

OAK RIDGE  
NATIONAL LABORATORY

MANAGED BY UT-BATTELLE  
FOR THE DEPARTMENT OF ENERGY

# FY 2010 Annual Report of GPRA-PMM Software Metric SC GG 3.1/2.5.2, Improve Computational Science Capabilities

September 2010



  
UT-BATTELLE

ORNL-27 (4-00)



**FY 2010 Annual Report of GPRA-PMM Software Metric SC GG 3.1/2.5.2,  
Improve Computational Science Capabilities**

Date Published: September 2010

Prepared for  
U.S. Department of Energy  
Office of Science  
Advanced Scientific Computing Research Program







## CREDITS

### Application Credits

#### *TD\_SLDA*

Aurel Bulgac, Kenneth J. Roche (University of Washington)

URL: <http://www.phys.washington.edu/groups/qmbnt/index.html>

#### *POP*

Phil Jones, Mathew Maltrud (Los Alamos National Laboratory)

URL: <http://climate.lanl.gov/Models/POP/>

#### *LS3DF*

Lin-Wang Wang (Lawrence Berkeley National Laboratory)

URL: <http://hpcrd.lbl.gov/~linwang/PEtot/PEtot.html>

#### *Denovo*

Thomas M. Evans (Oak Ridge National Laboratories)

URL: <http://www.ornl.gov/sci/scale/>

### Technical Team

#### *TD\_SLDA*

Hai Ah Nam (Oak Ridge National Laboratory)

#### *POP*

Trey White, Pat Worley, Ilene Carpenter (Oak Ridge National Laboratory)

#### *LS3DF*

Markus Eisenbach (Oak Ridge National Laboratory)

#### *DENOVO*

Wayne Joubert (Oak Ridge National Laboratory)

### Additional Credits

Kenneth J. Roche (University of Washington, Pacific Northwest National Laboratory)

Rebecca Hartman-Baker, Ricky Kendall, Douglas B. Kothe (Oak Ridge National Laboratory)

### DOE Program Contacts

Barbara Helland ([helland@ascr.doe.gov](mailto:helland@ascr.doe.gov))

Daniel Hitchcock ([daniel.hitchcock@science.doe.gov](mailto:daniel.hitchcock@science.doe.gov))

Lucy Nowell ([lucy.nowell@science.doe.gov](mailto:lucy.nowell@science.doe.gov))

Michael Strayer ([michael.strayer@science.doe.gov](mailto:michael.strayer@science.doe.gov))

Christine Chalk ([christine.chalk@science.doe.gov](mailto:christine.chalk@science.doe.gov))

### Additional Contacts

Rebecca Hartman-Baker ([hartmanbaktj@ornl.gov](mailto:hartmanbaktj@ornl.gov))

Douglas B. Kothe ([kothe@ornl.gov](mailto:kothe@ornl.gov))

Kenneth J. Roche ([k8r@u.washington.edu](mailto:k8r@u.washington.edu); [kenneth.roche@pnl.gov](mailto:kenneth.roche@pnl.gov))



# CONTENTS

## TABLE OF CONTENTS

Credits .....	iii
Contents .....	v
List of Tables .....	ix
List of Figures .....	xii
ABBREVIATED TERMS.....	1
<b>1. Metric Statement for Computational Effectiveness .....</b>	<b>3</b>
1.1 GPRA-PMM Metrics.....	3
1.2 FY10 GPRA-PMM Goals for the DOE ASCR Program .....	3
1.3 Quarterly Tasks Related to SC GG 3.1/2.5.2 .....	4
<b>2. Metric Results for Computational Effectiveness .....</b>	<b>5</b>
2.1 Target HPC System: jaguarpf.ccs.ornl.gov .....	5
2.2 Results Summary .....	5
2.2.1 TD-SLDA .....	6
2.2.2 POP .....	6
2.2.3 LS3DF .....	Error! Bookmark not defined.
2.2.4 Denovo .....	9
2.3 Conclusions .....	10
<b>3. Overview of Computational Science Capabilities and Analysis of Metric Results .....</b>	<b>12</b>
<b>3.1 TD_SLDA.....</b>	<b>13</b>
3.1.1 Introduction.....	13
3.1.2 Background and Motivation .....	13
3.1.3 Capability Overview .....	15
3.1.4 Science Driver for Metric Problem .....	15
3.1.5 The Model and Algorithm.....	16
3.1.6 Q2 Baseline Problem Results.....	19
3.1.7 Developments, Enhancements, and Q4 Benchmark Results.....	29
3.1.8 SLDA Summary of Results.....	36
<b>3.2 POP .....</b>	<b>39</b>
3.2.1 Introduction.....	39
3.2.2 Background and Motivation .....	39
3.2.3 Capability Overview .....	39
3.2.4 Science Driver for Metric Problem .....	41
3.2.5 The Model and Algorithm.....	41
3.2.6 Q2 Baseline Problem Results.....	42
3.2.7 Q4 Code Improvements.....	43
3.2.8 Q4 Metric Problem Results.....	43
3.2.9 Analysis.....	44
3.2.10 Summary and Conclusions .....	45
<b>3.3 LS3DF .....</b>	<b>46</b>
3.3.1 Introduction.....	46

3.3.2	Background and Motivation .....	46
3.3.3	Capability Overview .....	47
3.3.4	Science Driver for Metric Problem .....	49
3.3.5	The Model and Algorithm .....	51
3.3.6	Q2 Baseline Problem Results .....	52
3.3.7	Computational Performance Gains .....	55
3.3.8	Q4 Metric Problem Results .....	56
3.3.9	Interpretation of Results .....	56
3.3.10	Summary and Conclusions .....	57
<b>3.4</b>	<b>Denovo .....</b>	<b>59</b>
3.4.1	Introduction .....	59
3.4.2	Background and Motivation .....	59
3.4.3	Capability Overview .....	61
3.4.4	Science Driver for Metric Problem .....	62
3.4.5	The Model and Algorithm .....	63
3.4.6	Q2 Baseline Problem Results .....	66
3.4.7	Scalable Multigroup Transport Algorithms .....	67
3.4.7.1	Within-Group Solvers .....	68
3.4.7.2	Multigroup Solvers .....	68
3.4.7.3	Eigenvalue Solvers .....	69
3.4.8	Additional Code Optimizations .....	69
3.4.9	Q4 Problem Results .....	69
3.4.10	Conclusions .....	71
<b>4.</b>	<b>REFERENCES .....</b>	<b>72</b>
<b>Appendices: Benchmark Problem Environments .....</b>		<b>1</b>
<b>APPENDIX A.</b>	<b>Overview .....</b>	<b>A-1</b>
<b>A.1</b>	<b>Modules Available on the Target Architecture .....</b>	<b>A-2</b>
A.1.1	Modules Available in Q2 .....	A-2
A.1.2	Modules available in Q4 .....	A-7
<b>A.2</b>	<b>Machine Event Data Collection Routines .....</b>	<b>A-17</b>
A.2.1	KRP Machine Event Data Collection API .....	A-17
A.2.2	KRP Use Example .....	A-19
<b>APPENDIX B.</b>	<b>TD_SLDA .....</b>	<b>1</b>
<b>B.1</b>	<b>Input Settings .....</b>	<b>2</b>
<b>B.2</b>	<b>Compilation .....</b>	<b>3</b>
B.2.1	Q2 Compilation .....	3
B.2.2	Q4 Compilation .....	5
<b>B.3</b>	<b>Batch Scripts .....</b>	<b>7</b>
B.3.1	Q2 Batch Scripts .....	7
B.3.2	Q4 Batch Scripts .....	8
<b>B.4</b>	<b>Runtime Environment .....</b>	<b>9</b>
B.4.1	Q2 Runtime Environment .....	9
B.4.2	Q4 Runtime Environment .....	9
<b>APPENDIX C.</b>	<b>POP .....</b>	<b>11</b>
<b>C.1</b>	<b>Input Settings .....</b>	<b>12</b>

C.2	Compilation.....	13
C.3	Batch Script .....	14
C.4	Runtime Environment .....	15
<b>APPENDIX D. LS3DF.....</b>		<b>16</b>
D.1	Input Settings .....	17
D.2	Compilation.....	21
D.3	Batch Script .....	24
D.4	Runtime Environment .....	25
<b>APPENDIX E. Denovo.....</b>		<b>29</b>
E.1	Input Settings .....	30
E.2	Compilation .....	35
E.3	Batch Script .....	37
E.4	Runtime Environment.....	38

## LIST OF TABLES

Table 1. FY10 GPRA-PMM software summary of Q2 baseline performance simulations and data ..	11
Table 3.1.1: Results of instrumentation of Run 1 of the first benchmark problem. ....	20
Table 3.1.2: Results of instrumentation of Run 2 of the first benchmark problem. ....	21
Table 3.1.3: Results of instrumentation of Run 3 of the first benchmark problem. ....	22
Table 3.1.4: Total cost of unitary fermion gas system Q2 benchmark. ....	22
Table 3.1.5: Results for unitary Fermi gas system benchmark problem. ....	23
Table 3.1.6: Measured machine events for solver runs of second benchmark problem. ....	28
Table 3.1.7: Performance of time-dependent nuclear code on second benchmark problem. ....	28
Table 3.1.8: Aggregated performance results of second benchmark problem. ....	28
Table 3.1.9: Selected observable data computed (or input) by the nuclear solver or computed by the time-dependent nuclear code and reported after the last timestep in the second benchmark run. ....	29
Table 3.1.10: Performance results of Q4 238U benchmark solver problem. ....	35
Table 3.1.11: Performance results of Q4 238U benchmark time-dependent problem. ....	35
Table 3.1.12: Aggregated performance results of the Q4 238U solver and time-dependent benchmark runs. ....	36
Table 3.1.13: Selected observable data computed (or input) by the nuclear solver or computed by the time-dependent nuclear code and reported after the last timestep in the Q4 benchmark run. ....	36
Table 3.2.1: POP timers for the benchmark at three core counts. ....	42
Table 3.2.2: POP timers for the benchmark at three core counts with Q4 I/O improvements. ....	<b>Error!</b>
<b>Bookmark not defined.</b>	
Table 3.3.1: Benchmarking results for varying process counts and configurations. ....	54
Table 3.3.2: Timings of subroutines within code. ....	55
Table 3.3.3: Total LS3DF computation times (for 20 initial iterations plus 40 SCF iterations), flops, and percentage of performance to the theoretical limit. $N_g$ is the number of fragment groups, $N_c$ is the number of cores in each band group, and $N_b$ is the number of band groups. Within each fragment group, there are $N_c \times N_b$ processors, and $N_{tot} = N_g \times N_c \times N_b$ . ....	56
Table 3.3.4: The computational time for subroutine PEtot_F for each self-consistent field (SCF) iteration (containing 4 CG or 4 DIIS steps). The times for the other three subroutines (Gen_VF, Gen_dens, and Poisson) are unchanged from the Q2 results, and remain unchanged for the different methods in PEtot_F and different combinations of $N_g$ , $N_c$ , and $N_b$ . After averaging out the fluctuations, the times for Gen_VF, Gen_dens, and Poisson are 6, 9, and 22 seconds, respectively. Thus in total they account for 37 seconds for each SCF iteration. ....	56
Table 3.4.1: PWR900 core dimensions and configuration. ....	62
Table 3.4.2: Discretization of the Q2 problem. ....	66
Table 3.4.3: Wall clock timing measurements. All times are in minutes. The solver contains the within-group and two-grid times. The vast majority (99.3%) of the runtime is spent doing transport sweeps. ....	67
Table 3.4.4: Computational iteration costs for the Q2 benchmark problem. ....	67
Table 3.4.5: PAPI instrumentation for Q2 problem. ....	67
Table 3.4.6: Discretization of the Q4 problem. It has 22 times more DoF than the Q2 problem. ....	70
Table 3.4.7: Performance cases run during Q3 and Q4. The Q2 baseline is given as case 0 (see Tables 3.4.3 – 3.4.5.) ....	70
Table 3.4.8: Results of set 1 problem runs. % Peak is referenced to a maximum peak efficiency on Jaguar of 4 FP/Cycle/Core. ....	70
Table 3.4.9: Results of set 2 problem runs. % Peak is referenced to a maximum peak efficiency on Jaguar of 4 FP/Cycle/Core. ....	71





## LIST OF FIGURES

Figure	Page
Figure 3.1.1: Error after each self-consistent iteration in computing the first Q2 benchmark problem.	20
Figure 3.1.2: Energy of 300 particle system on $32^3$ lattice as function of time.	23
Figure 3.1.3: Eight instances from the 3-D movie made with VisIt of excitation of the stirring with ball and rod of a fermion unitary gas system.	25
Figure 3.1.4: Strong scaling study comparing parallel QR and parallel Cuppen's algorithms for fixed small Hermitian problem, $n=4096$ .	26
Figure 3.1.5: Execution time vs. Number of PEs for three problems for nuclear solver portion of software.	27
Figure 3.1.6: Response of 48Cr to Coulomb field generated by relativistic projectile.	29
Figure 3.1.8: Snapshots from unitary gas vortex formation calculation of the ball and rod excitation of the 300 particle dilute Fermi gas system executed in Q2. We have added a high resolution movie capability since Q2 to our analysis tools to be used for either unitary gas or nuclear systems. Currently we use VisIt to export JPEG files and then these are imported into Apple's iMovie software. We export MPEG-4 videos with annotation.	34
Figure 3.1.7: Raw solver machine event data (protons and neutrons).	35
Figure 3.1.8: Raw time-dependent code machine event data.	35
Figure 3.1.9: Ratios used in comparison of Q2 and Q4 solver performance.	36
Figure 3.1.10: Ratios used in comparison of Q2 and Q4 time-dependent code performance.	37
Figure 3.2.1: A tripole grid in which the geographic pole has been displaced into two continents.	40
Figure 3.2.2: Temperature at 15 m depth from a global 0.1-degree simulation using POP.	41
Figure 3.2.3: Raw Q2 data for POP run on 4096 processors.	<b>Error! Bookmark not defined.</b>
Figure 3.2.4: Raw Q4 data for POP run on 4096 processors.	<b>Error! Bookmark not defined.</b>
Figure 3.2.5: Ratios derived from POP raw data for performance analysis.	44
Figure 3.3.1: Program flow chart for conventional LDA method (left) and LS3DF (right).	48
Figure 3.3.2: Scalability of LS3DF vs. LDA.	48
Figure 3.3.3: Parallelization of problem domain.	49
Figure 3.3.4: Weak scaling of LS3DF on three leadership computing resources.	49
Figure 3.3.5: The benchmark ZnO nanorod and some of its fragments.	50
Figure 3.3.6: Schematic of LS3DF subdivision of domain space.	52
Figure 3.3.7: Convergence of the LS3DF method for ZnO nanorod.	53
Figure 3.3.8: Self-consistent potential along the center axis of ZnO nanorod.	53
Figure 3.4.1: Three levels of reactor geometries.	60
Figure 3.4.2: PWR900 $17 \times 17$ pin fuel assembly. The pins have been homogenized into 45 unique materials in each assembly. The labels show the material ids in a LEU assembly. Material ids cover the ranges [1,45] (LEU), [46, 90] (MEU), and [91, 135] (HEU). All assemblies have the same $\frac{1}{4}$ symmetry pattern.	63
Figure 3.4.3: PWR900 core model: (a) 2D radial view ( $xy$ -plane), and (b) 3D view showing axial ( $z$ -axis) geometry. The assembly enrichments are LEU (light blue), MEU (red/blue), and HEU (yellow/orange).	63
Figure 3.4.4: Close-up view of the Denovo spatial grid in the radial ( $xy$ ) plane. Each homogenized pin cell has a $2 \times 2$ spatial grid of $0.63 \times 0.63$ cm. The each cell has a 0.6 cm resolution in the axial ( $z$ ) direction.	66
Figure 3.4.5: Expanded solver taxonomy in Denovo.	67
Figure 3.4.6: Denovo multilevel energy decomposition. Energy is decomposed in sets and space-angle is decomposed in blocks.	68





## ABBREVIATED TERMS

2D, 3D, ...	two dimensional, three dimensional, ...
ALCF	Argonne Leadership Computing Facility
AMD	Advanced Micro Devices (computer chip company)
API	application programming interface
ASCAC	Advanced Scientific Computing Advisory Committee
ASCR	Advanced Scientific Computing Research (DOE program)
BCS	Bardeen-Cooper-Schrieffer (superfluid)
BdG	Bogoliubov-de Gennes
BEC	Bose-Einstein condensate
BLAS	basic linear algebra subroutines
BRLCAD	Ballistic Research Laboratory computer-aided design program (software)
CCSM	Community Climate System Model
CG	conjugate gradient method (solver)
CLE	Cray Linux Environment
CPU	central processing unit
DD	domain decomposition; diamond difference
DDLZ	diamond difference with linear-zero flux fixup
DDR2	double data-rate two
DFT	density functional theory
DIIS	direct inversion of the iteration space
DIMM	dual inline memory module
DOE SC	DOE Office of Science
DOE	U.S. Department of Energy
DoF	degree of freedom
DSA	diffusion synthetic acceleration
DVR	discrete variable representation
EDF	Electricité de France
eV	electron volt(s) (unit of energy)
F90, F95	Fortran 90, Fortran 95
FFT	fast Fourier transform
FFTW	fastest Fourier transform in the West (implementation of FFT)
FMO	fragment molecular orbital
FSM	folded spectrum method
FY	fiscal year
flop	floating point operation
GB	gigabyte(s)
GMRES	generalized minimal residual method (solver)
GNU	“GNU’s not Unix!” (open-source software)
GPRA	Government Performance and Results Act of 1993
GSL	GNU scientific libraries
HDF5	hierarchical data format version 5
HEU	highly enriched uranium
HPC	high-performance computing
I/O	input/output
IEEE	Institute of Electrical and Electronics Engineers
INCITE	Innovative and Novel Computational Impact on Theory and Experiment (DOE program)
KBA	Koch-Baker-Alcouffe
KPP	<i>k</i> -profile parameterization

L2, L3	level-two, level-three (cache)
LANL	Los Alamos National Laboratory
LAPACK	linear algebra package
LD	linear-discontinuous finite element
LDA	local density approximation
LEU	low enriched uranium
LS3DF	linearly scaling three-dimensional fragment method
MB	megabyte(s)
METIS	multilevel partitioning algorithms (software)
MEU	medium enriched uranium
MeV	megaelectron volt(s)
meV	millielectron volt(s)
MPI	Message Passing Interface Standard
NCAR	National Center for Atmospheric Research
NCCS	National Center for Computational Sciences
NERSC	National Energy Research Scientific Computing Center
NSTD	Nuclear Science and Technology Division (subunit at ORNL)
OMB	U.S. Office of Management and Budget
OLCF	Oak Ridge Leadership Computing Facility
ORNL	Oak Ridge National Laboratory
OpenMP	Open Message Passing (shared memory programming standard)
OS	operating system
PAPI	Performance Application Programming Interface
PART	Performance Assessment Rating Tool
PB	petabyte(s)
PE	processing element
PGI	Portland Group
POP	Parallel Ocean Program
POSIX	portable operating system interface [for Unix] (IEEE standard)
PWR	pressurized water reactor
Q1, Q2, ...	Quarter 1, Quarter 2, ...
QR	orthogonal matrix decomposition
QRPA	quasiparticle random phase approximations
R&D	research and development
SC	step characteristics (slice balance)
SCF	self-consistent field
SciDAC	Scientific Discovery through Advanced Computing (DOE program)
SILO	data storage and exchange library
SPRNG	scalable parallel pseudorandom number generator
SuperLU	library for direct solution of systems of linear equations
SWIG	simplified wrapper and interface generator
TD-SLDA	time-dependent superfluid local density approximation
TLD	trilinear-discontinuous finite element
TWD	theta-weighted diamond difference
UNEDF	universal nuclear energy density functional (collaboration)

# 1. METRIC STATEMENT FOR COMPUTATIONAL EFFECTIVENESS

## 1.1 GPRA-PMM METRICS

ASCR's GPRA-PMM Software Metric for Computational Effectiveness is designed to comply with Public Authorizations PL 95-91, "Department of Energy Organization Act," and PL 103-62, "Government Performance and Results Act."

The U.S. Office of Management and Budget (OMB)\* oversees the preparation and administration of the President's budget; evaluates the effectiveness of agency programs, policies, and procedures; assesses competing funding demands across agencies; and sets the funding priorities for the federal government. The OMB has the power of audit and exercises this right annually for each federal agency. According to the Government Performance and Results Act of 1993 (GPRA), federal agencies are required to develop three planning and performance documents:

1. Strategic Plan: a broad, 3 year outlook;
2. Annual Performance Plan: a focused, 1 year outlook of annual goals and objectives that is reflected in the annual budget request (*What results can the agency deliver as part of its public funding?*); and
3. Performance and Accountability Report: an annual report that details the previous fiscal year performance (*What results did the agency produce in return for its public funding?*).

OMB uses its Performance Assessment Rating Tool (PART) to perform evaluations. PART has seven worksheets for seven types of agency functions. The function of Research and Development (R&D) programs is included. R&D programs are assessed on the following criteria:

- Does the R&D program perform a clear role?
- Has the program set valid long term and annual goals?
- Is the program well managed?
- Is the program achieving the results set forth in its GPRA documents?

In Fiscal Year (FY) 2003, the Department of Energy Office of Science (DOE SC-1) worked directly with OMB to come to a consensus on an appropriate set of performance measures consistent with PART requirements. The scientific performance expectations of these requirements reach the scope of work conducted at the DOE national laboratories. The Joule system emerged from this interaction. In FY09 Joule was renamed PMM. PMM, or GPRA-PMM herein, enables the chief financial officer and senior DOE management to track annual performance on a quarterly basis. GPRA-PMM scores are reported as "success, goal met" (*green light* in PART), "mixed results, goal partially met" (*yellow light* in PART), and "unsatisfactory, goal not met" (*red light* in PART). GPRA-PMM links the DOE strategic plan† to the underlying base program targets.

## 1.2 FY10 GPRA-PMM GOALS FOR THE DOE ASCR PROGRAM

The DOE Advanced Scientific Computing Research (ASCR)‡ program has the following two annual performance measures as part of its PART requirements:

1. SC GG 3.1/2.5.1—Focus usage of the primary supercomputer at the National Energy Research Scientific Computing Center (NERSC) on capability computing, defined as the percentage of the computing time used by computations that require at least 1/8 of the total resource. FY10 performance metric: capability usage is at least 40%.

---

\* <http://www.whitehouse.gov/omb>

† <http://www.er.doe.gov/about/MissionStrategic.htm>

‡ <http://www.sc.doe.gov/ascr/About/about.html>

2. SC GG 3.1/2.5.2—Improve computational science capabilities, defined as the average annual percentage increase in the computational effectiveness (either by simulating the same problem in less time or simulating a larger problem in the same time) of a subset of application codes. FY10 performance metric: efficiency measure is  $\geq 100\%$ .

Ensuring compliance with these metrics, which are tracked on a quarterly basis, is an important milestone each fiscal year for the DOE ASCR Program Office as well as for the success of the overall DOE SC-1 open science computing effort. This document details the results of the effectiveness of the computational science capability (SC GG 3.1/2.5.2).

### 1.3 QUARTERLY TASKS RELATED TO SC GG 3.1/2.5.2

The GPRA-PMM effort to improve computational science capabilities is a yearlong effort requiring quarterly updates. The quarterly sequence of tasks for exercising this software metric is as follows.

**Quarter One (Q1) Tasks** (deadline: December 31). Identify a subset of candidate applications (scientific software tools) to be investigated on DOE SC supercomputers. Management (at DOE SC and national laboratories) decides upon a short list of applications and computing platforms to be exercised. The Advanced Scientific Computing Advisory Committee (ASCAC) approves or rejects the list. The Q1 milestone is satisfied when a short list of target applications and machines (supercomputers) is approved.

**Quarter Two (Q2) Tasks** (deadline: March 31). Problems that each chosen application must simulate on the target machines are determined. The science capability (simulation result) and computational performance of the implementation are benchmarked and baselined (recorded) on the target machines for the defined problems and problem instances. The Q2 milestone is satisfied when benchmark data—namely the machine operation count, execution time, and machine instance—is collected and explained. If an application is striving to achieve a new science result in addition to demonstrating improved performance, then providing a detailed discussion of its current (prior to Q2) capability, a discussion of why the capability is insufficient, and a description of why the new capability being developed satisfy the Q2 milestone.

**Quarter Three (Q3) Tasks** (deadline: June 30). The application software (its models, algorithms, and implementation) is enhanced for efficiency, scalability, science capability, etc. The Q3 milestone is satisfied when the status of each application is reported at the Q3 deadline. Corrections to Q2 problem statements are normally submitted at this time.

**Quarter Four (Q4) Tasks** (deadline: September 30). Enhancements to the application software continue as in Q3. The enhancements are stated and demonstrated on the machines used to generate the Q2 baseline information. A comparative analysis of the Q2 and Q4 data is summarized and reported. The Q4 milestone is satisfied if the enhancements made to the application software are in accordance with the efficiency measure as defined in Q2 (run-time efficiency, scalability, or new result).

## 2. METRIC RESULTS FOR COMPUTATIONAL EFFECTIVENESS

Each application is discussed and its baseline and metric problem described in the respective application sections. A brief description of the machine used for the application problems is given. A summary of measured results for each application is provided.

### 2.1 TARGET HPC SYSTEM: JAGUARPF.CCS.ORNL.GOV

The Cray XT5 high-performance computing (HPC) leadership system, Jaguar/XT5, at the Oak Ridge National Laboratory (ORNL) Leadership Computing Facility (OLCF) is used to exercise the DOE ASCR FY10 GPRA-PMM software metric.

Jaguar/XT5 has a total of 18,688 XT5 compute nodes or 224,256 processing elements (PEs). These dual-socket compute nodes are six-core AMD Opteron™ “Istanbul” chips operating at 2.6 GHz with 16 gigabytes (GB) of unbuffered memory per node, 6 megabytes (MB) of shared, 48-way associative L3 cache per chip, 512 kilobytes (KB) of 16-way associative L2 cache per core, and 64 KB instruction and 64 KB data two-way associative L1 caches per core. Each socket employs double data-rate two (DDR2) dual inline memory modules (DIMMs) at 800 MHz with, in the best case, 25.6 GB/s of local memory bandwidth per node.

Installed on the compute nodes is the Cray Linux Environment (CLE) version 2.2. CLE is a lightweight operating system (OS) designed to minimize the layers of OS between the application and the hardware. CLE supports many parallel programming models, including MPI 2.0 and OpenMP, the two models used by the applications in this report. Application developers have access to a variety of compilers and libraries, including five different C, C++, and Fortran programming environments (Cray, GNU, Intel, Pathscale, and PGI); highly optimized computing libraries, such as BLAS, LAPACK, and the Cray Scientific Libraries; and internal instrumentation libraries, such as PAPI, which was used to collect the machine event data for the metrics described in this report. A complete list of the software, which is accessible to users in the form of modules, is reproduced in the Appendix.

Jaguar/XT5 has 192 input/output (I/O) and login/service nodes. Each of these nodes consists of a 2.6 GHz dual-core AMD Opteron™ chip with 8 GB of memory per node. The I/O and service nodes are running a variant of SuSE Linux. Approximately 10 petabytes (PB) of disk space are available in the scratch file systems that support massive I/O parallelism through the Lustre file system software.\* HyperTransport links all nodes to Cray’s proprietary SeaStar2+ chips, which are used to construct a three-dimensional torus communication network between nodes. There are six switch ports per Cray SeaStar2+ chip, and each port has a bandwidth of 9.6 GB/s. The best-case bandwidth between the compute node and the SeaStar2+ interconnect chip is 6.4 GB/s. Thus, the injection bandwidth is half this, or 3.2 GB/s.

For further information, the NCCS website† describes the system and its software stack and is sufficiently detailed for the purposes of this report. For information on the Cray XT5 platform, see the Cray website.‡ For more information on the AMD Istanbul chip set, see the presentation by Brian Waldecker at the NCCS website.§

### 2.2 RESULTS SUMMARY

The FY10 studies aim to demonstrate *strong scaling*, where the problem complexity for an application is fixed and the time to execute the instance is reduced by demonstrating effective utilization

---

\* <http://www.lustre.org>

† <http://www.nccs.gov/computing-resources/jaguar/>

‡ <http://www.cray.com/Assets/PDF/products/xt/CrayXT5Brochure.pdf>

§ [http://www.nccs.gov/wp-content/uploads/2009/06/CrayORNL\\_120709.pdf](http://www.nccs.gov/wp-content/uploads/2009/06/CrayORNL_120709.pdf)

of an increased hardware allocation, or *weak scaling*, where the goal is to compute in the same wall-clock time a more complex problem on an increased hardware allocation (i.e., maintaining fixed work per processing element). In lieu of *or* in addition to these modes of enhancement, the process of computing a particular algorithm may be enhanced for *efficiency*, where the time to execute a fixed problem is reduced on a fixed hardware allocation. Application developers often develop an efficiency metric that is relevant to their problem to report enhancements –such as the rate of computing math operations on floating point numbers or the rate an external device such as the file system can be effectively utilized.

The *program binary* (a compiled/loaded executable constructed from the application source code) is the instantiation of the problem on the target machine, and the *computational complexity* of each problem instance is deduced directly by monitoring the values of the various program counters for the various functional units (e.g., floating point operations) activated during program execution. In other words, the required resources define the complexity of the problem and the work conducted to actually execute it. This measure of work is fairly basic from the hardware perspective and can be derived from system observables such as the number of processing elements (PEs) dedicated to executing the program, execution time, total number of instructions executed, the magnitude of the memory demand, etc.

In the following sections, each of the FY10 applications (TD-SLDA, POP, LS3DF, and Denovo) is introduced, and their computational effectiveness metric problem results and conclusions are summarized. A more detailed examination of each application can be seen in Section 3.

### 2.2.1 TD-SLDA

The time-dependent superfluid local density approximation (TD-SLDA) is the time dependent extension of the density functional theory (DFT) for superfluid systems. TD-SLDA can be applied to study a host of fundamental problems in the physics of fermionic superfluids: i) dynamics of nuclear large amplitude collective motion (including induced fission); ii) the dynamics of vortices in the neutron star crust and the elucidation of the starquake mechanism; iii) the generation and dynamics of vortices and the study of the quantum turbulence in fermionic superfluids; and iv) the study of dynamics of cold gases in various external conditions (static and time-dependent magnetic fields/Feshbach resonance, static and time-dependent optical lattices, reactions to various laser fields, etc.). Due to the very large number of coupled nonlinear partial differential equations, the equations in 3D can be solved (SLDA) and evolved (TD-SLDA) only on leadership computing resources.

Both the unitary Fermi gas and nuclear software capabilities were benchmarked in Q2 (see the SLDA section on Q2 benchmarks for details). While both sets of codes were under continual development this year, the nuclear solver and time evolution codes became the focus of our Q3 and Q4 enhancement efforts as we deemed the unitary codes to be in good shape for production. The goal of the nuclear software is to be capable of modeling systems in spatial domains from  $40^3$ - $75^3$  fm<sup>3</sup> and for times up to almost an attosecond. The total number of nucleons can reach thousands in some systems of interest.

In Q2, the capability to pass solutions computed by the nuclear solver software to the time dependent software was tested for the nucleus 198W on a  $40^3$  spatial lattice with 0.75fm lattice constant within a 100MeV energy cutoff. The solver executed on 73,728 PEs, and converged and wrote 16,412 (~64GB) quasi-particle wave functions to represent the ground state of the system after 6538.524s. The time dependent nuclear code executed on 16,414 PEs, read and distributed the static solutions and observables, executed 200 successful time steps, and exited after 2084.424s.

The Q4 problem was considerably more complex than the Q2 problem. Ground state solutions to 238U on a 40x40x64 spatial lattice with 1.25fm lattice constant and within the same 100MeV cutoff were constructed. The Q4 solver utilized 217,800 PEs, and converged and wrote 136,626 quasi particle wave functions (~834GB) after 18393.181s. The Q4 time dependent code executed on 136,628 PEs, read and distributed the solution data, evolved the system 200 time steps, and exited after 2031.541s.

For the solver and time dependent codes, both the number of mathematical operations on floating point numbers and the rate of executing writes and reads on a file in the Lustre file system are critical factors to the performance of the SLDA codes. In the Q2 solver,  $5.358236026051104e+16$  FP\_OPs were

executed at the rate 111,150,055 FP / s / PE. In the Q4 solver,  $9.416633279080292e+17$  FP\_OPs were executed at the rate 235,078,792 FP / s / PE. The enhanced code executed 17.575 times more floating point operations than the original solver and achieved 211.4967% of the weak scaling efficiency for floating point operations relative to the Q2 problem. In Q2, 64GB of solution data was constructed and the rate of conducting the assembly and writes of solution data was measured as 47.2013 MBps. In Q4 834GB of solution data was constructed and the rate that the new assembly and write routine achieved was rate 3482.5963 MBps. Thus, the Q4 solver constructed over 13 times as much solution data and the new I/O routine is nearly 75 times faster than the Q2 I/O algorithm for the problems tested. In a nutshell, the enhanced time dependent code read and distributed over 13 times more solution data, executed on just over 8 times the number of processes, computed nearly 12 times more floating point operations, evolved the same number of time steps as the Q2 code -all in slightly less time! While the rate of executing floating point operations remained essentially constant between Q2 and Q4, the rate of conducting I/O and distribution of solutions was over 41 times faster in Q4. We report a hyper-weak result for the time dependent code based on the ratio  $T(Q4) / T(Q2) := 1.026$ . The Q2 the solver utilized 32.87% of the entire machine, whereas in Q4 we exercised 97.12% of the total system. The Q2 time dependent code executed on  $\sim 7.32\%$  of the entire system, whereas the Q4 time dependent code executed on  $\sim 61\%$  of the system.

There were several major changes to the SLDA software. The solver was modified to form the entire BdG matrix prior to diagonalization (rather than in staged diagonalizations) and was redesigned to utilize a parallel write over the Lustre file system that imposes a transformation of the decomposed solutions to their reference global indexing scheme as required by the time dependent code. The enhanced time dependent code was modified to correct the lack of particle conservation. It was determined that all terms in the functional that contained an odd power of the gradient needed to be symmetrized on the lattice basis. While this step significantly increased the amount of mathematical operations on each wave function each time step, it improved the quality of the results restoring numerical conservation between the solver and time dependent codes as well as between time steps. To help counteract the increased number of operations per wave function, coefficients to derivatives within each time step per wave function were stored instead of being recomputed in Q4 where this was possible. The I/O routines were rewritten to perform optimized parallel reads and writes to files in the Lustre file system. Although not compared as part of the metric, a parallel check-point and restart capability was completed and tested in the Q4 nuclear code enabling unprecedented studies of long physical durations

### 2.2.2 POP

The Parallel Ocean Program (POP) [1] is an ocean general circulation model used for ocean and climate studies, available both as a standalone code and as the ocean component of the Community Climate System Model (CCSM). POP remains one of the primary ocean models in use for global climate and ocean research. The POP model is used for both climate change and oceanographic research. For climate change research, POP is coupled to atmosphere, land, and sea-ice models and run at a relatively coarse resolution to achieve maximum simulation throughput over centuries of simulation time. In oceanographic research, however, POP is run at a scale that is fine enough to resolve the mesoscale eddies that influence global ocean circulation over the course of simulated decades. Leadership computational capabilities have now reached a point where it is feasible to run the coupled climate model at a much finer scale. Using the mesoscale resolution of the POP ocean model coupled with more physically realistic models for the remaining components of the Earth system is expected to improve the climate simulations and provide more accurate projections of future climate change.

The community goal is to achieve a throughput of more than one simulated year per CPU-day for the fully coupled system. There are several factors to consider when designing improvements to POP's performance and scalability. The ocean model is but one component of the system, so careful attention to memory allocation is required. In addition, output for the climate-coupled model will be larger and occur more frequently than it does in the ocean-only model.

For both the Q2 and Q4 problems, we performed a computation using the ocean-only mode that is effectively identical to the expected requirements for the coupled simulation. The model used a tripole 0.1-degree global grid (3600×2400×42 grid points) and computed three simulated days with a timestep of ten minutes. To benchmark the high-frequency output time slice, the simulation will output data comparable to the data needed for the climate-coupled model after each simulated day. This is an order-of-magnitude increase in IO activity; for ocean-only models, data is output once every simulation-month. Thus, the enhancement aimed to improve the strong scaling capability of POP. The Q2 problem was executed on 4800, 9600, and 14,400 PEs of the Jaguar XT5 system in Q2. The horizontal subdomain block sizes used in Q2 were 30×60, 30×30, and 30×20, respectively. The Q2 runtimes were ~ 958s for 4800 PEs, ~ 1012s for 9600 PEs, and ~1450s for 14400 PEs. The Q2 I/O implementation was inadequate for this frequency of output, thus improving the I/O system became the primary focus of this code’s GPRA-PMM activities. Based on the small amount of work computed locally for the 14400 PE case on the Q2 problem, we decided to focus on a strong scaling result between the 4800 PE and 9600 PE cases.

On careful analysis, it was determined that the I/O phases of the code might be re-written to achieve a significant performance gain in efficiency. Thus in Q4 we first aimed to enhance the efficiency of the Q2 problem on 4800 PEs where 652.933508s of the total run time of 957.842493s, or 68.1671% of the walltime, was spent doing I/O. The simulation executed 3 simulated days. Each day observable data and movie related data were written to disk. The observables were formed in 8 3D fields and 19 2D fields. Each process managed a 60 × 30 × 42 fragment per 3D field and a 60 × 30 fragment per 2D field. The data type for all the I/O data is float. Thus, the volume of observable data written per day for the target problem is  $(8 \times 4 \times 30 \times 60 \times 42 \times 4800 + 19 \times 4 \times 30 \times 60 \times 4800)$  B / day = 11.4262104 GB / day, or about 35 GB for the Q2 problem and 1 file for observables per day. There are 60 movies formed each day from coordinate data. The 3600 × 2400 coordinate movie data is decomposed over a virtual 60 × 80 rectangular process grid (for 4800 PEs) where each process owns a 60 × 30 block of the global data set. The total volume of data written for movies in the Q2 problem is  $60 \times 4 \times 60 \times 30 \times 4800$  B / day = 1.931190491 GB / day or 5.793571472 GB for the entire benchmark problem.

