

A Data Parallel C Benchmark for Car-Parrinello Molecular Dynamics

P. Jeffrey Ungar

Pacific-Sierra Research Corp., Santa Monica, CA 90405

Abstract

The Car-Parrinello (CP) *ab initio* molecular dynamics (MD) technique has become an increasingly important tool to study condensed phase systems at the atomic level over the past decade. In order to apply it to ever larger and more complex systems, researchers have turned to parallel computing platforms to meet the heavy computational and memory demands of this method. In this article we provide an overview of CPMD and then focus on the design and performance of a benchmark in Data Parallel C that includes an appropriate mix of its most time-consuming operations. In particular, we compare its execution performance to a C version of the code that uses explicit message passing as well as the effort required to write and to maintain both programs.

1. Introduction

Classical molecular dynamics (MD) emerged in the 1950's and 1960's along with the earliest electronic computers. The standard MD method, which is based on the use of empirical interatomic potential functions that have been fitted to experimental data, has proven its worth as a tool in the study of polymers, biomolecules, inorganic materials, and many other systems. In the last ten years, *ab initio* molecular dynamics, which uses interatomic forces computed directly from the electronic structure rather than requiring an empirical interaction potential as input, has opened entirely new avenues in the investigation of chemically complex environments.

In the Car-Parrinello (CP) MD method,^{1,2} the electronic structure is modeled using the Kohn-Sham formulation³ of density functional theory (DFT),⁴ and the Kohn-Sham orbitals are represented using a plane wave basis. The key to the CPMD approach is that the plane wave expansion coefficients are treated as fictitious dynamical variables that are propagated adiabatically with respect to the nuclei. The net effect is that they describe the instantaneous ground state Born-Oppenheimer surface at each time step, so the need to solve the Kohn-Sham equations explicitly is avoided. The CPMD strategy is versatile enough to be fruitfully applied to studies of condensed matter systems as wide ranging as the reorientational dynamics of white phosphorus,⁵ and proton transport in liquid water.^{6,7}

The benefits of CPMD come with a price, however, which in this case is the large demand it makes on CPU speed and memory capacity. As researchers have undertaken ever more ambitious simulations of materials they have turned to parallel computers to obtain the needed resources. Unfortunately, parallelizing their codes has added yet another layer of complexity to programs that already involve not only many lines of code but also fairly subtle physics and mathematics. Since a factor of two in performance can mean a reduction of weeks or even months of time to complete a simulation, most have opted to use explicit message passing (via MPI,⁸ for example) for maximum efficiency. In this article, we show that using Data Parallel C (DPC) it is possible to lighten the bookkeeping burden of parallelizing CPMD without significantly degrading runtimes. We have not constructed a complete CPMD application, but we have put together a prototype

benchmark that captures an appropriate mix of computations. Work is already in progress on a second version that will incorporate all the machinery used in parallelizing the actual application.

The rest of this paper is organized as follows: in Sec. 2 we briefly describe the CP simulation method. In Sec. 3 we discuss some parallelization strategies used for CPMD and present the benchmark prototype. Sec. 4 shows the performance of the DPC benchmark compared to a C implementation using MPI. The use of the DEEP (development environment for parallel programming) analysis tool to verify communication bottlenecks is also mentioned. Finally, Sec. 5 contains conclusions and further discussion.

2. Car-Parrinello Simulation

We give here a brief review of the CPMD method for the casual reader. Detailed reviews of the method and its application are available in the literature.^{2,9,10}

The CP scheme expresses the energy of a system with N valence electrons in the Kohn-Sham formulation of density functional theory as

$$E_{KS}[\{\phi_i\}, \{\mathbf{R}_I\}] = \sum_i \langle \phi_i | -\frac{1}{2} \nabla^2 + V_{NL} | \phi_i \rangle + \frac{1}{2} \iint d\mathbf{r} d\mathbf{r}' \frac{n(\mathbf{r})n(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} + E_{xc}[n] + \int dr V_{loc}(r)n(r) + U(\{\mathbf{R}_I\}) \quad \text{Eq. 1}$$

Here $n(\mathbf{r})$ is the spatial electron density, $\phi_i(\mathbf{r})$ ($i = 1 \dots N$) are single-particle orbitals for the electrons, E_{xc} is the exchange-correlation energy, and $U(\{\mathbf{R}_I\})$ is the ion-ion Coulomb interaction energy. The action of the ions on the electronic degrees of freedom is represented above by a pseudopotential with a local part

$$V_{loc}(\mathbf{r}) = \sum_I v_{loc}^I(\mathbf{r} - \mathbf{R}_I) \quad \text{Eq. 2}$$

and a nonlocal part, V_{NL} . For an all-electron model, this is zero and

$$V_{loc}(\mathbf{r}) = \sum_I -Z_I / |\mathbf{r} - \mathbf{R}_I| \quad (\text{all-electron}) \quad \text{Eq. 3}$$

The terms thus express the total energy as a sum of the electronic kinetic energy, the electron-electron electrostatic energy, the exchange-correlation energy, the ion-electron energy, and the ion-ion electrostatic energy.

The orbitals and the electron density are expanded in a plane wave basis by representing all quantities on a grid in a periodic cell. This allows very efficient computations using fast Fourier transform (FFT) algorithms to compute convolution integrals, and so on. On the other hand, the rapidly varying core states need a prohibitive number of plane waves for accurate representation, so for practical purposes pseudopotentials are used to integrate out these oscillations. The idea is to construct a potential for which the calculated wave functions match the all-electron wave functions outside the ion cores, but vary smoothly inside the core regions. It is this construction that results in a potential containing the nonlocal operators that are implicit in V_{NL} . Various schemes have been developed to ensure good transferability to different environments and to maximize smoothness.

