Future Proofing WL-LSMS: Preparing for First Principles Thermodynamics Calculations on Accelerator and Multicore Architectures

Markus Eisenbach

Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, USA

ABSTRACT: The WL-LSMS code has a very good track record for scaling on massively parallel architectures and achieves a performance of approx. 1.8 PF on the current Jaguar system at ORNL. Yet the code architecture assumes a distributed memory with a single thread of execution per MPI rank, which is not a good fit for multicore nodes and the emerging accelerator based architectures. This paper presents the ongoing work to restructure the WL-LSMS code to take advantage of these new architectures and continue to work efficiently during the next decade.

KEYWORDS: Magnetism, Monte-Carlo, Matrix Inversion, GPU

1 Introduction

A large number of codes that get employed in basic research evolve at a rapid pace. Quite often new features, that are required to investigate new scientific directions, get implemented in an *ad hoc* fashion by the code's users and might not be coordinated. While this appears to be a code developer's and maintainer's nightmare, this is tempered by the large overlap of the user and developer communities, which in some cases might be identical. This uncoordinated evolution of the code base, while not ideal, is usually manageable as long as no other disruptive changes in the computing environment occur. The recent proliferation of multicore and accelerator based architectures represent such a disruptive event that has to be addressed and which potentially can necessitate fundamental changes in an actively evolving research code. In the present paper I will outline the experience in restructuring the WL-LSMS code to allow it to exploit the capabilities offered by these new architectural challenges. First I will outline the structure of the code and it's main components and the I will describe the changes that were made to the LSMS portion in moving from LSMS-1 to LSMS-3 to facilitate the use of accelerators and multithreading.

2 Structure of WL-LSMS

The WL-LSMS code uses a hybrid parallelization scheme. At the top level, the code parallelizes over concurrent random walkers, where we use a master-slave scheme, with a master that accumulates the density of states of the system, and the slaves that execute the random walks, each running its own instance of the LSMS method. The second parallelization level is the LSMS portion of the code, where domain decomposition is used with one atom per processing core. In typical production runs, the WL method would use a hundred to a few thousand concurrent walkers, and the LSMS portion would be parallelized over up to a few thousand processing cores. The method hence will scale to hundred thousand or

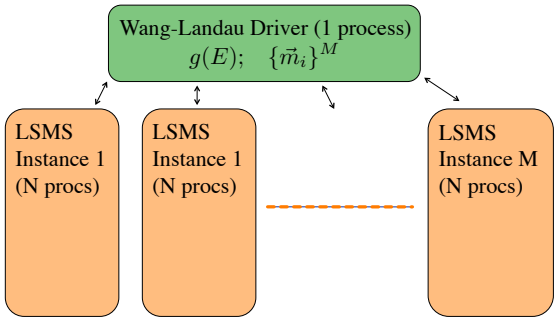

Figure 1: Parallelization strategy of the combined Wang-Landau/LSMS algorithm. The Wang-Landau process (Alg. 1) generates random spin configurations for M walkers and updates a single density of states $g(E)$. The energies for these N atom systems are calculated by independent LSMS processes (Fig. 2). This results in two levels of communication, between the Wang-Landau driver and the LSMS instances, and the internal communication inside the individual LSMS instances spanning N processes each.

Algorithm 1 Wang-Landau/LSMS algorithm

1: initialize logarithmic density of states $\ln g(E) \leftarrow 0$, histogram $h(E) \leftarrow 0$, modification factor $\gamma \leftarrow 1$, and the set of magnetic moment directions for the M walkers $\{\hat{e}\}_{1\ldots M}$
2: **repeat**
3: submit new random moment directions $\{\hat{e}^{\text{new}}\}$ to idle LSMS instances
4: receive new energy E_n^{new} from walker n
5: accept new set of directions $\{\hat{e}^{\text{new}}\}_n$ with probability $\min[1, g(E_n^{\text{old}})/g(E_n^{\text{new}})]$
6: **if** Move accepted **then**
7: $\{\hat{e}^{\text{old}}\}_n \leftarrow \{\hat{e}^{\text{new}}\}_n$
8: **end if**
9: update density of states $\ln g(E_n) \leftarrow \ln g(E_n) + \gamma$ and histogram $h(E_n) \leftarrow h(E_n) + 1$
10: **if** $h(E)$ flat **then**
11: $\gamma \leftarrow \gamma/2$, $h(E) \leftarrow 0$
12: **end if**
13: **until** $g(E)$ converged, *i.e.* $\gamma \approx 0$

millions of processing cores. The schematics of the parallelization structure are shown in fig. 1.