In Q4, we introduced a C routine that targets a single Lustre file for a parallel write. Instead of a single process writing after each gather, the Q4 code executes a targeted gather phase where first the data affiliated with each  $k$ -value and field are used to identify the process ID of the gathering process. After gathering the data to be written into a set of designated, disjoint I/O processes, then each I/O process executes a write with offset into the Lustre file. For the 2D fields, a single I/O PE was assigned for each 2D field. The gather and write proceed as just described in the case of 3D fields. We note that for the combined assembly and write of the 3D and 2D observable data, the Q4 algorithm is 7.595252132037274 times faster than the Q2 version for the fixed Q2 problem on 4800 PEs. Similarly, for the movie files, a group of designated I/O processes replace the single process write –in this case a single I/O process assumes management of a single movie. A gather phase is executed where the block decomposed data is sent to the process with MPI process ID equal to the movie index. The I/O processes then locally copy the data from the receive buffer into a write buffer thus restoring the global indices –transforming to a column major ordering of the coordinates in the 3600 × 2400 spatial grid. The set of I/O processes then write with offset to a single Lustre file. Looking at the performance numbers between Q2 and Q4, we note that the Q4 code is 7.946443175395545 times faster on the Q2 problem and 4800 PEs.

The strong scaling assertion is all but complete. We use the new I/O algorithm and execute the Q2 problem on 9600 PEs in ~290s, two times the PEs in the Q2 4800 PE baseline problem and ~3.3 times faster than the Q2 run time on 4800 PEs.

### 2.2.3 LS3DF

LS3DF [2, 3] is a code for *ab initio* density functional theory (DFT) calculations. It was designed for the study of nanosystems containing a few thousand to hundreds of thousands of atoms. LS3DF scales linearly with both the number of atoms in the system being studied and the number of processors used in

the computation of a given system. It achieves this scalability by using a divide-and-conquer method for solving the DFT equations. The physical system is divided into many fragments, each of which is computed upon by a small group of processors before the results from these fragments are gathered together to generate the results. It is this independent solution of different fragments that makes this algorithm so amenable to parallelization. Because the algorithm inside LS3DF is so naturally parallelizable, the code is already capable of performing DFT calculations on systems with hundreds of thousands of atoms. It can finish a self-consistent calculation across hundreds of thousands of processors within an hour.

The development of LS3DF has made the study of the internal electric field problem in nanosystems computationally feasible. Electric fields could occur within nanocrystals due to dipole moments. These internal electric fields can exert significant influence over the electronic structure, the electron wave function localization, the exciton binding energy and dissociation, and the carrier dynamics of the system. The internal electric field problem is not yet well understood due to the onerous computational demands of the system, but with LS3DF this knowledge is now within reach.

For the GPRA-PMM metric, we sought to improve the strong scaling capability of LS3DF. To test its scalability, we chose a 2776 atom, 24,220 valence electron ZnO nanorod problem modeled with realistic surface passivation for the Q2 and Q4 benchmarks. In this problem, we perform self-consistent calculations to compute the total charge density and potential energy of the system. Q2 results showed poor strong scaling due to inefficiencies in the subroutine that dominates the computations. It is this subroutine that was the focus of LS3DF's GPRA-PMM efforts.

We made three major developments in the code: (1) a band-index parallelization within the main subroutine; (2) a new algorithm for wave function optimization; and (3) an improved load balancing algorithm.

For Q4, the same problem was run again, and the performance markedly improved. We successfully reduced the computational time from the original 3.87 hours on 43,200 processors to 1.48 hours on 86,400 processors, a factor of 2.6 speed-up in wall time.

#### **2.2.4 Denovo**

Denovo [4, 5] serves as a general radiation transport application for nuclear and radiological sciences by finding accurate numerical solutions to the linear Boltzmann equation as described in Section 3.3.7. This application area includes, but is not limited to, nuclear reactor analysis, fusion, radiation shielding and protection, nuclear safeguards, radiation detection, and radiation therapy, diagnostics, and treatment planning. Nuclear reactor analysis requires accurate characterization of the neutron distribution in the reactor in order to determine power, safety, and fuel and component performance. In a steady-state operational reactor, the neutron field is characterized by six independent variables (three in space, two in angle, and one in energy), and the mean flight times of low-energy neutrons are in the millimeter to centimeter range. Thus, high-resolution solutions of the transport equation require tremendous computational resources. Traditionally, computational resources have not been sufficient to attack this problem at full resolution; so multi-level approximation schemes have been employed. However, current leadership systems such as Jaguar/XT5 open the door, for the first time ever, to attacking this problem from a full transport approach. To achieve this goal, Denovo has synthesized the last decade's worth of computational transport work into a modern, production-quality application that has the ability to support a full-core reactor analysis from an ab initio approach. Further details on the Denovo physical models, numerical algorithms, and software implementation are given in Section 3.3.7.

For both the Q2 baseline and Q4 metric problems, the power distribution in a full EDF PWR900 model core [6] is computed. The core (height 4.2 m) has  $17 \times 17$  (289) assemblies (height 3.6 m), of which 157 are fuel and 132 are reflector, with each fuel assembly consisting of  $17 \times 17$  fuel pins. Three different fuel enrichments (ranging from 1.5% to 3.25%) are modeled in the fuel assemblies, with the 289 fuel pins being "homogenized" with 45 "pin cells". A total of 135 different pin cell materials are used to

accommodate the three different enrichments. With this configuration, the Denovo application is used to solve for the k-eigenvalue and scalar flux throughout the core (see Equation 2 in Section 3.4).

For the Q2 baseline problem, a 2x2 spatial mesh array is used for each pin cell, yielding 578 mesh cells in the x and y directions (0.63 cm width) and 700 cells in the axial (z) direction (0.60 cm width), for a total of 233,858,800 cells. Denovo is asked to return solutions to a discretized Boltzmann equation consisting of one scalar unknown per cell, 168 angular directions per scalar unknown, and two energy groups (fast and thermal) per scalar unknown, for a total of  $7.86 \times 10^{10}$  unknowns (degrees of freedom). Solutions were obtained on 17,424 cores of Jaguar/XT5 using a  $k_{\text{eff}}$  tolerance of 0.001 and an eigenvector tolerance of 0.10. A total simulation time of 187.68 min (11,260.8 s) is required, with over 99% of the time being consumed by the sweep algorithm.

During the GPRA-PMM exercise, parts of the code were optimized, a new parallel decomposition was implemented, and several new solvers were added to the code, allowing Denovo to scale to hundreds of thousands of cores. Solving the Q2 problem on the same number of processors, but using improved KBA sweep ordering and block-size analysis, and a residual Krylov solver for within-group solves, the problem was solved in 11 minutes, a factor of 17 improvement.

An additional Q4 metric problem was also solved: rather than two energy groups, this problem incorporated 22 times more degrees of freedom, for a total of 44 energy groups and  $1.73 \times 10^{12}$  unknowns. This problem, which was infeasible before the improvements made during the GPRA-PMM exercise, was solved on 112,200 cores of Jaguar in just over twenty minutes in the best case. Denovo was exercised in a weak scaling manner – at a minimum, we sought to achieve a constant runtime for the same amount of work per core. Despite increasing the size of the problem by a factor of 22 while increasing the number of processors by only a factor of 6.4, we still achieve a reduction in runtime, resulting in a factor of 31 efficiency improvement in the best case. The Q2 the software utilized ~7.77% of the entire machine, whereas in Q4 we utilized 50.03% of the total system.

## 2.3 CONCLUSIONS

All four FY10 GPRA-PMM applications have successfully executed their Q2 baseline problems on the target Jaguar XT5 architecture, and achieved their stated performance goals in Q4. As dictated by their science drivers, two of the applications (POP and LS3DF) pursued strong scaling metric problems for Q4 and the other two (TD-SLDA and Denovo) accomplished weak scaling or efficiency results. Table 1 below summarizes the baseline problems and FY10 findings for each applicatio

Table 1. FY10 GPRA-PMM Summary of Enhancement Results for Q2, Q4 Benchmark Exercises

Application	TD-SLDA	POP	LS3DF	Denovo
<b>Problem</b>	<p><b>Q2 : Nuclear 198W study</b></p> <ul style="list-style-type: none"> <li>• Z=74, N=124</li> <li>• 40 x 40 x 40 lattice</li> <li>• 7,466 p-quasiparticle</li> <li>• 8,946 n-quasiparticle</li> <li>• 200 time steps</li> <li>• 0.75fm spacing</li> <li>• 100MeV cutoff</li> </ul> <p><b>Q4 : Nuclear 238U study</b></p> <ul style="list-style-type: none"> <li>• Z=92, N=146</li> <li>• 40 x 40 x 64 lattice</li> <li>• 67,118 p-quasiparticle</li> <li>• 69,508 n-quasiparticle</li> <li>• 200 time steps</li> <li>• 1.25fm spacing</li> <li>• 100MeV cutoff</li> </ul>	<p><b>3 simulated days, ocean-only model</b></p> <ul style="list-style-type: none"> <li>• 0.1-degree tripole global grid (3600x2400)</li> <li>• 42 vertical levels</li> <li>• 10 minute time steps</li> <li>• High-frequency output time slice</li> </ul>	<p><b>Self-consistent DFT calculation for ZnO nanorod</b></p> <ul style="list-style-type: none"> <li>• 2776 atoms</li> <li>• 24220 valence electrons, d-electrons in valence band</li> <li>• 720x300x300 numerical grid</li> </ul>	<p><b>Q2 : Full Core EDF PWR900 benchmark</b></p> <ul style="list-style-type: none"> <li>• 17x17 fuel assemblies</li> <li>• 17x17 fuel pins per assembly</li> <li>• 2x2 cells per pin cell</li> <li>• 3 fuel enrichments</li> <li>• 45 homogenized pin cell materials per assembly</li> <li>• 135 different pin cell materials</li> <li>• 233,858,800 (578x578x700) cells</li> <li>• 168 angles, 1 moment, 2 energy (fast and thermal) groups</li> <li>• <math>7.86 \times 10^{10}</math> total unknowns</li> </ul> <p><b>Q4 : Full Core EDF PWR900 benchmark</b></p> <ul style="list-style-type: none"> <li>• 168 angles, 1 moment, 44 energy (fast and thermal) groups</li> <li>• <math>1.73 \times 10^{12}</math> total unknowns</li> </ul>
<b>Hardware (cores)</b>				
<b>Q2</b>	(s)73,728; (td)16,414	4,800	43,200	17,424
<b>Q4</b>	(s)217,800; (td)136,628	9600	86,400	112,200
<b>Time (seconds)</b>				
<b>Q2</b>	(s)6538.5, (td)2084.4	957.8	13,932	11,260.8
<b>Q4</b>	(s)18393.2, (td)2031.5	290.3	5328	1121.6
<b>Metric target</b>	(s)Q2:Q4 efficiency $\geq 1.0$ ; (td)Q2:Q4 time $\geq 1.0$	Q2:Q4 time $\geq 2.0$	Q2:Q4 time $\geq 2.0$	Q2:Q4 efficiency $\geq 1.0$
<b>Metric result</b>	(s)Q2:Q4 efficiency = 2.11 (td)Q2:Q4 time = 1.026	Q2:Q4 time = 3.2992	Q2:Q4 time = 2.6	Q2:Q4 efficiency = 31

### **3. OVERVIEW OF COMPUTATIONAL SCIENCE CAPABILITIES AND ANALYSIS OF METRIC RESULTS**

## 3.1 TD\_SLDA

### 3.1.1 Introduction

A large number of quantum many-body systems become superfluid when their temperature is sufficiently low. The best-known examples are the electrons in a superconductor and the liquid Helium 4, the first being an example of a fermion system and the second an example of a bosonic system. Many other systems also demonstrate superfluid properties: electrons in high  $T_c$  superconductors, excitons in condensed matter systems, neutrons in the crust of neutron stars, neutrons and protons in atomic nuclei, quarks in high density matter (expected to exist in the cores of neutron stars), liquid Helium 3, and both fermionic and bosonic cold atoms in atomic traps. The great importance of superfluidity was rewarded over the years with at least ten Nobel prizes – a record number for any topic in physics. Superfluidity in some of these systems – bosonic superfluids (liquid Helium 4, cold bosonic atoms in traps, and excitons in condensed matter systems) – is related to the formation of a Bose-Einstein Condensate (BEC). In the fermionic superfluids (electrons, fermionic atoms in traps, nucleon and quarks) the nature of superfluidity is due to the formation of Cooper pairs of fermions and their subsequent formation of a BEC. In fermionic superfluids the critical temperatures for the onset of superfluidity span an astounding twenty orders of magnitude from quark matter to cold atomic gases. Despite these great differences in the values of their critical temperatures, these systems share a number of properties, so the methods used to study one system can be used to study the other systems.

We have developed a set of codes that allows the study of the structure and in particular the dynamics of a large class of fermionic superfluids: neutron superfluid in neutron star crust, cold atoms in traps, nuclear reactions with gamma rays, incident nucleons and other projectiles as well as induced nuclear fission.

The current level of understanding of the dynamics of the neutron star crust, in particular of the nature of the starquakes which lead observationally to the so-called glitches in the rotational spectra of pulsars, is low at best. With the Time-Dependent Superfluid Local Density Approximation (TD-SLDA) software, a systematic study of the dynamics of the vortex pinning and de-pinning dynamics, believed to be the source of the starquakes, is now possible.

While one cannot perform experiments on neutron stars, the physics governing the neutron superfluid in the neutron star crust shares many similarities with the physics of cold fermion atoms in traps, which today are extensively studied in many laboratories. One particular aspect of the cold-atom physics, the extreme tunability of the properties through the use of the so-called Feshbach resonance, is of great importance. Another extremely important aspect of the cold atom physics is the fact that these systems have the highest critical temperature (in appropriate units) of any known superfluid. These systems are also exactly at the middle of the BCS-BEC crossover and for this reason they share many properties with both fermionic and bosonic superfluids. Perhaps as a result of the fact that these systems are characterized by such a critical temperature, they also demonstrate a property so observed only in high  $T_c$  superconductors, namely the phenomenon of pseudo-gap. The presence of the pseudo-gap has been established in first principle calculations and subsequently put in evidence in experiments as well.

The theoretical time-dependent formalism used to describe the physics of nuclei and various nuclear reactions is similar to the formalism used to study the dynamics of neutron star crust, the main difference being the actual number of protons and neutrons and the size and geometry of the system. From the practical point of view, the study of various nuclear reactions and of induced nuclear fission are topics of great importance to energy production, homeland security and defense.

### 3.1.2 Background and Motivation

In condensed matter and chemistry calculations, the leading method is the Density Functional Theory (DFT). In the DFT method instead of solving the Schrödinger equation we solve an  $N$ -electron system requiring the solution of a partial differential equation in a  $3N$ -dimensional space with a system of  $N$  nonlinear, coupled 3D-equations. This simplification is achieved by introducing an energy density

functional, the variation of which provides the energy and the electron density spatial distribution of the ground state of the system. The existence of the energy functional was proven by Kohn, Hohenberg and Sham in 1964-1965 and this achievement was recognized with the Nobel Prize in chemistry to W. Kohn in 1998. Further theoretical developments and the extension of the DFT formalism have led today to the study of the properties of a large number of excited states of electron systems within the framework of the TD-DFT (Time-Dependent DFT) formalism.\*

This highly successful theoretical formalism is limited to so-called ‘normal’ systems, however, and is essentially impossible to apply to superconductors in particular. In the early 1980s, a nonlocal extension of the DFT for the study of superconductors was suggested, and has been implemented during the last few years by E. K. U. Gross and his group for the study of a few systems. The great success of DFT applied to normal electron systems was ensured by the introduction of LDA (Local Density Approximation) form of DFT by Kohn and Sham, which leads to local nonlinear coupled partial differential equations, as opposed to nonlocal nonlinear coupled integral-partial differential equations. This great advantage of DFT was essentially nullified in the extension of the DFT formalism implemented by Gross and collaborators.

Recently a local extension of the DFT method to superfluid systems has been introduced. The Superfluid LDA (SLDA) and its extension to time dependent phenomena (TD-SLDA) have been applied with notable success to the study of nuclei, cold atomic gases in traps, the structure of vortices in neutron stars and in cold atomic gases, and to the dynamics of superfluid cold fermionic atomic gases. TD-SLDA has been used in particular to study the generation and dynamics of the vortices in a cold atom system in 3D<sup>†</sup> and to study the Coulomb excitation of an atomic nucleus by an impinging relativistic heavy ion. This work was performed within the SciDAC-UNEDF collaboration.‡ Formally, the TD-SLDA equations resemble the time-dependent Bogoliubov-de Gennes equations; there are distinct parallels between the TD-LDA and TD Hartree equations. Due to the immense size of the system of coupled nonlinear partial differential equations, the TD-SLDA equations in 3D can be solved only on leadership computing resources.

The initial goal of the nuclear TD-SLDA code development was to provide a simple and straightforward solution of the QRPA (Quasiparticle Random Phase Approximation) equations, which describe a set of excited states of atomic nuclei. QRPA is formally the small amplitude limit, or the linear response, of a quantum system to an external time-dependent perturbation. The complexity of the QRPA is still too great to allow for the study of deformed nuclei, or even to obtain an accurate solution in the case of spherical nuclei where the complexity of these equations is significantly reduced. In its traditional formulation QRPA amounts to the diagonalization of a very large (non-Hermitian) matrix, and both the construction and the diagonalization are extremely time- and memory-consuming. Accurate solution of these equations for a deformed nucleus, even on the largest supercomputers, is not possible without drastic truncations that would prove difficult to control and evaluate. We instead suggest studying these problems with TD-SLDA, which can be implemented and solved on current leadership class computers.

In the case of nuclei the TD-SLDA describes the dynamics of the excitation by various probes both in the linear regime when the strength of the external probes is weak and in the nonlinear regime when the external probes generate strong fields. In particular, the QRPA and the second QRPA (the first correction to the linear response regime) as well as all the higher-order corrections can be easily incorporated into TD-SLDA. Within TD-SLDA simulations of the excitation of nuclei by gamma rays, nucleons, and even by heavy ions can easily be performed.§ Since in TD-SLDA the nuclear projectile and target structures and interactions are described within the same framework, there are no theoretical ambiguities arising from the poor understanding of reaction models; thus we can in principle hope for a theoretically consistent description and more controlled theoretical errors.

---

\* see <http://www.tddft.org/>

† see <http://www.phys.washington.edu/groups/qmbnt/index.html>

‡ see <http://unedf.org/>

§ see Coulomb excitation of <sup>48</sup>Cr at [http://www.phys.washington.edu/groups/qmbnt/nuclei\\_dynamics.html](http://www.phys.washington.edu/groups/qmbnt/nuclei_dynamics.html)

TD-SLDA can potentially provide the best theoretical framework for the study of induced nuclear fission. Nuclear fission is a topic of fundamental theoretical significance and great practical importance. Despite seventy years of intense research, the theoretical models for nuclear fission are still simplistic, based mostly on phenomenology and possessing little microscopic underpinning. The reasons are rooted in the enormous complexity of the underlying microscopic equations, and the intrinsic difficulty of solving them. We are adding the capability to directly address this problem for the first time into the TD-SLDA software. This is now possible thanks to considerable theoretical progress and the advent of petaflop supercomputers.

TD-SLDA can be applied to study a whole gamut of fundamental problems in the physics of fermionic superfluids. The study of the generation and dynamics of vortices in fermionic superfluids has been essentially beyond the reach of computers. So far only approximate methods have been used, such as the Landau-Ginzburg approach which is valid only near the critical temperature. The two-fluid hydrodynamics, another approximate approach developed by Landau in the 1940s to describe the motion of superfluids, is inappropriate for the study of quantized vortices as there is no Planck's constant in Landau's formulation of the two-fluid hydrodynamics. There are no known simple approximations valid in the small temperature regime – a regime of great interest in studies of liquid Helium 3 and neutron star physics. The whole field of quantum turbulence in fermionic superfluids currently lacks a theoretical framework for discussing the extensive set of experimental results generated in many laboratories worldwide. The physics of the pinning and depinning of quantum vortices, in particular in neutron star crust physics, also lacks an adequate theoretical framework. Only the Landau-Ginzburg approach has been used in the past, even though this approximation is invalid far from the critical regime.

We hope that the tools we have developed will be valuable to scientists studying the following physical phenomena at a minimum: i) the dynamics of nuclear large amplitude collective motion including induced fission; ii) the dynamics of vortices in the neutron star crust and the elucidation of the starquake mechanism; iii) the generation and dynamics of vortices and the study of the quantum turbulence in fermionic superfluids; and iv) the dynamics of cold gases in various external conditions (static and time-dependent magnetic fields/Feshbach resonance, static and time-dependent optical lattices, reactions to various laser fields, etc.).

### 3.1.3 Capability Overview

The goal of the nuclear TD-SLDA software is to compute on systems in spatial domains that range from  $40^3$  to  $75^3$  fm<sup>3</sup> over times up to nearly an attosecond. The total number of nucleons can reach thousands (perhaps even tens of thousands) for the neutron star crust case.

For the study of cold fermionic atomic gases, the goal is to study system sizes up to roughly  $100^3$  spatial lattice points and of the order of a million time steps. In physical terms this amounts to systems of up to  $10^5$  particles followed in time for up to hundreds of periods, where a period is roughly the time it takes a fermion to cross the system.

We will have to evolve from 20,000 to 500,000 3D complex wave functions in time. Putting this into perspective, other nuclear physicists have reported evolving at most a few hundred wave functions, and only in normal systems, when the pairing correlations are neglected.

### 3.1.4 Science Driver for Metric Problem

We want to study the generation and the dynamics of vortices in a unitary Fermi gas and try to establish whether quantum turbulence can be observed and under what conditions. Quantum turbulence is driven by the vortex-vortex crossing, in which vortex lines are reconnected in a process very similar to DNA recombination. In a 3D simulation of this system we will be able to provide a microscopic description of the formation of vortex rings and of the reconnection of vortex lines in fermionic superfluids for the first time.

For nuclei want to study one of a number of phenomena: i) the process of Coulomb excitation of an open shell nucleus by a relativistic heavy ion; ii) the gamma and/or nucleon excitation of an open shell nucleus; iii) gamma- and/or nucleon-induced nuclear fission.

### 3.1.5 The Model and Algorithm

The Superfluid LDA (SLDA), the extension of the DFT to superfluid systems, is based on the following energy density functional for the ground state for an unpolarized unitary Fermi gas:

$$E_{gs} = \int d^3r \left[ \frac{\hbar^2}{2m} \tau(\vec{r}) + g_{eff}(n(\vec{r})) |\kappa(\vec{r})|^2 + \varepsilon(n(\vec{r})) + V_{ext}(\vec{r}) n(\vec{r}) \right],$$

$$n(\vec{r}) = 2 \sum_n |v_n(\vec{r})|^2, \quad \tau(\vec{r}) = 2 \sum_n |\vec{\nabla} v_n(\vec{r})|^2, \quad \kappa(\vec{r}) = \sum_n v_n^*(\vec{r}) u_n(\vec{r}),$$

where  $n(\vec{r})$  is the number (normal) density,  $\tau(\vec{r})$  is the kinetic energy density, and  $\kappa(\vec{r})$  is the anomalous density. The anomalous density vanishes in the normal phase. These densities depend on the quasiparticle wave functions  $(u_n(\vec{r}), v_n(\vec{r}))$ .  $V_{ext}(\vec{r})$  is an arbitrary external potential in which the system might reside,  $\varepsilon(n(\vec{r}))$  is the normal part of the interaction energy density and  $g_{eff}(\vec{r})$  is the renormalized coupling strength of the pairing correlations. Varying the quasiparticle wave functions  $(u_n(\vec{r}), v_n(\vec{r}))$  we obtain the Bogoliubov-de Gennes-like equations

$$\begin{pmatrix} -\frac{\hbar^2 \Delta}{2m} + U(n(\vec{r})) + V_{ext}(\vec{r}) - \mu & \Delta(\vec{r}) \\ \Delta^*(\vec{r}) & \frac{\hbar^2 \Delta}{2m} - U(n(\vec{r})) - V_{ext}(\vec{r}) + \mu \end{pmatrix} \begin{pmatrix} u_n(\vec{r}) \\ v_n(\vec{r}) \end{pmatrix} = E_n \begin{pmatrix} u_n(\vec{r}) \\ v_n(\vec{r}) \end{pmatrix},$$

which must be solved self-consistently. They represent an infinite set of nonlinear coupled partial differential eigenvalue equations. Upon discretization, these equations become nonlinear coupled matrix equations that are solved iteratively. We begin with a guess for the densities, which allows us to evaluate various potentials (self-energy  $U(n(\vec{r}))$  and pairing potential  $\Delta(n(\vec{r}))$ ). After diagonalization, the relevant densities and potentials are reevaluated, and the process is repeated until convergence is achieved. The convergence rate can be accelerated by various techniques, such as Broyden's line search algorithm.

In the time-dependent case we consider the variation of a different functional,

$$L = \int dt d^3r \left[ i\hbar \sum_n v_n^*(\vec{r}, t) \frac{\partial v_n(\vec{r}, t)}{\partial t} \right]$$

$$- \int dt d^3r \left[ \frac{\hbar^2}{2m} \tau(\vec{r}, t) + g_{eff}(n(\vec{r}, t)) |\kappa(\vec{r}, t)|^2 + \varepsilon(n(\vec{r}, t)) + V_{ext}(\vec{r}, t) n(\vec{r}, t) \right],$$

$$n(\vec{r}, t) = 2 \sum_n |v_n(\vec{r}, t)|^2, \quad \tau(\vec{r}, t) = 2 \sum_n |\vec{\nabla} v_n(\vec{r}, t)|^2, \quad \kappa(\vec{r}, t) = \sum_n v_n^*(\vec{r}, t) u_n(\vec{r}, t).$$

All quantities are functions of time. In general the structure of this functional is more complicated, and Galilean invariance requires that we use a slightly different form for the kinetic energy density:

$$\tau(\vec{r}, t) \Rightarrow \tau(\vec{r}, t) - \frac{|\vec{p}(\vec{r}, t)|^2}{n(\vec{r}, t)},$$

$$\vec{p}(\vec{r}, t) = \frac{1}{2i} \sum_n \left[ v_n(\vec{r}, t) \vec{\nabla} v_n^*(\vec{r}, t) - v_n^*(\vec{r}, t) \vec{\nabla} v_n(\vec{r}, t) \right],$$

where  $\vec{p}(\vec{r}, t)$  is the time-dependent current density. This illustrates a qualitative difference between the static and the dynamic problems: the presence of currents in the latter case. For the sake of simplicity we here omit the equations of motion for the quasiparticle wave functions in their most general form, when an effective mass different from the bare of mass is present. The time-dependent evolution equations for the quasiparticle wave functions ( $u_n(\vec{r}, t)$ ,  $v_n(\vec{r}, t)$ ) become

$$i\hbar \frac{\partial}{\partial t} \begin{pmatrix} u_n(\vec{r}, t) \\ v_n(\vec{r}, t) \end{pmatrix} = \begin{pmatrix} -\frac{\hbar^2 \Delta}{2m} + U(n(\vec{r}, t)) + V_{ext}(\vec{r}, t) - \mu & \Delta(\vec{r}, t) \\ \Delta^*(\vec{r}, t) & \frac{\hbar^2 \Delta}{2m} - U(n(\vec{r}, t)) - V_{ext}(\vec{r}, t) + \mu \end{pmatrix} \begin{pmatrix} u_n(\vec{r}, t) \\ v_n(\vec{r}, t) \end{pmatrix}.$$

These represent an infinite set of nonlinear coupled partial differential equations of evolution.

In the nuclear case, the equations for both the static (SLDA) and the time-dependent (TD-SLDA) problems are characterized by additional degrees of complexity, one due to the coupling between the orbital motion, another due to the spin degrees of freedom, and more due to the existence of both protons and neutrons in the nuclear system. The energy density functional depends now on proton and neutron number densities, on proton and neutron kinetic energy densities, on the proton and neutron spin-current densities, and on the gradients of the proton and neutron densities. In the case of odd total proton or neutron numbers, the energy density functional can depend on the proton and/or neutron spin number densities as well. Consequently, while the formal matrix structures of both SLDA and TD-SLDA equations are similar for the unitary gas problem and for the nuclear problem, the structure of the nuclear problem is significantly more complex. The form of the nuclear Hamiltonian is described in greater detail in section describing the Q4 developments. The quasiparticle wave functions have four components in the nuclear case, instead of only two in the unitary gas. The absence of a spin-orbit coupling in the unitary gas case means that the four component quasiparticle wave functions can be reduced to only two components.

We represent the quasiparticle wave functions on a discrete three-dimensional spatial lattice with a lattice constant  $a$ ,  $N$  lattice points in each spatial direction (although the spatial dimensions of the lattice do not in general need to be identical), and periodic boundary conditions.

In the static (solver) SLDA code we calculate first- and second-order spatial derivatives using a matrix representation of the FFT (Fast Fourier Transform) according to the formulas given below, where  $F(x)$  is the interpolating function of the DVR (Discrete Variable Representation) method. Using this interpolating function  $F(x)$  we evaluate the corresponding matrix representation of the first and second derivatives, as shown in the following equations:

$$F(x) = \sum_{l=0}^{N-1} \frac{1}{N} \exp\left(\frac{ixk_l}{Na}\right) = \frac{1}{N} \frac{\sin\left(\frac{\pi x}{a}\right) \exp\left(-i\frac{\pi x}{Na}\right)}{\sin\left(\frac{\pi x}{Na}\right)}$$

$$k_l = -\frac{\pi}{a} + \frac{2\pi l}{Na}, \quad l = 0, \dots, N-1$$

$$x_n = na, \quad n = 0, \dots, N-1$$

$$F(x_n - x_m) = \delta_{nm}$$

$$\nabla_{nm} = F'(x_n - x_m) = \frac{\pi}{Na} (-1)^{n-m} \left[ (1 - \delta_{nm}) \cot\left(\frac{\pi(n-m)}{N}\right) - \frac{i}{N} \right]$$

$$\Delta_{nm} = (\nabla^2)_{nm} = F''(x_n - x_m) = \frac{\pi^2}{2N^2 a^2} \frac{(-1)^{n-m} (1 - \delta_{nm})}{\sin^2\left(\frac{\pi(n-m)}{N}\right)} - \frac{\pi^2}{3a^2} \left(1 + \frac{2}{N^2}\right) \delta_{nm}$$

For lack of a better notation, we have used above the symbols usually reserved for the gradient and Laplacian in 3D for the matrix representation of the first and second derivatives in 1D.

The eigenvalues of the above discrete matrices  $\nabla$  and  $\Delta$  are exactly  $ik_l$  and  $-k_l^2$ , as expected. For instance, using the DVR method, the discrete representation of the 1D Schrodinger equation

$$-\frac{\hbar^2}{2m} \frac{d^2\psi(x)}{dx^2} + U(x)\psi(x) = E\psi(x)$$

reads

$$-\frac{\hbar^2}{2m} \sum_{l=0}^{N-1} \Delta_{kl} \psi_l + U_k \psi_k = E\psi_k, \quad \psi_k = \psi(x_k), \quad U_k = U(x_k).$$

In SLDA we need to determine the full eigenvalue spectrum of the 3D Schrödinger equation. In the case of the nuclear problem we have to self-consistently solve the equivalent HFB (technically, Bogoliubov-de Gennes (BdG)) equations, of dimensionality  $4N^3 \times 4N^3$  in matrix form. For  $N=50$  lattice points in each spatial direction, this amounts to diagonalizing and determining the full spectrum of eigenvalues and eigenvectors of a  $500,000 \times 500,000$  Hermitian matrix several hundred times (until self-consistency is achieved). In Q2, we performed this diagonalization in two steps, by first diagonalizing the Hartree block, which is a Hermitian matrix of size  $2N^3 \times 2N^3$ , and then introducing a carefully chosen energy cut-off, a matrix that is typically smaller than  $2N^3 \times 2N^3$  in size, to evaluate the effect of the pairing correlations. See the Q4 section for changes made to the Q2 nuclear solver.

In the case of the unitary gas, where there is no spin-orbit interaction, we can proceed with the direct diagonalization of the BdG matrix of dimension  $2N^3 \times 2N^3$  or of smaller dimensions in cases where we consider homogeneous systems in one spatial direction. If the system is translationally invariant in one spatial direction, the quasiparticle wave functions have a simpler structure,

$$\begin{pmatrix} \mathbf{u}_n(\mathbf{x}, \mathbf{y}, t) \exp(i\mathbf{k}_n z) \\ \mathbf{v}_n(\mathbf{x}, \mathbf{y}, t) \exp(i\mathbf{k}_n z) \end{pmatrix},$$

and densities and potentials do not depend on the  $z$ -variable. Experimentally, this situation can be approximately realized in very elongated cigar shaped traps for example.

The generalization of these formulas to the 3D case of coupled nonlinear equations of many wave functions with several components is straightforward and easy to code. In the DVR method only the kinetic energy and spin-orbit interaction (spatial derivatives) are represented as matrices, while all local potentials appear as diagonal matrices, thus making the evaluation of all these quantum operators simple to implement numerically.

In the TD-SLDA method we evaluate only the action of the Hamiltonian on various wave functions, so diagonalization and determination of the eigenvalues is not necessary. To speed up the evaluation of first- and second-order derivatives, we use the FFTW (the Fastest Fourier Transform in the West) rather than a matrix representation, thereby avoiding matrix operations altogether. This allows us to evaluate spatial derivatives with extremely high accuracy and with essentially the same speed as a multi-step finite difference formula.

The time evolution of the TD-SLDA equations is performed using a multistep, fifth-order predictor-corrector-modifier Adams-Bashforth-Milne method:

$$p_{n+1} = \frac{y_n + y_{n-1}}{2} + \frac{h}{48} (119y'_n - 99y'_{n-1} + 69y'_{n-2} - 17y'_{n-3}),$$

$$m_{n+1} = p_{n+1} - \frac{161}{170} (p_n - c_n),$$

$$c_{n+1} = \frac{y_n + y_{n-1}}{2} + \frac{h}{48} (17m'_n + 55y'_n + 3y'_{n-1} + y'_{n-2}),$$

$$y_{n+1} = c_{n+1} + \frac{9}{170} (p_{n+1} - c_{n+1}).$$

We have selected this method for its unique combination of high accuracy and numerical stability, and economical function-evaluation footprint. It requires only two evaluations of the right hand side of the differential equations per time step, a number that cannot be reduced without going to a lower accuracy numerical method.

### 3.1.6 Q2 Baseline Problem Results

We wish to investigate the excitation of a superfluid unitary gas constrained to a can-shaped external potential through a process of stirring with a rod and ball. The goal is to study quantum turbulence in Fermi gas systems and in particular to investigate the features of vortex formation and dynamics. In the Q2 problem we generated two sets of performance benchmark data for the unitary gas capability described in the text. Each will be described briefly here.

**Problem 1.** The target problem is to execute a system with 5216 particles on an  $N_x * N_y * N_z = 50 \times 50 \times 100$  lattice for approximately 100,000 time steps. Due to the cost of executing the problem, we benchmarked this system for 2051 time steps to gather sufficient performance data as a reference for future enhancements and to demonstrate the full software capability. The solver was first executed to self-consistently construct the stationary solutions of the system that will be required for the time-dependent

analysis we aim to perform. The solver exploits the homogeneity in the  $z$ -component of the problem geometry and executes each  $k_z$  value independently (there are  $N_z / 2 + 1$  such values in general) within a self-consistent iteration. This amounts to forming 51 independent parallel work groups, each of which will simultaneously numerically diagonalize a reduced dimension ( $n = 2 * N_x * N_y$  versus  $2 * N_x * N_y * N_z$ ) matrix each self-consistent iteration. It should be clear that one achieves a nearly perfect strong scaling curve when computing  $O(N_z)$  diagonalizations simultaneously versus in sequence. In the run, the size of each work group was chosen to be 144 PEs amounting to a virtual  $12 \times 12$  process grid over which the matrix elements are block-cyclically distributed. Thus, the solver utilized a total of 7344 PEs and 129 iterations were performed to achieve convergence. The error after each iteration of this run is plotted in Figure 3.1.1. The system was numerically resolved to machine precision for the problem dimension.

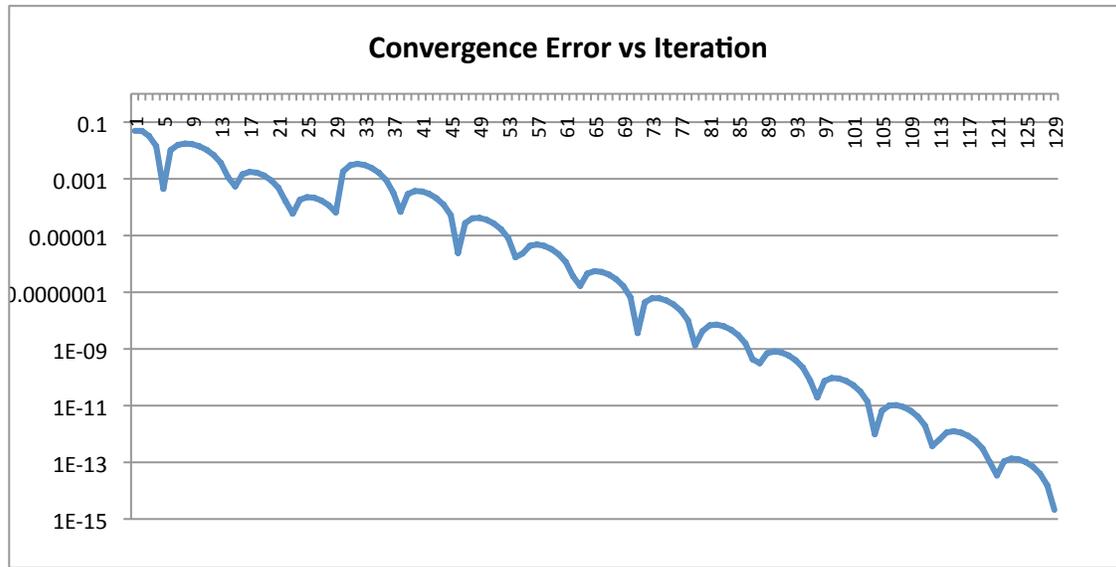


Figure 3.1.1: Error after each self-consistent iteration in computing the first Q2 benchmark problem.

