3%	rewald_	ewald.f
19.3%	58.6%	nbondma_	nbondm.f
16.6%	75.2%	nbonda_	nbonda.f
9.2%	84.4%	fill_ch_grid_	pme.f
2.5%	86.9%	mkimat_	upimag.f
1.8%	88.8%	cfftf1_	pmeutil.f
1.8%	90.6%	cfftb1_	pmeutil.f
1.5%	92.1%	grad_sumrc_	pme.f

Subroutine `rewald` was partially vectorized, achieving the contribution of 40%, and when it was fully vectorized, the score remains at the same level. Also, some work was done to optimize procedures: `fill_ch_grid_`, `nbondma_`,`nbonda_`.

2.3 Results

All single processor optimizations, including vectorization, resulted in the following test results for SSP processors. For comparison, PC reference time and X1 execution time before performing any optimizations was included.

Test set	X1 Time (before) SSP	X1 Time (after) SSP	PC Time
Molecular dynamics, 1000 steps Cytochrome in water, counterions, Ewald summation	1.24 h	35 min	39.5 min

As for today, only small fraction of the code was already optimized. The code selected for optimization (1 100 KB out of 17 500 KB) represents over 90% of calculation time for all usual CHARMM calculations. Currently, approximately 20% of the selected code was optimized, achieving significant speedup, though more improvement may be expected. We are planning to complete vectorization of the remaining part of the code. That should speed up CHARMM code on Cray X1 to the level competitive with the fastest PC CPUs on the market (including AMD Opteron and Intel Itanium 2). After finishing vectorization, parallel code tuning may be performed.

3 DFTB

Self-Consistent Charge Density-Functional Tight-Binding (SCC-DFTB) is the theoretical method allowing the description of atomic systems in terms of Kohn-Sham density functional theory. It was introduced by

Th. Frauenheim and his coworkers and first applied in solid state physics. Then it was extended also to molecular systems and applied in biophysics and chemistry. The formulation of the method allows for very efficient energy calculations for systems consisting of hundreds of atoms. This is due to the semi-empirical approach which involves the parameterization of the computationally demanding energy terms. The method was first implemented as Fortran 77 scalar code DFTB and then parallelized using parallel versions of the `lapack` library routines. Currently the code is being rewritten using Fortran 90, however the new version is not yet available. Basic features of the program are:

- singe point energy calculation,
- atomic structure optimization,
- molecular dynamics,
- IR spectra calculation,
- reaction pathway search (using external module `neb`).

The source code of the DFTB program consists of 15,466 physical lines of code (SLOC) from which 13,002 are written in pure Fortran 77 and 2,464 in its dialect `Ratfor`.

There were not any special difficulties with porting the non-parallel DFTB code on the Cray X1 platform. Compilation was made for both MSP and SSP, and with the third level of optimization switched on.

3.1 Initial profiling

Test set	X1 Time 1 MSP	PC Time 1 PE 2.66GHz
kinase energy point	470.871 s	618.79 s

Comparison was made for timing of PC computer Intel(R) Pentium(R) 4 CPU 2.66GHz. The X1 Time is presented as it was before optimization. Calculations were carried out on 1 MSP (4 SSP). The percentage time of contribution of procedures was:

Samp	**Cum.Samp**	**Function**	**Source**
81.9%	81.9%	atomenerg_	atomen.f
4.9%	86.8%	dnrm2_	LibSci library
1.1%	87.9%	dlaed4_	LibSci library
1.1%	89.0%	eglcao_	eglcao.f
1.0%	90.0%	__bcopy_prv	
0.9%	90.9%	daxpy_	LibSci library
0.8%	91.7%	dgemmnn@_	
0.7%	92.4%	dgemmnt@_	
0.7%	93.1%	_F90_FCD_ASG	
0.7%	93.8%	dscal_	LibSci library
0.6%	94.4%	dsymv_	LibSci library
0.4%	94.8%	ewevge_	ewevge.f
0.3%	95.2%	dgemv_	LibSci library

Apart from the library functions two procedures are on the list of the most time consuming items, thus they were the subject of the optimization: *atomenerg* and *eglcao*.