The Wang-Landau portion of the code uses the algorithm 1 to calculate the thermodynamic density of states necessary to investigate material behavior at finite temperature. Since this part already had been written with an interface in mind that allows the easy exchange of the energy calculation (the computationally intensive part performed by LSMS) no modifications were necessary in moving from LSMS-1 to LSMS-3 and the only part that is of concern for this study is the LSMS section of the code.

3 The LSMS Algorithm

For the energy evaluation, we employ the first principles framework of density functional theory (DFT) in in the local spin density approximation (LSDA). To solve the Kohn-Sham equations arising in this context, we use a real space implementation of the multiple scattering formalism. The Locally Self-consistent Multiple Scattering (LSMS) method calculates the electronic properties from first principles in reals space, but introduces some approximations that make the treatment of infinite systems possible. Furthermore this method results in a code that scales linearly with the size of the system.

The LSMS method is based on the observation that good convergence can be obtained by solving the Kohn-Sham equation of density functional theory at a given atomic site by considering not the whole system, but only a sufficiently large neighborhood, the local interaction zone (LIZ), of each site. The details of this method for calculating the Green function and the total ground state energy $E[n(\vec{r}), \vec{m}(\vec{r})]$ are described elsewhere [3, 4]. For the present discussion it is important to note that the computationally most intensive part is the calculation of the scattering path matrix τ for each atom in the system by inverting the multiple scattering matrix.

$$\tau = [I - tG_0]^{-1} t \tag{1}$$

The only part of τ that will be required in the subsequent calculation of site diagonal observables (*i.e.*

magnetic moments, charge densities, and total energy) is a small (typically 32×32) diagonal block of this matrix who's rank is $O(4k)$. This will allow us to employ the algorithm described in the next section for maximum utilization of the on node floating point compute capabilities.

For a more detailed description of the underlying algorithms see [10].

4 LSMS 3

4.1 LSMS 1 Background

The original LSMS-1 code was written to efficiently exploit the capabilities of distributed memory architectures of the 1990s. Due to the experience of the original programmers and the widely available software development environments available at the time this resulted in the use of Fortran 77 (with some additions of Fortran 90 features, such as dynamic memory allocation) and MPI for the communication layer. Additionally BLAS and LAPACK are used for the dense linear algebra that accounts for approximately 95% of the execution time. For the most part the use of `COMMON` blocks was avoided and all arguments for subroutine calls were passed explicitly. This resulted in unwieldy long subroutine calls in Fortrang 77 due to the lack of user defined data types, but it significantly improved the reusability of a large number of subroutines in the new LSMS 3 code. Consequently, the major architectural feature missing in LSMS 1 going forward to new hardware architectures, was the assumption of a on-to-one mapping between atoms and MPI ranks that limited the code's flexibility in adapting to a hybrid multi-threaded and distributed memory (OpenMP/MPI) paradigm or utilizing accelerators such as GPUs.

4.2 Overall Code Structure

As the refactoring of LSMS had to be done by one person, it was important to be able to reuse as much code from LSMS-1, either directly or with minimal modifications. Yet it was also desirable to use more expressive and economical coding styles by using abstractions that were not available in the original Fortran 77 code. To achieve these goals The new code employes both C++ and Fortran. The main skeleton of the code and the I/O sections are written in C++ while the main computational subroutines reuse the original LSMS-1 based Fortran code. The combination of pruning obsolete code and the greater expressiveness of C++ resulted in a reduction of the size of the LSMS main subroutine in LSMS-1, `lsms_main_as_subroutine.f`, from 2539 lines to 353 lines in LSMS-3's `lsmsClass.cpp`. Listing 1 shows the essential part of the standalone (without Wang-Landau) LSMS `main`. The data required to perform the LSMS calculation has been encapsulated in four classes that contain 1) system wide parameters, such as the total number of atoms or the energy contour and other method related parameters (`LSMSSystemParameters`), 2) the crystal structure, thus describing the physical system (`CrystalParameters`), 3) the data that is needed only on individual nodes, *e.g.* the atomic potentials (`LocalTypeInfo`) and 4) a class to hold the information needed do distribute the data and perform the communication (`LSMSCommunication`).