After the solutions converge, the wave function data is written to a single Lustre file, three relevant observables are written to a second Lustre file, and some scalar, problem-related metadata is written to a text file. All the write operations are performed with POSIX semantics. The size of these data sets is  $\langle NWF \rangle * N_x * N_y * N_z * 2 * \text{sizeof}(\text{double complex})$ . For this problem, the number of quasiparticle wave functions is  $\langle NWF \rangle = 103,917$ . Thus,  $8.31336 \times 10^{11}$  Bytes (over 774 GB) of binary data were written for the wave function data alone. The analysis data file has magnitude  $3 * N_x * N_y * N_z * \text{sizeof}(\text{double})$ , or 6000000 Bytes of binary data. The following table summarizes the aggregated machine events measured during problem execution. The results of Run 1 are separated into an initialization phase prior to the self-consistent iteration ( $N_z / 2 + 1$  grids are formed and the operators and initial matrix elements are constructed) and a second phase defined by the self-consistent iterations plus the data dump after convergence. The total runtime was 184m 54.515s (or 11,094.515s).

Machine Event (7344PEs)	Phase 1 (Initialization)	Phase 2 (Self-consistent iterations and data dump)
Instructions Retired	16231470834773	333492109460091733
Floating Point Operations	6323441040	3628444034788875
Time (s)	0.661899	11,084.637851

Table 3.1.1: Results of instrumentation of Run 1 of the first benchmark problem.

The next run of this Q2 benchmark, Run 2, was intended to demonstrate the capability of the time-dependent software to successfully load and initialize the physical system created by the solver, evolve the system in time, and to checkpoint the time dependent computation –a step necessary to continue the time evolution of long simulations while adhering to the scheduling policies of a single run on the target platform. The same lattice and physical problem parameters are utilized but the time dependent code executes on 103,917PEs of the target platform in an effort to exploit all the coarse grain parallelism in the time evolution algorithm –e.g. one quasiparticle wave function per MPI process. The total runtime for this run was 140m 40.914s (or 8,440.914s). This run can be described by three distinct phases of code execution. In Phase 1, the solutions and observables generated by the solver are read and distributed, the lattices are initialized, and the lattice operators and derivatives are computed to initialize the time evolution algorithm. During Phase 2 of Run 2, exactly one time step of the system is executed and exactly one set of analysis data is written to a single Lustre FILE. In Phase 3, the code checkpoints the necessary data required for a restart of the simulation and then code termination occurs. This step requires writing  $22 * \langle \text{NWF} \rangle * N_x * N_y * N_z * \text{sizeof}(\text{double complex}) + (2 * \text{sizeof}(\text{double}) + \text{sizeof}(\text{double complex})) * N_x * N_y * N_z$  BYTES of data. For the  $50 \times 50 \times 100$  and  $\langle \text{NWF} \rangle = 103,917$  problem this amounts to writing just over 8TB of data required for a restart. The write is organized into 24 I/O groups and thus the data from each group of approximately 4330PEs is collected into a lead process per I/O group for each relevant numerical data structure and written to a unique Lustre file, totaling 24 Lustre files per data structure written. There is one additional Lustre file for the potentials and pairing field. Run 2 demonstrates the connection between the solver and the time evolution software as well as the checkpoint capability. The results are reproduced in Table 3.1.2.

Machine Event (103,917PEs)	Phase 1 (Read and distribute data; initialization)	Phase 2 (Execute single timestep and write analysis data)	Phase 3 (Checkpointing and termination)
Instructions Retired	885594209315035579	7458651188051530	2645109807479392620
Floating Point Ops	40887312859337	150994975407499	729823
Time (s)	2049.428677	4.896880	6236.922754

**Table 3.1.2: Results of instrumentation of Run 2 of the first benchmark problem.**

Run 3 of the Q2 unitary gas benchmark can be considered as having two distinct execution phases. Phase 1 demonstrates the restart capability of the time dependent software by continuing the computation at the subsequent time step to the time step last executed by the previous time-dependent run – Run 2 in this benchmark example. The number of I/O groups used for restart is currently coded to be the same number of I/O groups used to checkpoint the problem – 24 in this example. The data is read from the respective Lustre files by a root process in each group and distributed to the processes in the same modulo class. Phase 2 evolves the system 2050 time steps, executes exactly 25 I/O events of the time dependent analysis data, and exits. The total runtime for Run 3 was 219m 30.400s (or 13,170.4s). The following table represents the aggregated machine events for these two phases of execution. The average aggregated floating point count per 82 time steps over the 25 I/O events was  $1.2941 \times 10^{16}$  with a standard deviation of  $3.10993 \times 10^{13}$ . The average time required to execute 82 time steps and conduct a single I/O write of observable data event was 254.933521s, or 3.108945378s if amortized across all time steps. This is the time that we would like to minimize if at all possible.

Machine Event (103,917PEs)	Phase 1 (Read restart data and initialize)	Phase 2 (Run 2050 time steps)
Instructions Retired	2821900819478064487	1.59386e18
Floating Point Ops	26731359080	3.23524e17
Time(s)	6540.557430	6373.339007

**Table 3.1.3: Results of instrumentation of Run 3 of the first benchmark problem.**

The total cost of the  $50 \times 50 \times 100$  unitary fermion gas system Q2 benchmark on the target platform can be determined by aggregating results for each machine event and noting that the CPU time results are weighted by the total number of PEs involved in the particular run and converted to units of hours (time-dependent runs are bolded):

Machine Data	Run 1	Run 2	Run 3	Total
Instructions	3.335083409e17	<b>3.538162667e18</b>	<b>4.415760819e18</b>	8.287431827e18
Floating Point Ops	3.628450357e15	<b>1.91882289e14</b>	<b>3.235240267e17</b>	3.273443593e17
Wall Time(s)	11,085.29975	<b>8,291.248311</b>	<b>12,913.89644</b>	32,290.4445
CPU \$(hours)	22,614.01149	<b>239,333.7919</b>	<b>372,770.3823</b>	634,718.1857
PEs	7,344	<b>103,917</b>	<b>103,917</b>	

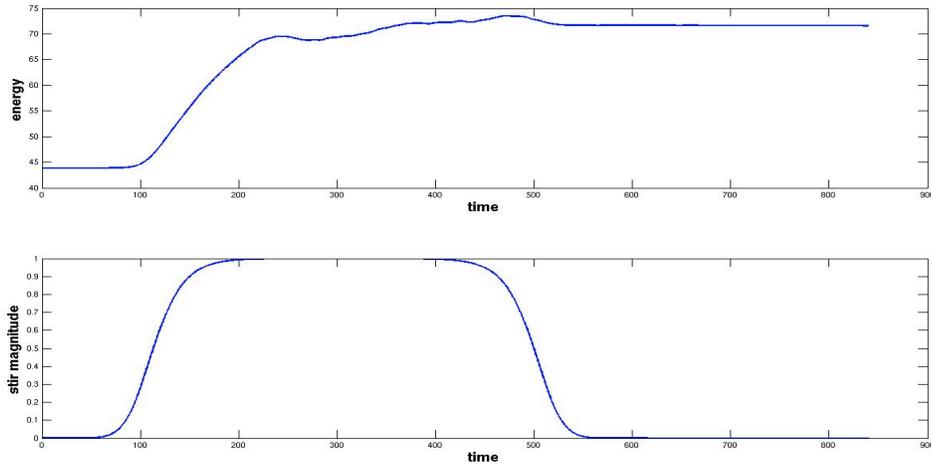
**Table 3.1.4: Total cost of unitary fermion gas system Q2 benchmark.**

Our two primary goals are to improve the efficiency of computing a time step per wave function, and to use the optimized version of the software to study a similar but more complex system on a larger lattice completing  $O(10^5)$  time steps on at least twice the PEs utilized in Q2. If we can identify more coarse-grained parallelism in the code, we may execute a strong scaling result for this system in Q4. Otherwise, we will pursue a weak scaling result based on the nuclear code base.

**Corollary of Problem 1.** A problem with reduced complexity but directly related to the benchmark problem above allows us to demonstrate the full software capability by completing a related simulation. We executed the ball and rod excitation of a unitary Fermi gas system with 300 particles on a  $32 \times 32 \times 32$  lattice for a total of 104,132 time steps including 1509 I/O events of analysis data. The total simulation advanced much like the previous benchmark problem. In Run 1 the self-consistent solver generated stationary solutions for the system. Run 2 used the results of the solver to initialize the time dependent code, executed a total 87271 time steps including 1264 completed I/O events, and check pointed the computation for restart. Run 3 restarted the time-dependent calculation at time step 87272, executed 16861 additional time steps including 245 I/O events, and exited cleanly. Below is a brief table summarizing the measured time data for these runs (with time-dependent runs in bold). The machine event data for the runs are also available.

	Run 1	Run 2	Run 3	Total
PEs	612	<b>9458</b>	<b>9458</b>	
Time(s)	451.081	<b>34,909.812</b>	<b>11,020.127</b>	46,381.02
CPU \$(hours)	76.68377	<b>91,715.83386</b>	<b>28,952.32255</b>	120,744.8401

**Table 3.1.5: Results for unitary Fermi gas system benchmark problem.**

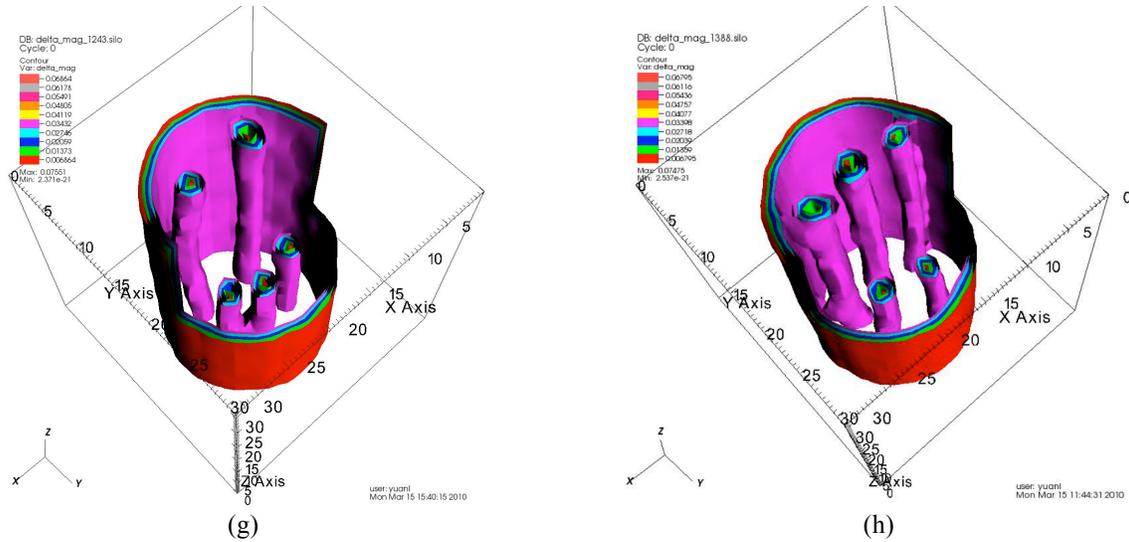


**Figure 3.1.2: Energy of 300 particle system on  $32^3$  lattice as function of time.**

Figure 3.1.2 depicts the energy of the 300 particle system and the magnitude of the excitation on a  $32^3$  lattice as a function of time. The system is excited adiabatically through the stirring of a rod and ball. The excitation is adiabatically turned off and the system is left to evolve. Three full stirs occur in total.

In Figure 3.1.3, the magnitude of the pairing field as a function of time is shown. Several magnitudes of the observable and a segment of the geometry have been removed from the plot to enable the viewing the interior of the system. The plots are ordered in row-major format (Silo frame numbers 0, 208, 256, 661, 722, 1095, 1243, 1388). Figure 3.1.3(a) depicts the unexcited system. In Figure 3.1.3(b) the ball and rod are being introduced. In Figure 3.1.3(c) the system response to the low-magnitude stirring strength begins to appear. Figure 3.1.3(d) shows two vortices that are already well formed. One of the vortices is being impinged upon by the ball, which has higher angular frequency than the vortex. Figure 3.1.3(e) shows two vortices in close proximity situated between the ball and rod. In Figure 3.1.3(f) the system depicts clearly five well-formed vortices, two of which are still entangled. There is no external excitation at this time or in subsequent frames. In Figure 3.1.3(g) the five vortices continue to circulate and the separation of the two entangled vortices is nearly complete. In Figure 3.1.3(h) there are clearly five disconnected vortices. To relate the plots in Figure 3.1.3 to the times in Figure 3.1.2, multiply the Silo frame number by the iterations/IO event and multiply this product by the time step (iterations/IO event = 69, time step = 0.008067).





**Figure 3.1.3: Eight instances from the 3-D movie made with VisIt of excitation of the stirring with ball and rod of a fermion unitary gas system.**

**Comment on Eigendecompositions for the SLDA static solvers.** It is worth repeating here that for the self-consistent solvers we have to diagonalize matrices of dimension  $N = 2 N_x \times N_y \times N_z$  (or  $N = 4 N_x \times N_y \times N_z$  in a self-consistent iteration) possibly hundreds of times to achieve convergence in the general case (as mentioned gross reductions in complexity can be exploited if the problem specifics allow one to exploit some symmetry properties). We were originally utilizing a parallel QR-based subroutine to achieve the diagonalizations. However, there exists a much more efficient method due to Cuppen that applies a divide-and-conquer procedure to achieve more precise numerics at nearly four times the speed on all problems we have tested on the target architecture. The method reduces the original problem to two independent symmetric tridiagonal eigenvalue problems of dimension  $k$  and  $N - k$ . This reduction can be repeated until a stopping condition is satisfied, at which point QR iteration is applied on a much smaller system. The subroutine `pzhheevd()` in the Cray Scientific Library implements the divide-and-conquer procedure but from our experience is not as broadly used as the QR-based `pzhhev()` subroutine.

We performed a strong scaling study comparing the parallel QR and parallel Cuppen's algorithms for a fixed small Hermitian problem of dimensionality  $4096 \times 4096$ . First, we intentionally pushed the problem until further parallelization yielded no further reduction in runtime. Figure 3.1.4 shows that the divide-and-conquer algorithm runtime is a quarter of the parallel QR runtime, the algorithm produces more accurate results, and it does not introduce additional floating-point operations as more processes are used. The trend of increased floating point computations and increased runtimes gets worse on larger matrices and larger process counts for the parallel QR. Because of its advantages, we now use Cuppen's algorithm.

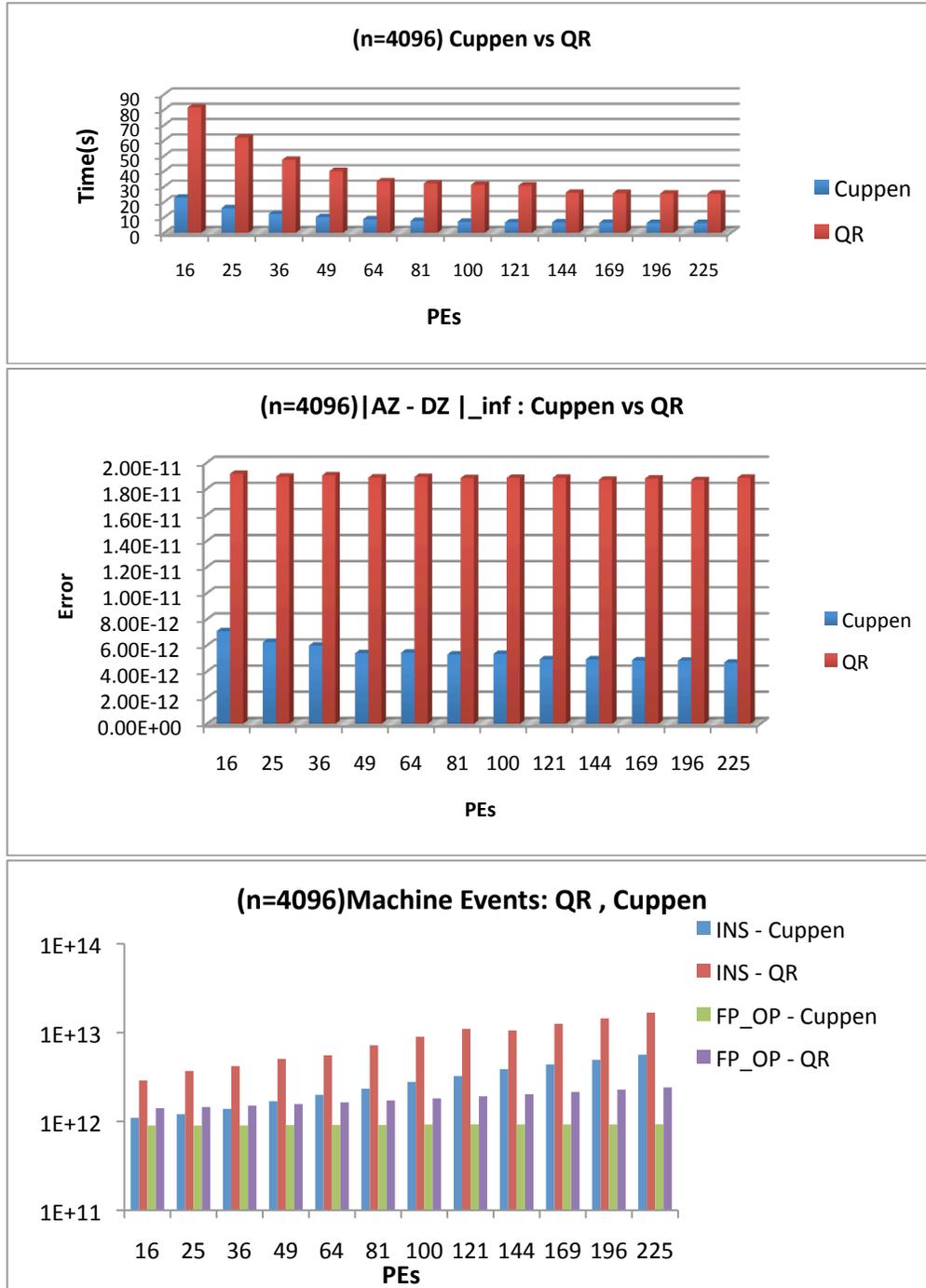


Figure 3.1.4: Strong scaling study comparing parallel QR and parallel Cuppen's algorithms for fixed small Hermitian problem, n=4096.

Figure 3.1.5 shows the execution time compared to the number of PEs on Jaguar Cray XT5 for three problems solved with the nuclear solver software. Each iteration is dominated by the cost to diagonalize the large Hartree matrices and perform another smaller diagonalization. The problems shown correspond almost exactly to executing the solver on  $30^3$ ,  $40^3$ , and  $50^3$  lattices, respectively. The  $50^3$  lattice problem (the problem of dimension 250,000 in the figure) defines a practical limit for the nuclear solver on the target architecture, since both the proton and neutron work groups have to diagonalize such a matrix

simultaneously. Thus in this example the total number of PEs for the  $50^3$  problem would be  $318 \times 318 + 318 \times 318 = 202,248$ , where the virtual rectangular process grid required by the diagonalization routines in each particle group would be 318 rows by 318 columns. (In fact, we could reduce the time of the computation slightly by executing at close to full-machine scale on  $334 \times 334$  grids.)

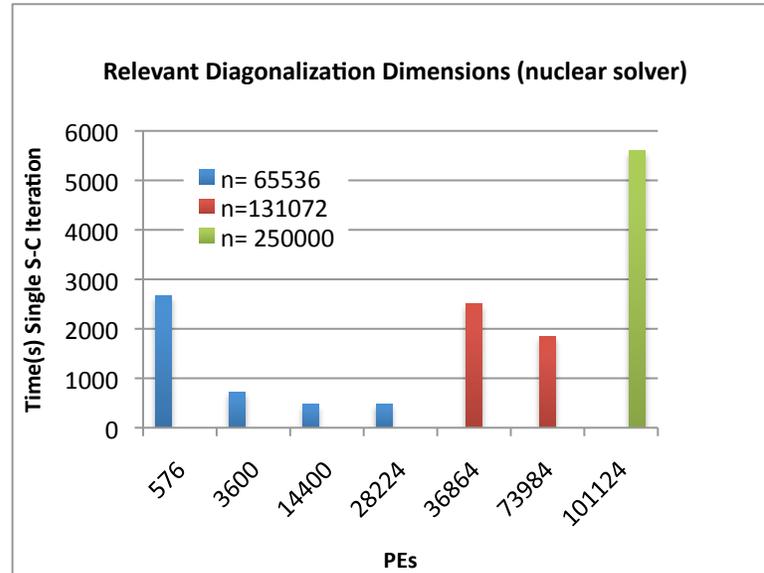


Figure 3.1.5: Execution time vs. Number of PEs for three problems for nuclear solver portion of software.

**Problem 2.** The nuclear problem is significantly more complex in both the solver and time-dependent implementations. In the Q2 benchmark we wanted to demonstrate the connection between the solver application and the time-dependent code, and test the accuracy of the time evolution of the nuclear system in the absence of any excitation. Owing to the cost of achieving full self-consistency in the general nuclear solver and desiring to test the newly developed time dependent capability, we chose a supposed spherical nucleus, Tungsten-198 ( $Z = 74$ ,  $N = 124$ ), and exploited existing software for spherical systems and knowledge of the system to hand-code a fairly accurate potential for the system into the solver code, therefore requiring the execution of only a single iteration for *self-consistent* convergence. The solver and time-dependent nuclear codes executed the problem on a  $40^3$  lattice with constant lattice spacing of 0.75fm in each spatial dimension and energy cutoff at 100MeV. The solver computed 7466 proton-dedicated and 8946 neutron-dedicated quasiparticle wave functions to an accuracy  $1 \times 10^{-8}$  MeV. The solutions are complex, four-component wavefunctions. Thus, the solver writes  $\langle NWF \rangle * 4 * N_x * N_y * N_z * \text{sizeof}(\text{double complex})$  BYTES to be used by the time-dependent nuclear code. For the benchmark problem this equates to about 64GB of wave function data. In addition to the wave function data, a small text file is created for problem-specific scalar data that is needed by the time-dependent code in order to match the problem specifics of the solver. The code was executed a second time and profiled again without the I/O turned on, thus isolating the cost of writing the wave function data needed by the time dependent code. For Run 1, the solver executed for 110m 36.589s (or 6636.589s) on 73728 PEs of the target architecture. Run 2 executed for 87m 52.916s (or 5272.916s). Since we are interested in the dynamics of nuclear systems, Run 1 is the relevant Q2 benchmark for the solver. The number of PEs is  $73,728 = 2 \times 192 \times 192$  because both protons and neutrons diagonalize a unique  $128,000 \times 128,000$  Hermitian matrix simultaneously in two disjoint  $192 \times 192$  virtual rectangular process grids. The relevant metrics for the solver are both the number of iterations required to achieve self-consistency and the cost per self-consistent iteration. Table 3.1.6 summarizes the measured machine events for the solver benchmark runs, which are separated into an initialization phase followed by the loop (in this case only a

single iteration of the loop) until self-consistency is achieved, plus the option of writing wave functions for the time-dependent code (Phase 2):

Machine Event (73,728PEs)	Run1 - Phase 1	Run 1 (I/O) - Phase 2	Run 2 – Phase 1	Run 2 – Phase 2
Instructions Retired	23346203445224852	1782654886997981840	23819899169287440	1401740648364960545
Floating Point Ops	25603102378	53582334657408658	25603102378	53459232137049469
Time(s)	160.635993	6377.887773	161.168643	5019.676603

**Table 3.1.6: Measured machine events for solver runs of second benchmark problem.**

In Q2, we reported the time-dependent nuclear code results in two phases. Phase 1 is the initialization phase during which the wave functions constructed by the solver software are read from disk and distributed in the proton and neutron groups respectively, all the relevant wave function data needed for time propagation are initialized, and the lattice parameters, potentials, densities, and energy are computed prior to entering the time stepping loop. Phase 2 of execution is the time evolution of the system including the write of the relevant time dependent densities and currents required for additional analysis. Neither the Q2 solver nor the Q2 time-dependent nuclear codes exploit collective I/O over the Lustre file system – Fortran semantics are used. The benchmark executed on as many processes as quasiparticle wave functions generated by the solver, 16,412PEs, plus two PEs used to compose and exchange densities between the neutron and proton communicators.. The time-dependent code completed execution in 35m 9.581s (or 2109.518s). The performance results are reproduced in Table 3.1.7. In the benchmark, 200 time steps were executed and < 2MB of densities (and no currents) were written. Thus, the cost to evolve the entire set of 16,412 wave functions on 16,414 PEs for a single time step is estimated as 4.792598095 s / ts. Said another way, if the job were executed on a single PE the time to evolve this system a single time step at this rate would require no less than 22 hours, and the entire benchmark would require nearly 4,400 hours or about a half year!

Machine Event (16,414PEs)	Phase 1	Phase 2
Instructions Retired	76088401412884728	48871991142253943
Floating Point Ops	2020709113712	1601488874412091
Time (s)	1125.904417	958.519619

**Table 3.1.7: Performance of time-dependent nuclear code on second benchmark problem.**

Table 3.1.8 summarizes the aggregated performance results of the Q2 benchmark of the current nuclear solver (Run1) and time-evolution software technology (Run 2, in bold):

Machine Data	Run 1	<b>Run 2</b>	Total
Instructions	1.806001090443207e+18	<b>1.249603925551387e+17</b>	1.930961483e+18
Floating Point Ops	5.358236026051104e+16	<b>1603509583525803</b>	5.518586984e+16
Wall Time (s)	6538.523766	<b>2084.424036</b>	8622.947802
CPU \$(hours)	133,908.9667	<b>9502.6575</b>	143411.6242
PEs	73,728	<b>16,412</b>	

**Table 3.1.8: Aggregated performance results of second benchmark problem.**

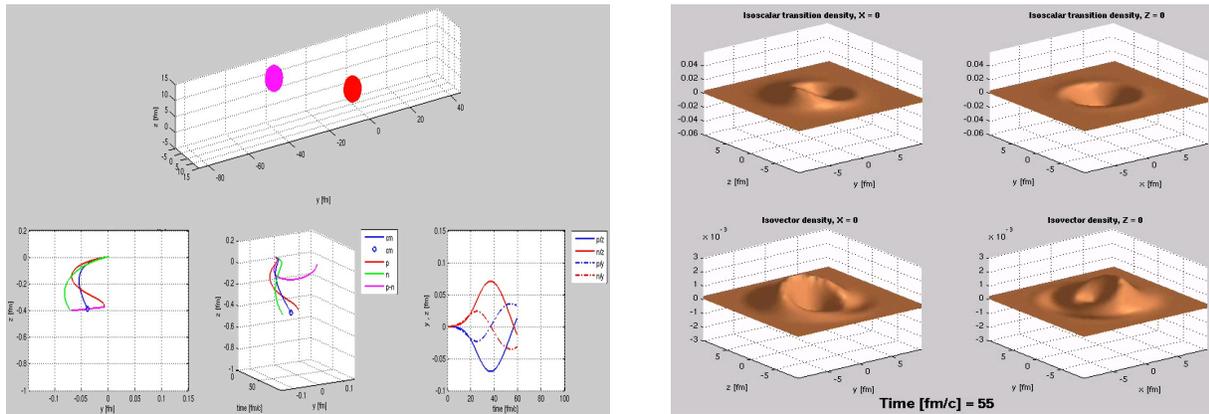
Table 3.1.9 reproduces selected observable data that was either computed or input by the nuclear solver, or subsequently computed by the time-dependent nuclear code and outpt after the last timestep in the second Q2 benchmark run. There is an error somewhere in the Q2 codes, since the number of

particles in the system should remain constant. However, the number of particles was not constant and in particular the number of proton particles is wrong. Another goal for Q4 is to demonstrate that the nuclear portion of the code is free of such numerical problems. We will identify and correct the problem by the Q4 deadline. The computing capability is still remarkable and unprecedented.

Observable	Nuclear Solver	Nuclear Time Evolver
E tot	-1521.753838292821	-1521.865932819248
E kin	3762.407463	3762.407463
Laplacian(Rho)	307.692969	307.692969
Rho * Tau	1275.879974	1275.879974
Gamma	13676.468327	13676.468327
Spin orbit	-89.878283	-89.878283
Coulomb	733.210176	733.210176
J (current)	-0.000001	-0.000003
Neutron pairing (n-p)	-3.668164309096726	-3.644882803655416
Proton pairing (p-p)	-8.477893427168159	-8.613267322628873
Proton particle number	74	76.49921931460780
Neutron particle number	124	124.7264648073978
A = Z + N	198	201.2256841220056

**Table 3.1.9:** Selected observable data computed (or input) by the nuclear solver or computed by the time-dependent nuclear code and reported after the last timestep in the second benchmark run.

The time-evolution code was demonstrated last summer at the UNEDF all-hands meeting. In the first application of TD-SLDA to the nuclear system, we considered the response of  $^{48}\text{Cr}$  to a Coulomb field generated by a relativistic projectile. The projectile interacts with the target from 20 fm to -20 fm. The impact parameter is  $z = 10$  fm, and the projectile moves in the  $yz$ -plane. The plots in Figure 3.1.6 are snapshots of a movie we made of the experiment. On the left is a schematic of the process and the evolution in time of the system, and proton and neutron centers-of-mass. On the right is the variation of isovector and isoscalar density of the target nucleus seen from the  $yz$ - and  $xy$ -planes.



**Figure 3.1.6:** Response of  $^{48}\text{Cr}$  to Coulomb field generated by relativistic projectile.

### 3.1.7 Developments, Enhancements, and Q4 Benchmark Results

In Q3 and Q4, we turned our primary focus to the nuclear code capabilities. We have completed the nuclear solver capability, addressed some numerical and efficiency issues in the time dependent code, have worked on the performance and scalability of the I/O capabilities connecting the solver to the time dependent code, and developed a scalable and high-performance checkpoint and restart function for extremely large nuclear data sets.

For the nuclear problem the static solver and TD-SLDA have the capability of using a general functional of the form that is highly non-linear and non-local in effect:

$$\mathcal{E}_T(\vec{r}) = \frac{1}{2M_n}\tau_n(\vec{r}) + \frac{1}{2M_p}\tau_p(\vec{r}) - \Delta(\vec{r})\nu_c(\vec{r})$$

$$\sum_{T=0} \left( C_T^\rho \rho_T^2 + C_T^{\Delta\rho} \rho_T \Delta\rho_T + C_T^\tau \rho_T \tau_T + C_T^j \vec{j}_T^2 + C_T^{\nabla J} \rho_T \vec{\nabla} \cdot \vec{J}_T + C_T^{\nabla j} \vec{s}_T \cdot \nabla \times \vec{j}_T \right)$$

where  $C_T$  are real coefficients fitted to reproduce observables (like binding energies and radii) over a large number of nuclei. Note that the Galilean invariance requires the following relations:

$$C_T^j = -C_T^\tau$$

$$C_T^{\nabla J} = C_T^{\nabla j}.$$

There are several sets of  $C_T$  coefficients corresponding to different functionals, which describe a large number of nuclei with varying degrees of accuracy. In the present code we have introduced two such sets (called SLY4 and SKM\*) in the solver, and one in the time-dependent code (SLY4). There is no particular reason we have not extended yet to cover all the functionals, but SLY4 is widely used in similar calculations. The one-body Hamiltonian is obtained by the minimization of the energy functional. It has the form:

$$h(\vec{r}) = -\vec{\nabla} \frac{1}{2m^*(\vec{r})} \vec{\nabla}$$

$$+ 2C_T^\rho \rho_T(\vec{r})$$

$$+ 2C_T^{\Delta\rho} \Delta\rho(\vec{r})$$

$$+ C_T^\tau \tau(\vec{r})$$

$$+ C_T^{\nabla J} \left[ \vec{\nabla} \cdot \vec{J}_T(\vec{r}) + \frac{1}{i} \left( \vec{\nabla} \rho_T(\vec{r}) \right) \cdot \left( \vec{\sigma} \times \vec{\nabla} \right) \right]$$

$$+ C_T^j \frac{1}{i} \left[ 2\vec{j}_T(\vec{r}) \cdot \vec{\nabla} + \left( \vec{\nabla} \cdot \vec{j}_T(\vec{r}) \right) \right]$$

$$+ C_T^{\nabla j} \left[ \vec{\sigma} \cdot \left( \vec{\nabla} \times \vec{j}_T(\vec{r}) \right) + \frac{1}{i} \left( \vec{\nabla} \times \vec{s}_T(\vec{r}) \right) \cdot \vec{\nabla} \right]$$

Taking into account the superfluid part of the functional one obtains the Hartree-Fock-Bogoliubov equation :

$$\begin{pmatrix} h_{++} - \mu & h_{+-} & 0 & \Delta \\ h_{-+} & h_{--} - \mu & -\Delta & 0 \\ 0 & -\Delta^* & -(h_{++} - \mu) & -h_{+-} \\ \Delta^* & 0 & -h_{-+} & -(h_{--} - \mu) \end{pmatrix} \begin{pmatrix} u_+ \\ u_- \\ v_+ \\ v_- \end{pmatrix} = E \begin{pmatrix} u_+ \\ u_- \\ v_+ \\ v_- \end{pmatrix}$$

However, because for even-even nuclei the mean field solution does not break the time-reversal invariance, one can show analytically that the terms involving the spin densities and the currents vanish. On the other hand, all the terms are included in the time-dependent code, and one first check that the interface between the two nuclear codes is correct is obtaining vanishing spin densities and small currents. The currents are not exactly zero because the gradient operator is not real in our implementation, but the errors decrease with increasing the momentum cutoff (or equivalently with decreasing the lattice spacing)

as they numerically should. Despite the small errors in the currents, it is important to use the current implementation of the operators on the lattice because of the compatibility with the Fourier transform. Recall that the two codes have a different but compatible implementation. In the solver, we employ a matrix approach in which all the operators are represented as matrices utilizing the DVR described in section 3.1.5. The matrix to be diagonalized is efficiently generated for both neutrons and protons in parallel and adhering to a block cyclic decomposition over two respective virtual process grids. In the time-dependent code, the operators are applied directly on the wave functions using FFTW. After Q2, the solver was redesigned. In Q2 the HFB solutions were obtained in two steps: first the Hartree matrix of dimension  $2N_{xyz}$  ( $2*N_x*N_y*N_z$ ) was diagonalized to obtain the H (HF) orbitals:

$$\begin{pmatrix} h_{++} - \mu & h_{+-} \\ h_{-+} & h_{--} - \mu \end{pmatrix} \begin{pmatrix} \phi_+^{(i)}(\vec{r}) \\ \phi_-^{(i)}(\vec{r}) \end{pmatrix} = \varepsilon_i \begin{pmatrix} \phi_+^{(i)}(\vec{r}) \\ \phi_-^{(i)}(\vec{r}) \end{pmatrix}$$

In the second step, only the orbitals up to a chosen cutoff ( $|\varepsilon_i - \mu| < E_{\text{cut}}$ ) were used as a basis in which one expands the u and v components, e.g.,

$\begin{pmatrix} u_+(\vec{r}) \\ u_-(\vec{r}) \end{pmatrix} = \sum_i C_i \begin{pmatrix} \phi_+^{(i)}(\vec{r}) \\ \phi_-^{(i)}(\vec{r}) \end{pmatrix}$ , where the complex coefficients  $C_i$  are obtained by a second diagonalization which includes the pairing part. Such an approach is efficient when the energy cutoff is relatively small. However, when the cutoff is large, as required in our calculations, the dimension of the second diagonalization becomes larger than  $2N_{xyz}$ , thus the whole calculation required two diagonalizations of dimension  $2N_{xyz}$ . This approach induces errors during the time evolution in the form of particle non-conservation. Therefore, in the Q4 implementation, we diagonalize directly a dimension  $4N_{xyz}$  matrix that includes *both* pairing and normal contributions at the same time.

The one-body Hamiltonian  $h(\vec{r})$  is Hermitian, but numerical errors can make the matrix non-hermitian without additional caution. Thus, to avoid this possible source of errors, we symmetrize the terms involving odd powers of gradients so that the Hamiltonian is numerically Hermitian in our basis on the lattice. One finds the Laplacian operator includes effective mass corrections:

$$-\vec{\nabla} \frac{\hbar^2}{2m^*(\vec{r})} \vec{\nabla} v(\vec{r}) = -\frac{1}{2} \left[ \frac{\hbar^2}{2m^*(\vec{r})} \vec{\nabla}^2 v(\vec{r}) + \vec{\nabla}^2 \left( \frac{\hbar^2}{2m^*(\vec{r})} v(\vec{r}) \right) - \left( \vec{\nabla}^2 \frac{\hbar^2}{2m^*(\vec{r})} \right) v(\vec{r}) \right]$$

The spin-orbit terms require special treatment:

$\vec{W} \cdot (\vec{\sigma} \times \vec{\nabla} v(\vec{r})) = \frac{1}{2} \left[ \vec{W} \cdot (\vec{\sigma} \times \vec{\nabla} v(\vec{r})) + \vec{\sigma} \cdot (\vec{\nabla} \times (\vec{W} v(\vec{r}))) - \vec{\sigma} \cdot (\vec{\nabla} \times \vec{W}) \right]$ . The term involving the current density becomes:

$$2\vec{j}_T(\vec{r}) \cdot \vec{\nabla} \phi(\vec{r}) + \left( \vec{\nabla} \cdot \vec{j}_T(\vec{r}) \right) \phi(\vec{r}) = \vec{j}_T(\vec{r}) \vec{\nabla} \phi(\vec{r}) + \vec{\nabla} \cdot \left( \vec{j}_T(\vec{r}) \phi(\vec{r}) \right).$$
 A similar

result holds for the  $\vec{\nabla} \times \vec{s}_T$  terms. While in Q2 the solver used the symmetrized operators, they were not symmetrized in the time dependent code. The symmetrization has been implemented and tested in the Q4 time dependent code. This numerical enhancement has introduced a considerable amount of extra work not performed in the Q2 code. The extra work is partially offset by another change in the Q4 code. In Q2 we were computing derivatives of the v components six times during a time step: four times for computing currents and twice when the evolution was performed. In the current implementation, we calculate the derivatives two times (each time step requires two applications of the Hamiltonian, thus

requiring the two calculations), save and reuse them when necessary. We have also eliminated a series of assignments in the most time-consuming sections of the code by more efficiently grouping the calculations together.