3.2 Development

In the subroutine *atomenerg* the total electronic energy is calculated as the sum of the contributions of every atomic orbital. In the original code this summation was done separately for the set of atomic orbitals located on each atom. Due to the fact of multi-nested loops and short runs of each of it, the efficient vectorization and multi-streaming were not possible:

```
1        c       do i=1,ndim
1        c        do mu=1,ind(l+1)-ind(l)
1        c         do ls=1,nn
1        c          do nu=1,ind(ls+1)-ind(ls)
1        c           eat = eat + occ(i) * a(ind(l)+mu,i) * a(ind(ls)+nu,i)
1        c      &           * hamil(ind(l)+mu,ind(ls)+nu)
1        c          end do
1        c         end do
1        c        end do
1        c       end do
```

The upper loop was restructured, and the summation is taken over entire set of atomic orbitals regardless of the atoms. This change significantly improved vectorization and multi-streaming.

```
1            !dir$ nointerchange, prefervector, preferstream
1 MV-------<         do nu=ind(1)+1,ind(nn+1)
1 MV r-----<           do i=1,ndim
1 MV r r---<             do mu=ind(l)+1,ind(l+1)
1 MV r r                   eat = eat + occ(i) * a(mu,i) * a(nu,i) * hamil(mu,nu)
1 MV r r--->             end do
1 MV r----->           end do
1 MV------->         end do
```

In the procedure `eglcao`, Hamiltonian and Overlap matrices are to be updated:

```
1            c       do j = 1,nn
1            c         indj  = ind(j)
1            c         indj1 = ind(j+1)
1            c         do k = 1,j
1            c           indk  = ind(k)
1            c           indk1 = ind(k+1)
1            c
1            c           do n = 1,indk1-indk
1            c             do m = 1,indj1-indj
1            c               a(indj+m,indk+n) = hamil(indj+m,indk+n)
1            c               b(indj+m,indk+n) = overl(indj+m,indk+n)
1            c             end do
1            c           end do
1            c         end do
1            c       end do
```

and,

```
253.  1   c      do i = 1,nn
254.  1   c       do li = 1,lmax(izp(i))**2
255.  1   c        do j = 1,i
256.  1   c         do lj = 1,lmax(izp(j))**2
257.  1   c              a(ind(i)+li,ind(j)+lj) = a(ind(i)+li,ind(j)+lj)
258.  1   c     &        +0.5*overl(ind(i)+li,ind(j)+lj)*(shift(i)+shift(j))
259.  1   c         end do
260.  1   c        end do
261.  1   c       end do
262.  1   c      end do
```

To optimize the performance new auxiliary array was declared, and initialization of the Hamiltonian and Overlap matrices was added. Introducing the redundant upper triangle assignments we were able to restructure the loop achieving better efficiency of calculations.

```
221.  1            !dir$ nointerchange, preferstream, prefervector
222.  1 MV-------<       do n=ind(1)+1,ind(nn+1)
223.  1 MV r-----<         do m=ind(1)+1,ind(nn+1)
224.  1 MV r                 a(m,n) = hamil(m,n)
225.  1 MV r                 b(m,n) = overl(m,n)
226.  1 MV r----->         end do
227.  1 MV------->       end do
```

```
268.  1            !dir$ nointerchange, preferstream, prefervector
269.  1 m--------<       do i=1,nn
270.  1 m MVw----<         do li=ind(i)+1,ind(i+1)
271.  1 m MVw                tmpshift(li) = shift(i)
272.  1 m MVw---->         end do
273.  1 m-------->       end do
274.  1
275.  1            !dir$ nointerchange, preferstream, prefervector
276.  1 MV-------<       do n=ind(1)+1,ind(nn+1)
277.  1 MV r-----<         do m=ind(1)+1,ind(nn+1)
278.  1 MV r                 a(m,n) = a(m,n) + 0.5*overl(m,n)*(tmpshift(m)+tmpshift(n))
279.  1 MV r----->         end do
280.  1 MV------->       end do
```

3.3 Results

The table below presents code profile after optimizations:

Samp	Cum.Samp	Function	Source
27.4%	27.4%	dnrm2_	LibSci library
8.1%	35.6%	dlaed4_	LibSci library
4.9%	40.5%	daxpy_	LibSci library
4.9%	45.4%	dgemmnt@_	
4.7%	50.1%	__bcopy_prv	
4.5%	54.6%	_F90_FCD_ASG	
3.8%	58.4%	dgemmnn@_	
3.6%	62.0%	dscal_	LibSci library
3.2%	65.1%	dtrsm_llt@_	LibSci library
2.9%	68.1%	dsymv_	LibSci library
2.6%	70.7%	dgemmtn@_	
2.1%	72.7%	bcopy	
2.0%	74.7%	dgemv_	
1.9%	76.6%	ewevge_	ewevge.f
1.2%	77.8%	dlasr_	
1.1%	78.9%	dsyr2k_	LibSci library
1.0%	79.9%	repen_	LibSci library
1.0%	80.9%	dtrsv_	LibSci library

It is clearly shown that the main contribution is due to Linear Algebra procedures and system functions witch are already optimized for Cray X1.