Listing 1: Structure of `main`

```
  LSMSSystemParameters lsms;
  LSMSCommunication comm;
  CrystalParameters crystal;
  LocalTypeInfo local;

// Initialize communication and accelerator

// Read the input file

  communicateParameters(comm,lsms,crystal);

  local.setNumLocal(distributeTypes(crystal,
      comm));
  local.setGlobalId(comm.rank,crystal);

  buildLIZandCommLists(comm,lsms,crystal,
      local);

  loadPotentials(comm,lsms,crystal,local);

  setupVorpol(lsms,crystal,local,
      sphericalHarmonicsCoeficients);

  calculateCoreStates(comm,lsms,local);
```

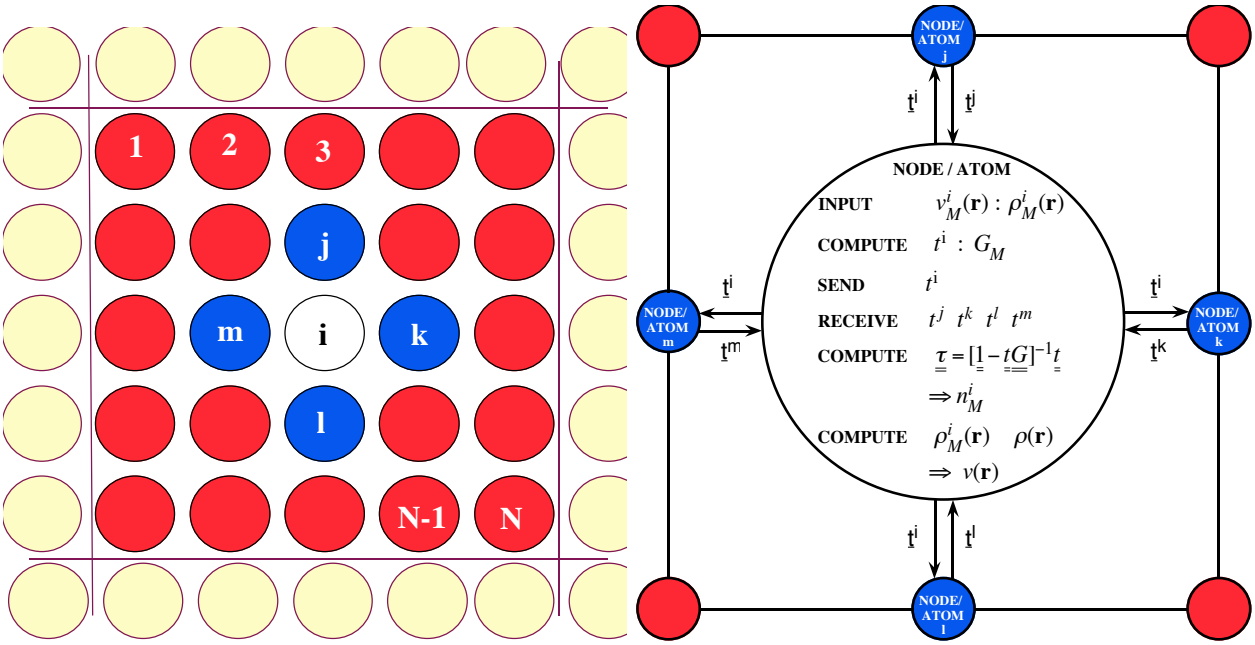

Figure 2: Schematic, Left: LIZ centered at processor/atom i; Right: message passing and computation.

```
  energyContourIntegration(comm,lsms,local);
  calculateChemPot(comm,lsms,local,eband);
```