In the Q4 codes we employ a more accurate calculation of the Coulomb potential. In both Q2 nuclear codes the long-range part of Coulomb potential is computed analytically by considering a Gaussian distribution of charge with the same number of particles as the proton distribution. The Gaussian distribution is placed in the proton center of mass. This approach is particularly accurate when the proton distribution is spherical or almost spherical. In the presence of large deformations like when the nuclei are stretched with dipole and quadrupole fields, however, the deformation becomes too large to yield a correct description of the long range of the Coulomb potential. This is why in the current implementation we have added a second Gaussian distribution; both distributions are displaced symmetrically from the charge center, thus allowing a better description of the Coulomb field. The new implementation is present in both the static and the dynamic codes. In the future, we might need to add more Gaussian distributions in order to accurately describe non-axial deformations.

We have added, in the Q4 TD implementation, the possibility to turn on absorbing boundary conditions. While they are not necessary for low-amplitude excitations, such a feature is essential for simulations involving fragments that break and acquire kinetic energy so that they are pushed to the boundaries. Because of the periodicity of the lattice, the fragments exiting the box on one side, enter the box on the symmetric side in the absence of absorption thus making the analysis complicated and changing the process initially studied. In the presence of the absorbing boundary conditions, the fragment(s) do not reappear making possible the study of the remaining fragment.

As a test of the implementation, we have taken a fixed potential and obtained the eigenvectors, which we then feed into the time-dependent code. In the absence of external excitations, the solution is stationary, and, except for a time-dependent complex phase, the eigenvectors should remain the same. This is exactly what we observe, obtaining the time derivative zero within numerical errors. Therefore we are confident that the changes to the Q4 have been correctly implemented.

## I/O

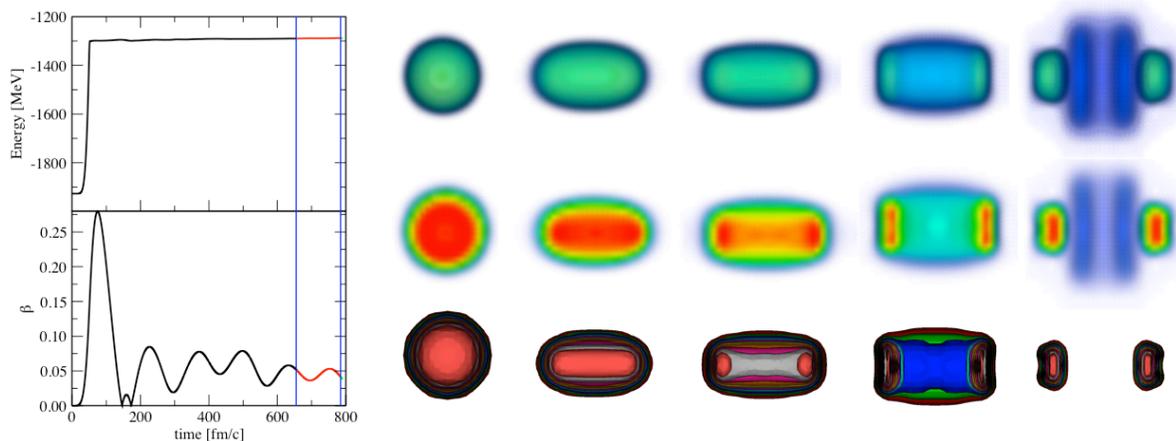
There have been four major I/O algorithms developed and tested since the Q2 codes were benchmarked: a new routine to store the solutions generated from the ground state solver to disk, a new routine to read and distribute the ground state solutions from disk into the time dependent code, a new routine to checkpoint the time dependent data, a new routine to read and distribute the time dependent data needed for restarting time dependent computations. Each routine is described briefly now.

In the Q2 solver the converged solutions were block cyclically decomposed over two disjoint rectangular ( $P$  rows  $\times$   $Q$  columns) virtual process grids—one for proton solutions, one for neutron solutions. The Q2 algorithm executed a loop in each communicator over the indices of the solutions to be retained whereby each decomposed solution was transformed into its global form by reduction to the root process at which time the root process would write the solution via Fortran semantics. The remaining processes waited for the root process to return from writing to continue the process until each solution had been assembled and written. Clearly, per communicator, there are  $P \times Q - P$  processes that do nothing but contribute a zero value during the reduction (assembly) phase, and  $P \times Q - 1$  processes that do nothing but wait on the writing process during the write. We modified the Q4 I/O algorithm in several ways. First, we form either 88 or  $Q$  I/O groups, whichever is larger, and assign the process that has the lowest canonical process rank from the group to be the root process responsible for leading the assembly of solutions to be written and performing the actual write of the solutions to an offset region in the solutions file. The actual I/O is now achieved using directly some Lustre semantics to prepare the file system to perform a parallel write event, then the write is executed in parallel. Thus, per communicator, there are  $P \times Q - N_{\text{igroups}} * P$  processes that do nothing but contribute a zero value during the reduction (assembly) phase, and  $P \times Q - N_{\text{igroups}}$  processes that wait on write.

Next, the Q2 time dependent nuclear code defined a proton and neutron communicator space and assigned a single process to execute the read and subsequent point-point distribution of the ground state solutions to the set of MPI processes allocated for the problem prior to initializing a time dependent run. Thus, each  $N_p - 2$  MPI processes waited per solution per communicator while while the root process and the process that owned the particular solution conducted an exchange. The Q4 code forms Niogroups and uses Lustre semantics to attack the read and distribution of solutions problem in parallel employing modulo classes over the Niogroups and point-point communications. Thus, in each communicator,  $N_p/Niogroups - 2$  processes wait per solution per I/O group per communicator while while the root process of the modulo class (1-1 with the I/O group) and the process that owns the particular solution conduct an exchange.

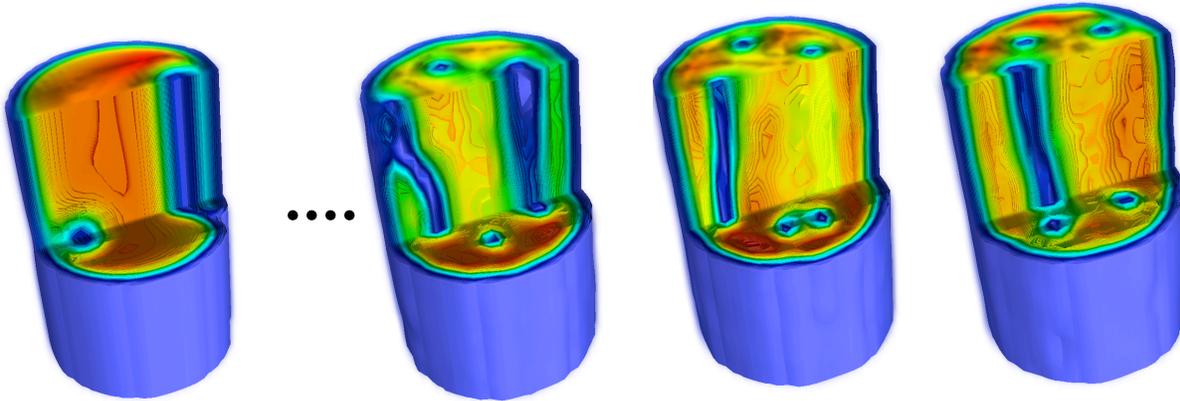
Furthermore, we have completed the nuclear time-dependent checkpoint and restart capability. At the end of Q2 we were able to read the solutions from the dft solver and iterate as many time steps possible in a single execution event. But, without a check point and restart capability, we were severely limited to short wall time scale simulations. We are now capable of very sophisticated nuclear studies that require 10K to 100K timesteps (and more if the simulation demands it and we are permitted the allocation). The algorithm is very similar to that employed in the unitary gas codes except that there are more I/O groups defined and that the total volume of data needed to continue an interrupted problem is  $44 * NWF * N_x * N_y * N_z * \text{sizeof}(\text{double complex})$ , a factor of two larger than the unitary gas code.

**Sample Result from Q4 Nuclear Codes** Figure 3.1.7 depicts a new result obtained with the enhanced and corrected Q4 nuclear codes. The energy and deformation parameter are plotted as a function of time for a quadropole excitation experiment on  $^{280}\text{Cf}$  on  $32^3$  lattice run and 43,380 qpws. The solver wrote roughly 85GB of solution data. In one execution event, the time-dependent code read and distributed the groundstate solutions (85GB), executed a total 10,951 time steps, completed 1095 I/O events of analysis data, and check-pointed over 932GB of data needed to continue the calculation. In a second execution event, the time dependent code restarted (reading and distributing over 932GB of data) at time step 10,952, executed 2,121 additional time steps plus 212 additional I/O events, and check pointed (>932GB) again. In the third and final run of the simulation experiment, the time dependent code restarted (again, reading and distributing over 932GB of data) at time step 13,073, executed 101 additional time steps plus 10 additional I/O events, and exited cleanly. In total 13,173 time steps were taken including 1317 I/O events of analysis data. The figure is intended to show the correctness of our nuclear restart capability (the vertical lines separate restart events). The nucleus is stretched to deformation and then let go to evolve. LACM ensues with subsequent induced fission process revealed.



**Figure 3.1.7: (left) Sample problem result demonstrating the capability of the nuclear codes to work together and resolve a highly non-trivial excitation. The nucleus is stretched to deformation and then let go to evolve.**

The energy and deformation parameter are plotted as a function of time for the quadropole excitation experiment on  $^{280}\text{Cf}$  - a  $32^3$  lattice run and 43,380 qpws. Three checkpoints and two restarts are plotted. The plot is intended to show the correctness of our nuclear restart capability (the vertical lines separate restart events). (right) Quadropole excitation of  $^{280}\text{Cf}$  nucleus. LACM ensues with subsequent induced fission process revealed.



**Figure 3.1.8:** Snapshots from unitary gas vortex formation calculation of the ball and rod excitation of the 300 particle dilute Fermi gas system executed in Q2. We have added a high resolution movie capability since Q2 to our analysis tools to be used for either unitary gas or nuclear systems. Currently we use VisIt to export JPEG files and then these are imported into Apple's iMovie software. We export MPEG-4 videos with annotation.

**Q4 Problem** In Q4 we again demonstrate the connection between the solver and the time-dependent code. First we execute the solver. Next we test the accuracy of the time evolution of the nuclear system in the absence of any excitation for 200 time steps. We wish to study  $^{238}\text{U}$  ( $Z = 92$ ,  $N = 146$ ). As in Q2, we computed a potential for the system and hand-coded the solver code with it thus requiring the execution of only a single iteration for convergence of the ground state solutions. The solver and time-dependent nuclear codes executed the problem on a  $40 \times 40 \times 64$  lattice with constant lattice spacing of 1.25 fm in each spatial dimension and energy cutoff at 100 MeV. The solver computed 67,118 proton-dedicated and 69,508 neutron-dedicated quasi-particle wave functions to an accuracy  $1 \times 10^{-8}$  MeV. Thus, the solver wrote  $\langle NWF \rangle * 4 * N_x * N_y * N_z * \text{sizeof}(\text{double complex})$  bytes of solution data to be used by the time-dependent nuclear code -this equates to about 833.8989258 GB of ground state data. In addition to the wave function data, a small text file is created for problem-specific scalar data that is needed by the time-dependent code in order to match the problem specifics of the solver. The Q2 solver machine event results were separated into two phases, so we present the Q4 results in two phases to make comparison to the Q2 I/O results possible: phase 1 includes initialization and the self-consistent step (that includes the diagonalizations); phase 2 is the I/O write phase. It is noted that there are several minutes we do not report in the numbers in Table 3.1.10 below – the difference between the ‘time’ function and the phases of computing measured internally. The difference is in the collection and organization of machine event data, the load and exit times of the binary on over 97% of the entire machine, and the memory munging and lattice cleansing routine at the end of the Q4 solver. We use the wall time result from the ‘time’ function in the final analysis.

real 306m33.181s				
user 1m1.172s				
sys 0m2.520s				
	time	ins	fp	dcm
h:	263532	23818606378382	35517036900	11608961526

D:	12852934280	5417482530533416527	438602965016290213	1229776528957498
SC:	4740803452	1382169375376583812	31667517838858134	282025495545468
WR:	245194223	78045799390130327	34603299588819	8045835615805
	time	ins	fp	dcm
h:	263995	23807612809267	35517036900	11562647556
D:	11407030138	4754089970506414484	438597416050424247	1183927475665207
SC:	6186768475	2012390402868940080	32795357968382774	315267396285921
WR:	236927985	83437414579954467	35895770295819	9678774688982

**Figure 3.1.7: Raw solver machine event data (protons and neutrons).**

Machine Event (217,800PEs)	Phase 1	Phase 2
Instructions Retired	1.356617990550454e+19	1.614832139700848e+17
Floating Point Ops	9.416633279080292e+17	70499069884638
Time (s)	17594.062608	245.194223

**Table 3.1.10: Performance results of Q4 238U benchmark solver problem.**

The time-dependent nuclear code executes in two phases as in Q2. Phase 1 is the initialization phase during which the wave functions constructed by the solver software are read from disk and distributed in the proton and neutron groups respectively, all the relevant wave function data needed for time propagation are initialized, and the lattice parameters, potentials, densities, and energy are computed prior to entering the time stepping loop. Phase 2 of execution is the time evolution of the system including the write of the relevant time dependent densities and currents required for additional analysis. 200 time steps were executed in Q4 on the 238U ground state solutions, and all densities and currents were written each 10 time steps (< 85MB). We present the raw data for the time-dependent code in Figure 3.1.8, and the machine event results of the Q4 benchmark of the enhanced nuclear solver and time-evolution software technologies in Table 3.1.11.

real	33m51.541s			
user	0m27.878s			
sys	0m1.740s			
	time	ins	fp	dcm
init:	32228731	9045196589846562	91057643393	363095113632
I/O:	330729676	163810654394246847	55971027102	1848702550452
t_loop:	1386519066	537846559382408604	18891863365740396	528423132128308

**Figure 3.1.8: Raw time-dependent code machine event data.**

Machine Event (136628PEs)	Phase 1	Phase 2
Instructions Retired	1.728558509840934e+17	537846559382408604
Floating Point Ops	147028670495	18891863365740396
Time (s)	362.958407	1386.519066

**Table 3.1.11: Performance results of Q4 238U benchmark time-dependent problem.**

Machine Data	Solver	Time Dependent	Total
Instructions	1.372766311947462e+19	<b>7.10702410366502e+17</b>	1.443836552984112e+19
Floating Point Ops	9.417338269779139e+17	<b>1.889201039441089e+16</b>	9.606258373723249e+17
Wall Time (s)	18393.181	<b>2031.541</b>	20424.722

CPU \$(hours)	1112787.4505	<b>77101.495485555556</b>	1189888.945985555556
PEs	217800	<b>136628</b>	

**Table 3.1.12: Aggregated performance results of the Q4 238U solver and time-dependent benchmark runs.**

Table 3.1.13 displays the same set of selected observable data as in Q2 that was either computed or input by the nuclear solver, or subsequently computed by the time-dependent nuclear code and output after the last timestep in the second Q2 benchmark run. There is a small discrepancy in pairing because in the solver we renormalize the pairing with the initial potentials, while in the first time step of the time dependent code we renormalize with the new potentials. If the ground state solutions were perfectly converged, then the effect would be nullified – reduced to machine precision numerical error. Also, there is a small current in the time dependent system that is not present in the static code. The error in the coulomb energy comes from the fact that in the static code we use the correct number of protons, 92.0, to calculate Coulomb, but in the time-dependent code the number, 92.02266357, computed from the density is used. To correct the difference simply multiply the time-dependent code result by 92/92.02266357.

Observable	Nuclear Solver	Nuclear Time Evolver
E_tot	-1737.68175351	-1737.439398333015
E_kin	4429.134164	4429.134164
Laplacian(Rho)	341.54	341.542143
Rho * Tau	1500.276387	1500.276387
Gamma	16244.888350	16244.888350
Spin orbit	-97.826035	-97.826035
Coulomb	984.727855	984.970436
J (current)	0.0000000...	-0.000441
Neutron pairing (n-p)	-2.781041295199195	(-2.781745279856377, -1.4717138513362090E-021)
Proton pairing (p-p)	-6.9377623480194758E-002	(-6.8458565020630524E-002, 1.2030492640979142E-021)
Proton particle number	92	92.02266357252937
Neutron particle number	146	146.0462954802713
A = Z + N	238	238.0689590528006

**Table 3.1.13: Selected observable data computed (or input) by the nuclear solver or computed by the time-dependent nuclear code and reported after the last timestep in the Q4 benchmark run.**

### 3.1.8 SLDA Summary of Results

#### Comparison of Q2 and Q4 Solvers

PEs	= 2.9541015625 ( 217800 / 73,728 )
TIME	= 2.813047968968719 ( 18393.181 / 6538.523766 )
INS	= 7.601137780102748 ( 1.372766311947462e+19 / 1.806001090443207e+18 )
FP_OP	= 17.575445023312077 ( 9.417338269779136e+17 / 5.358236026051104e+16 )
QPWFs	= 8.324762368998294 ( 136626 / 16412 )
CMPLX / WF	= 1.6 ( 4x40x40x64 / 4x40x40x40 )
TIME IO(wr)	= 0.180527320357703 ( 245.194223 / 1358.21117 ) (ref. Table 3.1.6)
BYTES IO(wr)	= 13.31961979039727 ( 4x40x40x64x136626x16 / 4x40x40x40x16412x16 )

**Figure 3.1.9: Ratios used in comparison of Q2 and Q4 solver performance.**

Here we provide a scaling analysis of the performance of the Q2 and Q4 codes to generate ground state solutions for 198W (Q2) and 238U (Q4). In weak scaling we define the scaling factor to be  $K = (T(Q4)/T(Q2)) \times (PE(Q4)/PE(Q2)) := 8.310029400517944$ . We can then compare the measure of work executed completing the Q2 and Q4 codes – essentially the number of floating point operations executed, the number of instructions retired, and the number of bytes of useful data stored to disk for the time-dependent code. The ratio of retired instructions is  $INS(Q4)/INS(Q2) = 7.601137780102748$  ( $1.372766311947462E19/1.806001090443207e+18$ ) thus achieving 91.4694% of weak scaling relative to the Q2 problem. Similarly, the ratio of executed floating point operations is  $FP\_OP(Q4)/FP\_OP(Q2) = 17.575445023312077$  ( $9.417338269779136e17/5.358236026051104e16$ ) –achieving 211.4967% of weak scaling for floating point operations relative to the Q2 problem (  $111,150,055$  FP / s / PE in Q2 vs  $235,078,792$  FP / s / PE in Q4). For the amount of solution data written we find  $BYT(Q4)/BYT(Q2) = 13.31961979039727$  ( $4 \times 40 \times 40 \times 64 \times 136626 \times 16 / 4 \times 40 \times 40 \times 40 \times 16412 \times 16$ ) which is 160.2837% of the number of BYTES in the solution set from Q2. This number should come as no surprise because it basically scales with the lattice dimensions which we see have the ratio  $1.6X = 40 \times 40 \times 64 / 40 \times 40 \times 40$  between Q4 and Q2. We note also that lost in the interpretation of instructions retired are two very significant enhancements we made to the solver since Q2. First, instead of solving smaller diagonal sub-blocks as in Q2, in Q4 the algorithm was redesigned to diagonalize the entire BdG matrix each self-consistent step in both the proton and neutron communicators at the same time. The Q4 matrix diagonalized has dimension  $N = 409,600$ . The complexity of the direct factorizations operation scales as  $O(N^3)$  and so we expect an increase in floating point work executed during the diagonalizations to be  $\sim 1.6^3 = 4.096$ . This algorithm is extremely rich in L3 BLAS and as such a single instruction can execute multiple floating point computations. Looking at the number of instructions per floating point operation in Q2 we find  $33.705142544349398$  INS / FP\_OP. However, in Q4 we measure  $14.577009688106458$  INS / FP\_OP ( $2.312212399217227X$ ). This change in complexity of the diagonalization accounts only for part of the larger Q4 floating point count. The remaining difference is due largely to the remainder of the self-consistent step when the computation of potentials and densities are formed by operations on a larger set of vectors than in Q2 and an increased complexity per vector operation. Second, for network operations such as for inter-process communication and I/O, the amount of data transferred per instruction is lost in the raw measurement of hardware events on the chip. The rate of conducting I/O was addressed head-on in this GPRA-PMM enhancement campaign resulting in a new parallel Lustre write routine that back transforms the block cyclically decomposed solutions into wavefunctions and writes them to disk in parallel. Using Table 3.1.6 and Table 3.1.10 we can compare the rate of conducting the writes. In Q2 this number was measured as  $47.2013$  MBps. The Q4 assembly and write routine achieved the write rate of  $3482.5963$  MBps. Comparing the two rates we note that the Q4 I/O routine is nearly 75X faster than the Q2 solver I/O write routine. The last significant comment is to note that in Q2 the solver utilized 32.87% of the entire machine whereas in Q4 we exercised 97.12% of the total system.

### Comparison of Q2 and Q4 Time-Dependent Codes

PEs	=	8.323869867186548	(	136628	/	16414	)
Time	=	0.974629425161743	(	2031.541/	2084.424036)		
INS	=	5.687421396767019	(	7.10702410366502e+17/	1.249603925551387e+17)		
FP	=	11.781663538842758	(	1.889201039441089e+16/1603509583525803)			
QPWFs	=	8.324762368998294	(	136626	/	16412	)
CMPLX / WF	=	1.6	(	4x40x40x64	/	4x40x40x40)	
Time IO(rd)	=	0.322370532986372	(	362.958407/1125.904417)			
BYTES IO(rd)	=	13.31961979039727	(	4x40x40x64x136626x16	/	4x40x40x40x16412x16)	

**Figure 3.1.10: Ratios used in comparison of Q2 and Q4 time-dependent code performance.**

Here we provide a scaling analysis of the performance of the Q2 and Q4 codes to use the ground state wave functions computed by the solvers in Q2 and Q4 to time evolve 198W (Q2) and 238U (Q4) for 200 time steps. We define the weak scaling factor to be  $K = (T(Q4)/T(Q2)) \times (PE(Q4)/PE(Q2)) :=$

8.112688503777179. We again compare the measure of work executed completing the Q2 and Q4 time dependent problems –the number of floating point operations executed, the number of instructions retired, and the rate of reading useful data from disk and distributing it over the MPI processes. The ratio of retired instructions is  $INS(Q4)/INS(Q2) = 5.687421396767019$  thus achieving 70.1052% of weak scaling relative to the Q2 problem. The ratio of executed floating point operations is  $FP\_OP(Q4)/FP\_OP(Q2) = 11.781663538842758$  ( $1.889201039441089e+16/1603509583525803$ ) –achieving 145.2251% of weak scaling for floating point operations relative to the Q2 problem. The floating point performance in the time loop remained essentially (98%) constant. The number of degrees of freedom for the problem was 13.31961979039727X more complex in Q4 versus Q2. Similarly, the amount of solution data read and distributed is the same as the amount written such that  $BYT(Q4) / BYT(Q2) = 13.31961979039727$  ( $4 \times 40 \times 40 \times 64 \times 136626 \times 16 / 4 \times 40 \times 40 \times 40 \times 16412 \times 16$ ). Using Table 3.1.7 and Table 3.1.11 we can compare the rate of conducting the read, distribute, and initialization in the time dependent codes. Assuming most of the time is spent in I/O we approximate that in Q2 this number was 56.9403 MBps. In Q4, for the same phase of computing and assumptions, we measure 2352.6456 MBps. Comparing the two rates we note that the Q4 algorithm for this I/O-distribute-initialize phase is  $\sim 41.32$  times faster than the Q2 equivalent routine. Thus, it is largely because the I/O is so much faster that the Q4 time dependent code achieves a hyper-weak scaling trend. We note, however, that time to compute a time step is the essential feature. We would expect that if we ran the Q2 routine on the Q4 problem that the wall time per timestep in the case that we map one wave function to one MPI process would be at least 1.6X larger. That is we predict that the Q2 code would execute at the rate of  $(time / ts(Q2)) \times 1.6 := 7.668156952$  s/ts. However, the enhanced Q4 algorithm executes the Q4 problem at the rate of 6.93259533 s/ts. Thus, the Q4 time stepping algorithm outperforms the Q2 time stepping algorithm achieving a hyper-weak scaling result, 110.61% of the expected scaled value when compared to the Q2 performance. The Q2 time dependent code executed on  $\sim 7.32\%$  of the entire system, whereas the Q4 time dependent code executed on  $\sim 61\%$  of the system.

## 3.2 POP

### 3.2.1 Introduction

The Parallel Ocean Program (POP) is an ocean general circulation model used for ocean and climate studies. It was written in the early 1990s for the Connection Machine by Smith, Dukowicz, and Malone [1], derived from previous models by Semtner [7] and Bryan [8]. In 2001, POP was officially adopted as the ocean component of the Community Climate System Model (CCSM) and the model is released to the public both as a standalone code through Los Alamos National Laboratory (LANL) and as part of the CCSM through the National Center for Atmospheric Research (NCAR). The model has continued to evolve in its numerical algorithms, computational implementation and physical parameterizations with developments from LANL, NCAR, and a broad external developer and user community. In more recent versions, ocean ecosystem models and biogeochemical processes have been added to the physical model of the ocean. POP remains one of the primary ocean models in use for global climate and ocean research. The last public release of the ocean-only model was POP 2.0.1 in January 2004. Updated versions have been included in the 2004 release of CCSM 3.0 and the 2010 release of CCSM 4.

### 3.2.2 Background and Motivation

The POP model is used for both climate change research and oceanographic research and thus has two broad modes of operation. For climate change research, POP is coupled to atmosphere, land and sea-ice models for a complete simulation of the Earth system. Because climate change simulations must be integrated for centuries, these simulations are at a relatively coarse resolution (1 degree or roughly 100 km). In addition, some of these simulations include many additional tracers and reactions to simulate the response of ocean ecosystems and biogeochemical feedbacks, especially for the global carbon and sulfur cycles. The computational focus of these simulations is achieving the maximum throughput at coarse resolution, a low resolution that does not provide enough degrees of freedom to scale the model to large processor counts.

The use of POP for oceanographic studies has focused on the impact of mesoscale eddies on the global circulation. Eddies are typically tens of kilometers in size, so these simulations have been run at resolutions of 0.1 degrees (roughly 10 km). Because of the large workload and smaller timesteps at such resolutions, these simulations have typically been run for only decades of simulation time. However, the ocean simulations at this resolution are far more realistic than the coarse resolution climate simulations.

Computational capabilities have now reached the point where these two efforts are merging and there is a funded effort to run a coupled climate model at very high resolutions. An eddy-resolving ocean at roughly 10 km resolution will be coupled to an atmospheric model at roughly 25 km resolution. Each component of this model is more accurate and has shown greater realism, so the coupling of these models is expected to provide a much better simulation of the Earth's climate and more accurate projections of future climate change.

### 3.2.3 Capability Overview

A complete description of the POP model is described in the *POP Reference Manual* [9, 10]. The model solves the full three-dimensional Navier-Stokes equations of fluid motion under the hydrostatic and incompressible approximations. Temperature and salinity are transported and a nonlinear equation of state is used to provide density and pressure as a function of local temperature and salinity of ocean water. Several advanced mixing parameterizations are provided to simulate subgrid-scale mixing processes in both the horizontal and vertical directions. A wide variety of output is available, including scalar diagnostics, snapshots of many fields, time-averages of many fields, floats, drifters and transports across various straits.

POP utilizes a finite-difference discretization of most spatial operators. Most of these are second-order centered-in-space, although a third-order upwinding scheme and some limiters are available for the advection terms and for enforcing positive-definite transport. The model is integrated in time using a time-split algorithm. Most of the three-dimensional vertically varying (baroclinic) terms are advanced in time using a fully explicit second-order centered leapfrog scheme. However, the fastest mode of the system is a vertically uniform (barotropic) gravity wave that is advanced implicitly using a free-surface formulation that solves for the surface pressure. A preconditioned conjugate gradient solver is used for this mode.



**Figure 3.2.1: A tripole grid in which the geographic pole has been displaced into two continents.**

The equations are solved on the surface of the Earth. POP supports generalized coordinates and a variety of different horizontal grids. For global problems, we avoid grid convergence near the geometric poles by displacing those poles into land points using either displaced-pole grids [11] or tripole grids [12] (see Figure 3.2.1). All of these grids are logically rectangular, except at domain boundaries or tripole branch cuts. The vertical grid is an Eulerian depth coordinate with variable resolution that uses higher resolution in the surface layers to resolve the ocean mixed layer. Realistic topography is used with partial bottom cells to provide a better resolution of steep topography; no-slip conditions are imposed at land boundaries.

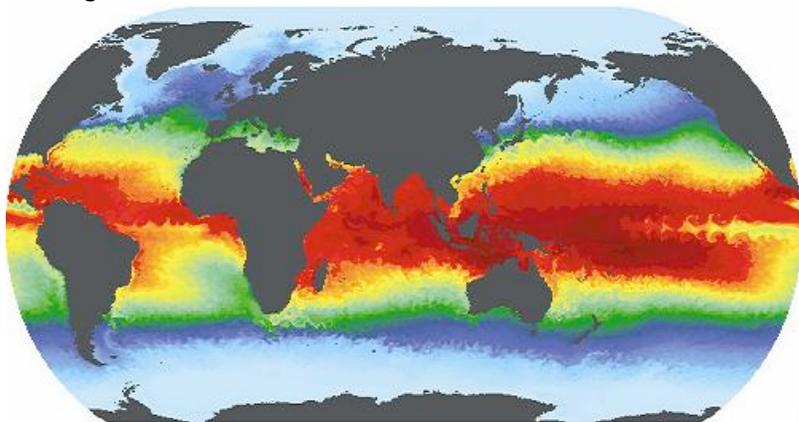
Ocean simulations are forced at the surface by atmospheric fields or fluxes, including wind stress, heat and water fluxes. These are either computed from climatological data sets for ocean-only simulation, or passed to POP from the CCSM flux coupler in the case of fully coupled simulations. For regional simulations, sponge layers are used at open lateral boundaries and a varying restoring to ocean climatology is imposed.

The parallel implementation of the model follows a domain decomposition approach. The horizontal domain is subdivided into Cartesian blocks. The size of the blocks can be made arbitrarily large or small to optimize for performance. Blocks consisting entirely of land points are eliminated, and the remaining blocks are distributed among nodes using either a Cartesian distribution or one of two methods that provide a load-balanced distribution. Multiple blocks can be distributed per node to provide opportunities for a hybrid threaded/message-passing model, with threading used across the multiple blocks assigned to a node. Small blocks generally lead to better cache performance, more land elimination, and better load

balancing. However a point of diminishing returns is reached due to increased communication requirements for the high surface-to-volume ratio. Large blocks can be used if large vector lengths are required (e.g., on machine architectures like that of the Earth Simulator). The vertical direction is not decomposed, so vertical columns are always retained locally. A hybrid OpenMP model was implemented for the 2.0.1 release, but lack of use in previous architectures has left this implementation broken and untested. That will be one focus of this GPRA-PMM effort.

### 3.2.4 Science Driver for Metric Problem

As mentioned above, high-resolution ocean simulations have shown a dramatic increase in realism due to the resolution of mesoscale eddies [13]. Improvements include better representation of western boundary currents like the Gulf Stream and frontal systems like the Azores front. Resolving eddies is also important for the dynamics of the Southern oceans and for simulating the eddy pumping of nutrients in biogeochemical simulations. The near-surface temperature from an eddy-resolving simulation is shown in Figure 3.2.2.



**Figure 3.2.2: Temperature at 15 m depth from a global 0.1-degree simulation using POP.**

Until recently, we have been able to afford to perform such high-resolution simulations at the scale of only a few decades and for the ocean only. We are now in the process of developing a fully coupled, high-resolution configuration of the CCSM using the eddy-resolving POP model coupled to a 25 km resolution atmosphere model. This model will be run for century-scale climate change simulations. For these simulations, a throughput of more than one simulated year per CPU day is required for the fully coupled system. In addition, because the ocean must share memory with other components, careful attention to memory allocation is required. Finally, in coupled climate simulations, we will need to output a more complete set of diagnostic fields and to evaluate extreme events, will need to output data more frequently in time, at least for specific time slices.

### 3.2.5 The Model and Algorithm

For this effort, we will use the latest LANL POP repository version (current version at 1 Feb 2010). We will configure POP in an ocean-only mode that is effectively identical to the anticipated coupled simulation and similar to previous ocean-only simulations. The model will use a tripole 0.1-degree global grid (3600×2400×42 grid points). Tracer advection is performed using a centered spatial discretization and biharmonic lateral mixing is chosen for both tracers and momentum. Vertical mixing is performed using the k-profile parameterization (KPP). The model will be forced at the surface using monthly normal-year forcing. The simulation will proceed for 3 simulated days with a time step of 10 minutes and will output many fields each simulated day to benchmark a high-frequency output time slice.

### 3.2.6 Q2 Baseline Problem Results

For the initial baseline, we performed the benchmark problem described above for three different machine configurations. The configurations differed only in core count, with 4800, 9600 and 14400 PEs used for each simulation. The horizontal subdomain block sizes used in each are 30×60, 30×30, and 30×20, respectively. A standard Cartesian distribution of blocks is used with one block per core; load balancing distributions at these block sizes are not as efficient due to the high surface to volume ratio for halo updates. The programming model was pure MPI for each (the original threading implementation was not functional for the initial baseline and will be one focus of performance improvement in this study). The compiler and environment variables are shown in an appendix.

Table 3.2.1 shows wall time results of the Q2 POP code on the fixed benchmark problem. The baroclinic portion of the code is computationally intensive and fully explicit in time, so it scales well between 4800 and 9600 cores. At 14400 cores, the subdomain size is only 600 points (20×30 Cartesian patch) and the work per core is becoming very small. The barotropic solver is a two-dimensional elliptic solve using a preconditioned conjugate gradient algorithm, and the compute time is dominated by either the global reductions or the implicit barrier that those reductions impose on the algorithm. The matrix operator is only a two-dimensional, nine-point stencil, and does not generate enough computation to overlap communications. This is a known performance issue in POP but will not be the focus of this enhancement exercise.

Previous production simulations of this ocean-only configuration have typically output only at monthly intervals, as that has been adequate for the ocean timescales. For this benchmark, we are performing more frequent output, which will be required for some of the planned fully coupled simulations. It is clear that the current parallel I/O implementation is not adequate for this application. The current I/O implementation is only parallel in the number of vertical levels, and much of the frequent output required will be for two-dimensional surface fields and no parallelism is available in the current I/O library. We will be implementing new parallel I/O libraries to improve this performance as part of this year’s GPR-PM activities. As such, we will focus on a strong scaling result between the 4800 PE and 9600 PE cases. The machine events and timing results for these two cases are summarized in Tables 3.2.2 and 3.2.3 for the Q2 benchmark problem.

PEs	4800	9600	14400
Wall Time (s)	957.842493	1011.535276	1450.240505

**Table 3.2.1: Q2 POP wall times for the benchmark at three processor core counts.**

4800 PEs, Q2	Time(s)	INS	FP OP
Barotropic	220.285649	3362619394734242	10914798749862
Baroclinic	84.623336	638046552543018	123489441332158
T avg	554.416994	10459543609613288	22070416032
Movie	98.516514	1838543581529579	15638400
TOTALs	957.842493	1.629875313842013e+16	134,426,326,136,452

**Table 3.2.2: Q2 POP machine events, 4800 PE benchmark.**

9600 PEs, Q2	Time(s)	INS	FP OP
Barotropic	161.178819	5237555077073871	11696471278395
Baroclinic	48.676842	741726106983124	133265275114487
T avg	680.398445	26639716932532043	24868717776
Movie	121.281170	4598600128921236	31276800
TOTALs	1011.535276	3.721759824551027e+16	144,986,646,387,458

**Table 3.2.3: Q2 POP machine events, 9600 PE benchmark.**

### 3.2.7 Q4 Code Improvements

Because POP users regard the wall time to execute a problem to be the most significant measure of productivity, the initial plan was to embark on a strong scaling pursuit to add hardware to minimize the wall time spent for computing the benchmark problem defined in Q2. However, upon inspection of the code we realized that the I/O phases of the code might be re-written to achieve a significant performance gain in efficiency. We consider first focusing on the 4800 PE Q2 benchmark where 652.933508s of the total run time of 957.842493s, or 68.1671% of the walltime, was spent doing I/O. The simulation executed 3 simulated days. Each day observable data and movie related data were written to disk. The observables were formed in 8 3D fields and 19 2D fields. Each process manages a  $60 \times 30 \times 42$  fragment per 3D field and a  $60 \times 30$  fragment per 2D field. The data type for all the I/O data is float. Thus, the volume of observable data written per day for the target problem is  $(8 \times 4 \times 30 \times 60 \times 42 \times 4800 + 19 \times 4 \times 30 \times 60 \times 4800)$  B / day = 11.4262104 GB / day, or about 35 GB for the benchmark at 1 observables file per day. There are 60 movies formed each day from coordinate data. The  $3600 \times 2400$  coordinate movie data is decomposed over a virtual  $60 \times 80$  rectangular process grid where each process owns a  $60 \times 30$  block of the global data set. The total volume of data written for movies is  $60 \times 4 \times 60 \times 30 \times 4800$  B / day = 1.931190491 GB / day or 5.793571472 GB for the entire benchmark problem.