The equations of motion for CPMD are generated through the use of a Lagrangian that has been extended to include the electronic wave functions together with the atomic coordinates. The Lagrangian takes the form

$$L = \mu \sum_i \langle \dot{\phi}_i | \dot{\phi}_i \rangle + \frac{1}{2} \sum_I M_I \mathbf{R}_I^2 - E_{KS}[\{\phi_i\}, \{\mathbf{R}_I\}] \quad \text{Eq. 4}$$

where M_I are the masses of the ions (nuclei). The Lagrangian is subject to the orthonormality constraints

$$\langle \phi_i | \phi_j \rangle = \delta_{ij} \quad \text{Eq. 5}$$

which are incorporated via Lagrange multipliers Λ_{ij} leading to the equations of motion

$$\begin{aligned} \mu |\ddot{\phi}_i\rangle &= -\frac{\delta E_{KS}}{\delta \langle \phi_j |} + \sum_j \Lambda_{ij} |\phi_j\rangle \\ M_I \ddot{\mathbf{R}}_I &= -\frac{\partial E_{KS}}{\partial \mathbf{R}_I} \end{aligned} \quad \text{Eq. 6}$$

The parameter μ is a fictitious mass that is selected to be small enough so that the electronic parameters adjust to changing ionic coordinates quickly. Under appropriate conditions, these equations of motion will then be a good approximation to Born-Oppenheimer dynamics in which the forces on the ions are calculated using the instantaneous ground state wave functions.

Although we will not describe specific forms, key to the success of the Kohn-Sham formulation is the use of a local approximation to the exchange correlation functional $E_{xc}[n]$, which makes the evaluation of it and the corresponding exchange-correlation potential fast. Even the simplest approximations have been found to yield surprisingly good bond lengths and vibration frequencies for covalent or metallic bonding. More sophisticated functionals are available that also give good descriptions of weakly bonded systems.

3. CPMD Benchmark

Even though many details were glossed over, from the preceding overview of the CPMD method it is clear that the machinery required to assemble a production CP code is very involved. The fundamental code to evaluate the functionals and forces in the equations of motion is the tip of the iceberg. At a minimum, one also needs an atomic code to generate pseudopotentials, special code to determine a fully-relaxed initial state, MD integration algorithms that accommodate constraints and have checks on conserved quantities and temperature regulation, and much more. Fortunately, there is no need to implement a full-

blown CPMD code to make a meaningful benchmark if we can abstract the most important computations. With knowledge of the technique from the top down, we can construct a benchmarking test bed from the *bottom up*.

Essentially, the most demanding parts of CPMD simulation boil down to two kinds of operations: 3-d FFT's and orbital algebra. The FFT's arise from transforming to whichever basis provides the fastest computation at each stage in the calculation. The orbital manipulations include linear transformations and matrix element (inner product) calculations. In the following, we describe the representation used for the orbitals and the density, and show the sequence of operations required to calculate the total energy and perform an MD step. We then discuss the operations included in the benchmark to model the actual application.

Operations

All continuous quantities in the CPMD simulation are, in principle, represented as values on a discrete 3-d spatial grid. The periodic boundary conditions imposed on the MD supercell mean that the single-particle Kohn-Sham orbitals can be expanded in plane waves according to the Bloch's theorem as

$$\phi_j^{\mathbf{k}}(\mathbf{r}) = e^{i\mathbf{k}\cdot\mathbf{r}} \sum_{\mathbf{g}} c_j^{\mathbf{k}}(\mathbf{g}) e^{i\mathbf{g}\cdot\mathbf{r}} \quad \text{Eq. 7}$$

The \mathbf{r} 's range over the discrete spatial grid points and the \mathbf{g} 's are the set of reciprocal lattice vectors corresponding to the "real" space grid. The wave vector \mathbf{k} labels a point within the Brillouin zone (BZ) of the reciprocal lattice. The sum is truncated by only including plane waves up to a maximum kinetic energy $E_{cut} \geq \frac{1}{2}|\mathbf{k} + \mathbf{g}|^2$. The pseudopotentials used to represent the core-valence interactions in large part determine the minimum cutoff energy that will yield acceptable results. The net effect of E_{cut} is to set the minimum dimensions of the grid along the axes of the periodic cell. To adequately represent the orbital wave functions this would typically mean a grid in the 24×24×24 to 40×40×40 range, but all quantities related to the electron *density* require twice this resolution since the density is the square of the wave functions:

$$\rho(\mathbf{r}) = \sum_{j,\mathbf{k}} w_{\mathbf{k}} f_{j,\mathbf{k}} |\phi_j^{\mathbf{k}}(\mathbf{r})|^2 \quad \text{Eq. 8}$$

Here $f_{j,\mathbf{k}}$ are occupation numbers for the orbitals, and $w_{\mathbf{k}}$ are weights for integrating over the BZ. At this stage, we will only explicitly consider the $\mathbf{k} = 0$ point, since that is often selected for finite temperature simulations where crystal lattice symmetry is broken, and it has little effect in the design of the benchmark.

The following table is adapted from Galli and Parrinello²:

FOURIER SPACE		REAL SPACE	
$\{ c_i^{\mathbf{k}}(\mathbf{g}) \}$	$\xrightarrow{N \text{ FFT}}$	$\{ \phi_i^{\mathbf{k}}(\mathbf{r}) \}$	
\downarrow			
$\sum_{\mathbf{g},i,\mathbf{k}} \mathbf{k} + \mathbf{g} ^2 c_i^{\mathbf{k}}(\mathbf{g})$	E_K		\downarrow
V_{NL}^{PS}	E_{PS}^{NL}		
$\tilde{\rho}(\mathbf{g})$	$\xleftarrow{\text{FFT}}$	$\rho(\mathbf{r})$	
\downarrow			
$\tilde{V}_H(\mathbf{g})$	E_H		\downarrow
$+$			
$\tilde{V}_L^{PS}(\mathbf{g})$	E_L^{PS}	$V_{xc}(\mathbf{r})$	E_{xc}
\downarrow		$+$	
$\tilde{V}_{LH}(\mathbf{g})$	$\xrightarrow{\text{FFT}}$	$V_{LH}(\mathbf{r})$	
			\downarrow
		$V_{LOC}(\mathbf{r})$	

$$\begin{array}{ccc}
[V_{LOC} \circ c_i^{\mathbf{k}}](\mathbf{g}) & \xleftarrow{N \text{ FFT}} & \{V_{LOC}(\mathbf{r})\phi_i^{\mathbf{k}}(\mathbf{r})\} \\
\downarrow & & \\
\partial E / \partial c_i^{\mathbf{k}}(\mathbf{g})^* = \hat{H}c_i^{\mathbf{k}}(\mathbf{g}) & &
\end{array}$$