Listing 2: Class to store data for local atoms

```
class LocalTypeInfo {
public:
  void setNumLocal(int n);
  void setGlobalId(int rank,
      CrystalParameters &crystal);

  int num_local;
  std::vector<int> global_id;
  std::vector<AtomData> atom;
  std::vector<int> n_per_type;

  int lDimTmatStore;
  Matrix<Complex> tmatStore;
  std::vector<int> tmatStoreGlobalIdx;
};
```

4.3 Input

The major user visible change is the choice of a new input file format. The LSMS-1 code reads its main input line by line from a text file and expects the input parameters on the correct line in the correct order. This can make it difficult to track down errors in the input and also the rigidity of this format makes extensions to the input format challenging. To provide flexibility and extensibility of the input, LSMS-3 uses a **Lua** [11], a scripting language that is popular for its simple API and light weight, based input.

This change is not necessary for the change to multicore and accelerator based architectures, yet the value added by this new interface change will contribute to the future adaptability of the code. The input to LSMS-3 consists of a Lua script that is executed at the start of the code on MPI rank 0. An example for a 1024 iron run is given in listing 3. After executing the script the code will read the values of the Lua that contain the system information, such as `energyContour` for the energy integration parameters or `bravais` and `site` for the crystal structure. The script allows the user to specify calculations for the input parameters. In the example given the lattice constant of iron is defined in one place as `a =`

5.42 and can then reused to specify the lattice structure in terms of this parameter, which improves both the readability of the input file and allows for easy change of this parameter. Finally the example shows the power of the scripting language approach for filling in default parameters without the need to provide complicated mechanism inside the scientific code itself that would have to account for multiple usage scenarios.

Listing 3: Input file for 1024 atom iron calculation

```
systemid="Fe1024"
system_title = "Iron␣test␣for␣LSMS␣3"
pot_in_type=1
num_atoms=2
nspin=3

xRepeat=8
yRepeat=8
zRepeat=8
makeTypesUnique=1

energyContour = {npts=31,grid=2,ebot=-0.3,
    etop=0.0,eitop=0.825,eibot=0.025}

a = 5.42

bravais = {}
bravais[1]={a,0,0}
bravais[2]={0,a,0}
bravais[3]={0,0,a}

site_default={lmax=3,rLIZ=12.5,rsteps
    ={89.5,91.5,93.2,99.9},atom="Fe",Z=26,Zc
    =10,Zs=8,Zv=8,rad=2}

site = {}
for i =1,num_atoms do site[i]={} end

site[1].pos={0,0,0}
site[1].evec={0,0,1}
site[2].pos={0.5*a,0.5*a,0.5*a}
site[2].evec={0,0,1}

-- set site defaults
for i =1,num_atoms do
  for k,v in pairs(site_default) do
    if(site[i][k]==nil) then site[i][k]=v
        end
  end
end
```

4.4 Communication

A major change in the code structure in moving from LSMS-1 to LSMS-3 involved the distribution of work across MPI ranks. LSMS-1 assumes a on-to-one mapping between atoms and MPI ranks and does not allow for further parallelism beyond the atom level. (The additional Wang-Landau parallelism that sits on top of the LSMS part is not effected by this.) Consequently the main driving force for the refactoring of LSMS-3 was the desire to allow greater flexibility in the distribution of work and allow in addition to the original scheme the possibility to assigne multiple atoms to a MPI rank and use OpenMP on a multicore node to further distribute the work or to utilize accelerators such as GPUs that are usually have a different number available then the number of cores (eg. a node with twelve CPU cores and one GPU).

This change significantly complicates the communication pattern to distribute the t matrices inside the LIZ as shown in figure 2. The original path taken in LSMS-2, an ongoing Fortran 90 rewrite of LSMS to implement new scientific capabilities such as full potential and k space calculations, was to use a GET based on-sided communication scheme, since the sites from which t matrices are required can be easily calculates whereas the sites that require a given atom's t matrix are harder to calculate. Unfortunately on many distributed memory architectures this communication scheme incurs a unacceptably large performance penalty.