In Q2, the observable data for each computed day was gathered in the interior of nested loops over the fields and  $k$ -values to a single lead process one  $4800 \times 60 \times 30$  block at a time, then written (Fortran I/O) to a single open file by the gathering process. Thus, during each loop iteration,  $NP - 1$  PEs wait while a single process writes to the disk a very small amount of data. In Q4, we have introduced a C routine that targets a single Lustre file for a parallel write. We define NIOPE processes to participate in the parallel write. For the benchmark problem we define 42 PEs per 3D field to be part of the I/O group so that  $NIOPEs := 8$  (3D fields / day)  $\times$  42 (  $k$ -values / fields )  $\times$  1 ( PE /  $k$ -value) = 336 IOPEs / day. Now, instead of a single process writing after each gather, the Q4 code executes a targeted gather phase where first the data affiliated with each  $k$ -value and field are used to identify the process ID of the gathering process. After gathering the data to be written into the set designated, of disjoint processes, then  $NP - NIOPEs$  wait one time while each IO process executes a write with offset into the Lustre file. Finally, the Lustre file parameters and number of processes to include in the write for the 4800 PE run were determined by executing an oracle code to execute a micro-kernel of the POP I/O phases. We conducted basically an exhaustive search over sensible combinations of NIOPEs, number of OSTs, and stripe size setting. For the 2D fields, one can reuse the described algorithm assuming there is only a single  $k$ -value. Thus, we employ a single I/O PE for each 2D field and execute the gather and write as just described in the case of 3D fields. We note that for the combined assembly and write of the 3D and 2D observable data, the Q4 algorithm is 7.595252132037274 times faster than the Q2 version for the 4800 PE problem.

Next, for the movie files, we again introduce a set of designated, disjoint I/O processes to replace the single process write –in this case we assign a single I/O process per movie. We first execute a gather phase where the block decomposed data is sent to the process with MPI process ID equal to the movie index. The I/O processes then locally copy the data from the receive buffer into a write buffer thus restoring the global indices –transforming to a column major ordering of the coordinates in the  $3600 \times 2400$  spatial grid. At this point,  $NP - NMOVIE$  PEs wait while  $NMOVIE$  PEs write with offset to a single Lustre file. Looking at the performance numbers between Q2 and Q4, we note that the Q4 code is 7.946443175395545 times faster on the same problem and 4800 PEs.

As a last point, the microkernel code also included a correctness check to insure that all the index mappings were correct and in compliance with the POP demands.

### 3.2.8 Q4 Metric Problem Results

The enhancement described was designed to make the 4800 PE run more efficient to lead the way for the strong scaling result when comparing the Q2 4800 PE performance results to the Q4 9600 PE

performance results. The raw machine-event data for the enhanced Q4 codes on the Q2 benchmark problems are listed in Tables 3.2.4 and 3.2.5.

4800 PEs, Q4	Time(s)	INS	FP_OP
Barotropic	162.845484	2493523139608176	10918903717734
Baroclinic	81.234007	611926226154622	123489442062604
T_avg	72.995206	1369947333186195	22070417409
Movie	12.397561	228560389936546	15640101
TOTALs	329.472258	4,703,957,088,885,539	134,430,431,837,848

**Table 3.2.4: Q4 POP machine events, 4800 PEs, Q2 problem.**

9600 PEs, Q4	Time(s)	INS	FP_OP
Barotropic	143.867992	4352776136294947	11696471278395
Baroclinic	47.994133	755616085382567	133265275114487
T_avg	84.648207	3180959264572214	24868719153
Movie	13.812455	505002308418671	31278501
TOTALs	290.322787	8,794,353,794,668,399	144,986,646,390,536

**Table 3.2.5: Q4 POP machine events, 9600 PEs, Q2 problem.**

### 3.2.9 Analysis

The TIME ratio in Figure 3.2. shows that we have exceeded a factor of two in speedup comparing the Q2 and Q4 codes run on the same problem for the 4800PE case. The ratio FP\_OP is very close to one, meaning that the number of floating point operations remained essentially the same; this makes sense because the changes in the code were purely in the I/O implementation. We have enhanced the efficiency of the code by more than a factor of two! While the new I/O algorithms were written for the Q2 problem instance and 4800 PEs, it is easy to generalize them to most POP problems with little additional work. For lack of resources, we stopped once the speed-up required to satisfy the GPRA-PMM metric was reached. We derive the following ratios from the raw data in Tables 3.2.2 and 3.2.4 as part of the analysis of POP's performance.

PES	: 1
TIME	: 0.343973315454068 (329472258 / 957842493)
INS	: 0.288608401448646 (4703957088885539 / 1.629875313842013e+16)
FP_OP	: 1.000030542390869 (134430431837848 / 134426326136452)

**Figure 3.2.5: Ratios derived from POP raw data for performance analysis of 4800 PE runs in Q2 and Q4.**

As described in the previous section, the primary performance and scalability bottleneck in the chosen benchmark configuration was the I/O implementation. Therefore, the focus of the GPRA-PMM effort was a new parallel I/O implementation that could use the I/O subsystem more efficiently. The strong scaling argument uses the results from Tables 3.2.5 and 3.2.2. Figure 3.2.6 shows the results of the strong scaling comparison between the Q4 enhanced code executed on 9600 PEs versus the Q2 POP code on 4800 PEs.

PES	: 2
TIME	: 0.3031007593855 (290.322787 / 957.842493)

INS	:	0.539572181993355	(8794353794668399 / 1.629875313842013e+16)
FP_OP	:	1.078558423469556	(144986646390536 / 134426326136452)

**Figure 3.2.6: Ratios derived from POP raw data for performance analysis of Q2 4800 PE and Q4 9600 PE runs.**

### 3.2.10 Summary and Conclusions

For upcoming high-resolution production simulations of the climate system at high resolution, we needed to improve the throughput of the POP ocean component. The primary bottleneck for this model was the parallel I/O implementation that had limited parallelism (across vertical levels only) for 3D fields, and no parallelism for 2D fields. For climate statistics and climate extremes we anticipated very frequent output of many diagnostic fields and the current implementation was not only limiting performance, but also effectively preventing such frequent output. We implemented a far more efficient I/O scheme that was developed by K. Roche, which significantly improved our model throughput for this high I/O configuration. We met the metric goal of a factor of two increase in model throughput for the POP ocean model. To wit, on doubling the number of PEs on the Q2 baseline problem, the run time was 3.299232908645226 faster than the Q2 run time.

### 3.3 LS3DF

#### 3.3.1 Introduction

LS3DF [2, 3] is a code for *ab initio* density functional theory (DFT) calculations that scales linearly. It is designed to study nanosystems containing a few thousand to hundreds of thousands of atoms. This relatively new code, developed by Dr. Lin-Wang Wang's group at Lawrence Berkeley National Laboratory, is based on a new divide-and-conquer charge density patching algorithm that cancels out the artificial boundary effects due to subdivision. As a result of this patching scheme, the results of LS3DF computations can be as accurate as conventional direct DFT calculation methods, while running thousands of times faster on systems of tens of thousands of atoms [14]. LS3DF won a special Gordon Bell prize award in 2008 for algorithm developments [14]. It has been run successfully on the full Jaguar Cray XT5 machine at NCCS with 150,000 processors in November 2008, attaining 33% of the machine's theoretical peak speed. It has also been run successfully on the full Intrepid IBM BlueGene/P machine at ALCF with 164,000 processors at 40% of the theoretical peak speed. It has been used in two DOE INCITE projects to study the electronic structure of nanosystems, and to simulate the critical components of nano solar cells. In typical production LS3DF runs, the number of processors used ranges from 8000 processors to 50,000 processors. The LS3DF code has been ported to many platforms, including Cray XT5, IBM BlueGene/P and IBM SP machines. The kernel of LS3DF is based on the PEtot planewave DFT code [15] developed in Dr. Wang's group, which by itself can be used to calculate systems with up to a thousand atoms. The current version of the LS3DF code performs only self-consistent DFT calculations without moving the atoms. An upgraded version of the code with the atomic relaxation capability is under development.

#### 3.3.2 Background and Motivation

As computational power has grown to petascale, the optimal algorithms for doing simulations of a given problem, as well as the scale of the physical problems that can be simulated, have both changed. Nanosystems often contain hundreds of thousands of atoms, and are beyond the scope of traditional *ab initio* DFT computational methods. However, many problems must be simulated from first principles, e.g., the total energy, the dipole moment, the band alignment, and the atomic positions. The main reason these large systems are out of reach when using traditional DFT methods is the cubic scaling of DFT methods with the system size [16]. This has motivated the development of linear scaling methods. Linear scaling methods are possible, especially for semiconductor systems, due to the locality of the quantum mechanical properties of the material (the nearsightedness principle [17]). On the other hand, the long-range portion of the interaction (the electrostatic interaction) is classical, and can be calculated quickly based on the charge density.

In the past fifteen years, many linear scaling methods have been developed. There are three main approaches: (1) truncated localized orbital methods [18]; (2) truncated density matrix methods [19]; and (3) divide-and-conquer methods [20]. The localized orbital methods sometimes suffer from an energy local minimum problem, although some schemes have been proposed to overcome this [21]. The truncated density matrix method often uses a local basis set, and is popular in quantum chemistry calculations.

The LS3DF method belongs in the divide-and-conquer category. The advantage of the divide-and-conquer method is its straightforward implementation without worrying about numerical instability, and its natural suitability to large-scale parallelization. A disadvantage is the relatively large prefactor in the linear scaling coefficient due to the multiple calculations of a same spatial point. Nevertheless, numerical tests show that the crossover between the linear scaling divide-and-conquer method and the  $O(N^3)$  scaling traditional DFT method is at about 500 atoms, which is essentially the same crossover system size as other linear scaling methods.

In the divide-and-conquer approach, a large system is divided into many fragments, and each fragment can be calculated by a small group of computer processors independently. The results for

different fragments are gathered together to generate the results of the original system. The independent execution of different fragments makes the computation amenable to large-scale parallelization. Indeed, the LS3DF can scale to hundreds of thousands of processors. Similarly, Professors Vashishta and Nakano's group at the University of Southern California has implemented another version of the divide-and-conquer scheme that can also scale to hundreds of thousands of processors [22]. In the early divide-and-conquer scheme proposed by Professor W.T. Yang [20], only a central part of a fragment is used in patching the system to generate the result of the original global system. Spatial partition functions are used in that task. Unfortunately, the use of partition functions has caused some technical problems, including the non-unique treatment of the kinetic energy, the metallic-like treatment for fragments state occupation, and the nonexistence of a variational formalism for the total energy. All these issues make the result not very accurate compared with the direct DFT results. LS3DF has overcome these problems with a spatial divide and patching scheme. In this scheme, fragments of different sizes are used, both positive and negative, and the fragments are patched together in such a way that the artificial boundary effects caused by the subdivision will be cancelled out between different fragments. Because no partition function is used, a variational formula exists for the total energy and the fragment Kohn-Sham equation. The LS3DF method is also similar to the fragment molecular orbital (FMO) method [23]. However, FMO is designed for organic chain molecules, and cuts the system only along the chains and uses fragment dimmers to correct the interface energies. It thus cannot be used for systems like inorganic nanocrystals. In addition, the FMO cancels only the division boundary effects between two fragments (thus it does not cancel out the effects of the edges and corners of a 3D fragment), while LS3DF cancels out all artificial boundary effects. In the LS3DF method, the division of the system into fragments is done based on atomic spatial positions. It is thus ideal for 3D systems like the nanocrystal, but also suitable for organic systems.

### 3.3.3 Capability Overview

LS3DF performs DFT calculations on systems with thousands or hundreds of thousands of atoms. With hundreds of thousands of computer processors, it can finish a self-consistent calculation within an hour. The kernel of the LS3DF for the fragment calculations is based on the standard planewave pseudopotential codes. It uses the norm conserving pseudopotentials. The total energy error compared with the direct DFT calculation is about 1 meV per atom. The charge density difference is less than 0.1% [2, 3]. For a converged calculation, it provides the total charge density and total potential, but no global eigenfunctions and eigenstates. To calculate eigenfunctions and eigenstates, one can take the potential from the LS3DF method and perform a folded spectrum method (FSM) [24] using the PEsCan code [25]. The Poisson equation is solved based on the global charge density. It can be solved using periodic boundary conditions or open boundary conditions. The open boundary condition is useful for calculations involving isolated systems like a quantum dot. Self-consistency is achieved iteratively via potential mixing schemes. Pulay and Kerker potential mixings are used. The LS3DF code divides the global system into many fragments. The fragment division is based on a real space grid, which is provided by the user. The grid cell corresponds to the smallest fragment size: the larger the fragment size, the more accurate the results. For good accuracy, the smallest fragment in a typical computation corresponds to roughly eight atom cells. LS3DF can be used to calculate the atomic force based on Hellmann Feynman theory. The LS3DF-calculated forces differ from the results of the direct DFT method by only  $10^{-5}$  a.u.

Currently, the LS3DF code cannot be used to relax the atomic position. A newer version of the code allowing atomic relaxation is under development. The current version of the LS3DF code allows the calculation of the semiconductor or insulator systems where a band gap exists between the occupied states and unoccupied states. Tests are underway to show whether the LS3DF method can be used for metallic systems, in which the electron occupation for each fragment is determined by a global Fermi level.

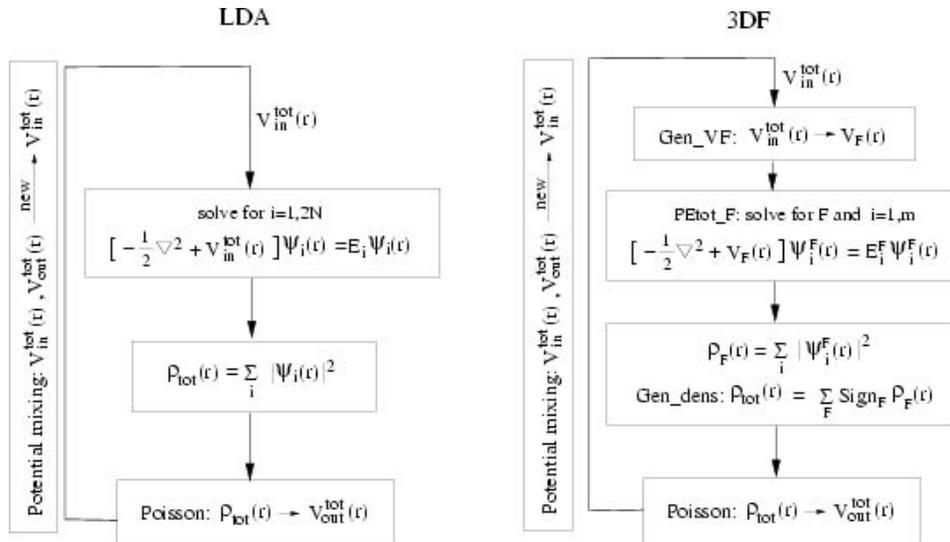


Figure 3.3.1: Program flow chart for conventional LDA method (left) and LS3DF (right).

The LS3DF code is based on the planewave DFT PEtot code [15]. Flow charts comparing the original LDA code and the LS3DF code are shown in Figure 3.3.1. The LS3DF code consists of several components: PEtot\_F, which divides the number of processors into processor groups, and the calculation of the fragment wavefunctions  $\psi_{F,i}$  of each group for a given fragment potential  $V_F(r)$ . It also calculates the fragment charge density  $\rho_F(r)$  from the wavefunction  $\psi_{F,i}$ ; Gen\_dens patches together the fragment charge densities  $\rho_F(r)$  to generate the total charge density  $\rho_{tot}(r)$  of the whole system. The Poisson generates the LDA total potential  $V_{tot}(r)$  from the total charge density  $\rho_{tot}(r)$ . This step solves the Poisson equation for the whole system using a global FFT. It also uses the Pulay scheme to mix the resulting LDA potential that is used in the next iteration. Finally, Gen\_VF generates the fragment potential  $V_F(r)$  from the input total potential  $V_{tot}(r)$ .

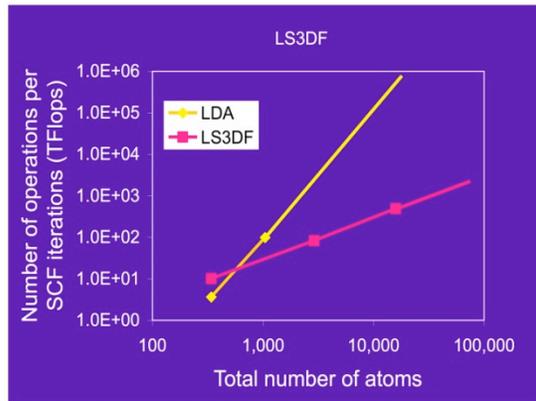
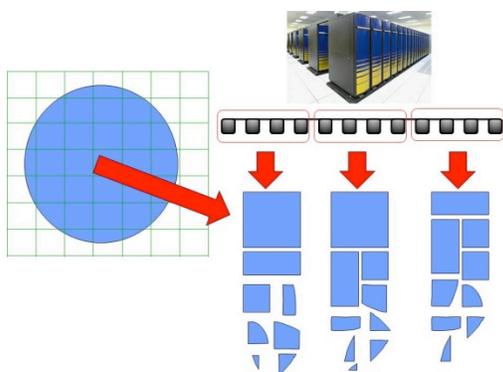
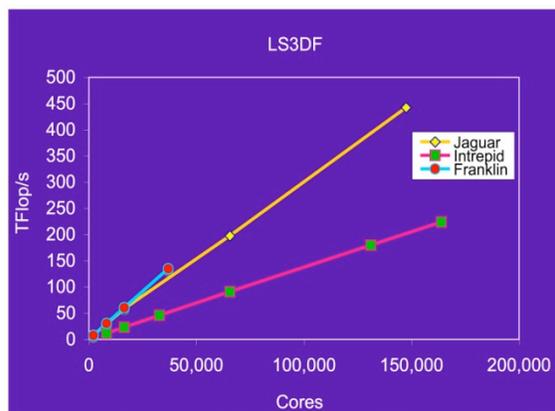


Figure 3.3.2: Scalability of LS3DF vs. LDA.



**Figure 3.3.3: Parallelization of problem domain.**

The biggest advantage of the LS3DF method is its linear scaling with respect to both the size of the physical system and the number of computer processors. The linear scaling of the computational cost for LS3DF method to the size of the system is illustrated in Figure 3.3.2; LS3DF calculations become cheaper than direct DFT calculations when the size of the system exceeds about 500 atoms.



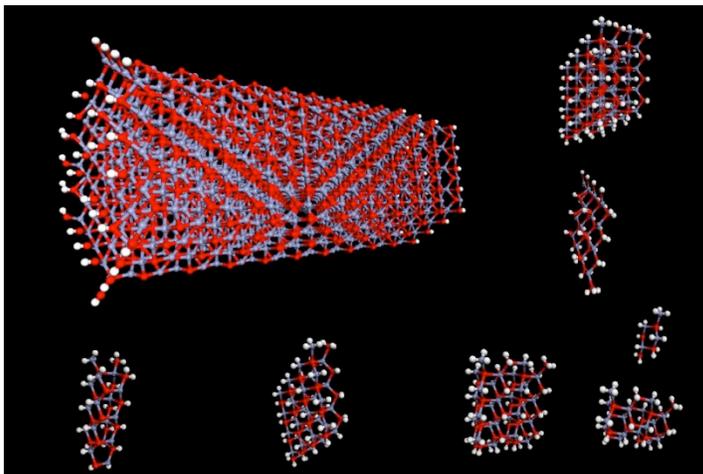
**Figure 3.3.4: Weak scaling of LS3DF on three leadership computing resources.**

A schematic of the LS3DF parallelization scheme is shown in Figure 3.3.3, where computer processors are divided into different groups, while each group calculates upon a few fragments. The assignment of the fragment to the processor groups are done statically at the beginning of the calculation based on the estimated computing time for each fragment to reach a load balance. Finally, a weak scaling study of the code on a bulk semiconductor alloy ZnTe:O system is shown in Figure 3.3.4. We see that it scales linearly to the full machine sizes available at the time of the test.

### 3.3.4 Science Driver for Metric Problem

Due to possible dipole moments of the nanocrystals, there could be large internal electric fields in such systems. The internal electric field can significantly change the electronic structure, the electron wave function localization, the exciton binding energy and dissociation, and the carrier dynamics. All these are critically important for solar cell performance [26]. Despite more than a decade of study [27-29], the internal electric field problem in a composite nanocrystal and its consequence on the electron wave functions are still poorly understood. Experimentally, this is due in part to the lack of adequate experimental probes that are able to make direct measurements of the electric field inside a nanocrystal. The situation is also complicated by the sensitivity of the internal electric field to the nano environment, including surface and interface dipoles, piezoelectric effects, bulk dipole moment, charged dopants, surface trapped charges, compensation charge, and surface ligands. For example, the piezoelectric and

dipole effects are more complicated in a nanostructure than in bulk superlattices due to the complicated geometries, and a single dopant or surface charge can completely change the outlook of the internal electric field. The band alignment problem, effects of surface passivation, and the change of dielectric screening have also made the internal electric field problem a challenge to model and simulate. Although the traditional continuous models for piezoelectricity have been used for large epitaxial embedded quantum dots, the use of such models for smaller colloidal nanosystems is not established. This is particularly true given the recently found importance of higher order piezoelectric effects [30]. All these make it necessary to use *ab initio* DFT methods to study the electric field effect microscopically at the atomic level. Unfortunately, the relevant nanosystems often contain a few thousand to tens of thousands of atoms, which are beyond the capability of conventional DFT methods. This has prevented the use of the modern *ab initio* methods for studying the electric field problem in nanosystems. However, the situation has changed with the development of LS3DF method: these problems are now computationally feasible. We have thus applied the LS3DF method to study this longstanding nanoscience problem.



**Figure 3.3.5: The benchmark ZnO nanorod and some of its fragments.**

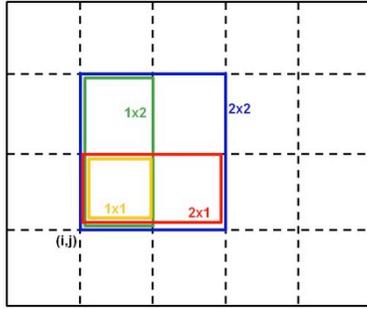
For our benchmark problem, we have chosen a ZnO nanorod (Figure 3.3.5). This is a nanorod modeled with realistic surface passivation. The ZnO rod is chosen due to its widespread use in many different applications, excellent experimental sample quality, and a clear surface passivation model. ZnO has been used in many nano solar cell designs, and is a widely used semiconductor material. Despite its importance, little is known about the total dipole moment and internal electric field of a ZnO nanorod. For bulk wurtzite structure, there is a small bulk dipole moment per unit cell ZnO. There have been many experimental and theoretical studies of the dipole moment of the ZnO polar surface, and the resulting surface passivation and compensation charges. However, there is no such study for nanosystems. Surface passivation often plays an important role in determining nanocrystal properties. Unfortunately, for many nanocrystals, we know very little about their surface passivations. For ZnO nanorod, the side surface is self-healed without organic ligands. The surface atomic relaxation pattern is well understood. The Zn and O atoms on the (10-10) and (1120) surfaces form a dimer structure with Zn moving inward and O moving outward. The resulting surface states are slightly above the bulk band gap. For the bottom (O-terminated) and top (Zn-terminated) dipole surfaces, we have used H and OH group to passivate them. Previous studies indicate that H and OH are very efficient passivation agents, and they are also ubiquitous. The quantum rod atomic structure is shown in Figure 3.3.5. There are 2776 atoms in the system, with 24220 valence electrons in the DFT calculations. Zn *d*-electron is included in the valence electrons. Our goal is to perform self-consistent calculations for this system to find the total charge density and potential of the system.

Our LS3DF computations find a large dipole moment and a very tilted potential profile. If the tilting is larger than the energy band gap, such potential can cause the electron to flow from one side to another. This is a charge self-compensation effect (much like in a bare ZnO polar surface). As a result, large tilting cannot happen, and it makes the study of the dipole moment effect complicated. One advantage of the LS3DF method is its ability to prevent such charge flow from one side to another despite a severely tilted potential. This is because the fragments are calculated independently, and a fragment does not see the whole quantum rod. As a result, the electric break down effect can be prevented. If a global Fermi energy is used in LS3DF, we can make it possible for the charge to flow from one side to another, but here, we deliberately prevent such charge flow in order to study the dipole moment and internal electric field effect.

### 3.3.5 The Model and Algorithm

LS3DF is based on the observation that the total energy of a system can be decomposed into two pieces: the quantum mechanical part (the wave function kinetic energy and the exchange correlation energy), and the classical electrostatic part. The electrostatic energy (Coulomb energy) is long range, while the quantum mechanical energy is local in nature (nearsighted). While the long-range Coulomb interaction can be solved efficiently by the Poisson equation even for million-atom systems, the quantum mechanical part presents a challenge. To overcome this challenge, we will take advantage of quantum-mechanical locality by using a spatial decomposition divide-and-conquer method. While there are previous methods [20, 22] based on this divide-and-conquer concept, they all rely on positive spatial partition functions to divide and patch the spaces. There are intrinsic difficulties inherent to these positive partition function techniques, especially for dividing the kinetic energies, and making a variational formalism. In contrast, the division-patching method in LS3DF avoids these problems, resulting in a much more accurate algorithm (with accuracy essentially the same as the original full-system LDA method).

The division of space (and hence the atoms in it) of the LS3DF method is shown in Figure 3.3.6. For simplicity, we have illustrated this division in a two-dimensional system (rather than three-dimensional), but the concept is the same in any dimensionality. If the whole system is placed inside a box, one can first divide the box into an  $m_1 \times m_2$  two-dimensional grid. Now, at each grid point  $(i, j)$ , one can define four fragments (pieces) enumerated by their left lower corners  $(i, j)$ . In terms of the unit grid, the sizes of these four fragments are:  $2 \times 2$ ,  $2 \times 1$ ,  $1 \times 2$ , and  $1 \times 1$ . Let us denote the charge density of these fragments as  $\rho_{2 \times 2}(i, j)$ ,  $\rho_{2 \times 1}(i, j)$ ,  $\rho_{1 \times 2}(i, j)$ , and  $\rho_{1 \times 1}(i, j)$ . If all these fragment charges have been calculated (e.g., by different computer processors), then the total charge of the system can be summed as:  $\rho_{tot} = \sum_{i,j} [\rho_{2 \times 2}(i, j) - \rho_{2 \times 1}(i, j) - \rho_{1 \times 2}(i, j) + \rho_{1 \times 1}(i, j)]$ . As a result, all the edges and corners of the fragments in the summation will cancel out. In Figure 3.3.6, if we compute the area of the fragments covering each grid cell  $(i, j)$ , we obtain  $2 \times 2 - 2 \times 1 - 1 \times 2 + 1 = 1$ , i.e., exactly the area needed to fill the entire box. On the other hand, adding up the number of fragment edges along the borders of the box (fragment boundary line), we have  $8 - 6 - 6 + 4 = 0$ , and summing the number of corners, we have  $4 - 4 - 4 + 4 = 0$ . Thus, the extraneous areas, edges, and corners all cancel out. This scheme also works in the three-dimensional system, where each grid cell  $(i, j, k)$  contains eight fragments with sizes (and the +/- signs in the summation)  $2 \times 2 \times 2(+)$ ,  $2 \times 2 \times 1(-)$ ,  $2 \times 1 \times 2(-)$ ,  $1 \times 2 \times 2(-)$ ,  $2 \times 1 \times 1(+)$ ,  $1 \times 2 \times 1(+)$ ,  $1 \times 1 \times 2(+)$ , and  $1 \times 1 \times 1(-)$  respectively.



**Figure 3.3.6: Schematic of LS3DF subdivision of domain space.**

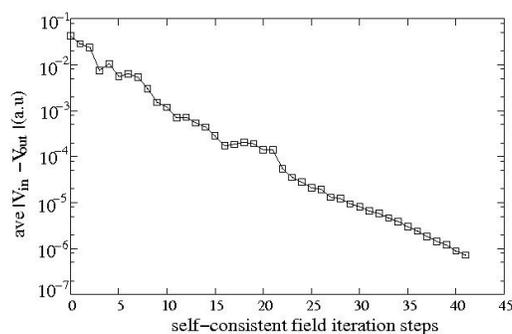
Note that only the fragment charge densities are patched together, not the fragment wave functions. The global charge is needed to solve the Poisson equation of the whole system for the long-range Coulomb interactions. This global charge will also be used to calculate the exchange-correlation energy under the density functional formula (e.g., the local density approximation, LDA). On the other hand, the fragment wave functions  $\{\psi_F, i\}$  are used only to generate the fragment charge density and the fragment kinetic energy (which will also be summed up using the same formula with positive and negative signs for fragments of different sizes). The remaining energy terms in the LDA energy expression will be calculated in terms of the total charge density  $\rho_{tot}(r)$ . Since calculating  $\{\psi_F, i\}$  is the most time-consuming part, and the total computational cost is proportional to the number of fragments, the method scales linearly with the size of the entire system. It also makes the whole calculation easily parallelizable. Typically the set of processors is divided into many subsets, and each subset is assigned a number of fragments, over which computations are performed in sequence.

The accuracy of the LS3DF method depends on the size of the fragment: the larger the size, the better the accuracy compared with the original direct DFT calculation. Thus, to achieve a given accuracy, we use a fixed fragment size. A typical  $1 \times 1 \times 1$  fragment contains 8 atoms; thus the largest  $2 \times 2 \times 2$  fragment contains 64 atoms. Some fragments of the ZnO quantum rod are shown in Figure 3.3.5. Using such fragments for the computation, the resulting LS3DF total energy differs from the direct whole-system DFT calculations by only a few meV per atom, which is smaller than many other numerical errors in a typical DFT calculation, e.g., the plane wave cutoff error, and the pseudopotential error. Thus, for most practical purposes, the LS3DF method can be considered numerically equivalent to the direct DFT method. Due to the linear scaling of the LS3DF method, however, it can be thousands of times faster.

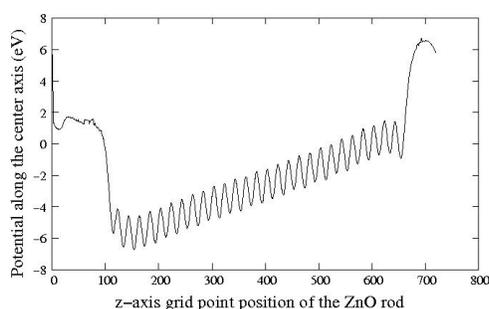
In a variational formalism like LS3DF, the total energy minimization is achieved by self-consistent iterations, where a total system input potential is used to generate an output potential through the LS3DF formalism. When the input and output potential become the same, self-consistency is reached.

### 3.3.6 Q2 Baseline Problem Results

The LS3DF calculation of the 2776 atom, 24220 valence electron ZnO nanorod shown in Figure 3.3.5 converges after forty self-consistent field iterations. This is indicated by the exponential decay of the input and output potential difference as the iteration progresses, as shown in Figure 3.3.7. A 50 Ryd plane wave kinetic energy cutoff is used to describe the electron wave functions, and an open boundary condition is used to solve the global Poisson equation. This assures that there is no dipole-dipole interaction between image quantum rods in a periodic boundary condition. The supercell is divided into an  $m_1 \times m_2 \times m_3 = 18 \times 6 \times 6$  grid, as depicted in Figure 3.3.6. In real space, the supercell is discretized on a numerical grid of dimension  $720 \times 300 \times 300$ . This is the FFT grid for the global Poisson solver.



**Figure 3.3.7: Convergence of the LS3DF method for ZnO nanorod.**



**Figure 3.3.8: Self-consistent potential along the center axis of ZnO nanorod.**

A large dipole moment and internal potential is found, as shown in Figure 3.3.8. Such internal electric fields will strongly localize the electron at one side and a hole at the other side, thus dramatically altering the carrier dynamics in the system. Further analysis and comparative studies with other quantum dots with different side surface terminations reveal that the dipole moment comes from the Zn-O dimerization at the quantum rod side surface. These results suggest ways to manipulate the internal electric field by changing the side surface passivation, e.g., via application of different organic ligands. Note that the tilting of the internal potential from one side of the rod to the other is about 6 Volts, which is larger than the ZnO band gap (3.3 eV). If such a large tilting occurs in a physical system, the occupied valence electron at one side will flow to the conduction band state at the other side. Thus, there will be a self-compensation effect. Under the current mode of calculation in our LS3DF method, however, this large tilting is possible because we occupy each local fragment with a fixed number of electrons. This prevents electrons from flowing from one side to another while still allowing the dipole moment to exist. In the LS3DF calculation, we can also use a global Fermi energy to determine the electron occupation of individual fragments. If that is done, there will be charge flow from one side to the other, hence self-compensation will occur. We observe charge compensation in the direct DFT calculation (which we performed upon smaller systems), and there is no algorithm to prevent charge compensation from occurring. The ability to prevent charge compensation in the LS3DF method provides a means to study the total dipole moment effect without the additional complication of the charge flow, which depends on other factors like the surface electronic states. A full physical picture can be obtained by comparing the calculated results with and without the charge compensation.

LS3DF calculations on problems such as the ZnO nanorod consist of two stages. In the first stage, each fragment is calculated self-consistently by using a small box periodic boundary condition for each fragment. Then fragment charge densities and potentials for all the fragments are obtained. These initial fragment charge densities are used to generate the initial global charge density, and the initial global potential. For a given fragment, the global potential in the real-space domain of that fragment is

abstracted from the initial fragment potential to yield the fragment surface passivation potential [3]. This finishes the first stage of the LS3DF calculation. We performed twenty local self-consistent iterations for each fragment to yield the initial fragment charge density. Note that, at this stage, the self-consistent iteration for each fragment is performed independently without communications between fragments. An artificial periodic boundary condition is used over a small box on each fragment.

In the second stage, the flow charge is described in Figure 3.3.1, and the fragment wave functions are iterated over the fragment potentials, and the updated fragment charge densities are patched to generate the global charge density. This global charge density is used to solve the global Poisson equation to yield the global potential, which is used as the input (after potential mixing) for the next iteration. Here, the self-consistency is done over the global charge density and the potential. The forty iterations shown in Figure 3.3.7 are these global self-consistent iterations.

Two variables characterize the computation:  $N_g$ , the number of groups, and  $N_c$ , the number of cores within each group. The total number of cores equals  $N_g N_c$ . In our benchmark calculations, we varied both parameters, one at a time. In Table 3.3.1, the total compute times (from beginning to the end) required for the 20 initial iterations plus the 40 global self-consistent field (SCF) iterations are shown.

$N_{tot}$ (core)	10,800	21,600	43,200	8,640	17,280
$N_g$ (group)	108	108	108	108	216
$N_c$ (core/group)	100	200	400	80	80
Total time (hrs)	5.36	4.21	3.87	5.92	4.05
Tflops/sec	19.91	26.35	20.80	17.84	26.08
% peak	17.7%	11.7%	4.6%	19.2%	14.5%

**Table 3.3.1: Benchmarking results for varying process counts and configurations.**

From Table 3.3.1, one can see that the parallel scaling is far from perfect, and much worse than that shown in Figure 3.3.4. The main reason for this is that the scaling shown in Figure 3.3.4 is weak scaling, in which the system size increases with the number of groups, while here we have performed a strong scaling study near the point at which strong scaling for both  $N_g$  and  $N_c$  has reached its limit. Furthermore, a spatially homogeneous semiconductor alloy (ZnTe:O) is used for the test shown in Figure 3.3.4, which makes load balancing much easier and allows the use of some special fine-tuned algorithms for Gen\_VF and Gen\_dens. In the current benchmark problem, the system is inhomogeneous, consisting of the ZnO rod and vacuum. As a result, there is a large variation in fragment sizes. When  $N_g$  reaches 100 or 200, each group will have only a few fragments, with one or two large  $2 \times 2 \times 2$  fragments. This makes load balancing difficult. The resulting load balance problem shows up when  $N_g$  goes from 108 to 216 while  $N_c$  is fixed at 80. For a perfectly load balanced case, the compute time should be halved; in reality, it has only reduced to 68%. From Table 3.3.1, it is also apparent that the  $N_c$  has also reached its scaling limit. As  $N_c$  increases from 100 to 200 while  $N_g$  is kept constant, the reduction in compute time is only 20%, not the ideal 50%. When  $N_c$  is increased further, the compute time actually rises. Unfortunately, the program cannot be run with  $N_c$  much smaller than 80, due to memory constraints. In the current system, the  $d$ -electron of the Zn atom is kept in the valence band. As a result, the largest fragment has about a thousand electrons. Storing all the wave functions for all the fragments in a given processor group requires substantial memory. The overall floating point performance is less than 20% of the theoretical limit, indicating room for improvement.