It shows the sequence of steps involved in the calculation of the total energy and the forces acting on the $c_i(\mathbf{g})$ coefficients for the orbitals. Some of the calculations are most efficiently performed in real space, while the rest are simplest in Fourier space. In the table, E_K is the electronic kinetic energy; V_{NL}^{PS} and E_{NL}^{PS} are the nonlocal pseudopotential and corresponding energy; V_L^{PS} and E_L^{PS} are the local pseudopotential and pseudopotential energy; V_H and E_H are the electronic Hartree (electrostatic) potential and energy; V_{LOC} is the combined local potential; and \hat{H} represents the complete Hamiltonian operator (a matrix). The only truly time-consuming operations not present in the table but required to perform an MD time step are those necessary to satisfy the orthonormality constraints on the wave functions. In most CPMD integration schemes, the overlap matrix

$$O_{ij} = \langle \phi_i | \phi_j \rangle = \sum_{\mathbf{g}} c_i^*(\mathbf{g})c_j(\mathbf{g})$$

or a related quantity will be computed. The computations involved in evaluating the nonlocal pseudopotential as they are usually implemented (via a separable form that is nonlocal in the angular part) involve similar effort.

Design

On the whole, much of the time required to perform a CPMD step is spent performing 3-d FFT's to calculate the density and the net forces. The rest is spent calculating the nonlocal pseudopotential terms, overlaps, or similar quantities. Therefore, we can construct a

useful benchmark by concentrating on these operations provided we keep the strategies used in the real application in mind.

Typical CPMD codes exploit the spherical cutoff for the orbital wave functions yet still use a spatial grid with enough resolution to represent the electron density. Operations on the orbitals in Fourier space are only calculated for the $c(\mathbf{g})$'s within the limit, which is about 6.5% of the number of points on the spatial grid. Furthermore, simulations using $\mathbf{k} = 0$ need only store half of these since we can choose purely real orbital wave functions, which means $c(-\mathbf{g}) = c^*(\mathbf{g})$. In any case, the net effect is to speed up the overlap and nonlocal pseudopotential calculations, and to require fewer 1-d FFT's to transform orbitals to and from real space. This last advantage results from embedding the orbital coefficients on the dense, zero-filled spatial grid, and only transforming sequences containing nonzero data. Note that one can also pack *two* real orbitals together for transformation.

For the present benchmark prototype, we do not use all of these techniques, but we do try to maintain the balance between real space and Fourier space computations. For example, this version does not use a spherical cutoff for the orbitals, but does use a grid (containing the $c(\mathbf{g})$'s) half as dense as the density grid. This is about a factor of two less efficient. On the other hand, we use a source code FFT that is about 2 to 3 times slower than that found in an optimized library, so this mostly compensates. We also use complex orbitals rather than real ones, but not being able to transform two orbitals at a time is balanced by performing overlaps in Fourier space with twice the number of coefficients that real orbitals would require.

The prototype benchmark does the following: it initializes a set of orbitals to random normalized states, performs Gram-Schmidt orthonormalization and checks the overlap matrix, transforms all orbitals to real space to calculate the density, and finally transforms them back while checking the total charge (which *should* be the same!). After the initialization, this set corresponds to the kind and the balance of computations done in a single CPMD step. In order to approximately match a CPMD production run, we use 100 occupied valence orbitals and a spatial grid for the density of $64 \times 64 \times 64$ points (which means $32 \times 32 \times 32$ for each wave function).

Parallelization Strategies

There are two independent data decomposition strategies that can be used to parallelize CPMD: spatial slab decomposition and orbital distribution. We consider these in turn.

The spatial slab decomposition in parallel CPMD splits up the grids used for the electron density and the Fourier space coefficients along a single dimension. As an example, Figure 1 illustrates how we may distribute the $64 \times 64 \times 64$ density grid across 4 processors, each owning a $16 \times 64 \times 64$ slab. (The pieces owned by each processor are labeled with P_n .) For the purposes of the prototype benchmark, the orbital coefficients on the $32 \times 32 \times 32$ grid would then be split up into $8 \times 32 \times 32$ slabs. This slab decomposition gives very good scaling for all computations occurring purely on either grid.

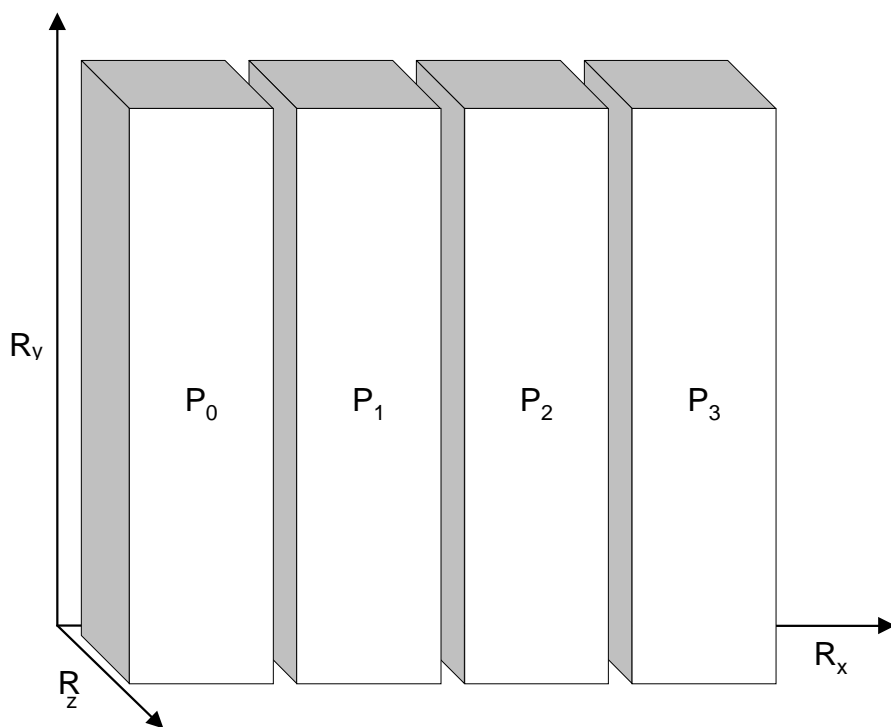


Figure 1. Distribution of the real space grid along the x-axis

The challenge entailed by this strategy is in transforming data from one representation to the other. In order to transform a function from the real space grid to Fourier space, we