Additionally the analysis of the average vector length and number of floating operations is presented:

	Average vec. length	**Total FP ops.**
Before optimization	25.46	339.098 M/s
After optimization	48.54	3 213.275 M/s

The restructurization of the code resulted in the following speed up of the SCC-DFTB program:

Test set	X1 Time (before) 1 MSP	X1 Time (after) 1 MSP	PC Time 1 PE 2.66GHz
Kinase energy point	470.871 s	48.29 s	618.79 s

The parallel version of SCC-DFTB has been already ported to Cray X1 architecture, we plan to increase the efficiency of the parallel code.

4 CPMD

Car-Parrinello Molecular Dynamics (CPMD) is the method of performing the dynamics of atomic systems with *ab initio* potential. This approach exploits the separation of fast electronic and slow nuclear motion using special form of the equations of motion based on the Lagrangian. From the energetic point of view CPMD is similar to DFTB since it incorporates the density functional for solving the electronic problem, using different basis set however. The method is well implemented in the CPMD code written in Fortran 90 and parallelized using MPI. It has been developed many years afo and extended set of features is now available in the code. Some of them are the following:

- single point energy calculation,
- wave function and atomic structure optimization,

- Car-Parrinello molecular dynamics (in ground and excited states),
- path-integral dynamics,
- many electronic properties.

The source of the CPMD code consists of 149,226 lines of code with more than 98% being written in Fortran 90.

The CPMD supports a number of computational platforms including few older Cray supercomputers (T3E, T90, YMP). However the suitable configuration files for Cray X1e were not available. Therefore, appropriate makefiles had to be prepared. Due to platform dependence, a few sections of the code (containing e.g system calls) needed adjustment as well.

For general performance evaluation four tests were chosen from tutorial inputs provided with the CPMD (see table below for details).

Test	Description
ex_B2H6	Energy calculation for diborane.
ex_H20	Energy calculation for 32 water molecules.
ex_Si64	Energy calculation for silicon super cell with 64 atoms.
ex_c120	Energy calculation for 120 atoms of carbon.

For the code compiled in SSP mode with the standard optimization option -O2 the following characteristics were obtained (for comparison execution time on Opteron 246 included):

Test	X1 SSP Time	Average vector length	Total FP Ops	PC Time
ex_B2H6	194 s	42.7	1 009 M/s	334 s
ex_H20	784 s	58.3	1 501 M/s	3 014 s
ex_Si64	82 s	58.1	1 193 M/s	240 s
ex_c120	3 092 s	58.7	1 942 M/s	19 032 s

Calculating the same tests with the code compiled in MSP mode (optimization option -O2) yielded the following results:

Test	X1 MSP Time	Average vector length	Total FP Ops	PC Time
ex_B2H6	119 s	33.2	1 429 M/s	334 s
ex_H20	389 s	58.0	2 939 M/s	3 014 s
ex_Si64	59 s	55.4	1 637 M/s	240 s
ex_c120	1 089 s	52.7	4 220 M/s	19 032 s

The above tests show that the CPMD makes use of X1e vector capabilities and provides reasonable performance. However, the parallel version of the CPMD code is not scaling well, therefore we are planning parallel optimizations.

5 VASP

The Vienna Ab-initio Software Package 4.6 (VASP) original source package was not ported to Cray X1 platform. The makefiles delivered were unable to use Cray `ftn` compiler. The last supported Cray vector supercomputer was Cray Y-MP. The code did not compile after adjusting makefiles.

As Cray Inc. has previously worked on porting VASP to Cray X1, we received information on how to modify the source code to compile on Cray X1 (modification of compiler options, directives) and to speed the code up (just by directives modifications). The received information was result of porting VASP 4.4 to Cray X1, instead of the 4.6 version, and therefore were not directly used.

5.1 Initial performance

After completing porting VASP to Cray X1 platform, the following test results were obtained. For comparison, PC execution time was included.

Test set	# CPU	PC Time	X1 SSP Time
Hg	1	222 s	981 s
Hg38	1	408 s	1 277 s
Cu	1	342 s	1 044 s

The tests were conducted with SSP binaries. The PC reference time is the time of test execution on single Opteron 244 computer.

The performance of the code was ranging from 100 MFlops to 300 MFlops, depending on the kind of the test.