Table 3.3.2 shows a breakdown of the compute times for different subroutines, and their floating-point performance compared to the theoretical limit. The durations indicate the relative importance of each subroutine, while the peak percentage (efficiency) indicates the performance of each piece. For the SCF iterations, the value for a single, representative iteration has been reproduced (in this case, there are 40 SCF iterations). We observed only minor fluctuations (a few percent) from one iteration to another. There are four subroutines in each SCF iteration, as shown in Figure 3.3.1. The most time-consuming part

is PEtot\_F, where the fragment wave functions are calculated. This part has an efficiency of about 20%. Since it dominates the calculation, improving the performance of this part is vital. For the three remaining subroutines, the low efficiency is somewhat an artifact of their limited scalability. Only a fraction of the cores perform these calculations, while the remaining cores idle. In all the benchmark runs, only  $m_1 \times m_2 \times m_3 = 648$  cores are used to carry out Gen\_VF and Gen\_dens subroutines, while only 720 cores are used to run Poisson. This configuration has contributed to the extremely low efficiencies of these subroutines. Gen\_VF generates the fragment potential from the global potential and the passivation potential, while Gen\_dens patches up the fragment charge density to yield the global charge density. Both of these subroutines perform minimal floating-point operations; most of the time is spent on data communications. This is evident by comparing the total number of floating point operations to the total number of machine instructions measured in these subroutines. In Gen\_VF, the number of floating-point operations is only about 0.0013% of the total number of machine instructions; in Gen\_dens, this number is 0.25%. On the other hand, for PEtot\_F, the percentage of the floating-point operation instruction relative to the total machine instructions is about 60%. For the Poisson solver, we have used an open boundary condition. This is realized by doubling the size of the supercell in each direction, then performing the FFT on the larger box. The efficiency of this subroutine is also very poor, indicating room for improvement. Time is probably taken up by subroutines other than the FFT, for example in the box doubling routines. Right now, PEtot\_F takes about 90% of the time for each SCF iteration, while the other three inefficient subroutines take up the remaining ten percent. Only after we substantially improve the performance of the PEtot\_F (e.g., by strong scaling) does it become important to improve the other three subroutines. Even then, we would strive only to reduce the amount of time spent in these subroutines.

N <sub>tot</sub> (core)		10,800	21,600	43,200	8,640	17,280
N <sub>g</sub> (group)		108	108	108	108	216
N <sub>c</sub> (cores/group)		100	200	400	80	80
Initial PEtot_F (20 iter)	Time(sec)	6714	4653	4860	6494	3872
	% peak	16.2%	10.4%	5.0%	17.6%	14.8%
Initial Gen_dvr	Time(sec)	215	438	305	229	247
	% peak	0.064%	0.016%	0.011%	0.075%	0.035%
Gen_VF (1 iter)	Time(sec)	6.33	5.94	5.55	4.81	5.26
	% peak	0.0004%	0.0003%	0.0001%	0.0009%	0.0004%
PEtot_F (1 iter)	Time(sec)	299	220	177	328	223
	% peak	20.8%	14.7%	9.5%	23.5%	17.3%
Gen_dens (1 iter)	Time(sec)	8.67	10.82	8.30	9.11	8.75
	% peak	0.07%	0.06%	0.056%	0.07%	0.07%
Poisson (1 iter)	Time(sec)	23.3	23.8	18.0	23.3	21.8
	% peak	0.0038%	0.0018%	0.0012%	0.0047%	0.0025%

**Table 3.3.2: Timings of subroutines within code.**

### 3.3.7 Computational Performance Gains

The fastest calculation reported in Q2 to converge the ZnO quantum rod is 3.87 hours using 43,200 processors. In Q4, we have calculated the exact same ZnO quantum rod system. We have made three improvements to the code: (1) introduced a wave function index parallelization within the PEtot\_F subroutine; (2) implemented a new algorithm: the direct inversion of the iteration space (DIIS) method, in addition to the conjugated gradient (CG) method, in the PEtot\_F subroutine to converge the wave functions; (3) developed a better formula to estimate the computational time of each fragment, which allows a better static assignment of fragments into fragment groups. This improves the load balance between different fragment groups. After these improvements, with a better strong scaling, by using more 86,400 processors, we have reduced the computational time (while having the same physical results) to 1.5 hours. Compared to the best Q2 results of 3.87 hours, this represents a speedup of 2.6 times. This is achieved by using more processor groups (432 groups instead of 108 groups, which become useful due to

our better load balance), and two band-index groups ( $N_b=2$ ) under the new wave function index parallelization scheme. The largest processor count, 86,420 processors, constitutes more than 25% of the whole Jaguarpf machine.

### 3.3.8 Q4 Metric Problem Results

The Q4 benchmark results are listed in Table 3.3.3 and Table 3.3.4. They should be compared with Table 3.3.1 and Table 3.3.2. Like the Q2 tests, the Q4 calculations consist of 20 initial steps of self-consistent calculations for each independent fragment, followed by 40 self-consistent iterations for the whole system. Note that for the initial 20 iterations, we always use the CG method in PETot\_F. However, for the following 40 iterations, we have tested both the DIIS method and the CG method. We have also tested different band groups  $N_b$  for the band index parallelization. Note that there could be significant fluctuations in the benchmarking results due to the general status of the machine, the processor assignment (mapping), and the way the rest of the machine is used. We have thus taken the liberty of reporting the best results we have obtained from different runs, or from the fluctuations between different iterations steps. These best results represent the ideal performance when the machine is not adversely impacted by other jobs of other users. In reality, the performance results might fluctuate by about 10% from run to run.

$N_{tot}$ (cores)	$N_g$ (groups)	$N_c$ (cores/band)	$N_b$ (band groups)	Method	Total time (hours)	Tflops/sec	% peak
8,640	108	80	1	DIIS	5.59	17.0	18.9%
8,640	108	80	1	CG	5.76	18.3	20.3%
17,280	108	80	2	CG	4.56	23.2	12.9%
17,280	216	80	1	CG	3.33	31.7	17.6%
34,560	216	80	2	CG	2.37	44.6	12.4%
69,120	216	80	4	CG	2.23	47.3	6.6%
69,120	432	80	2	CG	1.58	66.8	9.3%
86,400	432	100	2	CG	1.48	71.4	7.9%

**Table 3.3.3: Total LS3DF computation times (for 20 initial iterations plus 40 SCF iterations), flops, and percentage of performance to the theoretical limit.  $N_g$  is the number of fragment groups,  $N_c$  is the number of cores in each band group, and  $N_b$  is the number of band groups. Within each fragment group, there are  $N_c \times N_b$  processors, and  $N_{tot} = N_g \times N_c \times N_b$ .**

$N_{tot}$ (cores)	$N_g$ (groups)	$N_c$ (cores/band)	$N_b$ (band groups)	Method	PETot_F time (1 iter, sec)	Tflops/sec	% peak
8,640	108	80	1	DIIS	299	19.8	22.0%
8,640	108	80	1	CG	314	23.2	25.7%
17,280	108	80	2	CG	221	33.0	18.3%
17,280	216	80	1	CG	174	42.0	23.3%
34,560	216	80	2	CG	112	65.2	18.1%
69,120	216	80	4	CG	96	76.0	10.5%
69,120	432	80	2	CG	68	107.4	14.9%
86,400	432	100	2	CG	61	119.6	13.3%

**Table 3.3.4: The computational time for subroutine PETot\_F for each self-consistent field (SCF) iteration (containing 4 CG or 4 DIIS steps). The times for the other three subroutines (Gen\_VF, Gen\_dens, and Poisson) are unchanged from the Q2 results, and remain unchanged for the different methods in PETot\_F and different combinations of  $N_g$ ,  $N_c$ , and  $N_b$ . After averaging out the fluctuations, the times for Gen\_VF, Gen\_dens, and Poisson are 6, 9, and 22 seconds, respectively. Thus in total they account for 37 seconds for each SCF iteration.**

### 3.3.9 Interpretation of Results

We first note that the DIIS is only slightly faster than the CG method in our test. Furthermore, for the same number of iterations, DIIS has a lower floating-point operation count than CG. As a result, in terms of flops and percentages of the theoretical limit, the DIIS is slightly worse than the CG result. However, because DIIS is a different algorithm, and its convergence properties are different from CG, it is difficult to compare the CG and DIIS in a fair fashion. We also found that in our current system, sometimes DIIS convergence is not stable. Thus, we have not tested the DIIS extensively and compared its result with the CG result. But we do know that, for large systems, DIIS can be extremely useful (e.g., for standalone PETot run). Thus, the usefulness of DIIS might be dependent upon the systems we are studying, and the size of the fragments. DIIS performs particularly well on larger fragment sizes.

While the band index parallelization ( $N_b$ ) does help, the strong scaling is not perfect and shows a saturation when  $N_b=4$ . For the band parallelization, when a wave function orthogonalization or subspace diagonalization is needed, the wave functions from different subband groups must carry out an inter-group dot product. That increases the communication time, hence reduces the parallel efficiency. One advantage of band index parallelization is to increase the memory available since more processors are used within one fragment group. Note that, if all the processors within each fragment group are assigned to  $N_c$  (the G-space parallelization), the FFT might not have a high parallel efficiency. Furthermore, if real space nonlocal projections are used, a large  $N_c$  will fragmentize the real space projector data, and slow down that part of the calculation.

In our Q4 calculation, an additional major improvement we made in the code resulted in better load balance between different fragment groups. In order to do that, one needs a better model to predict the computational time for each fragment. We have improved our original model, and obtained a better fit of the formula. As a result, we are able to scale to larger number of fragment groups (e.g., 432 groups in our test).

Note that, from the 17,280 processors used in Q2 (the best performance case in Q2) to 86,420 processors in this Q4 result, the processor number has increased by a factor of 5, while the time has only improved by a factor of 2.7. Thus the parallel efficient is only about 54%. Part of the reason is that the PETot\_F time (the part for quantum mechanical calculation) has been reduced close to the times of the other parts (the classical calculation part). In the case of 86,420 processors, the PETot\_F time for each SCF iteration is 62 seconds, while the sum of the other three parts (Gen\_VF, Gen\_dens, Poisson) is 37 seconds. Thus, in order to have an even better strong scaling performance, we have to improve these three classical parts. For the Q2 to Q4 performance improvement activity, we have not touched these three parts. Future improvement is likely, in particular to rewrite the algorithms to use more processors. Currently, a fixed number of processors (a very small percentage of the total number of processors) are used for these three parts regardless of the total number of processors. This points out the possible future improvement of the LS3DF code.

### 3.3.10 Summary and Conclusions

We have used the LS3DF code to calculate a 2776-atom ZnO nanorod with 24,220 valence electrons. The calculation result shows a strong dipole moment, which has significantly tilted the internal potential of the rod. Further investigation revealed that this dipole moment comes from the side surface of the ZnO wurtzite nanostructure.

In the Q2 benchmark, the original production LS3DF code was used. At the largest processor count (43,200), it took 3.87 hours to converge. Twenty initial self-consistent steps are used for the independent fragments, and 40 self-consistent steps are used for the whole system. From Q2 to Q4, we have made three major developments: (1) a band-index parallelization with the PETot\_F subroutine; (2) the DIIS algorithm in PETot\_F for wave function optimization; (3) an improved fragment-to-fragment group assignment algorithm to improve load balance between fragment groups.

Our Q2 to Q4 tests are based on improving strong scaling. As has already been demonstrated [14], the LS3DF code can exhibit almost perfect weak scaling to hundreds of thousands of processors, and tens of thousands of atoms. Using the improvements described in the above paragraph (in particular (1) and (3)), we have demonstrated strong scaling to a significant fraction of Jaguar: we have successfully reduced the computational time from the original 3.87 hours on 43,200 processors to 1.48 hours on 86,400 processors, a factor of 2.6 improvement in timing from doubling the number of processors. *LS3DF has met the metric because the ratio of the Q2 to Q4 timings exceeds two.*

## 3.4 DENOVO

### 3.4.1 Introduction

Denovo [4, 5] is a linear radiation transport application for nuclear and radiological sciences. This application area includes, but is not limited to nuclear reactor analysis, fusion, radiation shielding and protection, nuclear safeguards, radiation detection, and radiation therapy, diagnostics, and treatment planning.

The science and engineering driver for the current work is nuclear reactor analysis. Nuclear reactor analysis requires accurate characterization of the neutron distribution in the reactor in order to determine power, safety, and fuel and component performance. In a steady-state operational reactor, the neutron field is characterized by six independent variables (three in space, two in angle, and one in energy), and the mean flight times of low-energy neutrons are in the millimeter to centimeter range. Thus, high-resolution solutions of the transport equation require tremendous computational resources. Traditionally, computational resources have not been sufficient to attack this problem at full resolution, so multi-level approximation schemes have been employed. However, the Jaguar XT5 leadership system enables us to attack this problem from a full transport approach. To achieve this goal, Denovo has synthesized the last decade's worth of computational transport work into a modern, production-quality code that can begin to attack full-core reactor analysis from an *ab initio* approach.

### 3.4.2 Background and Motivation

The transport of neutrons in matter is governed by the linear Boltzmann transport equation,

$$\frac{1}{v} \frac{\partial \psi(\mathbf{x}, \Omega, E, t)}{\partial t} + \hat{\Omega} \cdot \nabla \psi(\mathbf{x}, \Omega, E, t) + \sigma(\mathbf{x}, E) \psi(\mathbf{x}, \Omega, E, t) = q_c(\mathbf{x}, \hat{\Omega}, E, t) +$$

$$\iint \sigma_s(\mathbf{s}, \hat{\Omega}' \cdot \hat{\Omega}, E' \rightarrow E) \psi(\mathbf{x}, \Omega', E', t) d\Omega' dE' + \frac{\chi(\mathbf{x}, E)}{4\pi} \iint v \sigma_f(\mathbf{x}, E') \psi(\mathbf{x}, \hat{\Omega}', E', t) d\Omega' dE', \quad (1)$$

Here,  $\psi$  is the radiation intensity,  $\sigma$  is the total interaction cross-section,  $\sigma_s$  is the scattering cross-section,  $\sigma_f$  is the fission cross-section, and  $v$  is the number of neutrons emitted per fission event. The fission energy-spectrum is defined by  $\chi$ . In non-multiplying material,  $\sigma_f = 0$ ; however, when the fission is nonzero, Equation 1 has no steady-state solution. In this case, we must solve the eigenvalue form of Equation 1,

$$\hat{\Omega} \cdot \nabla \psi(\mathbf{x}, \Omega, E) + \sigma(\mathbf{x}, E) \psi(\mathbf{x}, \Omega, E) - \iint \sigma_s(\mathbf{s}, \hat{\Omega}' \cdot \hat{\Omega}, E' \rightarrow E) \psi(\mathbf{x}, \Omega', E') d\Omega' dE' = \frac{1}{k} \frac{\chi(\mathbf{x}, E)}{4\pi} \iint v \sigma_f(\mathbf{x}, E') \psi(\mathbf{x}, \hat{\Omega}', E', t) d\Omega' dE', \quad (2)$$

The value  $k$  is the effective- $k$  ( $k_{\text{eff}}$ ) of the system and is a measure of the neutron production per generation. In order to maintain a sustainable, stable reactor,  $k_{\text{eff}} = 1$ . When  $k_{\text{eff}} < 1$  the system is sub-critical and when  $k_{\text{eff}} > 1$  the system is super-critical. The goal of reactor design is to keep  $k_{\text{eff}}$  as close to unity over as wide a range of operating conditions as possible.

Most of reactor core design deals with the solution of Equation 2, although solving Equation 1 is required to perform shielding analysis and assess component performance. In general the full radiation field is not required for practical analysis; instead, our approach focuses upon the first two angular moments of the radiation field,

$$\phi(\mathbf{x}, E) = \int_{4\pi} \psi(\mathbf{x}, \Omega, E) d\Omega, \quad (3)$$

$$\mathbf{J}(\mathbf{x}, E) = \int_{4\pi} \hat{\Omega} \psi(\mathbf{x}, \Omega, E) d\Omega. \quad (4)$$

Equation 3 is the scalar intensity (or scalar flux) and Equation 4 is the radiation current. The scalar flux is used to calculate the spatial power distribution in the reactor core,

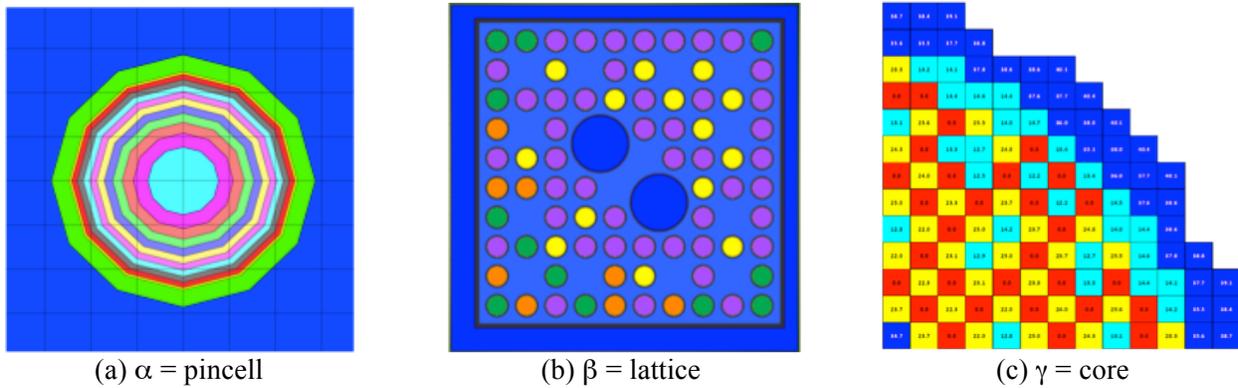
$$P \propto \iint \sigma_f(\mathbf{x}, E) \phi(\mathbf{x}, E) dV dE. \quad (5)$$

Finally, the *reactivity* is an important measure of reactor operation,

$$\rho = \frac{k_{\text{eff}} - 1}{k_{\text{eff}}}. \quad (6)$$

The reactivity is measured in dimensionless units of  $\Delta k / k$  or pcm ( $10^{-5} \Delta k / k$ ).

Most reactor analysis requires the solution of Equation 2. A survey of the literature shows that many transport approximations have been used throughout the years to calculate the  $k$ -eigenvalue and power distribution.



**Figure 3.4.1: Three levels of reactor geometries.**

Present reactor transport methods use an **inconsistent** three-level homogenization approach (each within distinct simulation codes) in modeling radiation transport in the core of a nuclear reactor. Figure 3.4. depicts the spatial domains of this multi-scale challenge:  $\alpha$  use of a fine-mesh in one spatial-dimension in an approximate small subset (pincell) of the reactor core with a first-principles representation of the energy spectrum;  $\beta$ , use of a coarser-mesh with a two-dimensional transport solution in a larger subset (lattice) of the core with grouped representation of the energy spectrum, provided by the previous step; and  $\gamma$ , use of a very coarse mesh in a three-dimensional diffusive transport of neutrons in the full homogenized core of the reactor with a very coarse representation of the energy spectrum provided by the previous step. The first two steps ( $\alpha, \beta$ ) require  $10^7$  to  $10^8$  degrees of freedom (DoF) with  $10^2$  to  $10^3$  independent calculations each on single processor machines. Recent work at ORNL has demonstrated that, with moderate computational resources, this can be reduced to an **inconsistent** two-step approach, where step (A) utilizes the energy fidelity of  $\alpha$  with the spatial domain of  $\beta$  and step (B) uses the energy fidelity of  $\beta$  and the spatial domain of  $\gamma$ . In this approach, each step would require  $10^{11}$  to  $10^{13}$  DoF per step with  $10^2$  independent high-order (A) calculations for every timestep. With access to petascale computational resources, researchers at ORNL are using Denovo to develop a first-of-a-kind capability that integrates the two-level approach within a single application framework and incorporates a mathematically **consistent** algorithm that is extensible to novel reactor concepts. However, the orders-of-magnitude increase in fidelity of each of the two steps requires substantially more computational resources than algorithms currently utilize today. Therefore, a novel approach to parallelization of the transport equation must be developed.

### 3.4.3 Capability Overview

Denovo solves the time-independent form of Equations 1 and 2 using the discrete ordinates ( $S_N$ ) method. It also features a Monte Carlo module that can be used to solve the multigroup equations on the  $S_N$  spatial grid with continuous angular treatment. The fundamental features of Denovo can be summarized as:

- 3-D, Cartesian orthogonal structured (nonuniform) grids
- Steady-state fixed-source, Equation 1, and eigenvalue, Equation 2, modes
- Spatial domain decomposition (DD) parallelism using the Koch-Baker-Alcouffe (KBA) sweep algorithm [31]
- Krylov and source-iteration within-group solvers
- Multigroup with optional thermal upscattering
- Diffusion Synthetic Acceleration (DSA) preconditioning of within-group solves and transport two-grid acceleration of Gauss-Seidel iteration for upscatter groups
- Rebalance eigenvalue acceleration
- Forward and adjoint modes
- Multiple spatial differencing schemes:
  - diamond difference (DD)
  - diamond difference with linear-zero flux fixup (DDLZ)
  - theta-weighted diamond difference (TWD)
  - step characteristics (slice balance) (SC)
  - linear-discontinuous finite element (LD)
  - trilinear-discontinuous finite element (TLD)
- Reflecting, vacuum, and surface source boundary conditions
- First collision and distributed external fixed sources
  - Domain-replicated parallel ray-tracing uncollided flux solver for point sources
  - Domain-replicated and decomposed Monte Carlo uncollided flux solver for point and distributed sources
- Multiple input front-ends, including Python bindings
- Support for multiple outputs, including HDF5 formatted SILO files that can be directly read by VisIt.

Denovo builds and runs on most flavors of Linux (i386 and x86\_64), MacOS (32 and 64-bit), and Windows (serial only). It has been used extensively on OLCF's Jaguar leadership systems (Cray XT4 and XT5). The code is actively compiled and tested using the following compiler families: GNU (gcc/g++/gfortran), Intel (icc/icpc/ifort), and PGI (pgcc/pgCC/pgf90).

Denovo is written primarily in C++, but it contains FORTRAN and C computational kernels as well as shared object bindings for Python (along with direct Python code). It uses a GNU-based configure/make system with an integrated testing environment. Current (March 2010) lines-of-code statistics are

C++ Executable Code	18,570
F95 Executable Code	931
<b>Total Executable Code</b>	<b>19,501</b>
C++ Unit Test Code	33,773
Python Unit Test Code	4,424
<b>Total Unit Test Code</b>	<b>38,197</b>

Denovo is developed using an Agile Software Process that emphasizes unit testing at the point of code construction. As seen from the code statistics, the majority of code in the Denovo package is test code that is distributed among 169 separately compilable unit-tests.

Denovo makes extensive use of external packages for many features. The base code requires the following third-party libraries:

- BLAS/LAPACK
- GNU GSL
- Trilinos

The following libraries provide optional capabilities and are not required, although the associated functionality will be unavailable without them:

- MPI (parallel communication)
- SILO (formatted output for VisIt)
- HDF5 (output)
- BRLCAD (CAD-based meshing capabilities)
- SuperLU/METIS (optional direct sparse solver for DSA, uses PCG from Trilinos by default)
- SPRNG (Monte Carlo module and Monte Carlo first-collision source)
- Python/SWIG (required to build Python bindings)

Finally, Denovo makes use of the following tools (although they are not necessarily required to build/run the code):

- Graphviz (optional for inline documentation)
- Doxygen (required for inline documentation)
- Emacs (optional for development environment)
- Python (required for testing framework)
- Sphinx (required to build PyKBA documentation)
- TeXinfo (required for build and developer documentation)
- LaTeX (required for methods documentation)

Denovo uses Subversion for configuration management, and it is hosted on the NSTD GForge server (<http://nstdsrv.ornl.gov>).

### 3.4.4 Science Driver for Metric Problem

In order to advance the state-of-the-art for next generation reactor designs and to improve performance of existing designs while preserving operational safety margins, transport calculations with higher geometric fidelity, solution accuracy, and physical model realism are required. In particular, there is a strong need to move away from the semi-empirical, inconsistent multi-level approaches described in Section 3.4.2 because the experimental facilities required to validate that process are no longer in existence. The ultimate goal is to perform fully resolved, three-dimensional transport simulations of the entire core. While full-core, pin-resolved transport simulations are still beyond the scope of existing computer architectures, we can begin to attack the problems using three-dimensional transport. Within a decade, exascale architectures should permit a class of full-core, pin-resolved transport simulations that will surpass the current goal of an inconsistent two-step procedure.

With these goals in mind, a full-core pressurized water reactor (PWR) benchmark problem has been chosen as the metric problem. The core is an EDF PWR900 [6] model core with the dimensions outlined in Table 3.4.1.

Core Height	Assembly Height	Lattice Pitch	Assemblies
4.2 m	3.6 m	1.26 cm	17×17 (289) 157 fuel 132 reflector

**Table 3.4.1: PWR900 core dimensions and configuration.**

The core contains 289 (17×17) total assemblies, of which 157 are fuel and 132 are in the reflector. Each assembly contains a 17×17 array of homogenized fuel pins as shown in Figure 3.4.2. Three different fuel enrichments ranging from 1.5% to 3.25% (LEU, MEU, HEU) are used in the assemblies. Each assembly contains 45 homogenized pin-cells. Summing 45 pin-cells per assembly with 3 enrichment levels yields 135 total materials. The core problem geometry is shown in Figure 3.4.3.

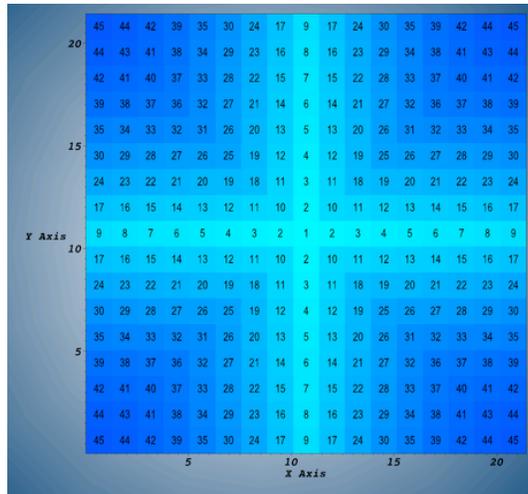


Figure 3.4.2: PWR900 17×17 pin fuel assembly. The pins have been homogenized into 45 unique materials in each assembly. The labels show the material ids in a LEU assembly. Material ids cover the ranges [1,45] (LEU), [46, 90] (MEU), and [91, 135] (HEU). All assemblies have the same ¼ symmetry pattern.

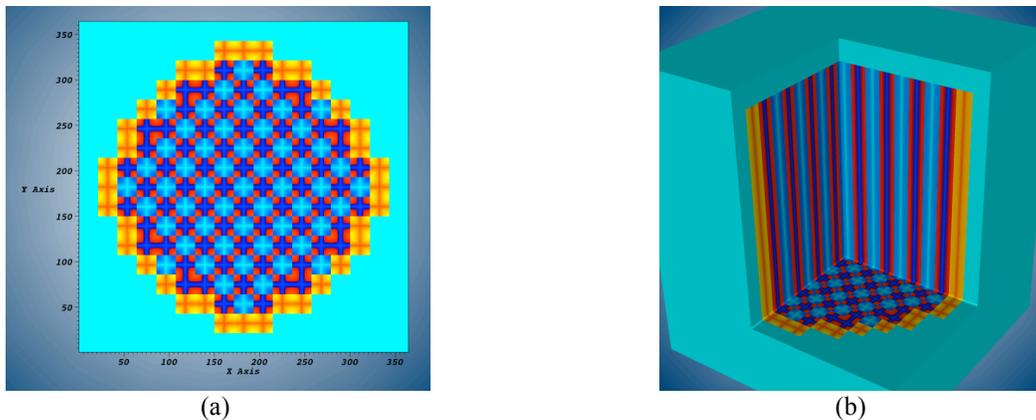


Figure 3.4.3: PWR900 core model: (a) 2D radial view ( $xy$ -plane), and (b) 3D view showing axial ( $z$ -axis) geometry. The assembly enrichments are LEU (light blue), MEU (red/blue), and HEU (yellow/orange).

Denovo is used to solve Equation 2 for the  $k$ -eigenvalue and the scalar flux throughout the core. Using the scalar flux, the pin power distribution, fission source, and groupwise power distributions can be analyzed. The ability to solve pin-homogenized, whole-core problems with transport, as opposed to diffusion or other low-order approximations, is the first step towards fully predictive reactor core modeling and simulation. To achieve first-principle predictive capability, each pin would be fully resolved in the three-dimensional, whole-core model. This objective cannot be approached, however, until we have demonstrated the ability to solve pin-homogenized, three-dimensional reactor problems with full transport.

### 3.4.5 The Model and Algorithm

Denovo solves Equations 1 and 2 using the discrete ordinates ( $S_N$ ) method, a finite-element collocation scheme in angle in which the transport equation is solved at discrete angles. The scattering source is represented by an expansion in spherical harmonics and the energy is discretized using the multigroup approximation (Petrov-Galerkin finite element). In order to preserve particle balance (conservation) the discrete angles are chosen from a quadrature set that is capable of integrating the

spherical harmonics in the scattering expansion. The resulting system of discrete equations for Equation 2 takes the form

$$\hat{\Omega}_n \cdot \nabla \psi_n^g + \sigma^g \psi_n^g = \sum_{g'=0}^G \sum_{l=0}^L \sigma_{sl}^{gg'} \left[ \phi_{l0}^{g'} Y_{l0}^e(\Omega_n) + \sum_{m=1}^l \left( \phi_{lm}^{g'} Y_{lm}^e(\Omega_n) + \vartheta_{lm}^{g'} Y_{lm}^o(\Omega_n) \right) \right] - \frac{1}{k} Y_{00}^e(\Omega_n) \chi^g \sum_{g'=0}^G \nu \sigma_{f}^{g'} \phi_{00}^{g'}, \quad (7)$$

where the moments of the angular flux are defined as

$$\phi_{lm} = \sum_{n=0}^N w_n Y_{lm}^e(\Omega_n) \psi_n, \quad m \geq 0, \quad (8)$$

$$\vartheta_{lm} = \sum_{n=0}^N w_n Y_{lm}^o(\Omega_n) \psi_n, \quad m > 0. \quad (9)$$

From Equation 8 the scalar flux of Equation 3 is defined to be proportional to the (0, 0) angular flux moment:

$$\phi = \int \psi \, d\Omega = \frac{1}{Y_{00}^e} \sum_{n=0}^N w_n Y_{00}^e \psi_n = \sqrt{4\pi} \phi_{00}. \quad (10)$$

In operator form (7), (8), and (9) are written

$$\mathbf{L}\Psi = \mathbf{M}\mathbf{S}\Phi - \frac{1}{k} \mathbf{M}\chi \mathbf{f}^T \Phi, \quad (11)$$

$$\Phi = \mathbf{D}\Psi. \quad (12)$$

In block matrix form we have

$$\begin{pmatrix} \mathbf{L} & -\mathbf{M}(\mathbf{S} + \frac{1}{k} \chi \mathbf{f}^T) \\ -\mathbf{D} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \Psi \\ \Phi \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \quad (13)$$

Taking the Schur Complement gives

$$(\mathbf{I} - \mathbf{T}\mathbf{S})\Phi = \frac{1}{k} \mathbf{T}\chi \mathbf{f}^T \Phi, \quad (14)$$

where  $\mathbf{T} = \mathbf{D}\mathbf{L}^{-1}\mathbf{M}$ . Recall that the full eigenvector matrix,  $\Phi$ , is a function of energy, space, and angle (moments),

$$\Phi = \left( [\phi]_0 \quad [\phi]_1 \quad \dots \quad [\phi]_G \right)^T, \quad (15)$$

where each  $[\cdot]$  indicates a block matrix over space and moments for a given energy group. The matrices in Equation 14 have the following form

$$\left[ \mathbf{I} - \begin{pmatrix} [\mathbf{T}]_0 & 0 & 0 & 0 \\ 0 & [\mathbf{T}]_1 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & [\mathbf{T}]_G \end{pmatrix} \begin{pmatrix} [\mathbf{S}]_{00} & [\mathbf{S}]_{01} & \dots & [\mathbf{S}]_{0G} \\ [\mathbf{S}]_{10} & [\mathbf{S}]_{11} & \dots & [\mathbf{S}]_{1G} \\ \vdots & \vdots & \ddots & \vdots \\ [\mathbf{S}]_{G0} & [\mathbf{S}]_{G1} & \dots & [\mathbf{S}]_{GG} \end{pmatrix} \right] \begin{pmatrix} [\phi]_0 \\ [\phi]_1 \\ \vdots \\ [\phi]_G \end{pmatrix} =$$

$$\frac{1}{k} \begin{pmatrix} [\mathbf{T}]_0 & 0 & 0 & 0 \\ 0 & [\mathbf{T}]_1 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & [\mathbf{T}]_G \end{pmatrix} \begin{pmatrix} [\mathbf{F}]_{00} & [\mathbf{F}]_{01} & \dots & [\mathbf{F}]_{0G} \\ [\mathbf{F}]_{10} & [\mathbf{F}]_{11} & \dots & [\mathbf{F}]_{1G} \\ \vdots & \vdots & \ddots & \vdots \\ [\mathbf{F}]_{G0} & [\mathbf{F}]_{G1} & \dots & [\mathbf{F}]_{GG} \end{pmatrix} \begin{pmatrix} [\phi]_0 \\ [\phi]_1 \\ \vdots \\ [\phi]_G \end{pmatrix}. \quad (16)$$

The fission matrix is a rank-one matrix assembled from the fission spectrum and cross-section vectors,

$$[\mathbf{F}]_{gg'} = [\chi]_g [\nu\sigma_f]_{g'}. \quad (17)$$

Traditionally, the energy-dependent eigenvector of flux moments is not required because reactor power via Equation 5 is the pertinent quantity in reactor analysis. In this case, the eigenvector is defined as the total fission source,

$$\mathbf{R} = \mathbf{f}^T \Phi, \quad (18)$$

and Equation 14 becomes

$$[\mathbf{f}^T (\mathbf{I} - \mathbf{TS})^{-1} \mathbf{T}\chi] \mathbf{R} = k\mathbf{R}. \quad (19)$$

Defining  $\mathbf{A} = \mathbf{f}^T (\mathbf{I} - \mathbf{TS})^{-1} \mathbf{T}\chi$ , Equation 19 has the common eigenvalue form  $\mathbf{A}\mathbf{R} = k\mathbf{R}$ .

The traditional method for solving Equation 19 is power iteration,

$$\mathbf{R}^{n+1} = \frac{1}{k} \mathbf{A}\mathbf{R}^n, \quad (20)$$

where the operation requires the solution to a multigroup fixed source problem,

$$(\mathbf{I} - \mathbf{TS})\Phi^{n+1} = \mathbf{T}\chi\mathbf{f}^T \Phi^n = q_f^n. \quad (21)$$

The solution strategy for the multigroup problem depends largely on the structure of  $\mathbf{S}$ . The most straightforward approach is to use Gauss-Seidel iteration,

$$\begin{aligned} (\mathbf{I} - [\mathbf{T}]_g [\mathbf{S}]_{gg}) [\phi]_g^{l+1, n+1} &= q_f^n + [\mathbf{T}]_g \left( \sum_{g'=0}^{g-1} [\mathbf{S}]_{gg'} [\phi]_{g'}^{l+1, n+1} + \sum_{g'=g+1}^G [\mathbf{S}]_{gg'} [\phi]_{g'}^{l, n+1} \right) = q_g, \\ g &= 0, \dots, G, \end{aligned} \quad (22)$$

When  $\mathbf{S}$  is lower-triangular (no upscattering), Gauss-Seidel iteration over energy will converge in a single iteration. For each group in the Gauss-Seidel iteration, a full space-angle transport equation must be solved. The default eigenvalue solution procedure in Denovo is as follows:

1. Outer eigenvalue power iterations to solve Equation 20
2. Gauss-Seidel iteration over energy to solve (21) for each eigenvalue iteration
3. GMRES Krylov iteration over space-angle to solve (22) for each group in the Gauss-Seidel iteration

Additionally, Denovo provides Two-Grid Transport Acceleration [5] to accelerate the Gauss-Seidel iteration over energy when upscatter is present. DSA is also available as a preconditioner for GMRES. Finally, Denovo provides rebalance eigenvalue acceleration for Equation 20 [31].

The innermost part of each step in the aforementioned process is the solution of

$$\mathbf{T}\phi = \mathbf{D}\mathbf{L}^{-1}\mathbf{M}\phi = q. \quad (23)$$

The transport streaming plus removal operator,  $\mathbf{L}$ , is a lower-left triangular operator for each angle that can be directly inverted by upwinding in the direction of neutron travel. The resulting solution is termed a transport “sweep” in the nuclear engineering literature. In the general numerical methods parlance this is a wavefront solution. Regardless of terminology, each eigenvalue iteration requires many transport sweeps. This is the most time-consuming part of the transport solution, and in general, 95-99% of the calculation time is spent doing transport sweeps. The upwinding, recursive nature of transport sweeps prevents easy linear scaling algorithms from being implemented. Parallel block-Jacobi methods

can be used, but these become very inefficient when the angular intensity is changing rapidly at processor boundaries. Plus, the transport sweep is only the innermost of several levels of iteration; thus, it is advantageous to use direct inversions at this level whenever possible. Denovo uses the well-known Koch-Baker-Alcouffe (KBA) wavefront algorithm [6] to invert  $\mathbf{L}$ .

### 3.4.6 Q2 Baseline Problem Results

The PWR900 problem has been described in Section 3.4.4. For the Q2 baseline we are solving this problem using the discretization parameters described in Table 3.4.2. The Q2 problem uses a  $2 \times 2$  spatial mesh in each pin cell. This yields 578 mesh cells in each radial direction, and each cell has a width of 0.63 cm. In the axial direction the model has 700 cells yielding 0.6 cm resolution. With this grid discretization, each mesh cell has a nearly uniform aspect ratio ( $0.63 \times 0.63 \times 0.6$ ). The spatial mesh is illustrated in Figure 3.4.4. For the Q2 baseline, Denovo uses the step-characteristics method that has one spatial unknown per cell.

Cells	Unknowns per cell	Angles	Moments	Groups	Total unknowns
233,858,800	1	168	1	2	$7.86 \times 10^{10}$

Table 3.4.2: Discretization of the Q2 problem.

The pin-cell cross sections are collapsed into two groups: fast and thermal. The fission spectrum is 1.0 in the fast group and zero in the thermal group. All cross sections use  $P_0$  scattering (1 angular moment). With these parameters, the total number of unknowns required by the Q2 baseline problem is  $7.86 \times 10^{10}$ .