Algorithm 2 The construction of the LIZ communication lists

for all atoms i in the crystal **do**
  build the local interaction zone $\mathrm{LIZ}_i = \{j | dist(\mathbf{x}_i, \mathbf{x}_j) < r_{\mathrm{LIZ}}\}$ of atom i
  for all atoms j in LIZ_i **do**
    add atom j to the list R_i of data to receive for atom i (`tmatFrom`)
    add atom i to the list S_j of data to send from atom j (`tmatTo`)
  end for
end for
remove duplicate entries from S_j and R_i

To achieve this the code first constructs the connections between the atomic sites in the crystal. This has to be performed in a sufficiently general way to not restrict the structures that can be investigated. The steps take to calculate these lists are sketched out in algorithm 2. For all atoms in the system two lists are generated, R_i of the remote sites that are required by site i to generate its scattering matrix and S_i of the sites that need data from site i to construct their scattering matrix. Care must be taken to take periodic boundary conditions into account and to remove duplicates from these lists, that can result from both the boundary conditions and the overlap of interaction zones. From this information the data structures `tmatTo` and `tmatFrom` in listing 4 are filled that list for each remote node the data for the atomic sites that are identified by their global index `globalIdx` that need to be exchanged and where it has to be stored locally (`tmatStoreIdx`).

Listing 4: Data structures for communication support

```
class TmatCommType {
public:
  int remoteNode;
  int numTmats;
  std::vector<int> tmatStoreIdx;
  std::vector<int> globalIdx;
  std::vector<MPI_Request>
      communicationRequest;
};

class LSMSCommunication {
public:
  int rank;
  int size;
  MPI_Comm comm;

  int numTmatTo, numTmatFrom;
  std::vector<TmatCommType> tmatTo, tmatFrom
      ;
};
```

This data struct needs to be constructed only once at startup and does not result in a major memory requirement, since it needs only be kept only for the sites that are local to a particular MPI rank. The actual communication of the atom data needed to build the scattering path matrices is provided by the three functions shown in listing 5 that allow the implementation of non blocking communication. In particular `expectTmatCommunication` can be called before the calculation of the individual t matrices to pre-post non blocking receives. The t matrix calculation itself requires no inter node communication and can easily exploit multithreaded intra node parallelism across the node local atomic sites. The routine `sendTmats` sends the t matrices after they have been calculated and `finalizeTmatCommunication` waits for the non-blocking communications to be finished before the inverse scattering path τ (eq. 1) matrix is constructed.

Listing 5: Functions for t matrix communication

```
void expectTmatCommunication(
    LSMSCommunication &comm,LocalTypeInfo &
    local);
void sendTmats(LSMSCommunication &comm,
    LocalTypeInfo &local);
void finalizeTmatCommunication(
    LSMSCommunication &comm);
```

Additionally the code provides encapsulations of commonly used communication to isolated the explicit use of a specific external API such as MPI. The example of a global sum shown in listing 6 also illustrates the expressiveness of templated C++ for type-general implementation of functions.

Listing 6: Functions using MPI communication can be templated on the communicated type using traits

```
template<typename T>
void globalSum(LSMSCommunication &comm,T &a)
{
  T r;
  MPI_Allreduce(&a,&r,1,TypeTraits<T>::
      mpiType(),MPI_SUM,comm.comm);
  a=r;
}
```

Here the use of traits (listing 7) provides the mechanism to obtain the type specific parameters that need to be passed to function calls thus leading to a significantly higher reusability of code and less code that needs to be maintained.

Listing 7: Type trait for generic functions using MPI

```
template<>
class TypeTraits<double>
{
public:
```

```
  inline static MPI_Datatype mpiType(void) {
      return MPI_DOUBLE;}
};
```

4.5 Matrix Inversion

The most computationally intensive part of the LSMS calculation is the matrix inversion to obtain the multiple scattering matrix τ. (eq. 1) The amount of computational effort can be reduced by utilizing the fact that for each local interaction zone only the left upper block (τ_{00}) of the scattering path matrix τ is required. LSMS uses an algorithm that reduces the amount of work needed while providing excellent performance due to its reliance on dense matrix-matrix multiplications that are available in highly optimized form in vendor or third party provided implementations (*i.e* `ZGEMM` in the BLAS library).