5.2 Vectorization

As the performance of the generally optimized code is far from satisfactory and vectorization ratio is low (approximately 10% of instructions in the average run were vector instructions), we have decided to review the code and to rewrite crucial parts, allowing better vectorization.

In the table below, we present the results of the vectorization of functions, that were specified as important for the VASP performance on Cray X1:

Function name	Type	Source filename	Importance	Result
dgemv_	B	n/a	75689	`libsci` used
eddav@david_	V	davidson.F	62834	Vectorized
zgemmcn@_	B	n/a	40623	`libsci` used
fpassm_	V	fft3dlib.F	34329	Partial vectorization
ipassm_	V	fft3dlib.F	22259	Partial vectorization
eccp@hamil_	V	hamil.F	18978	Vectorized
setylm_aug_	V	us.F	13859	Vectorized
overl1_	V	dfast.F	12175	Vectorized
dlaebz_	L	n/a	9983	`libsci` used
fornl_	V	nonl.F	9330	Vectorized
eddrmm@rmm_diis_	V	rmm-diis.F	6561	Partial vectorization
hamiltmu_	V	hamil.F	5466	Partial vectorization
cnormn@dfast_	V	dfast.F	5411	Vectorized
vnlac0@nonl_	V	nonl.F	4661	Partial vectorization
cnorma@dfast_	V	dfast.F	3595	Vectorized
map_forward_	V	fftmpi_map.F	3463	Partial vectorization
setdij_	V	us.F	3416	Vectorized
map_scatter_	V	fftmpi_map.F	3245	Partial vectorization
fexcg__	V	xcgrad.F	2649	Partial vectorization
fftwav_	V	fftmpi.F	2484	Inlined
racc0mu@nonlr_	V	nonlr.F	2124	Vectorized
dlagts_	L	n/a	2060	`libsci` used
rnlpr@nonlr_	V	nonlr.F	1978	Vectorized
strenl_	V	nonl.F	1914	Vectorized
denmp@densta_	V	dos.F	1910	Partial vectorization
map_gather_	V	fftmpi_map.F	1882	No vectorization
rpromu@nonlr_	V	nonlr.F	1748	Partial vectorization

overl_	V	dfast.F	1658	Partial vectorization
rhosyg_	V	symmetry.F	1498	Partial vectorization
ranmar_	V	random.F	1334	Partial vectorization
set_dd_paw@paw_	V	paw.F	1284	Partial vectorization
elmin_	V	electron.F	1021	Partial vectorization
deple_	V	us.F	1019	Vectorized

Legend:

Function name – function name in the form reported by Cray Performance Analysis Tools.

Type – type of the function:

- V – original VASP function,
- B – BLAS function,
- S – LAPACK function.

Source filename – location of the function source in the VASP source tree.

Importance – level of importance of described function (higher is more important), calculated as the sum of weighted CPU time occupied by the function for all tests.

Result – result of development efforts.

All remaining functions, not included in the table, represent all together less than 10% of CPU time for each test.

The vectorization has influenced memory requirements of VASP, though by 20% maximum. It would be possible to speed the code up by the factor of 1.7, but it would swing memory requirements up 3-4 times. The initial MSP performance was, for some tests, lower than of SSP version, due to the bugs in the code. W have corrected such bugs and added compiler mutlistreaming directives.

5.3 Parallel Scalability

VASP includes MPI-based parallel version, though its scalability on Cray X1 is limited. Adjusting MPI system settings do not solve the scalability problem.

The reason of poor scalability is that `MPI_allreduce` and `MPI_alltoallv` functions behavior on Cray X1 – the load generated by such functions is not well balanced between nodes.

We have attempted to use SHMEM communication that was supported in VASP in earlier releases. The obtained results are very similar to the MPI – lack of balance of load generated by collective communication.

Due to the lack of success with current code we have started rewriting some communication functions in Co-Array Fortran to improve scalability. That work is not completed yet.

5.4 Results

All single processor optimizations, including vectorization, resulted in the following test results. For comparison, PC reference time was included.

Test set	X1 SSP Time	X1 MSP Time	PC Time
Cu	346 s	279 s	366 s
Cu2	481 s	406 s	470 s
Hg	201 s	173 s	257 s
HgIspin	389 s	339 s	465 s
HgLreal	375 s	358 s	527 s
AlNiNi	1 440 s	1 165 s	1 560 s
Ni	269 s	213 s	183 s

The created SSP binaries obtain performance from 800 MFlops to 1 550 MFlops (25-50% of peak).

The MSP binaries performance is far from satisfactory, and the development of them was dropped in favor of MPI version.