first do the FFT's along y and z since these directions are local to each processor. Then the data must be redistributed to make the x -axis local so that all the FFT's along x can be done. Figure 2 shows a possible distribution of the Fourier space grid across processors.

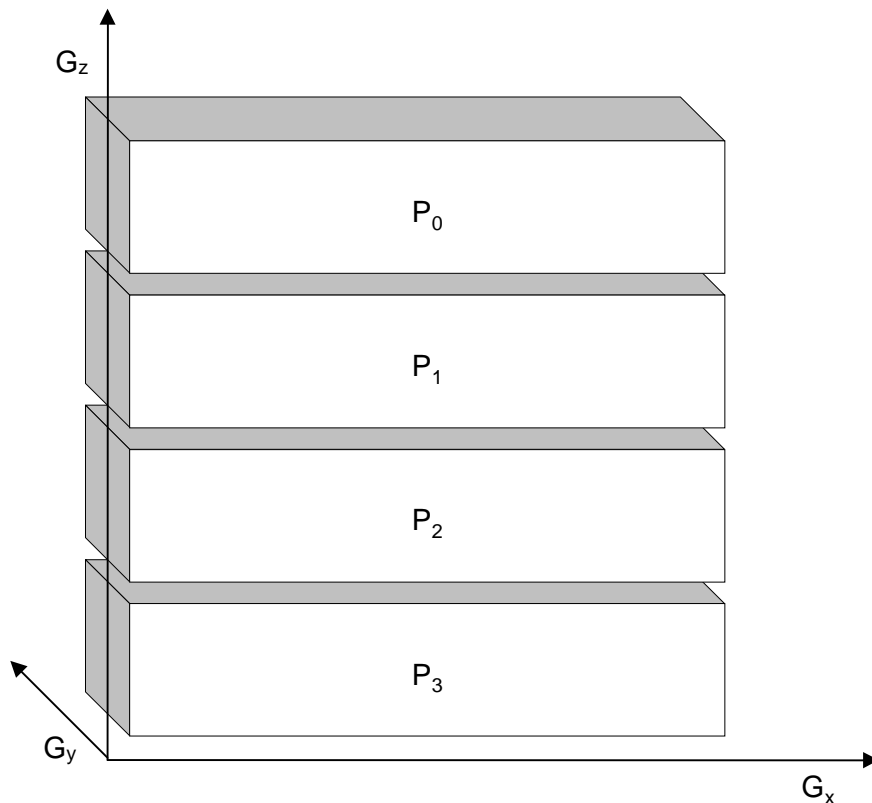


Figure 2. Distribution of Fourier space along z -axis.

The communication overhead of transposing the data across processors limits the scaling efficiency of this method as the number of processors is increased for constant problem size. The practical limit is in the neighborhood of 16 processors, depending on the system simulated and the computing platform.

The other parallelization strategy is to distribute the orbitals themselves across processors while keeping the spatial grids local. Since no nonlocal transposition is required, all the FFT operations are now completely parallelized. For example, Figure 3 shows 8 orbitals split across 2 processors. Each processor will only need to compute FFT's for the orbitals it owns. On the other hand, calculations like the overlap matrix or nonlocal pseudopotential terms will now require data motion to bring in orbitals owned

by other processors. Nevertheless, this strategy is competitive with the first, giving acceptable scaling up to 8–16 processors in many situations.

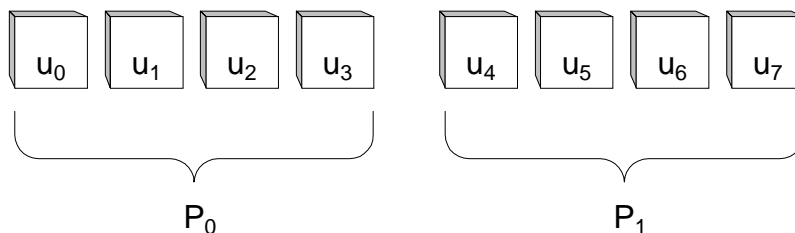


Figure 3. Orbital distribution.

The key to scaling up to larger numbers of processors is to realize that these two strategies are completely orthogonal to one another; that is, we can employ *both* at the same time, and achieve the product of the performance gains.

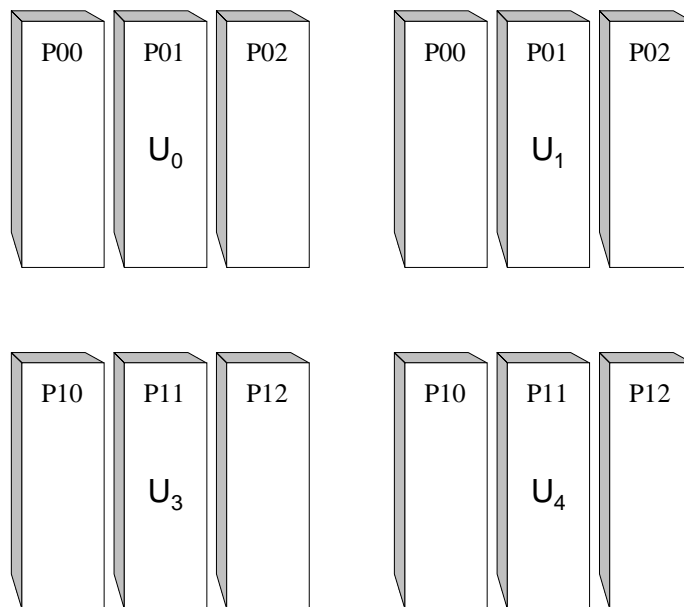


Figure 4. Combined spatial and orbital distribution.