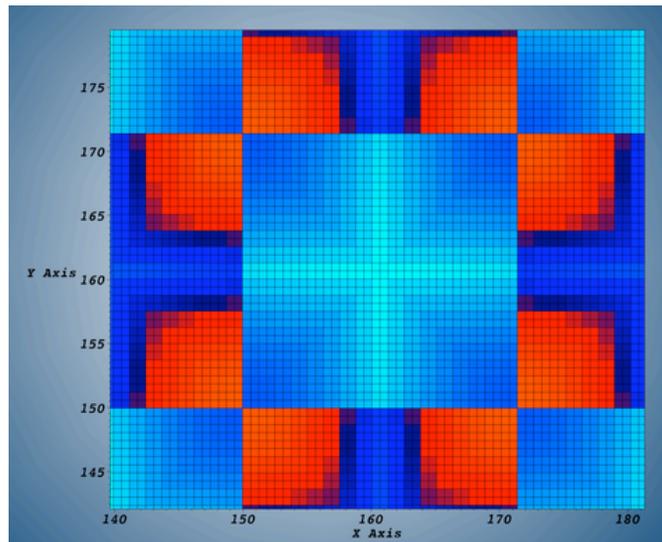


Figure 3.4.4: Close-up view of the Denovo spatial grid in the radial ( $xy$ ) plane. Each homogenized pin cell has a  $2 \times 2$  spatial grid of  $0.63 \times 0.63$  cm. The each cell has a 0.6 cm resolution in the axial ( $z$ ) direction.

The Q2 benchmark was run with a  $k_{\text{eff}}$  tolerance of  $1 \times 10^{-3}$  and an eigenvector tolerance of 0.1. The problem was run on Jaguar/XT5 on 17,424 cores with a parallel decomposition of  $132 \times 132$  domains in  $(x, y)$  with 175  $z$ -blocks. The runtimes (wall clock) for the Q2 benchmark are given in Table 3.4.3. Computational solver costs of the problem are shown in Table 3.4.4. PAPI instrumentation output is given in Table 3.4.5.

Total Time	Setup	Solver	Sweep	Two-Grid	Within-Group
187.68	0.76	186.47	186.34	50.68	133.38

**Table 3.4.3: Wall clock timing measurements. All times are in minutes. The solver contains the within-group and two-grid times. The vast majority (99.3%) of the runtime is spent doing transport sweeps.**

Eigenvalue Iterations	Gauss-Seidel Iterations	Within-group Iterations	Two-Grid Iterations	Total GMRES Iterations	Source Sweeps	Total Sweeps
7	21	636	246	882	56	938

**Table 3.4.4: Computational iteration costs for the Q2 benchmark problem.**

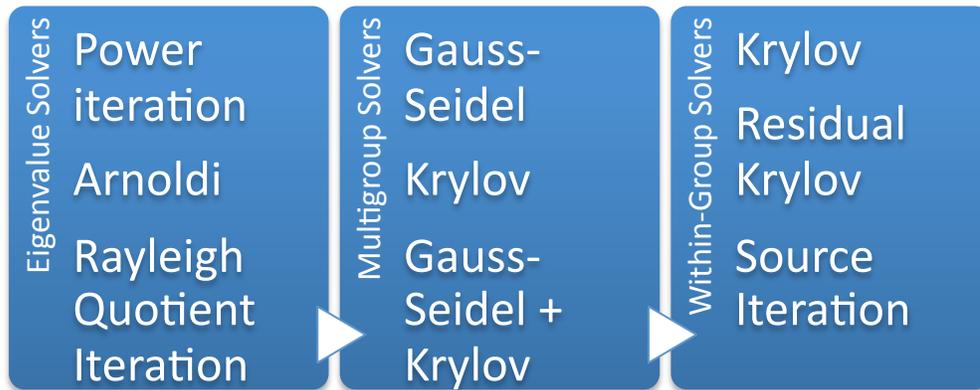
FP PAPI	INS PAPI	L2 Cache Misses	FLOPS	Cycles	FP/Cycle/Core
3.983e15	7.101e17	3.197e13	3.560e11	2.909e13	7.859e-3

**Table 3.4.5: PAPI instrumentation for Q2 problem.**

### 3.4.7 Scalable Multigroup Transport Algorithms

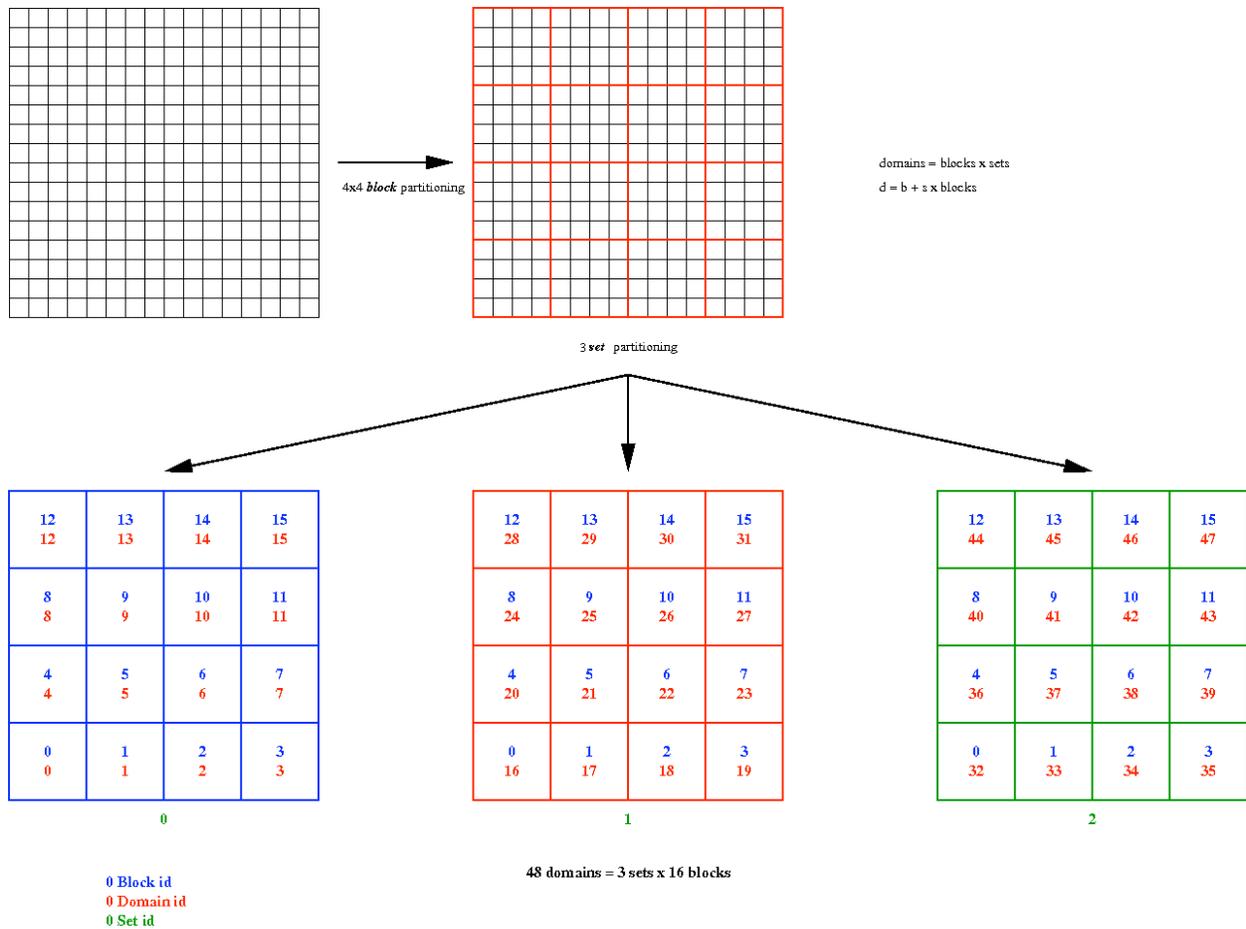
The standard power iteration method with Gauss-Seidel multigroup iteration and within-group Krylov iteration utilizing KBA sweeps is insufficient to scale to  $O(100K)$  processors. This is due to the communication latency incurred by KBA. As the KBA blocks become very small, the amount of time spent passing messages between blocks dominates the time spent solving each block. Scoping studies on Jaguar have shown that KBA tracks its theoretically predicted efficiency when the block is set greater than  $\sim 1500$  cells. Thus, for any given problem, the maximum number of cores is predetermined by the minimum block size. Even a 500M cell problem will be limited to 15,000 - 20,000 cores under these restrictions. To utilize the full resources of Jaguar we must find additional variables to parallelize. Using a new set of advanced solvers in Denovo, a multilevel decomposition over energy provides the necessary parallelism.

To scale to  $O(100K)$  cores, we have implemented an expanded solver taxonomy in Denovo that is divided into Within-Group Solvers, Multigroup Solvers, and Eigenvalue Solvers. These are arranged in levels as in to Figure 3.4.5.



**Figure 3.4.5: Expanded solver taxonomy in Denovo.**

The newly implemented Krylov multigroup solvers allow a multilevel decomposition in energy-space-angle as illustrated in Figure 3.4.6. In this decomposition space is partitioned into *blocks*. Energy is partitioned into *sets*. Each set contains the full mesh (all of the blocks) such that KBA sweeps never cross set boundaries. Every (block, set) combination is termed a *domain*. The total number of domains is



**Figure 3.4.6: Denovo multilevel energy decomposition. Energy is decomposed in sets and space-angle is decomposed in blocks.**

currently the same as the number of MPI processes in a parallel job. The old (Q2) Denovo space-angle decomposition can be thought of as a single-set energy decomposition over  $N_p$  blocks.

Having described the multilevel energy decomposition, the solver taxonomy can be described in more detail. In particular, the implementation of multigroup Krylov solvers enables the decomposition over energy.

### 3.4.7.1 Within-Group Solvers

All of the within-group solvers are parallelized over space because, by definition, they require no coupling between energy groups. Thus, they operate only within a set, not across sets.

Solver	Preconditioning	Parallelization
Direct Krylov	DSA	Interblock KBA
Residual Krylov	DSA	Interblock KBA
Source Iteration		Interblock KBA

### 3.4.7.2 Multigroup Solvers

Multigroup solvers are used independently to solve fixed-source problems, or they can be used in the inner iterations of eigenvalue problems. The multigroup solvers are parallelized over space-angle (blocks) and energy (sets), although not all solvers support energy parallelization. For example, the Gauss-Seidel solver does not support parallelization over energy.

Solver	Preconditioning	Parallelization
Gauss-Seidel	TTG	Single-set energy partitioning
Gauss-Seidel/Krylov		Gauss-Seidel over downscatter groups replicated on each set, Krylov iteration over upscatter groups using multiset energy partitioning
Krylov	Multigrid energy, LU	Multiset energy partitioning

### 3.4.7.3 Eigenvalue Solvers

The parallelization is largely determined by the choice of multigroup solver. Some eigenvalue solvers can solve both energy-dependent and energy-independent eigenvectors, and this choice dictates the parallelization strategy.

Solver	Eigenvector	Multigroup Solvers
Power Iteration	Energy independent	Gauss-Seidel, Krylov, and Gauss-Seidel/Krylov
Arnoldi	Energy independent/ energy dependent	Krylov and Gauss-Seidel/Krylov
Rayleigh Quotient Iteration	Energy dependent	Krylov

### 3.4.8 Additional Code Optimizations

In addition to the new solvers and parallel decompositions introduced in Section 3.4.7, we have also optimized the KBA space-angle sweep algorithm in Denovo. We have implemented three optimizations:

- fixed an ordering error in octant-level pipelining
- improved the quadrant-level pipelining
- applied latency analysis for KBA sweeps

The first two optimizations refer to the sweep ordering. The Q2 baseline and characterization runs revealed an error in angular pipelining. Instead of running pairs of (+Z) octants, Denovo was implementing the octant-based pipelining in a haphazard way. The end result was that a given process could not start work at the end of its angular sweep. We have fixed the pipeline ordering of angles so that a core can immediately begin work on the -Z angles as soon as its +Z angles have been completed.

An additional optimization resulted from the latency analysis alluded to in Section 3.4.7. Mainly, we discovered that best results with the code are attained when the cell-block size is greater than 1500 cells. Although this adds a constraint on the attainable scalability, it improves performance for a given number of cores, and we are able to amortize the cost of sweeps by applying our new multilevel energy decomposition.

### 3.4.9 Q4 Problem Results

The Q2 baseline problem was executed using 2 energy groups. The Q4 problem targeted 44 energy groups, resulting in a factor of 22 increase in the total DoF. However, before continuing, we define the weak-scaling efficiency that we will use in the remainder of this report,

$$\varepsilon = \frac{\tau_{\text{ref}}}{\tau_p} \left( \frac{\text{DoF}_p}{\text{DoF}_{\text{ref}}} \right), \quad \tau = t \times N_p. \quad (24)$$

Here,  $t$  is the wall-clock time and  $N_p$  is the number of processors (cores). The “ref” result refers to the Q2 baseline problem shown in Tables 3.4.3 – 3.4.5. The “P” subscript refers to the target problem.

The first set of runs was performed on the Q2 problem to demonstrate the performance improvements that resulted from code optimizations and new solvers. The second set of cases consisted of a 44-group version of the Q2 problem as described in Table 3.4.6. The full set of cases is described in Table 3.4.7.

Cells	Unknowns per cell	Angles	Moments	Groups	Total unknowns
233,858,800	1	168	1	44	$1.73 \times 10^{12}$

Table 3.4.6: Discretization of the Q4 problem. It has 22 times more DoF than the Q2 problem.

Problem	Case	Description
Q2	0	Baseline problem run using power iteration with transport two-grid preconditioned Gauss-Seidel multigroup solver. The parallel decomposition was $132 \times 132$ with 175 z-blocks.
Q2	1	Same solvers and decomposition as 0, but included improved KBA sweep ordering and block size analysis resulting in 10 z-blocks.
Q2	2	Same as case 1 but used a residual Krylov solver for within-group solves.
Q2	3	Used the multigroup Krylov solver with a multilevel energy decomposition of 2 sets. The mesh decomposition was $102 \times 100$ with 10 z-blocks.
Q2	4	Used the Arnoldi eigenvalue solver with a Krylov multigroup solver partitioned over 2 sets. The mesh decomposition was $102 \times 100$ with 10 z-blocks.
Q4	5	Power iteration with a Gauss-Seidel/Krylov multigroup solver partitioned over 11 sets. The mesh decomposition was $102 \times 100$ with 10 z-blocks.
Q4	6	Same as case 5 but used a full-partitioned Krylov multigroup solver partitioned over 11 sets.
Q4	7	Arnoldi solver with a Gauss-Seidel/Krylov multigroup solver partitioned over 11 sets. The mesh decomposition was $102 \times 100$ with 10 z-blocks.
Q4	8	Same as case 7 but used a full-partitioned Krylov multigroup solver partitioned over 11 sets.

Table 3.4.7: Performance cases run during Q3 and Q4. The Q2 baseline is given as case 0 (see Tables 3.4.3 – 3.4.5.)

The results in set 1 are shown in Table 3.4.8. Results for set 2 are shown in Table 3.4.9. All results are referenced to the Q2 baseline problem (case 0). Looking at the set 1 results, we see that the sweep-order optimizations and block-size analysis produce a significant savings without any additional solver improvements. These improvements result in a factor of 12 gain over the baseline case both in terms of parallel efficiency and percent peak utilization. The multigroup Krylov solver options allow the use of the multilevel parallel decomposition. Decomposing the Q2 problem over 2 sets allows a more efficient space-angle decomposition while still using roughly the same amount of computing resource. This solver/parallel decomposition option (case 3) produces a factor of 52 efficiency gain over the baseline case. Using the Arnoldi eigenvalue solver instead of power iteration is even more efficient and results in a factor of 77 gain over the baseline case (case 4).

Case	Cores	Solver Runtime (m)	FP PAPI	L2 Cache Misses	FP/Cycles/Core	% Peak	$\epsilon$
1	17,424	15.15	3.983e15	4.050e12	0.097	2.419	12.311
2	17,424	11.00	2.664e15	3.113e12	0.089	2.227	16.952
3	20,400	3.03	1.233e15	2.193e12	0.128	3.196	52.563
4	20,400	2.05	8.838e14	1.462e12	0.136	3.390	77.738

Table 3.4.8: Results of set 1 problem runs. % Peak is referenced to a maximum peak efficiency on Jaguar of 4 FP/Cycle/Core.

Case	Cores	Solver Runtime (m)	FP PAPI	L2 Cache Misses	FP/Cycles/Core	% Peak	$\epsilon$
5	112,200	38.88	6.544e16	1.065e14	0.098	2.439	16.622
6	112,200	36.30	5.090e16	1.125e14	0.080	2.003	17.552

Case	Cores	Solver Runtime (m)	FP PAPI	L2 Cache Misses	FP/Cycles/Core	% Peak	$\epsilon$
7	112,200	25.81	4.656e16	7.836e13	0.103	2.577	24.684
8	112,200	20.36	2.861e16	6.680e13	0.080	2.007	31.290

**Table 3.4.9: Results of set 2 problem runs. % Peak is referenced to a maximum peak efficiency on Jaguar of 4 FP/Cycle/Core**

All of the results in set 1 were generated using roughly the same computing resource as the Q2 baseline. Our analysis of KBA has demonstrated that this is the maximum amount of resource that could be used on a problem of this space-angle size. Using the standard power iteration plus Gauss-Seidel multigroup solver option with KBA sweeps will be unable to solve a similar sized problem with more energy groups because adding more domains to the KBA partitioning will not result in greater efficiency. Yet, the science driver for nuclear energy is greater energy resolution on full core simulations. The combination of Denovo’s new Krylov solvers and the multilevel parallel decomposition can break through this barrier because additional computing resource can be applied in the decomposition over energy. The problems in set 2 are designed to demonstrate this capability. Table 3.4.9 shows that the multilevel energy decomposition allows Denovo to solve a 44-group version of the baseline problem effectively on 112,200 cores. Using the baseline case as a reference, even though it would be impossible to effectively run this size problem using the solvers in the baseline problem, we see that the new solvers and multilevel parallel decomposition results in factors of 16 – 31 efficiency gains over the baseline problem. For each method, Denovo is achieving approximately 2 – 2.5 % peak efficiency.

### 3.4.10 Conclusions

We have implemented several code optimizations, a new parallel decomposition, and several new solvers that allows Denovo to scale to O(100K) cores. The combination of these solver and code-level improvements has resulted in factors of 16 – 31 parallel performance efficiency over a Q2 baseline problem run with Denovo’s original solver suite. Furthermore, these improvements were attained while increasing the machine utilization from 17,424 to 112,200 cores. Additionally, machine peak efficiency has increased from 0.196% for the baseline case to between 2 and 2.5% for the new solvers and multilevel parallel decomposition.

Denovo was exercised in a weak scaling manner – at a minimum, we sought to achieve a constant runtime for the same amount of work per core. Despite increasing the size of the problem by a factor of 22 while increasing the number of processors by only a factor of 6.4, we still achieve a reduction in runtime, resulting in a factor of 31 efficiency improvement in the best case. *Denovo has met the metric because the parallel efficiency has risen.*

## 4. REFERENCES

- [1] J. K. Dukowicz, R. D. Smith and R. C. Malone, "A reformulation and implementation of the Bryan-Cox-Semtner ocean model on the Connection Machine," *J. Atmos. Ocean Tech.*, vol. 14, pp. 294-317, 1993.
- [2] L. W. Wang, Z. Zhao, and J. Meza, *Phys. Rev. B*, vol. 77, 2008.
- [3] Z. Zhao, J. Meza, and L.W. Wang, *J. Phys: Condens. Matt.*, vol. 20, 2008.
- [4] T. M. Evans, A.S. Stafford, and K.T. Clarno, "Denovo -- A New Three-Dimensional Discrete Ordinates Code in SCALE," *Nuc. Tech.*, August 2010.
- [5] T. M. Evans, K.T. Clarno, and J.E. Morel, "A Transport Acceleration Scheme for Multigroup Discrete Ordinates with Upscattering," *Nuc. Sci.Eng.*, July 2010.
- [6] R. S. Baker, and K.R. Koch, "An S<sub>N</sub> Algorithm for the Massively Parallel CM-200 Computer," *Nuc. Sci.Eng.*, vol. 128, pp. 312-320, 1998.
- [7] A. J. Semtner, "Finite-difference formulation of a world ocean model," in *Advanced Physical Oceanographic Numerical Modelling*, J. J. O'Brien, Ed., ed Mass.: Dordrecht Reidel, 1986, pp. 187-202.
- [8] K. Bryan, "A numerical method for the study of the world ocean," *J. Comp. Phys.*, vol. 4, pp. 347-376, 1969.
- [9] R. D. Smith, and P. Gent, "Reference Manual for the Parallel Ocean Program (POP)," *Los Alamos Technical Report*, vol. LAUR-02-2484, 2002.
- [10] R. D. Smith, et al., "The Parallel Ocean Program (POP) Reference Manual," *Los Alamos Technical Report*, 2010.
- [11] R. D. Smith, S. Kortas and B. Meltz, "Curvilinear coordinates for global ocean models," *Los Alamos Technical Report*, vol. LA-UR-95-1146, 1995.
- [12] R. J. Murray, "Explicit generation of orthogonal grids for ocean models," *J. Comp. Phys.*, vol. 126, pp. 251-273, 1996.
- [13] M. E. Maltrud, and J. L. McClean, "An eddy resolving global 1/10 degrees ocean simulation," *Ocean Modelling*, vol. 8, pp. 31-54, 2005.
- [14] L. W. Wang, B. Lee, H. Shan, Z. Zhao, J. Meza, E. Strohmaier, and D. Bailey, *Proc. 2008 ACM/IEEE Conf. Supercomp. (ACM Gordon Bell)*, p. Article 65, 2008.
- [15] L. W. Wang. *PEtot Home Page*. Available: <http://hpcrd.lbl.gov/~linwang/PEtot/PEtot.html>
- [16] F. Gygi, et al., *Proc. 2005 ACM/IEEE Conf. Supercomp. (ACM Gordon Bell)*, 2005.
- [17] W. Kohn, *Phys. Rev. Lett.*, vol. 76, 1996.
- [18] G. Galli, and M. Parrinello, *Phys. Rev. Lett.*, vol. 69, 1992.
- [19] X. P. Li, R.W. Nunes, and D. Vanderbilt, *Phys. Rev. B*, vol. 47, 1993.
- [20] W. Yang, *Phys. Rev. Lett.*, vol. 66, 1991.
- [21] J.-L. Fattebert, and F. Gygi, *Phys. Rev. B*, vol. 73, 2006.
- [22] F. Shimojo, R.K. Kalia, A. Nakano, and P. Vashishta, *Comput. Phys. Commun.*, vol. 167, 2005.
- [23] K. Kitaura, E. Ikeo, T. Asada, T. Nakano, and M. Uebayasi, *Chem. Phys. Lett.*, vol. 313, 1999.
- [24] L. W. Wang, and A. Zunger, *J. Chem. Phys.*, vol. 100, 1994.
- [25] A. Canning, L.W. Wang, A. Williamson, and A. Zunger, *J. Comp. Phys.*, vol. 160, 2000.
- [26] L. W. Wang, *Energy & Env. Sci.*, vol. 2, 2009.
- [27] S. A. Blanton, R.L. Leheny, M.A. Hines, and P. Guyot-Sionnest, *Phys. Rev. Lett.*, vol. 79, 1997.
- [28] L. S. Li, and A.P. Alivisatos, *Phys. Rev. Lett.*, vol. 90, 2003.
- [29] M. Shim, and P. Guyot-Sionnest, *J. Chem. Phys.*, vol. 111, 1999.
- [30] G. Bester, X. Wu, D. Vanderbilt, and A. Zunger, *Phys. Rev. Lett.*, vol. 96, 2006.

[31] E. E. Lewis, and W.F. Miller, Jr., *Computational Methods of Neutron Transport*.  
LaGrange Park, IL: American Nuclear Society, Inc., 1993.







## **APPENDICES: BENCHMARK PROBLEM ENVIRONMENTS**



## **APPENDIX A. OVERVIEW**

We present in this appendix detailed information about the build and run time environments for the various benchmarks executed in Q2 and Q4 on the Cray XT5 system at ORNL's OLCF. An example follows where the source code is presented as well as the build and execution process invoked to execute instrumented code on the target machine.

## A.1 MODULES AVAILABLE ON THE TARGET ARCHITECTURE

### A.1.1 MODULES AVAILABLE IN Q2

(as of February 25, 2010)

```
----- /sw/tools/modulefiles -----
-----
mycraypat  nccsld      nccsld-test nccsld_dev  swadm      swtools
----- /sw/tools/modulefiles -----
-----
mycraypat  nccsld      nccsld-test nccsld_dev  swadm      swtools
----- /opt/modulefiles -----
-----
Base-opts/2.2.31
Base-opts/2.2.41(default)
MySQL/5.0.45
PrgEnv-cray/1.0.1(default)
PrgEnv-gnu/2.2.31
PrgEnv-gnu/2.2.41(default)
PrgEnv-intel/1.0.0
PrgEnv-pathscale/2.2.31
PrgEnv-pathscale/2.2.41(default)
PrgEnv-pgi/2.2.31
PrgEnv-pgi/2.2.41(default)
acml/4.3.0(default)
alps/1.2.0(default)
apprentice2/4.4.0.1
apprentice2/5.0.0
apprentice2/5.0.1(default)
cce/7.0.3
cce/7.0.4
cce/7.1.1
cce/7.1.2
cce/7.1.3
cce/7.1.4.111
cce/7.1.5(default)
cce/7.1.6
cce/7.2.0.131
cce/7.2.0.131.save
cray/MySQL/5.0.64-1.0000.2342.16.1
dwarf/8.6.0
dwarf/9.5.0(default)
elf/0.8.10(default)
fftw/2.1.5.1
fftw/3.1.1
fftw/3.2.0
fftw/3.2.1
fftw/3.2.2
fftw/3.2.2.1(default)
gcc/4.1.2
gcc/4.2.0.quadcore
gcc/4.3.2
gcc/4.4.1
gcc/4.4.2(default)
```

hdf5/1.8.2.2  
hdf5/1.8.2.3  
hdf5/1.8.3.0  
hdf5/1.8.3.1(default)  
hdf5/1.8.4.0  
hdf5-parallel/1.8.2.2  
hdf5-parallel/1.8.2.3  
hdf5-parallel/1.8.3.0  
hdf5-parallel/1.8.3.1(default)  
hdf5-parallel/1.8.4.0  
intel/11.0.081  
intel/11.0.083  
intel/11.0.084  
intel/11.1.046(default)  
libfast/1.0  
libfast/1.0.2  
libfast/1.0.4  
libfast/1.0.5  
libfast/1.0.6(default)  
moab/5.3.4  
moab/5.3.6(default)  
modules/3.1.6(default)  
modules/3.1.6.5  
mrnet/2.0.1.1(default)  
netcdf/3.6.2  
netcdf/4.0.0.2  
netcdf/4.0.0.3  
netcdf/4.0.1.0  
netcdf/4.0.1.1(default)  
netcdf/4.0.1.2  
netcdf-hdf5parallel/4.0.0.2  
netcdf-hdf5parallel/4.0.0.3  
netcdf-hdf5parallel/4.0.1.0  
netcdf-hdf5parallel/4.0.1.1(default)  
netcdf-hdf5parallel/4.0.1.2  
pathscale/3.2(default)  
pathscale/3.2.99  
petsc/2.3.3a  
petsc/3.0.0.1  
petsc/3.0.0.4  
petsc/3.0.0.6  
petsc/3.0.0.7  
petsc/3.0.0.8(default)  
petsc-complex/2.3.3a  
petsc-complex/3.0.0.1  
petsc-complex/3.0.0.4  
petsc-complex/3.0.0.6  
petsc-complex/3.0.0.7  
petsc-complex/3.0.0.8(default)  
pgi/10.0.0  
pgi/7.2.5  
pgi/8.0.1  
pgi/8.0.3  
pgi/8.0.4  
pgi/8.0.5  
pgi/8.0.6  
pgi/9.0.1  
pgi/9.0.2

pgi/9.0.3  
pgi/9.0.4(default)  
torque/2.4.1b1-snap.200905191614(default)  
torque-chester/2.3.2-snap.200807092141  
torque-chester/2.4.1b1-snap.200903301736(default)  
torque-istanbul/2.4.1b1-snap.200905191614(default)  
torque-jaguar/2.4.1b1-snap.200905191614(default)  
torque-jaguarpf/2.4.1b1-snap.200905191614(default)  
torque-rizzo/2.3.2-snap.200807092141  
torque-rizzo/2.4.1b1-snap.200903301736(default)  
xt-asyncpe/2.0  
xt-asyncpe/2.0.34  
xt-asyncpe/2.1  
xt-asyncpe/2.3  
xt-asyncpe/2.4  
xt-asyncpe/3.1.20  
xt-asyncpe/3.2  
xt-asyncpe/3.3  
xt-asyncpe/3.4  
xt-asyncpe/3.5(default)  
xt-asyncpe/3.6  
xt-atp/1.0(default)  
xt-craypat/4.4.0.4  
xt-craypat/5.0.0  
xt-craypat/5.0.1(default)  
xt-craypat/craypat  
xt-lgdb/1.2(default)  
xt-libsci/10.3.1  
xt-libsci/10.3.2  
xt-libsci/10.3.3  
xt-libsci/10.3.4  
xt-libsci/10.3.5  
xt-libsci/10.3.7  
xt-libsci/10.3.8  
xt-libsci/10.3.8.1  
xt-libsci/10.3.9  
xt-libsci/10.4.0  
xt-libsci/10.4.1(default)  
xt-mpt/2.1.50HD  
xt-mpt/3.1.0  
xt-mpt/3.1.1  
xt-mpt/3.1.2  
xt-mpt/3.2.0  
xt-mpt/3.3.0  
xt-mpt/3.4.0  
xt-mpt/3.4.1  
xt-mpt/3.4.2  
xt-mpt/3.5.0  
xt-mpt/3.5.1(default)  
xt-mpt/4.0.0  
xt-os/2.2.31  
xt-os/2.2.41  
xt-papi/3.6.1a  
xt-papi/3.6.2  
xt-papi/3.6.2.2(default)  
xt-pe/2.2.31  
xt-pe/2.2.41  
xt-service/2.2.31

xt-service/2.2.41  
 xt4/1.0(default)  
 xt5/1.0(default)  
 xtpe-target-catamount  
 xtpe-target-cn1

----- /opt/modules/3.1.6 -----

-----  
 modulefiles/modules/dot                    modulefiles/modules/modules  
 modulefiles/modules/module-cvs    modulefiles/modules/null  
 modulefiles/modules/module-info   modulefiles/modules/use.own

----- /sw/xt5/modulefiles -----

-----  
 DefApps                                    liblut/0.9.6  
 MiscApps                                  liblut/0.9.7  
 adios/0.9.10                              liblut/0.9.8  
 adios/0.9.10\_phdf5                      liblut/0.9.9(default)  
 adios/0.9.12                              liblut/1.0.0  
 adios/0.9.13(default)                    libxml2/2.7.6(default)  
 adios/1.0                                  libxslt/1.1.26(default)  
 adios/1.1.0                                m4/1.4.11(default)  
 arpack/2008.03.11(default)              matlab/7.5  
 atlas/3.8.2                                matlab/7.7  
 atlas/3.8.2-fPIC-dualcore                matlab/7.8(default)  
 atlas/3.8.3                                mercurial/1.0.2  
 atlas/3.8.3-fPIC-dualcore(default)      mercurial/1.3(default)  
 autoconf/2.63(default)                   metis/4.0(default)  
 automake/1.10.1(default)                mpe2/1.0.6(default)  
 aztec/2.1(default)                        mpip/3.1.2(default)  
 banner/1.3.2(default)                    mumps/4.7.3\_par(default)  
 blas/ref(default)                        mumps/4.9\_par  
 blas/ref-dualcore                        namd/2.6  
 blas-goto/1.0                             namd/2.7b1(default)  
 blas-goto/2.0(default)                   ncl/5.0.0(default)  
 bugget/2.0(default)                      nco/3.9.4(default)  
 cdo/1.3.1                                 nco/3.9.8  
 cdo/1.3.2(default)                        ncview/1.93c(default)  
 cmake/2.6.1(default)                     ncview/1.93g  
 cmake/2.6.2                                ncview/1.93g-netcdf4  
 cmake/2.6.4                                nedit/5.5(default)  
 cmake/2.8.0                                netcdf/3.6.2  
 cpmd/3.13.1                                netcdf/4.0.0  
 cpmd/3.13.2(default)                     netcdf/4.0.0\_par  
 ddt/2.4.1                                 nose/0.10.4(default)  
 ddt/2.4.1-11140                            omp/1.4a1-dtr103  
 ddt/2.5.1-12323(default)                omp/1.4a1r21772  
 doxygen/1.5.6                             omp/1.7a1r22177  
 doxygen/1.5.8                             omp/1.7a1r22229  
 doxygen/1.5.9                             omp/default(default)  
 doxygen/1.6.1(default)                    omp/experimental  
 dragon/1.0a(default)                     omp/routed-pgi-uos  
 ferret/6.1(default)                        omp/routing-pgi  
 ferret/6.3                                 omp/routing-pgi-uos  
 fftpack/5-r4i4                            omp/routing-test  
 fftpack/5-r8i4                            omp/routing-test-hexcore  
 fftpack/5-r8i8(default)                 omp/standard(default)  
 fftw/3.1.2                                p-netcdf/1.0.2

```

fftw/3.1.2-dualcore
fftw/3.2
fftw/3.2-dualcore
fftw/3.2.1
fftw/3.2.2
fftw/3.2.2-dualcore
fpmpi/1.0
fpmpi/1.1(default)
fpmpi_papi/1.0
fpmpi_papi/1.1(default)
gamess/2008Mar04
gamess/2009Jan12(default)
ghostscript/8.64(default)
git/1.6.0
git/1.6.0.4
git/1.6.2.4
git/1.6.4(default)
globalarrays/4.0.8(default)
globalarrays/4.1.1
globalarrays/4.2
globus/4.2.1(default)
gmake/3.81(default)
gnuplot/4.2.3
gnuplot/4.2.4(default)
gnuplot/4.2.5
gptl/3.4.1
gptl/3.4.3
gptl/3.4.7
gptl/3.5
gptl/3.5.1
gptl/3.5.2
gptl/3.6(default)
gptl_pmpi/3.6(default)
grace/5.1.21
grace/5.1.22(default)
gromacs/3.3.3
gromacs/4.0.5(default)
gsl/1.11
gsl/1.11-dualcore
gsl/1.12
gsl/1.12-dualcore(default)
gv/3.6.7(default)
hdf5/1.6.7
hdf5/1.6.7_par
hdf5/1.6.8
hdf5/1.6.8_par
hdf5/1.8.1
hdf5/1.8.1_par
hdf5/1.8.2
hdf5/1.8.2_par
hpctoolkit/4.9.2(default)
hypre/2.0.0(default)
idl/6.4(default)
imagemagick/6.4.2(default)
imagemagick/6.5.4
iota/0.2.1
iota/0.2.2(default)
java-jdk/1.5.0.06
p-netcdf/1.0.3(default)
p-netcdf/1.1.1
parmetis/3.1
parmetis/3.1.1(default)
perl-gd/2.44(default)
petsc/2.3.3-debug
petsc/3.0.0-custom
petsc/3.0.0-debug
petsc-complex/2.3.3-debug
petsc-complex/3.0.0-debug
pgplot/5.2(default)
pltar/0.8.9(default)
pltar/0.9.0
pspline/1.0(default)
python/2.5.2
python/2.5.2-netcdf(default)
qt/4.3.4(default)
ruby/1.8.7
ruby/1.9.1(default)
silob/4.7(default)
silob/test
spdcp/0.3.6
spdcp/0.3.7
spdcp/0.3.8(default)
spdcp/0.3.9
sprng/2.0b(default)
stagesub/1.0.2
stagesub/1.0.3(default)
subversion/1.4.6
subversion/1.5.0(default)
sundials/2.3.0(default)
superlu/3.0(default)
superlu_dist/2.2(default)
swig/1.3.36(default)
gzip/2.1(default)
gzip/2.1.tpb
tau/2.17.2
tau/2.17.3
tau/2.18.1(default)
tau/2.19
tkdiff/4.1.4(default)
totalview/8.6.0-1(default)
totalview/8.7.0-1
trilinos/10.0.4
trilinos/8.0.3
trilinos/9.0.2(default)
udunits/1.12.4
udunits/1.12.9(default)
umfpack/5.1.1(default)
valgrind/3.3.1
valgrind/3.4.1(default)
vampir/2.1.0(default)
vampirtrace/5.8(default)
vim/7.1
vim/7.2(default)
visit/1.11.1(default)
wgrib/1.8.0.12o(default)
wgrib/1.8.0.13b

```

```

java-jdk/1.6.0.06(default)
java-jre/1.5.0.06(default)
lammps/22Jul09
lammps/4Mar08
lammps/May08(default)
lapack/3.1.1(default)
lapack/3.1.1-dualcore
lapack/3.1.1-fPIC
libgd/2.0.35(default)
workflow/fspdemo
workflow/fullelm-2.0
workflow/fullelm-dart-1.0
workflow/gem-monitor-1.0
workflow/gtc-monitor-1.0
workflow/s3d-monitor-1.0
workflow/xgc-monitor-2.0
workflow/xgc-monitor-3.0(default)
xt-find/0.2.0(default)

```

```

----- /opt/cray/xt-asyncpe/3.5/modulefiles -----
-----
xtpe-barcelona      xtpe-mc12          xtpe-shanghai
xtpe-istanbul       xtpe-mc8           xtpe-target-native