The obtained parallel scalability is relatively poor in comparison to the Opteron cluster interconnected with Gigabit Ethernet:

Test set	# CPUs	X1 SSP Time	X1 Speedup	PC Time	PC Speedup
HgLreal	1	375 s	1.00	527 s	1.00
	2	251 s	1.49	293 s	1.80
	4	190 s	1.98	163 s	3.23
	8	116 s	3.23	91 s	5.79
	16	90 s	4.17	–	–

The PC tests were performed on 2 Quad Opteron servers.

In our opinion it would be difficult to improve single SSP CPU performance of VASP without rewriting large parts of the code and rearranging its structure, or increasing memory requirements 3-4 times.

The scalability of the parallel version of the code is not appropriate and we continuously work on it. As similar performance problems occurred both for MPI and SHMEM, without any visible reason in the VASP communication code, it may be a bug in both VASP itself and the Cray Message Passing Library, and it requires more investigation.

6 Siesta

Siesta (*Spanish Initiative for Electronic Simulations with Thousands of Atoms*) is a general purpose package for performing electronic structure calculations in solids within the framework of density functional theory (DFT). In contrary to majority of solid state codes (e.g. VASP, Wien2k, Pwscf), which apply plane wave basis set, it uses a linear combination of numerical atomic orbitals. Therefore, molecular systems can be easily handled as well. For the core electrons the code implements a pseudo potential approach with the norm-conserving pseudo potentials of the Kleinman-Bylander type. Siesta is designed for fast calculations involving large systems, especially implementation of so called linear-scaling methods is particularly desired and unique feature.

Siesta is written in Fortran 90. It is parallelized with the MPI library. On platforms not providing parallel linear algebra routines the use of Scalapack is recommended.

The porting procedure consisted of the following steps:

- Adapting makefiles to satisfy X1 environment requirements. In this case files provided for T3E served as a basis.
- Disabling parts of the code responsible for Cray non-standard floating point arithmetic.

- Linking scientific library containing Cray Blas and Lapack routines.

One of the main problems, which we encountered during optimization of Siesta, was preparation of the satisfactory set of tests for large systems. Since Siesta is relatively new code, among users of ICM there were no groups which could serve us with necessary expertise. So initially we based on the tutorial tasks provided with the code. Later on the group of Siesta co-author prof. Pablo Ordejon was contacted. On the basis of obtained information a set of more computationally demanding jobs was prepared (LargeNiCo, LargeSi, LargeSiKP LargeSiNColl). Used tests are to serve optimization purposes i.e. enable for selecting the most time-consuming routines and examining to what extent particular option influences calculation method. Prepared inputs are not meant to be valuable from the scientific point of view.

6.1 Initial performance

The results of initial tests for ported code, compiled with -O2 option in SSP mode are shown in the following table (PC time was measured on a single node of ICM halo cluster, which uses Opteron 246 platform, package was compiled with IFORT v. 8.1 and MKL library):

Test	X1 tot FP ops [M/s]	X1 time [s]	PC time [s]	X1/PC time
MgO	21	46	3	17.35
H2O	44	63	4	15.72
Fe	114	1 639	197	8.32
FeLda	111	1 395	163	8.55
FeNoSpin	95	417	45	9.24
FeNodes	120	2 233	288	7.77
SiH	84	1 227	135	9.06
SiHHarris	56	753	69	10.87
SiHPbe	75	1 895	197	9.63
SiHNColl	263	2 667	764	3.49
LargeNiCO	67	10 652	1 215	8.77
LargeSi	85	4 685	683	6.86
LargeSiKP	300	5 649	2 273	2.49
LargeSiNColl	390	2 985	1 383	2.16

Similarly, the results of initial tests for ported code, compiled with -O2 option in MSP mode are shown in the table below:

Test	X1 tot FP ops [M/s]	X1 time [s]	PC time [s]	X1/PC time
MgO	18	54	3	20.21
H2O	34	79	4	19.55
Fe	253	633	197	3.21
FeLda	253	631	163	3.87
FeNoSpin	152	266	45	5.91
FeNodes	304	909	288	3.16
SiH	51	2 218	135	16.37
SiHHarris	45	1 214	69	17.53
SiHPbe	58	2 694	197	13.68
SiHNColl	216	3 170	764	4.15
LargeNiCO	52	13 924	1 215	11.46
LargeSi	42	9 831	683	14.39
LargeSiKP	176	9 564	2 273	4.21
LargeSiNColl	329	3 384	1 383	2.45

The results of initial tests can be summarized as follows:

- Both SSP and MSP versions of the code work considerably slower then PC versions.
- Total FP ops performance is very distant from theoretical limit of 3200 Mflops/sec (SSP) and 12800 Mflops/sec (MSP).