For example, we could distribute orbitals across groups of processors with the spatial grids being split into slabs across the members of each group. A case with 4 orbitals

distributed across 2 groups of 3 processors is shown in Figure 4. (The processor owning each part of each orbital is labeled on the figure as Pmn , where m and n denote the processor row and column, respectively.) This two-fold parallelization makes it possible to use well over 100 processors with good efficiency.

Prototype Benchmark Implementation in DPC

The current version of our benchmark uses the spatial slab decomposition since this is probably the most common single level of parallelization in existing CPMD codes. The power of the Data Parallel C language actually makes it fairly easy to extend this to combined orbital-slab distribution, so it was also the logical first step to take. We now show DPC code fragments that illustrate how the benchmark calculation is set up and performed.

DPC shapes can be used to set up the fundamental grids used in the benchmark as shown in the following fragment:

```
#include <dpce.h>
...
#define NGX      32          /* orbital grid x-dimension */
#define NGY      32          /* orbital grid y-dimension */
#define NGZ      32          /* orbital grid z-dimension */
#define NRX      (2*NGX)    /* density grid x-dimension */
#define NRY      (2*NGY)    /* density grid x-dimension */
#define NRZ      (2*NGZ)    /* density grid x-dimension */
#define NO       100        /* number of complex orbitals */
/*
 * set up a complex type
 */
typedef struct {
    double re;
    double im;
} dcomplex;
/*
 * shapes of parallel variables for real and Fourier space grids, and
 * intermediate grids.
 */
shape [NGZ] [NGX] [NGY] OrbGrid;      /* orbital coefficient grid */
shape [NRX] [NRY] [NRZ] RhoGrid;     /* electron density grid */
shape [NRX] [NRZ] [NRY] FS1Grid;     /* 23-transposed grid */
shape [NRX] [NGZ] [NGY] FS2Grid;     /* packed 23-transposed grid */
shape [NGZ] [NRX] [NGY] FS3Grid;     /* 12-transposed FS2Grid */
/*
 * Global arrays
 */
dcomplex:OrbGrid Phi[NO];             /* NO complex orbitals in g-space */
double:RhoGrid Rho;                  /* total electron density in real-space */
```

Distributing the data across 4 processors on the first dimension is now as simple as compiling with VAST/DPC with `-Pprocs4:1:1`. Initializing the orbitals to random states is straightforward using DPC parallel intrinsics and parallel variable syntax. For example,

```
#include <dpce.h>
#include <stdlib.h>
#include <math.h>
...
const int nG=NGX*NGY*NGZ;
...
for ( i=0; i<NO; i++ ) {
    double a;
    phg[i].re = ((double)prand(OrbGrid))/RAND_MAX-0.5;
    phg[i].im = ((double)prand(OrbGrid))/RAND_MAX-0.5;
    phg[i].re -= ( +=phg[i].re )/nG;
    phg[i].im -= ( +=phg[i].im )/nG;
    a = sqrt(1/(+= phg[i].re*phg[i].re+phg[i].im*phg[i].im));
    phg[i].re *= a;
    phg[i].im *= a;
}
```

Note that the loop shown above is one over orbitals *only*. Each statement involving the parallel variable `phg` with shape `OrbGrid` actually implies operation on all $32 \times 32 \times 32$ elements. This code initializes the real and imaginary parts of each orbital to zero-mean random numbers, then normalizes them. Gram-Schmidt orthonormalization (to simulate typical orbital calculations) can be programmed as

```
dcomplex:OrbGrid ug;
for ( i=0; i<NO; i++ ) {
    dcomplex c[NO];
    double a;
    ug = phg[i];
    for ( j=0; j<i; j++ ) {
        c[j].re=0;
        c[j].im=0;
    }
    for ( j=0; j<i; j++ ) {
        c[j].re = (+= phg[j].re*ug.re + phg[j].im*ug.im);
        c[j].im = (+= phg[j].re*ug.im - phg[j].im*ug.re);
    }
    for ( j=0; j<i; j++ ) {
        ug.re -= (c[j].re*phg[j].re - c[j].im*phg[j].im);
        ug.im -= (c[j].re*phg[j].im + c[j].im*phg[j].re);
    }
    a = 1/sqrt(+= ug.re*ug.re + ug.im*ug.im);
    phg[i].re = a*ug.re;
    phg[i].im = a*ug.im;
}
```

Once again the loops are only over the orbitals; parallel variable syntax implicitly handles all the elements at once. Each orbital is taken in turn and projected onto all preceding orbitals (if any). These projections are subtracted from the orbital and the result is normalized. The overlap matrix calculation is similar in that it is composed of inner products of all pairs of orbitals.

Getting the net charge density in real space requires transforming each orbital and accumulating its contribution:

```
static dcomplex:RhoGrid ur;
double q;

rh = 0;
for ( i=0; i<NO; i++ ) {
    transGtoR(phg[i],ur);
    rh += ur.re*ur.re+ur.im*ur.im;
}
q = ( += rh);
printf("----- net charge is: %e\n",q/nR);
```

For the purposes of the benchmark, we actually transform each orbital to real space and back to get the correct FFT operations count (but use the transformed one to add to the density, of course). The code to transform an orbital from Fourier space to the density grid involves several steps. First the incoming orbital is unpacked onto an intermediate grid and transformed on the middle (unpacked) index:

```
void transGtoR(dcomplex:OrbGrid ug,dcomplex:RhoGrid ur)
/*****
/* transform g-space orbital (ug) to r-space (ur) on the density
  grid.
*/
{
    static dcomplex:RhoGrid ugy1;
    static dcomplex:FS1Grid ugy2,ugyz1;
    static dcomplex:FS2Grid ugyz2;
    static dcomplex:FS3Grid ugyz3,ug1;
    ...
    /* unpack x */
    ug1.re = 0;
    ug1.im = 0;

    [0:] [0:NGX/2-1] [0:]ug1 = [0:] [0:NGX/2-1] [0:]ug;
    [0:] [NRX-NGX/2:] [0:]ug1 = [0:] [NGX/2:] [0:]ug;
    ... /* <inverse FFT along x> */
```