The method employed in LSMS to calculate the required block of the inverse relies on the well known expression for writing the invers of a matrix in term of inverses and products of subblocks:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} U & V \\ W & Y \end{pmatrix}$$

where

$$U = (A - BD^{-1}C)^{-1}$$

and similar expressions for V, W, and Y. This this method can be applied multiple times to the subblock U until the desired block τ_{00} of the scattering path matrix is obtained.

The Fortran based implementation of this algorithm for CPUs, that is used in all versions of LSMS, is shown in listing 8.

Listing 8: CPU version of the matrix block inversion function `zblock_lu`

```
      subroutine zblock_lu(a,lda,blk_sz,nblk
          ,ipvt,mp,idcol,k)
...
c Do block LU
      n=blk_sz(nblk)
      joff=na-n
      do iblk=nblk,2,-1
      m=n
      ioff=joff
      n=blk_sz(iblk-1)
      joff=joff-n
c invert the diagonal blk_sz(iblk) x blk_sz(
    iblk) block
      call zgetrf(m,m,a(ioff+1,ioff+1),lda,
          ipvt,info)
c calculate the inverse of above multiplying
     the row block
c blk_sz(iblk) x ioff
      call zgetrs('n',m,ioff,a(ioff+1,ioff
          +1),lda,ipvt,
     &     a(ioff+1,1),lda,info)
      if(iblk.gt.2) then
      call zgemm('n','n',n,ioff-k+1,na-ioff,
          cmone,a(joff+1,ioff+1),lda,
     &     a(ioff+1,k),lda,cone,a(joff+1,k),
         lda)
      call zgemm('n','n',joff,n,na-ioff,
          cmone,a(1,ioff+1),lda,
     &     a(ioff+1,joff+1),lda,cone,a(1,
         joff+1),lda)
      endif
      enddo
      call zgemm('n','n',blk_sz(1),blk_sz(1)
          -k+1,na-blk_sz(1),cmone,
     &     a(1,blk_sz(1)+1),lda,a(blk_sz(1)
         +1,k),lda,cone,a,lda)
      end
```

The code performs LU factorizations and linear solves by utilizing the Lapack routines `zgetrf` and `zgerts` respectively and the BLAS routine `zgemm` for the matrix multiplications. The reliance on these common library routines for the implementation of the main computational kernel of LSMS has ensured performance portability in the past, when moving between different CPU platforms. This approach is also enabling the easy port to accelerators if the necessary libraries are available. Listing 9 shows the implementation of `zblock_lu` using CULA [12]. The only changes needed were the replacement of the BLAS and LAPACK routines by their cuBLAS and CULA counterparts `cublas_zgemm`, `cula_device_zgetrf` and `cula_device_zgetrs` as well as the data movement to and from the device (`cublas_set_matrix` and `cublas_get_matrix`.

Performing these calculations using a different linear algebra library, such as Magma [13] only requires similar changes, enabling the easy comparison of other libraries and choosing the best performing library for a given combination of computer architec-

ture and physical simulation with a simple recompilation of the code.