Blas and Lapack from Cray libraries were linked already in the porting phase. In addition, instead of provided one-dimensional Fast Fourier Transform from Numerical Recipes, the sci-lib `zzfft` was introduced during optimization stage.

6.2 Vectorization

During our work with Siesta the code of the following functions was modified in order to improve the use of X1 vector instructions: `diagk`, `vmat`, `rhoofd`, `dfscf`, `spher_harm`, `atmfuncs`, `atom`, `basis_io`, `bessph`, `radfft`.

In the above cases typical vectorization techniques were applied, namely: adding compiler directives, loop reordering, splitting loops into vector and non-vector parts etc. However, it has to be emphasized that in general Siesta code vectorizes poorly, due to frequent dependencies in the loops and rare use of typical vector/matrix operations.

During optimization the special version of `diagk` routine with large memory consumption was prepared. This procedure calculates the eigenvalues and eigenvectors, density and energy-density matrices, and occupation weights of each eigenvector for tasks which require Brillouin zone sampling.

The modified version has the following advantages:

- Instead of two Lapack diagonalization calls, it uses only one.
- Calculation of hamiltonian matrix for each k-point takes place only once.
- Calculation of the hamiltonian matrix vectorizes fully over k-points.

The main disadvantage is large memory space required for temporary storage of hamiltonians and wave functions, which to a large extent limits its practical applicability. Since the memory requirements grow linearly with the number of k-points they can easily extend the amount of memory installed on X1 platform even for systems of moderate size.

However, for really large systems, presumably most suited for efficient calculations on X1 platform, only Γ-point calculations are feasible anyway. In this case the `diagk` routine is unused.

6.3 Results

The comparison of results before and after optimizations is shown in the table below (the results do not include the `diagk` procedure with high memory usage):

Test	Before X1 tot FP ops [M/s]	After X1 tot FP ops [M/s]	Ratio	Before X1 time [s]	After X1 time [s]	Ratio
MgO	21	22	**1.09**	46	42	**1.10**
H2O	44	82	**1.87**	63	31	**2.02**
Fe	114	149	**1.31**	1 639	610	**2.69**
FeLda	111	127	**1.14**	1 395	597	**2.33**
FeNoSpin	95	146	**1.54**	417	200	**2.08**
FeNodes	120	136	**1.13**	2 233	922	**2.42**
SiH	84	121	**1.45**	1 227	892	**1.37**
SiHHarris	56	72	**1.30**	753	607	**1.24**
SiHPbe	75	100	**1.33**	1 895	1 475	**1.29**
SiHNColl	263	443	**1.69**	2 667	1 532	**1.74**
LargeNiCO	67	136	**2.03**	10 652	5 068	**2.10**
LargeSi	85	188	**2.22**	4 685	2 119	**2.21**
LargeSiKP	300	458	**1.53**	5 649	3 700	**1.53**
LargeSiNColl	390	728	**1.87**	2 985	1 561	**1.91**

Additional results for `diagk` procedure with high memory usage can be found in the following table:

Test	Before X1 tot FP ops [M/s]	After X1 tot FP ops [M/s]	Ratio	Before X1 time [s]	After X1 time [s]	Ratio
Fe	114	367	**3.21**	1 639	279	**5.88**
FeLda	111	352	**3.16**	1 395	241	**5.79**
FeNoSpin	95	230	**2.41**	417	119	**3.49**
FeNodes	120	461	**3.83**	2 233	313	**7.14**
LargeSiKP	300	423	**1.41**	5 649	3 308	**1.71**

The results of tests for most efficient version of the code, compiled with -O2 option in SSP mode are shown in the following table (PC time was measured on a single node of ICM halo cluster, which uses Opteron 246 platform, package was compiled with IFORT v. 8.1 and MKL library). The results do not include the `diagk` procedure with high memory usage.