Transposition of the first two indicies, which is nonlocal assuming distribution on the first index, may be accomplished with a single statement:

```
[pcoord(FS3Grid,1)] [pcoord(FS3Grid,0)] [pcoord(FS3Grid,2)]ugyz2 = ugyz3;
```

Unpacking the result leaves two purely local FFT steps plus a local transpose to reach the desired result:

```
/* unpack y and z */
[0:] [0:NGZ/2-1] [0:NGY/2-1]ugyz1 = [0:] [0:NGZ/2-1] [0:NGY/2-1]ugyz2;
[0:] [0:NGZ/2-1] [NRY-NGY/2:]ugyz1 = [0:] [0:NGZ/2-1] [NGY/2:]ugyz2;
[0:] [NRZ-NGZ/2:] [0:NGY/2-1]ugyz1 = [0:] [NGZ/2:] [0:NGY/2-1]ugyz2;
[0:] [NRZ-NGZ/2:] [NRY-NGY/2:]ugyz1 = [0:] [NGZ/2:] [NGY/2:]ugyz2;

... /* <inverse FFT along z> */

/* transpose 1-2 */
[pcoord(FS1Grid,0)] [pcoord(FS1Grid,2)] [pcoord(FS1Grid,1)]ugy1 = ugy2;

... /* <inverse FFT along y> */
}
```

4. Performance Results

We tested the code for 100 orbitals on a $32 \times 32 \times 32$ Fourier space grid ($64 \times 64 \times 64$ for the real space density) on two different platforms: a 6-processor 75 MHz R8000 SGI Power Challenge, and a Fujitsu VPP300/5. The shared memory version of MPICH provided the underlying interface for the DPC communication support library on the Power Challenge. Fujitsu's MPI port was used on the VPP300, although VAST/DPC can also use Fujitsu's native message passing library, MPLib.

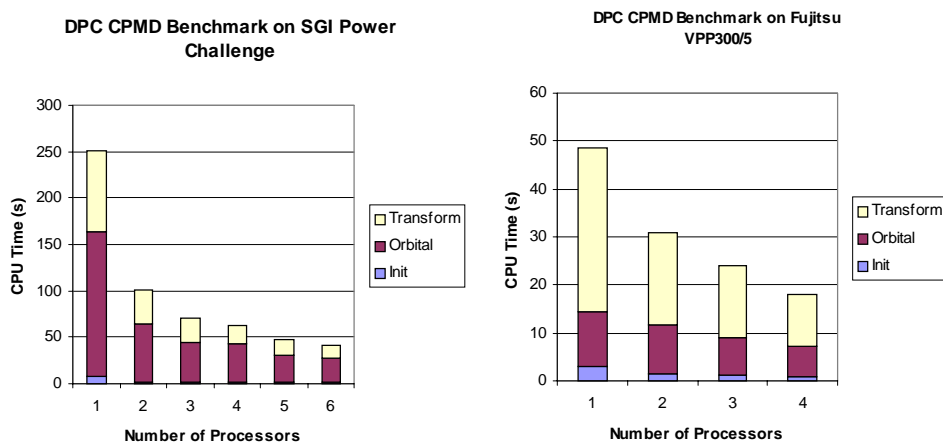


Figure 5. Timings for DPC benchmark.

Figure 5 shows the CPU time required to perform a single step using an increasing number of processors. Each bar is divided into the time spent doing transformations and the time doing orbital operations. The times on the Power Challenge are a little biased in favor of the orbital operations—a production CPMD code usually spends 50–75% of its time doing FFT's when running on one processor—but the balance is close enough to give useful information. On the VPP300, the balance is closer to the real application.

The scaling on the Power Challenge is excellent with the DPC benchmark achieving a speed increase of a factor of 6.0 on 6 processors (see Figure 6). Some of the results for fewer processors actually exhibit superlinear behavior due to better cache usage as a result of the distribution of the arrays. The scaling on the VPP300 is not as ideal, but still a respectable 2.8 on 4 processors. Curiously, the orbital operations scale worse than the FFT's. Inspection of the generated C code shows that some vector

efficiency has been lost due to the insertion of calls to do sum reductions across processors.

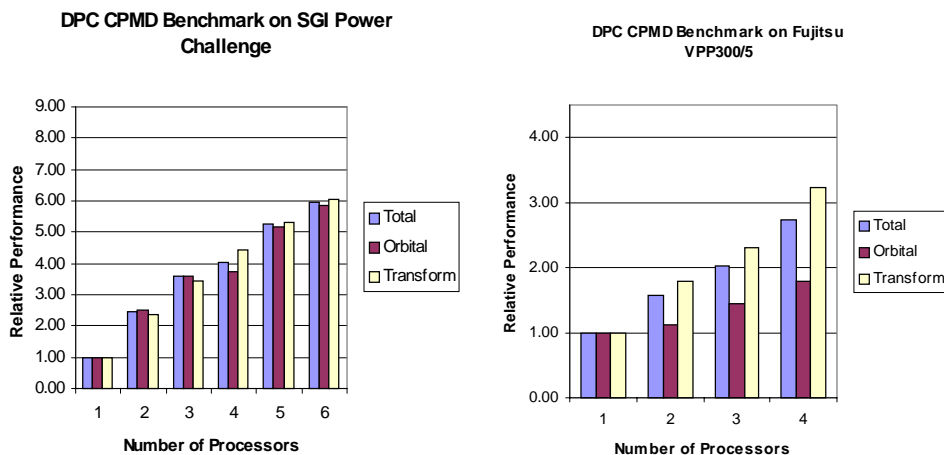


Figure 6. Scaling of DPC benchmark.