Listing 9: GPU version of the matrix block inversion function `zblock_lu` using CULA[12]

```
      subroutine zblock_lu(a,lda,blk_sz,nblk
          ,ipvt,mp,idcol,k)
...
!     copy matrix to device
      info = cublas_set_matrix(lda, na,
          sizeof_Z, a, lda, devA, lda)
...
c Do block LU
      n=blk_sz(nblk)
      joff=na-n
      do iblk=nblk,2,-1
      m=n
      ioff=joff
      n=blk_sz(iblk-1)
      joff=joff-n
c invert the diagonal blk_sz(iblk) x blk_sz(
    iblk) block
      info = cula_device_zgetrf(m,m,
     &       devA+idx2f(ioff+1,ioff+1,lda)*
         sizeof_Z,lda,devIPVT)
c calculate the inverse of above multiplying
     the row block
c blk_sz(iblk) x ioff
      info = cula_device_zgetrs('n',m,ioff,
     &     devA+idx2f(ioff+1,ioff+1,lda)*
         sizeof_Z,lda,devIPVT,
     &     devA+idx2f(ioff+1,1,lda)*sizeof_Z
         ,lda)
      if(iblk.gt.2) then
      call cublas_zgemm('n','n',n,ioff-k+1,
          na-ioff,cmone,
     &     devA+idx2f(joff+1,ioff+1,lda)*
         sizeof_Z,lda,
     &     devA+idx2f(ioff+1,k,lda)*sizeof_Z
         ,lda,cone,
     &     devA+idx2f(1,joff+1,lda)*sizeof_Z
         ,lda)
      endif
      enddo
      call cublas_zgemm('n','n',blk_sz(1),
          blk_sz(1)-k+1,na-blk_sz(1),
     &     cmone,devA+idx2f(1,blk_sz(1)+1,
         lda)*sizeof_Z,lda,
     &     devA+idx2f(blk_sz(1)+1,k,lda)*
         sizeof_Z,lda,cone,devA,lda)

      info = cublas_get_matrix(lda,blk_sz(1)
          ,sizeof_Z,devA,lda,a,lda)
      end
```

5 Conclusion

Actively evolving research codes that explore new physics, such as the WL-LSMS demonstrated here, benefit from occasional rewrites not only for porting to new architectures, but these rewrites are an excellent opportunity to prune abandoned features in the code base and organize the code structure in a more transparent way that will enable the future viability of the code for new generations of scientists.

Acknowledgments

This work was conducted at Oak Ridge National Laboratory (ORNL), which is managed by UT-Battelle for the U.S. Department of Energy (US DOE) under contract DE-AC05- 00OR22725.

About the Author

M. Eisenbach is a computational scientist at the National Center for Computational Sciences at Oak Ridge National Laboratory. He is one of the developers of the LSMS code as well as the main author of the WL-LSMS hybrid code.

References

[1] Martin, R. M., *Electronic Structure: Basic Theory and Practical Methods*, Cambridge, 2004.

[2] F. Wang, D. P. Landau, *Phys. Rev. Lett.* **86**, 2050 (2001).

[3] Yang Wang, G. M. Stocks, W. A. Shelton, D. M. C. Nicholson, Z. Szotek, and W. M. Temmerman, Phys. Rev. Lett. **75** 2867 (1995).

[4] M. Eisenbach, B. L. Györffy, G. M. Stocks, and B. Újfalussy, Phys. Rev. B **65**, 144424 (2002).

[5] G. M. Stocks, Y. Wang, D. M. C. Nicholson, W. A. Shelton, Z. Szotek, W. M. Temmerman, B. N. Harmon, V. P. Antropov, Mater. Res. Soc. Symp. Proc. **408**, 157 (1996).

[6] G. M. Stocks, B. Újfalussy, X. Wang, D.M.C. Nicholson, W. A. Shelton, Y. Wang, A. Canning, and B. L. Györffy, Philos. Mag. B **78**, 665 (1998).

[7] B. Újfalussy, X. Wang, D.M.C. Nicholson, W. A. Shelton, G. M. Stocks, Y. Wang, and B. L. Györffy, J. Appl. Phys. **85**, 4824 (1999).

[8] C.-G. Zhou, T. C. Schulthess, S. Torbrügge, D. P. Landau, *Phys. Rev. Lett.* **96**, 120201 (2006).

[9] M. Eisenbach, C.-G. Zhou, D. M. Nicholson, G. Brown, J. Larkin, T. C. Schulthess, SC'09: Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2009.

[10] M. Eisenbach, C.-G. Zhou, D. M. Nicholson, G. Brown, J. Larkin, T. C. Schulthess, Proceedings of CUG 2010.

[11] `http://www.lua.org`

[12] `http://www.culatools.com`

[13] `http://icl.cs.utk.edu/magma`