Test	X1 tot FP ops [M/s]	X1 time [s]	PC time [s]	X1/PC time
MgO	22	42	3	15.79
H2O	82	31	4	7.76
Fe	149	610	197	3.10
FeLda	127	597	163	3.66
FeNoSpin	146	200	45	4.44
FeNodes	136	922	288	3.21
SiH	121	892	135	6.59
SiHHarris	72	607	69	8.77
SiHPbe	100	1 475	197	7.49
SiHNColl	443	1 532	764	2.00
LargeNiCO	136	5 068	1 215	4.17
LargeSi	188	2 119	683	3.10
LargeSiKP	458	3 700	2 273	1.63
LargeSiNColl	728	1 561	1 383	1.13

SSP results after including version of the `diagk` routine with high memory usage (only tests actually using the `diagk` are shown):

Test	X1 tot FP ops [M/s]	X1 time [s]	PC time [s]	X1/PC time
Fe	367	279 (11 MB)	197 (23 MB)	2.41
FeLda	352	241 (11 MB)	163 (23 MB)	3.48
FeNoSpin	230	119 (10 MB)	45 (17 MB)	2.65
FeNodes	461	313 (14 MB)	288 (26 MB)	1.09
LargeSiKP	423	3 308 (718 MB)	2 273 (1 026 MB)	1.46

Note, that even with the optimization method involving excessively large memory cost the performance of the code on single SSP was still much lower than obtained on PC platform.

The results of tests for SSP version of the code compiled in MSP mode:

Test	X1 tot FP ops [M/s]	X1 time [s]	PC time [s]	X1/PC time
MgO	21	48	3	18.07
H2O	61	46	4	11.48
Fe	201	388	197	1.97
FeLda	201	387	163	2.37
FeNoSpin	183	161	45	3.58
FeNodes	248	514	288	1.79
SiH	106	1 276	135	9.42
SiHHarris	67	896	69	12.94
SiHPbe	101	1 761	197	8.94
SiHNColl	382	1 846	764	2.42
LargeNiCO	109	6 559	1 215	5.40
LargeSi	153	2 758	683	4.04
LargeSiKP	443	3 753	2 273	1.65
LargeSiNColl	879	1 262	1 383	0.91

From the presented tables it results that after vector optimization significant speed-up was obtained. However, performance remains very far from theoretical limits for X1. In addition, the vector characteristics of the code still indicate insufficient usage of X1 vector capabilities, as can be seen in the table below:

Test	Average vector length	Vector instructions
MgO	7.88	3.20%
H2O	14.30	5.99%
Fe	23.26	6.62%
FeLda	23.28	6.53%
FeNoSpin	24.34	7.00%
FeNodes	26.60	6.25%
SiH	12.19	11.74%
SiHHarris	8.59	8.86%
SiHPbe	7.50	13.92%
SiHNColl	24.05	19.41%
LargeNiCO	15.08	12.23%
LargeSi	17.48	16.38%
LargeSiKP	35.88	18.60%
LargeSiNColl	35.85	27.87%

This is mainly due to the following difficulties which we encountered in the vectorization phase:

- Frequent dependencies occurring in the code which prevent us from effective vectorization.
- Lack of more detailed documentation concerning algorithms and data structures used in the code. Deeper understanding of calculations could possibly enable us for redesigning the crucial parts of the code in a way advantageous for X1.
- Rare use of typical matrix/vector operations in the important routines.
- Relatively low percentage of time spent in library routines (Lapack, FFT etc.).

In the present stage, despite performed optimizations, the code still works considerably slower on SSP processor than on a single PC.

According to our experience with Siesta, this code is not suited to run efficiently on X1. Performed vectorization attempts brought significant speed-up, however afterwards the code still runs with performance very far from theoretical limit and works much slower on X1 SSP than on single Opteron 246 processor. Many parts of the code were found not to vectorize well. In addition there does not seem to be many possibilities for multistreaming optimizations.

Except the above, during our work with Siesta it appeared that current version of the package (1.3f1p) suffers from a few important shortcomings, which hinder its use:

- *Lack of pseudo potential data base.* Vast majority of solid state codes is provided with set of pre-tested pseudo potentials. Siesta provides only the tool for generating them. This results in additional expertise and effort needed for preparing the inputs.
- *Bugs in linear-scaling part of the code.* In the present version 1.3f1p the calculations made in the linear scaling mode give wrong results. According to the information from authors this part of Siesta will be completely rewritten in the new release of the package.
- *Imperfect parallelization.* Siesta is prepared to run in parallel on platforms providing MPI, it also makes use of parallel Scalapack library. However, on our PC cluster at ICM we were not able to observe hardly any scaling with number of processors. This agrees with experience of some other groups reported on the Siesta mailing list. Scaling of the code is said to be observed only on platforms with very efficient communication.

Bearing in mind all the above, we conclude that the practical utility of Siesta code on X1 platform is very limited, though it may change with the future Siesta releases.

7 GROMOS

The main focus of porting *GROningen MOlecular Simulation computer program package* (GROMOS), version 96, was on the **PROMD** program which is the main computational application from the package. The other programs included in the package play role of the help applications and are used for data preparation and data analysis and have limited CPU requirements.