In order to allow for a fair assessment of performance, we also tested a version of the benchmark in C with explicit MPI calls. The timings and scaling are shown in Figure 7 and Figure 8, respectively.

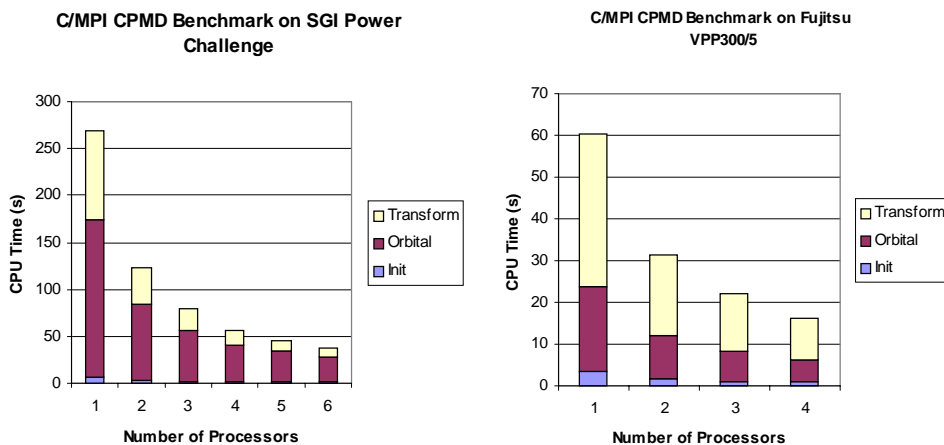


Figure 7. Timings for C/MPI benchmark.

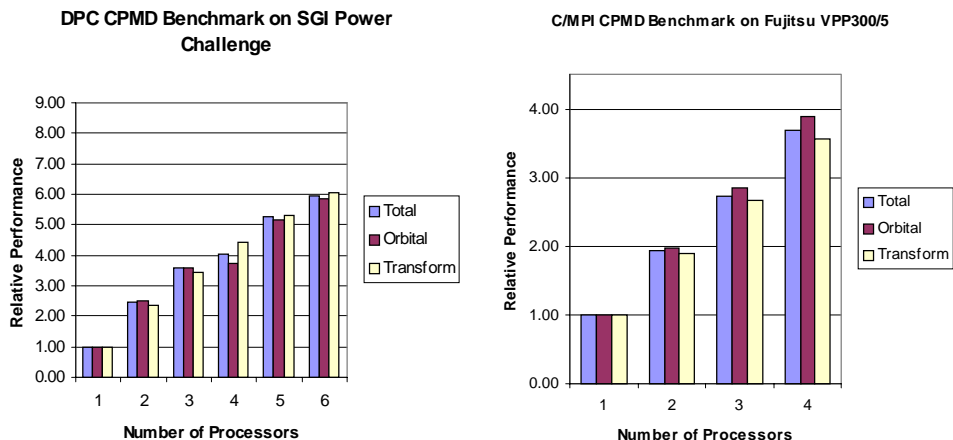


Figure 8. Scaling for C/MPI benchmark.

Overall, the timings are rather similar to the DPC results, but the scaling definitely shows superlinear behavior on the Power Challenge, particularly for the FFT operations. This results primarily from cache effects that are not washed out by communication overhead, which is very small for the C/MPI code. It is difficult at this point to draw any general conclusions comparing this aspect of the scaling to the DPC results, since there are rather complicated interactions between the C translation produced by VAST/DPC and the SGI C compiler. The scaling on the VPP300 is much closer to ideal than the DPC code with the C/MPI program speeding up by a factor of 3.7 on 4 processors. Nevertheless, the direct comparison of execution times for both platforms shown in Figure 9 is, perhaps, more to the point.

In spite of the better scaling of the C/MPI implementation, the race for performance is pretty much a dead heat across the range. (The VAST/DPC code is actually faster than the C/MPI program over half the range due to better cache usage on the Power Challenge, and better vectorization on the VPP300.) That leaves the communication overhead as the determining factor, and most of this is concentrated in the nonlocal transposition used to do the 3-d FFT. For 6 processors on the Power Challenge, this overhead amounts to 30% and 17% of the transformation time for the DPC and the C/MPI programs, respectively. The corresponding figures for 4 processors on the VPP300 are 8% and 24%. Consequently, the DPC program is likely to scale well to about 8

processors, while the C/MPI code should reach the practical limit of ~16 processors for the spatial slab scheme.

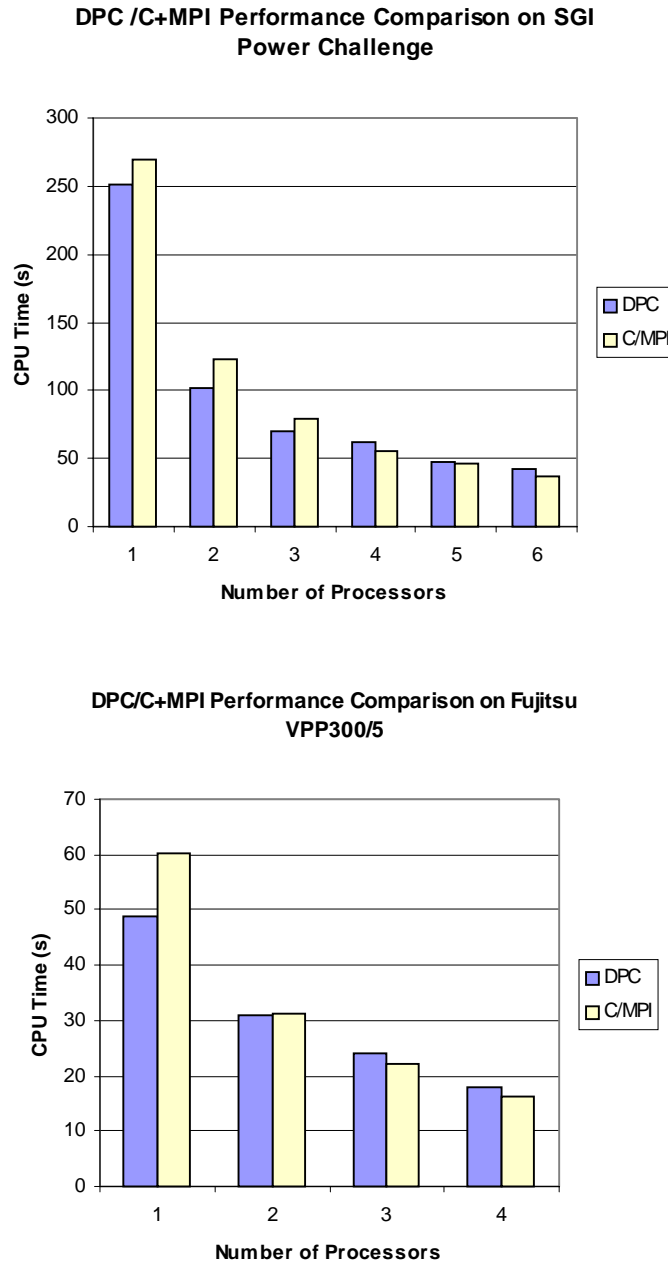


Figure 9. Comparison of the performance of the DPC benchmark to C with explicit MPI calls.

Static program analysis and execution profiling using DEEP confirmed that the nonlocal transpose is the primary limiting operation. The DPC compiler succeeded in scheduling exchanges of partial arrays, but had not aggregated these pieces into a single message per process first. We view this partly as an oversight since with a slight change in syntax it had no trouble generating a call to an internal library pack-and-exchange routine. Even without this capability, DEEP analysis allows the user to identify the hot spot and either program the communication by hand in a DPC nodal function, or call a supplied library routine to do the job more efficiently.

5. Discussion and Conclusions

The execution performance of the prototype DPC CPMD benchmark is competitive with MPI at least up to 6 processors. Furthermore, DEEP analysis of the benchmark and experience with production CPMD codes show that the communication bottleneck for the spatial slab scheme is the global transpose required to transform to and from Fourier space. This is true for the C/MPI benchmark as well. We have found that using a supplied library routine to perform the transpose essentially eliminates any real performance difference between the DPC and C/MPI implementations. Both should then achieve the typical practical performance limit of about 16 processors for the 1-d slab decomposition.

Execution performance is not the only criterion for comparison, however. A prime advantage of programming in DPC instead of C with explicit message passing is the ease of implementation and maintenance. For a programmer familiar with the application, it may take a day or two to assemble and debug the DPC benchmark. The C/MPI version, which at about 900 lines is *twice* as long, is likely to take a week or more assuming that a hand-coded transposition routine is *already* available.

Regarding this last point, one need only consider the difference between expressing a transpose in DPC as

```
[pcoord(S,1)] [pcoord(S,0)] [pcoord(S,2)] ut = u;
```

and using C plus MPI , which in about 70 lines uses all available MPI machinery to do a fast transpose (but is specific to the 1-d distribution on the first index). The bookkeeping required by the MPI program is a source of complication, and therefore, *bugs* that are notoriously difficult to find and fix in parallel programs. Furthermore, the specific distribution must be hardwired into the code at the beginning, while recompiling may be all that is necessary to change the decomposition in a DPC program. As an added benefit the parallel variable notation in DPC, which is reminiscent of Fortran 90 array operations, simplifies the expression of many numerical algorithms.

We should also point out that the DPC program is at least as portable as using C plus MPI. Either one can run on any platform that has a C compiler and supports MPI, but VAST/DPC can also use PVM or native message passing libraries on certain platforms.

Although the results obtained with the prototype CPMD benchmark in DPC and C/MPI are informative, there is room for many improvements short of implementing a production code. All of these are designed to make the benchmark as close to the “real thing” as possible, but no closer. The two main categories for improvements are parallelization strategy and node program optimization.

For the first, we need to implement the combined orbital and spatial slab decomposition to push the scalability to the limit. The second will mean using optimized vendor-supplied library routines for FFT's and some high-level BLAS, which will in turn require nodal functions to provide an appropriate interface. Using these libraries is expected to speed up the node programs by a factor of 2–3. This will make the communication overhead that much more important, but no more so for DPC than for C/MPI, and we are confident that we can achieve nearly equivalent execution speeds.

There are also some minor details needed to simulate the spherical wave vector cutoff used for the orbital Fourier coefficients, to take advantage of real orbitals, and so on. At that point very little will need to be added to convert the benchmark from a synthetic code representative of the application to a working component of a real CPMD simulation program.

Acknowledgments

Access to the computing resources at Prof. Michael Klein's Center for Molecular Modeling in the Department of Chemistry at the University of Pennsylvania and at Fujitsu America, Inc. is gratefully acknowledged.

References

- ¹ R. Car and M. Parrinello. *Phys. Rev. Lett.* **55**, 2471 (1985).
- ² G. Galli and M. Parrinello. *In Computer Simulations in Materials Science. Edited by M. Meyer and V. Pontikis. Kluwer, Dordrecht. 1991. p. 283.*
- ³ W. Kohn and L.J. Sham. *Phys. Rev.* **140**, A1133 (1965).
- ⁴ P. Hohenberg and W. Kohn. *Phys. Rev.* **136**, B89 (1964).
- ⁵ P.J. Ungar, K. Laasonen, and M.L. Klein. *Can. J. Phys.* **73**, 710 (1995).
- ⁶ M.E. Tuckerman, K. Laasonen, M. Sprik, and M. Parrinello. *J. Chem. Phys.* **103**, 150 (1995).
- ⁷ M.E. Tuckerman, K. Laasonen, M. Sprik, and M. Parrinello. *J. Phys. Chem.* **99**, 5749 (1995).
- ⁸ M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. *MPI: The Complete Reference. MIT Press, Cambridge. 1996.*
- ⁹ D.K. Remler and P.A. Madden. *Mol. Phys.* **70**, 921 (1990).
- ¹⁰ M.C. Payne, M.P. Teter, D.C. Allan, T.A. Arias, and J.D. Joannopoulos. *Rev. Mod. Phys.* **64**, 1045 (1992).