GROMOS supplies well known algorithms: Leap – Frog Verlet and 4-D L-F Verlet. Usually they serve for analyzing structural and conformational changes in big molecules – up to several thousand atoms in molecule. Owing to trajectory propagation all forces must be calculated, thus following interactions are taken into account: double bond, angles, dihedrals, improper and on bonded.

Since first four interactions types convey two-, three-, four- body terms are not computationally demanding. The non-bonded interactions are long-rage type and every step of forces calculations requires summing over all atoms in molecule and its environment. This part of simulations consumes significant amount of the CPU time.

GROMOS package contains 2 main versions of the non-bonded force subroutines: scalar and vector ones. The vector code however has been developed for old vector systems and its low performance is well known.

To some extend, the scalar code is more prospective since it was used for the development of the parallel version of the GROMOS. The parallelization has been performed with PFortran and Co-Array Fortran.

The GROMOS original source package was not ported to Cray X1 platform. The makefiles delivered were unable to use Cray `ftn` compiler. The last supported Cray vector supercomputer was Cray SV1, and the code compiled and worked correctly after adjusting makefiles.

Because of this we have performed profiling of both versions, the code has been compiled in SSP mode with flags `-Oaggress,scalar3,inline4,vector3`.

The performance of both versions (scalar and vector) was similar and we observed practically no difference in the computing time.

We have decided to start further work on scalar version since it was used for the parallelization of the GROMOS package.

7.1 Initial Performance

For the tests we have used reasonably size system (HIV proteasis in water, 50 000 atoms).

As expected most of the CPU time is spent in the subroutine `NONBML_MP` which evaluates non-bonded interactions:

Samp%	Cum.Samp%	Samp	Function
100.0%	100.0%	3106	Total
61.2%	61.2%	1902	nonbml_mp_
7.5%	68.8%	234	nbwith_mp_
2.4%	71.1%	73	_wrfmt
...			

Detailed profiling shows that most of the CPU time (40%) is spent in single loop `DO 827`. This loop is located inside loop `DO 810` which is a long one – loops over all atoms in the system – and is invoked NRAM times. In the original code, the `DO 810` is vectorized partially due to indirect addressing.

7.2 Development & Results

The main loop rearrangements allowed 20% code speedup.

We have decided to use parallel version of GROMOS, written in Co-Array Fortran, adjusting it to the Cray X1.

The following table presents comparison of initial performance of GROMOS on Cray X1 and reference PC (Opteron 246):

Test	X1 SSP time	PC time
jempci.sh	4.68 s	1.96 s
jempcl1.sh	58.63 s	36.99 s
jmdpc13.sh	65.45 s	48.47 s

Test results of the parallel scalability of GROMOS is presented in the following table:

# CPUs	X1 SSP time
1	329 s
2	179 s
4	108 s
8	99 s
12	91 s
16	100 s

The tests were performed for `test2` input.

The GROMOS binaries on Cray X1 are slower than binaries for the fastest PC processors. It seems that it would be impossible to improve vectorization without rewriting main computational kernel in program `promd`.

The parallel scalability is limited by the unbalanced calculation distribution between CPUs, not by the communication media or communication method. To improve the scalability, different calculation distribution algorithm should be deployed in the code (domain decomposition instead of force decomposition).

8 Summary

After working on multiple scientific codes on Cray X1e, we may draw some final conclusions.

It is possible to obtain reasonable performance of MD codes on Cray X1/X1e, very competitive to PC clusters. It may require though significant amount of development, depending on the code architecture, coding style and language constructs used.

The codes that were extremely difficult to optimize are Siesta and GROMOS. We have successfully optimized VASP, DFTB and CPMD, obtaining significant speedup. CHARMM development also brought reasonable speedup, though the size of the code requires enormous amount of development.

The common problem of all ported codes is parallel scalability. Due to MPI performance issues on Cray X1/X1e we have to consider porting communication layer to Co-Array Fortran.

9 About the Authors

The projects were led by Konrad Wawruch, coordinating, together with Lukasz Bolikowski, HPC development efforts at ICM. He may be reached at: ICM Warsaw University, Pawinskiego 5A blok D, 02-106 Warsaw, Poland, email: kwaw@icm.edu.pl. Wojciech Burakiewicz, Maciej Cytowski, Maria Fronczak, Mariusz Kozakiewicz, Michal Lopuszynski, Franciszek Rakowski and Joanna Slowinska are Software Developers at ICM.