```

### A.1.2 MODULES AVAILABLE IN Q4

(as of September 8, 2010)

```

----- /sw/tools/modulefiles -----
-----
mycraypat  nccsld      nccsld-test nccsld_dev  swadm      swtools

```

```

----- /sw/tools/modulefiles -----
-----
mycraypat  nccsld      nccsld-test nccsld_dev  swadm      swtools

```

```

----- /opt/cray/xt-asyncpe/3.7/modulefiles -----
-----
xtpe-barcelona      xtpe-mc12          xtpe-shanghai
xtpe-istanbul       xtpe-mc8           xtpe-target-native

```

```

----- /opt/modulefiles -----
-----
Base-opts/2.1.27HD
Base-opts/2.1.27HD.lusrelsave
Base-opts/2.1.29HD
Base-opts/2.1.29HD.lusrelsave
Base-opts/2.1.41HD
Base-opts/2.1.41HD.lusrelsave
Base-opts/2.1.50HD
Base-opts/2.1.50HD.lusrelsave
Base-opts/2.1.50HD_PS08
Base-opts/2.1.50HD_PS08.lusrelsave
Base-opts/2.2.27
Base-opts/2.2.27.lusrelsave
Base-opts/2.2.31
Base-opts/2.2.31.lusrelsave
Base-opts/2.2.31A
Base-opts/2.2.31A.lusrelsave
Base-opts/2.2.41
Base-opts/2.2.41.lusrelsave
Base-opts/2.2.41A(default)
Base-opts/2.2.41A.lusrelsave
PrgEnv-cray/1.0.0
PrgEnv-cray/1.0.1(default)
PrgEnv-gnu/2.1.27HD
PrgEnv-gnu/2.1.29HD
PrgEnv-gnu/2.1.41HD

```

PrgEnv-gnu/2.1.50HD  
PrgEnv-gnu/2.1.50HD\_PS08  
PrgEnv-gnu/2.2.27  
PrgEnv-gnu/2.2.31  
PrgEnv-gnu/2.2.31A  
PrgEnv-gnu/2.2.41  
PrgEnv-gnu/2.2.41A(default)  
PrgEnv-intel/1.0.0(default)  
PrgEnv-pathscales/2.1.27HD  
PrgEnv-pathscales/2.1.29HD  
PrgEnv-pathscales/2.1.41HD  
PrgEnv-pathscales/2.1.50HD  
PrgEnv-pathscales/2.1.50HD\_PS08  
PrgEnv-pathscales/2.2.27  
PrgEnv-pathscales/2.2.31  
PrgEnv-pathscales/2.2.31A  
PrgEnv-pathscales/2.2.41  
PrgEnv-pathscales/2.2.41A(default)  
PrgEnv-pgi/2.1.27HD  
PrgEnv-pgi/2.1.29HD  
PrgEnv-pgi/2.1.41HD  
PrgEnv-pgi/2.1.50HD  
PrgEnv-pgi/2.1.50HD\_PS08  
PrgEnv-pgi/2.2.27  
PrgEnv-pgi/2.2.31  
PrgEnv-pgi/2.2.31A  
PrgEnv-pgi/2.2.41  
PrgEnv-pgi/2.2.41A(default)  
acml/4.0.1a  
acml/4.1.0  
acml/4.2.0  
acml/4.3.0(default)  
acml/4.4.0  
apprentice2/5.0.0  
apprentice2/5.0.1  
apprentice2/5.0.2(default)  
apprentice2/5.1.0  
blcr/0.7.3  
cce/7.0.0  
cce/7.0.1  
cce/7.0.2  
cce/7.0.3  
cce/7.0.4  
cce/7.1.0  
cce/7.1.1  
cce/7.1.2  
cce/7.1.3  
cce/7.1.4.111  
cce/7.1.5(default)  
cce/7.1.6  
cray/MySQL/5.0.64-1.0000.2342.16.1  
cray/account/1.0.0-2.0202.18612.42.3  
cray/audit/1.0.0-1.0202.19561.0  
cray/csa/3.0.0-1\_2.0202.18623.63.1  
cray/job/1.5.5-0.1\_2.0202.18632.46.1  
cray/projdb/1.0.0-1.0202.18638.45.1  
dwarf/8.2.0  
dwarf/8.4.0

dwarf/8.6.0  
dwarf/8.8.0  
dwarf/9.5.0  
dwarf/9.9.0(default)  
elf/0.8.10  
elf/0.8.12(default)  
fftw/2.1.5  
fftw/2.1.5.1  
fftw/3.1.1  
fftw/3.2.0  
fftw/3.2.1  
fftw/3.2.2  
fftw/3.2.2.1(default)  
fftw/3.2.2.1.bak  
gcc/4.1.2  
gcc/4.2.0.quadcore  
gcc/4.2.3  
gcc/4.2.4  
gcc/4.3.2  
gcc/4.4.1  
gcc/4.4.2(default)  
gcc/4.4.3  
gcc/4.4.4  
gcc-catamount/3.3  
gnet/2.0.5  
hdf5/1.8.2.2  
hdf5/1.8.2.3  
hdf5/1.8.3.0  
hdf5/1.8.3.1(default)  
hdf5/1.8.4.0  
hdf5/1.8.4.1  
hdf5-parallel/1.8.2.2  
hdf5-parallel/1.8.2.3  
hdf5-parallel/1.8.3.0  
hdf5-parallel/1.8.3.1  
hdf5-parallel/1.8.4.0  
hdf5-parallel/1.8.4.1(default)  
intel/11.1.046(default)  
intel/11.1.064  
iobuf/1.0.6(default)  
java/jdk1.6.0\_17  
java/jdk1.6.0\_20(default)  
libfast/1.0  
libfast/1.0.2  
libfast/1.0.3  
libfast/1.0.4  
libfast/1.0.5  
libfast/1.0.6  
libfast/1.0.7(default)  
libfast/orig.1.0.3  
libscifft-pgi/1.0.0(default)  
mazama/4.0.0(default)  
moab/5.2.3  
moab/5.2.4  
moab/5.3.0  
moab/5.3.3  
moab/5.3.6(default)  
modules/3.1.6

modules/3.1.6.5(default)  
mrnet/2.0.1.1(default)  
mrnet/2.2.0.1  
netcdf/3.6.2  
netcdf/4.0.0.2  
netcdf/4.0.0.3  
netcdf/4.0.1.0  
netcdf/4.0.1.1  
netcdf/4.0.1.2  
netcdf/4.0.1.3(default)  
netcdf-hdf5parallel/4.0.0.2  
netcdf-hdf5parallel/4.0.0.3  
netcdf-hdf5parallel/4.0.1.0  
netcdf-hdf5parallel/4.0.1.1  
netcdf-hdf5parallel/4.0.1.2  
netcdf-hdf5parallel/4.0.1.3(default)  
pathscale/3.2  
pathscale/3.2.99(default)  
petsc/2.3.3a  
petsc/3.0.0  
petsc/3.0.0.1  
petsc/3.0.0.10(default)  
petsc/3.0.0.2  
petsc/3.0.0.3  
petsc/3.0.0.4  
petsc/3.0.0.6  
petsc/3.0.0.8  
petsc/3.1.00  
petsc-complex/2.3.3a  
petsc-complex/3.0.0  
petsc-complex/3.0.0.1  
petsc-complex/3.0.0.10(default)  
petsc-complex/3.0.0.2  
petsc-complex/3.0.0.3  
petsc-complex/3.0.0.4  
petsc-complex/3.0.0.6  
petsc-complex/3.0.0.8  
petsc-complex/3.1.00  
pgi/10.0.0  
pgi/10.1.0  
pgi/10.2.0  
pgi/10.3.0(default)  
pgi/10.4.0  
pgi/10.5.0  
pgi/6.2.5  
pgi/7.0.7  
pgi/7.1.6  
pgi/7.2.3  
pgi/7.2.4  
pgi/7.2.5  
pgi/8.0.1  
pgi/8.0.2  
pgi/8.0.3  
pgi/8.0.4  
pgi/8.0.5  
pgi/8.0.6  
pgi/9.0.1  
pgi/9.0.2

pgi/9.0.3  
pgi/9.0.4  
pkgconfig/0.15.0(default)  
torque/2.3.2-snap.200807092141  
torque/2.4.1b1-snap.200905191614(default)  
xt-asyncpe/1.0c  
xt-asyncpe/1.1  
xt-asyncpe/1.2  
xt-asyncpe/2.0  
xt-asyncpe/2.0.34  
xt-asyncpe/2.1  
xt-asyncpe/2.3  
xt-asyncpe/2.4  
xt-asyncpe/2.5  
xt-asyncpe/3.0  
xt-asyncpe/3.1  
xt-asyncpe/3.1.20  
xt-asyncpe/3.2  
xt-asyncpe/3.3  
xt-asyncpe/3.4  
xt-asyncpe/3.5  
xt-asyncpe/3.7(default)  
xt-asyncpe/3.8  
xt-asyncpe/3.9  
xt-asyncpe/4.0  
xt-atp/1.0(default)  
xt-boot/2.1.27HD  
xt-boot/2.1.29HD  
xt-boot/2.1.41HD  
xt-boot/2.1.50HD  
xt-boot/2.1.50HD\_PS08  
xt-boot/2.2.27  
xt-boot/2.2.31  
xt-boot/2.2.31A  
xt-boot/2.2.41  
xt-boot/2.2.41A  
xt-catamount/2.1.27HD  
xt-catamount/2.1.29HD  
xt-catamount/2.1.41HD  
xt-catamount/2.1.50HD  
xt-catamount/2.1.50HD\_PS08  
xt-craypat/5.0.0  
xt-craypat/5.0.1  
xt-craypat/5.0.2(default)  
xt-craypat/5.1.0  
xt-lgdb/1.2(default)  
xt-libc/2.1.27HD  
xt-libc/2.1.29HD  
xt-libc/2.1.41HD  
xt-libc/2.1.50HD  
xt-libc/2.1.50HD\_PS08  
xt-libsci/10.2.1  
xt-libsci/10.3.1  
xt-libsci/10.3.5  
xt-libsci/10.3.6  
xt-libsci/10.3.8  
xt-libsci/10.3.8.1  
xt-libsci/10.3.9

xt-libsci/10.4.0  
xt-libsci/10.4.1  
xt-libsci/10.4.2  
xt-libsci/10.4.3  
xt-libsci/10.4.4(default)  
xt-libsci/10.4.5  
xt-lustre-ss/2.1.27HD\_1.6.5  
xt-lustre-ss/2.1.29.HD\_ORNL.nic1\_1.6.5  
xt-lustre-ss/2.1.29HD\_1.6.5  
xt-lustre-ss/2.1.29HD\_ORNL.nic10\_1.6.5  
xt-lustre-ss/2.1.29HD\_ORNL.nic11\_1.6.5  
xt-lustre-ss/2.1.29HD\_ORNL.nic12\_1.6.5  
xt-lustre-ss/2.1.29HD\_ORNL.nic2\_1.6.5  
xt-lustre-ss/2.1.29HD\_ORNL.nic5\_1.6.5  
xt-lustre-ss/2.1.29HD\_ORNL.nic6\_1.6.5  
xt-lustre-ss/2.1.41HD\_1.6.5  
xt-lustre-ss/2.1.50HD.PS04.lus.1.6.5.steve.8062\_1.6.5  
xt-lustre-ss/2.1.50HD.PS04.lus.1.6.5.steve.8072\_1.6.5  
xt-lustre-ss/2.1.50HD.PS08.lus.1.6.5.steve.8099\_1.6.5  
xt-lustre-ss/2.1.50HD.PS08.lus.1.6.5.steve.8119\_1.6.5  
xt-lustre-ss/2.1.50HD\_1.6.5  
xt-lustre-ss/2.1.50HD\_PS04\_1.6.5  
xt-lustre-ss/2.1.50HD\_PS08\_1.6.5  
xt-lustre-ss/2.1.UP00\_ORNL.nic12\_1.6.5  
xt-lustre-ss/2.1.UP00\_ORNL.nic2\_1.6.5  
xt-lustre-ss/2.1.UP00\_ORNL.nic30\_1.6.5  
xt-lustre-ss/2.1.UP00\_ORNL.nic3\_1.6.5  
xt-lustre-ss/2.1.UP00\_ORNL.nic40\_1.6.5  
xt-lustre-ss/2.1.UP00\_ORNL.nic51\_1.6.5  
xt-lustre-ss/2.1.UP00\_ORNL.nic52\_1.6.5  
xt-lustre-ss/2.2.27.lus.1.6.5.steve.8154\_1.6.5  
xt-lustre-ss/2.2.27.lus.1.6.5.steve.8182\_1.6.5  
xt-lustre-ss/2.2.27.lus.1.6.5.steve.8189\_1.6.5  
xt-lustre-ss/2.2.27\_1.6.5  
xt-lustre-ss/2.2.31.lus.1.6.5.steve.8211\_1.6.5  
xt-lustre-ss/2.2.31A.lus.1.6.5.steve.8259\_1.6.5  
xt-lustre-ss/2.2.31A\_1.6.5  
xt-lustre-ss/2.2.31\_1.6.5  
xt-lustre-ss/2.2.41A\_1.6.5  
xt-lustre-ss/2.2.41\_1.6.5  
xt-lustre-ss/2.2.UP01.lus.1.6.5.steve.8265\_1.6.5  
xt-mpt/2.1.27HD  
xt-mpt/2.1.29HD  
xt-mpt/2.1.41HD  
xt-mpt/2.1.50HD  
xt-mpt/2.1.50HD\_PS08  
xt-mpt/3.0.1  
xt-mpt/3.0.2  
xt-mpt/3.0.4  
xt-mpt/3.1.0  
xt-mpt/3.1.0.4  
xt-mpt/3.1.0.6  
xt-mpt/3.1.0.7  
xt-mpt/3.1.1  
xt-mpt/3.1.2  
xt-mpt/3.2.0  
xt-mpt/3.3.0  
xt-mpt/3.4.0

```

xt-mpt/3.4.1
xt-mpt/3.4.2
xt-mpt/3.5.0
xt-mpt/3.5.1
xt-mpt/4.0.0(default)
xt-mpt/4.0.2
xt-mpt/4.0.3
xt-mpt/4.1.0.1
xt-mpt/4.1.1
xt-mpt/5.0.0
xt-os/2.1.27HD
xt-os/2.1.29HD
xt-os/2.1.41HD
xt-os/2.1.50HD
xt-os/2.1.50HD_PS08
xt-os/2.2.27
xt-os/2.2.31
xt-os/2.2.31A
xt-os/2.2.41
xt-os/2.2.41A
xt-papi/3.6
xt-papi/3.6.1a
xt-papi/3.6.2
xt-papi/3.6.2.2
xt-papi/3.7.2(default)
xt-papi/3.7.2.0.5
xt-pe/2.1.27HD
xt-pe/2.1.29HD
xt-pe/2.1.41HD
xt-pe/2.1.50HD
xt-pe/2.1.50HD_PS08
xt-pe/2.2.27
xt-pe/2.2.31
xt-pe/2.2.31A
xt-pe/2.2.41
xt-pe/2.2.41A
xt-service/2.1.27HD
xt-service/2.1.29HD
xt-service/2.1.41HD
xt-service/2.1.50HD
xt-service/2.1.50HD_PS08
xt-service/2.2.27
xt-service/2.2.31
xt-service/2.2.31A
xt-service/2.2.41
xt-service/2.2.41A
xtgdb/1.0.0(default)
xtpe-target-catamount
xtpe-target-cn1

```

```

----- /opt/modules/3.1.6 -----
-----
modulefiles/modules/dot          modulefiles/modules/modules
modulefiles/modules/module-cvs  modulefiles/modules/null
modulefiles/modules/module-info modulefiles/modules/use.own
----- /sw/xt5/modulefiles -----
-----

```

DefApps	lapack/3.1.1-dualcore
MiscApps	lapack/3.1.1-fPIC
adios/0.9.10	libgd/2.0.35(default)
adios/0.9.10_phdf5	liblut/0.9.6
adios/0.9.12	liblut/0.9.7
adios/0.9.13	liblut/0.9.8
adios/1.0	liblut/0.9.9(default)
adios/1.1.0	liblut/1.0.0
adios/1.1.amr1	libxml2/2.7.6(default)
adios/1.2	libxslt/1.1.26(default)
adios/1.2.1(default)	lsq/0.1.0
adios/1.2_jc	m4/1.4.11(default)
arpack/2008.03.11(default)	makedepf90/2.8.8(default)
atlas/3.8.2	matlab/7.5
atlas/3.8.2-fPIC-dualcore	matlab/7.7
atlas/3.8.3	matlab/7.8(default)
atlas/3.8.3-fPIC-dualcore(default)	mercurial/1.0.2
autoconf/2.63(default)	mercurial/1.3(default)
automake/1.10.1(default)	metis/4.0(default)
automake/1.11.1	mpe2/1.0.6(default)
aztec/2.1(default)	mpip/3.1.2(default)
banner/1.3.2(default)	mumps/4.7.3_par(default)
blas/ref(default)	mumps/4.9_par
blas/ref-dualcore	mxml/2.6(default)
blas-goto/1.0	namd/2.6
blas-goto/2.0(default)	namd/2.7b1(default)
boost/1.44.0	ncl/5.0.0(default)
bugget/2.0(default)	nco/3.9.4(default)
cdo/1.3.1	nco/3.9.8
cdo/1.3.2(default)	ncview/1.93c(default)
cmake/2.6.1(default)	ncview/1.93g
cmake/2.6.2	ncview/1.93g-netcdf4
cmake/2.6.4	nedit/5.5(default)
cmake/2.8.0	netcdf/3.6.2
cpmd/3.13.1	netcdf/4.0.0
cpmd/3.13.2(default)	netcdf/4.0.0_par
ddt/2.4.1	netcdf/4.1
ddt/2.4.1-11140	netcdf/4.1.1
ddt/2.5.1-12323(default)	netcdf/4.1.1_par
doxygen/1.5.6	netcdf/4.1_par
doxygen/1.5.8	nose/0.10.4(default)
doxygen/1.5.9	ompi/1.4a1-dtr103
doxygen/1.6.1(default)	ompi/1.4a1r21772
dragon/1.0a(default)	ompi/1.7a1r22177
emacs/23.1(default)	ompi/1.7a1r22229
esmf/4.0.0r_0	ompi/1.7a1r22760
esmf/4.0.0r_g	ompi/DT/routed-pgi-uos
esmf/4.0.0rp1_0(default)	ompi/DT/routing-pgi
esmf/4.0.0rp1_g	ompi/DT/routing-pgi-uos
esmf/4.0.0rp2_0	ompi/experimental(default)
esmf/4.0.0rp2_g	ompi/standard
ferret/6.1(default)	p-netcdf/1.0.2
ferret/6.3	p-netcdf/1.0.3(default)
fftpack/5-r4i4	p-netcdf/1.1.1
fftpack/5-r8i4	parmetis/3.1
fftpack/5-r8i8(default)	parmetis/3.1.1(default)
fftw/3.1.2	perl-gd/2.44(default)
fftw/3.1.2-dualcore	petsc/2.3.3-debug

```

fftw/3.2
fftw/3.2-dualcore
fftw/3.2.1
fftw/3.2.2
fftw/3.2.2-dualcore
fpmpi/1.0
fpmpi/1.1(default)
fpmpi_papi/1.0
fpmpi_papi/1.1(default)
gamess/2008Mar04
gamess/2009Jan12(default)
ghostscript/8.64(default)
git/1.6.0
git/1.6.0.4
git/1.6.2.4
git/1.6.4(default)
globalarrays/4.0.8(default)
globalarrays/4.1.1
globalarrays/4.2
globus/4.2.1(default)
gmake/3.81(default)
gnuplot/4.2.3
gnuplot/4.2.4(default)
gnuplot/4.2.5
gptl/3.4.1
gptl/3.4.3
gptl/3.4.7
gptl/3.5
gptl/3.5.1
gptl/3.5.2
gptl/3.6(default)
gptl/3.6.3
gptl_pmpi/3.6(default)
gptl_pmpi/3.6.3
grace/5.1.21
grace/5.1.22(default)
gromacs/3.3.3
gromacs/4.0.5(default)
gsl/1.11
gsl/1.11-dualcore
gsl/1.12
gsl/1.12-dualcore(default)
gv/3.6.7(default)
hdf5/1.6.7
hdf5/1.6.7_par
hdf5/1.6.8
hdf5/1.6.8_par
hdf5/1.8.1
hdf5/1.8.1_par
hdf5/1.8.2
hdf5/1.8.2_par
hpctoolkit/4.9.2(default)
hypre/2.0.0(default)
hypre/2.4.0b
hypre/2.4.0b-craypat
hypre/2.4.0b-debug
idl/6.4(default)
imagemagick/6.4.2(default)
petsc/3.0.0-custom
petsc/3.0.0-debug
petsc-complex/2.3.3-debug
petsc-complex/3.0.0-debug
pgplot/5.2(default)
pltar/0.8.9(default)
pltar/0.9.0
pspline/1.0(default)
python/2.5.2
python/2.5.2-netcdf(default)
qt/4.3.4(default)
ruby/1.8.7
ruby/1.9.1(default)
silo/4.7(default)
silo/test
spdcp/0.3.6
spdcp/0.3.7
spdcp/0.3.8(default)
spdcp/0.3.9
sprng/2.0b(default)
stagesub/1.0.2
stagesub/1.0.3(default)
subversion/1.4.6
subversion/1.5.0(default)
sundials/2.3.0(default)
superlu/3.0(default)
superlu_dist/2.2(default)
swig/1.3.36(default)
gzip/2.1(default)
gzip/2.1.tpb
tau/2.17.2
tau/2.17.3
tau/2.18.1(default)
tau/2.19
tcl_tk/8.5.8(default)
tkdiff/4.1.4(default)
totalview/8.6.0-1(default)
totalview/8.7.0-1
trilinos/10.0.4
trilinos/10.2.2
trilinos/10.4.0
trilinos/8.0.3
trilinos/9.0.2(default)
udunits/1.12.4
udunits/1.12.9(default)
umfpack/5.1.1(default)
valgrind/3.3.1
valgrind/3.4.1(default)
vampir/2.1.0
vampir/2.2.0(default)
vampirtrace/5.10b100802
vampirtrace/5.8
vampirtrace/5.8.2
vampirtrace/5.8_noatime
vampirtrace/5.9(default)
vampirtrace/5.9-beta
vampirtrace/5.9b100716
vim/7.1

```

```
imagemagick/6.5.4
iota/0.2.1
iota/0.2.2(default)
iota/0.2.3
iota/0.2.4
java/1.5.0.06
java/1.6.0.06
java-jdk/1.5.0.06
java-jdk/1.6.0.06(default)
java-jre/1.5.0.06(default)
lammps/22Jul09
lammps/4Mar08
lammps/9Sep10
lammps/May08(default)
lapack/3.1.1(default)

vim/7.2(default)
visit/1.11.1(default)
vmd/1.8.7(default)
wgrib/1.8.0.12o(default)
wgrib/1.8.0.13b
workflow/fspdemo
workflow/fullelm-2.0
workflow/fullelm-dart-1.0
workflow/gem-monitor-1.0
workflow/gtc-monitor-1.0
workflow/s3d-monitor-1.0
workflow/xgc-monitor-2.0
workflow/xgc-monitor-3.0(default)
workflow/xgc-monitor-3.1
xt-find/0.2.0(default)
```

## A.2 MACHINE EVENT DATA COLLECTION ROUTINES

The routines presented in this appendix were used by the FY10 applications to access machine hardware events captured by PAPI during process execution of the benchmarks described in the report. These particular routines were designed to be invoked from Fortran source code on the target architecture. The API of each routine is described briefly, a compilation is presented, and an example user code that calls the routines is presented along with the outcome of a small sample problem.

### A.2.1 KRP MACHINE EVENT DATA COLLECTION API

#### *krp-init.c :*

```
void krp_init_( int * iam , int * hw_counters , long long int * rcy
, long long int * rus , long long int * ucy , long long int * uus )
```

- *\*iam* , (in) the calling MPI process ID, (out) unchanged
- *\*hw\_counters*, (in) the number of event hardware counters on the target chipset, (out) unchanged
- *\*rcy*, (in) ignored, (out) current reading of real system cycles
- *\*rus*, (in) ignored, (out) current reading of real microseconds
- *\*ucy*, (in) ignored, (out) current reading of user (virtual) cycles
- *\*uus*, (in) ignored, (out) current reading of user (virtual) microseconds

The process *\*iam==0* returns specific hardware information such as number of processor cores in the SMP node, clock speed of the processors, and some chipset vendor and model specific information. Calling the routine marks (initializes) the cycle values *\*rcy*, *\*ucy* and the internal clock values *\*rus*, *\*uus*, and informs PAPI to monitor the following events: "PAPI\_TOT\_INS", "PAPI\_FP\_INS", "PAPI\_L2\_DCM".

#### *krp-rpt-init.c :*

```
void krp_rpt_init_( int * iam , MPI_Fint * commf , int * hw_counters
, long long int * rcy , long long int * rus , long long int * ucy ,
long long int * uus )
```

- *\*iam* , (in) the calling MPI process ID, (out) unchanged
- *\*commf*, (in) MPI communicator of the calling MPI processes, (out) unchanged
- *\*hw\_counters*, (in) the number of event hardware counters on the target chipset, (out) unchanged
- *\*rcy*, (in) previously initialized real system cycles, (out) current reading of real system cycles
- *\*rus*, (in) previously initialized real microseconds, (out) current reading of real microseconds
- *\*ucy*, (in) previously initialized user (virtual) cycles, (out) current reading of user (virtual) cycles
- *\*uus*, (in) previously initialized user (virtual) microseconds, (out) current reading of user (virtual) microseconds

The PAPI event information collected by each MPI process in *\*commf* since the previous call to either *krp\_init()* or *krp\_rpt\_init()* subroutines is gathered to *\*iam==0*. The gathering process sums the collected values for each PAPI event and prints the total value and its local value for each monitored event to STDOUT. Elapsed cycle and microsecond count differences are formed by taking the difference in the values passed *\*rcy*, *\*ucy*, *\*rus*, *\*uus* against the current values on the chip. Process *\*iam==0* gathers these differences and searches for the largest value for each observable and prints this number to STDOUT

–it is the slowest process in the set of processes that dictates the overall parallel performance. All the PAPI events and clock information are turned back on and the calling process' local values are set to the current chip values for future reference.

### *krp-rpt-init-sum.c :*

```
void krp_rpt_init_sum_( int * iam , MPI_Fint * commf , int *  
hw_counters , long long int * rcy , long long int * rus , long long  
int * ucy , long long int * uus , long int * rt_rus , long int *  
rt_ins , long int * rt_fp , long int * rt_dcm )
```

- *\*iam* , (in) the calling MPI process ID, (out) unchanged
- *\*commf*, (in) MPI communicator of the calling MPI processes, (out) unchanged
- *\*hw\_counters*, (in) the number of event hardware counters on the target chipset, (out) unchanged
- *\*rcy*, (in) previously initialized real system cycles, (out) current reading of real system cycles
- *\*rus*, (in) previously initialized real microseconds, (out) current reading of real microseconds
- *\*ucy*, (in) previously initialized user (virtual) cycles, (out) current reading of user (virtual) cycles
- *\*uus*, (in) previously initialized user (virtual) microseconds, (out) current reading of user (virtual) microseconds
- *\*rt\_rus*, (in) (if *\*iam==0* then running total of real microseconds for the processes in *\*commf*) (if *\*iam !=0* then running total of *\*iam*'s real microseconds), (out) (if *\*iam==0* then the updated running total of real microseconds for the processes in *\*commf*) (if *\*iam != 0* then the updated running total of *\*iam*'s real microseconds)
- *\*rt\_ins*, (in) (if *\*iam==0* then running total of retired instructions for the processes in *\*commf*) (if *\*iam !=0* then running total of *\*iam*'s retired instructions), (out) (if *\*iam==0* then the updated running total of retired instructions for the processes in *\*commf*) (if *\*iam != 0* then the updated running total of *\*iam*'s retired instructions)
- *\*rt\_fp*, (in) (if *\*iam==0* then running total of floating point instructions executed for the processes in *\*commf*) (if *\*iam !=0* then running total of *\*iam*'s floating point instructions executed), (out) (if *\*iam==0* then the updated running total of floating point instructions executed for the processes in *\*commf*) (if *\*iam != 0* then the updated running total of *\*iam*'s floating point instructions executed)
- *\*rt\_dcm*, (in) (if *\*iam==0* then running total of level 2 data cache misses for the processes in *\*commf*) (if *\*iam !=0* then running total of *\*iam*'s level 2 data cache misses), (out) (if *\*iam==0* then the updated running total of level 2 data cache misses for the processes in *\*commf*) (if *\*iam != 0* then the updated running total of *\*iam*'s level 2 data cache misses)

The PAPI event information collected by each MPI process in *\*commf* since the previous call to either *krp\_init()*, *krp\_rpt\_init()* or *krp\_rpt\_init\_sum()* subroutines is gathered to *\*iam==0*. The gathering process sums the collected values for each PAPI event and prints the total value and its local value for each monitored event to STDOUT. Elapsed cycle and microsecond count differences are formed by taking the difference in the values passed *\*rcy*, *\*ucy*, *\*rus*, *\*uus* against the current values on the chip. Process *\*iam==0* gathers these differences and searches for the largest value for each observable and prints this number to STDOUT –it is the slowest process in the set of processes that dictates the overall parallel performance. In addition, process *\*iam==0* maintains a running total for a specific collection of events over each process in *\*commf*: total retired instructions, total floating point instructions executed, total number of level 2 data cache misses, and wall time (real) in microseconds. All processes in *\*commf* \ *\*iam==0* maintain a local running sum over each of these events as well. All the PAPI events and clock information are turned back on and the calling process' local values are set to the current chip values for future reference. The routine is intended to be used

to isolate phases of execution within loop structures but has other obvious uses –i.e. when the loop limits are a single iteration.

***krp-rpt.c :***

```
void krp_rpt_( int * iam , MPI_Fint * commf , int * hw_counters ,
long long int * rcy , long long int * rus , long long int * ucy ,
long long int * uus )
```

- *\*iam* , (in) the calling MPI process ID, (out) unchanged
- *\*commf*, (in) MPI communicator of the calling MPI processes, (out) unchanged
- *\*hw\_counters*, (in) the number of event hardware counters on the target chipset, (out) unchanged
- *\*rcy*, (in) previously initialized real system cycles, (out) unchanged
- *\*rus*, (in) previously initialized real microseconds, (out) unchanged
- *\*ucy*, (in) previously initialized user (virtual) cycles, (out) unchanged
- *\*uus*, (in) previously initialized user (virtual) microseconds, (out) unchanged

The PAPI event information collected by each MPI process in *\*commf* since the previous call to either *krp\_init()*, *krp\_rpt\_init()*, or *krp\_rpt\_init\_sum()* subroutines is gathered to *\*iam==0*. The gathering process sums the collected values for each PAPI event and prints the total value and its local value for each monitored event to STDOUT. Elapsed cycle and microsecond count differences are formed by taking the difference in the values passed *\*rcy*, *\*ucy*, *\*rus*, *\*uus* against the current values on the chip. Process *\*iam==0* gathers these differences and searches for the largest value for each observable and prints this number to STDOUT –it is the slowest process in the set of processes that dictates the overall parallel performance. The PAPI event counters are stopped.

### A.2.2 KRP USE EXAMPLE

The sample user code is intended to demonstrate the use of the KRP wrapper routines only – please do not execute the binary for this program on a large MPI allocation. The program is designed to execute two different phases inside a loop structure and both phases compute a simple nested loop matrix multiply routine (the implementation is not even checked for correctness) such that  $C \leftarrow a A B + b C$ . where *a* and *b* are complex scalars and *A*, *B*, and *C* are complex matrices composed of random coefficients in [(-.5,-.5),(.5,.5)]. The user supplies the number of loop iterations and the dimensions of the matrices.

***f-usr-krp.f90 Listing :***

```
! kenneth.roche@pnl.gov ; k8r@uw.edu

! fortran code to demo interface to krp_*( ) machine event collection routines -PAPI wrappers
! intended for fy10 GPRA-PMM applications

!-----
! main ( ) {}
!-----

program f_usr_krp

  implicit none

  include 'mpif.h'
```

```

integer, parameter :: DPC = kind( ( 1.0D0 , 1.0D0 ) )

! mpi foo

integer :: ip , np

! problem related foo

integer :: i , j , k , ierr , isd

integer :: l , m , n , ll , mm , nn , its , nits

complex( DPC ) , allocatable , dimension( : ) :: a , b , c ! for complex mm multiply -
dimension n

complex( DPC ) , allocatable , dimension( : ) :: x , y , z ! for complex mm multiply -
dimension m

complex( DPC ) :: alpha , beta , ztmp , ztmp_

! papi related foo

integer :: hwc ! number of hardware counters

integer*8 :: krp_rus , krp_rcy , krp_uus , krp_ucy

integer*8 :: t_a , ins_a , fp_a , dcm_a

integer*8 :: t_b , ins_b , fp_b , dcm_b

!
!

call MPI_Init( ierr ) !ierr not checked :: FIXME
call MPI_Comm_rank( MPI_COMM_WORLD , ip , ierr )
call MPI_Comm_size( MPI_COMM_WORLD , np , ierr )

if ( ip .eq. 0 ) then

  print * , 'm l n'

  read * , m , l , n

  print * , 'm2 l2 n2'

  read * , mm , ll , nn

  print * , 'nits'

  read * , nits

endif

krp_rus = 0
krp_rcy = 0
krp_uus = 0
krp_ucy = 0

call krp_init( ip , hwc , krp_rcy , krp_rus , krp_ucy , krp_uus )
call MPI_Bcast( m , 1 , MPI_INTEGER , 0 , MPI_COMM_WORLD , ierr )
call MPI_Bcast( l , 1 , MPI_INTEGER , 0 , MPI_COMM_WORLD , ierr )
call MPI_Bcast( n , 1 , MPI_INTEGER , 0 , MPI_COMM_WORLD , ierr )
call MPI_Bcast( mm , 1 , MPI_INTEGER , 0 , MPI_COMM_WORLD , ierr )
call MPI_Bcast( ll , 1 , MPI_INTEGER , 0 , MPI_COMM_WORLD , ierr )

```

```

call MPI_Bcast( nn , 1 , MPI_INTEGER , 0 , MPI_COMM_WORLD , ierr )
call MPI_Bcast( nits , 1 , MPI_INTEGER , 0 , MPI_COMM_WORLD , ierr )
! buffers for C <- a A B + b C
! if A[m,l], B[l,m] , C[m,n] then aAB + bC has complexity :: 8mnl + 13mn
! (ask me if this is not clear -is trivial)
allocate ( a( m * l ) , STAT = ierr )
if ( ierr .ne. 0 ) then
    write( * , * ) 'ERROR: cannot ALLOCATE a()'
endif
allocate ( b( l * n ) , STAT = ierr )
if ( ierr .ne. 0 ) then
    write( * , * ) 'ERROR: cannot ALLOCATE b()'
endif
allocate ( c( m * n ) , STAT = ierr )
if ( ierr .ne. 0 ) then
    write( * , * ) 'ERROR: cannot ALLOCATE c()'
endif
allocate ( x( mm * ll ) , STAT = ierr )
if ( ierr .ne. 0 ) then
    write( * , * ) 'ERROR: cannot ALLOCATE x()'
endif
allocate ( y( ll * nn ) , STAT = ierr )
if ( ierr .ne. 0 ) then
    write( * , * ) 'ERROR: cannot ALLOCATE y()'
endif
allocate ( z( mm * nn ) , STAT = ierr )
if ( ierr .ne. 0 ) then
    write( * , * ) 'ERROR: cannot ALLOCATE z()'
endif
! initialize the data
if ( ip .eq. 0 ) print * , '.... initialize the arrays'
isd = 7
call get_rnd( m * l , a , isd )
call get_rnd( l * n , b , isd )
call get_rnd( m * n , c , isd )
isd = 3
call get_rnd( mm * ll , x , isd )
call get_rnd( ll * nn , y , isd )

```

```

call get_rnd( mm * nn , z , isd )

alpha = ( 1.25D0 , -1.D0 )

beta = ( -.25D0 , .5D0 )

if ( ip .eq. 0 ) print * , '.... entering loop'

t_a = 0
ins_a = 0
fp_a = 0
dcm_a = 0

t_b = 0
ins_b = 0
fp_b = 0
dcm_b = 0

call krp_rpt_init( ip , MPI_COMM_WORLD , hwc , krp_rcy , krp_rus , krp_ncy , krp_uus )

do its = 1 , nits ! this is the loop over which we wish to accumulate results
  if ( ip .eq. 0 ) print * , '.... entering phase 1 work'
  ! phase 1 work - zgemm(m,l,n) -just choose a permutation -don't care about performance
here
  do i = 1 , m
    do j = 1 , n
      do k = 1 , l
        ztmp = ztmp + a( i + ( k - 1 ) * m ) * b( k + ( j - 1 ) * l )
      end do
      ztmp_ = beta * c( i + ( j - 1 ) * m )
      c( i + ( j - 1 ) * m ) = alpha * ztmp + ztmp_
      ztmp = ( 0.D0 , 0.D0 )
    end do
  end do

  call krp_rpt_init_sum( ip , MPI_COMM_WORLD , hwc , krp_rcy , krp_rus , krp_ncy , krp_uus
, t_a , ins_a , fp_a , dcm_a )

  if ( ip .eq. 0 ) print * , '.... entering phase 2 work'
  ! phase 2 work - zgemm(mm,ll,nn)
  do i = 1 , mm
    do j = 1 , nn
      do k = 1 , ll
        ztmp = ztmp + x( i + ( k - 1 ) * mm ) * y( k + ( j - 1 ) * ll )
      end do
      ztmp_ = beta * z( i + ( j - 1 ) * mm )
      z( i + ( j - 1 ) * mm ) = alpha * ztmp + ztmp_
      ztmp = ( 0.D0 , 0.D0 )
    end do
  end do

```

```

        end do

        call krp_rpt_init_sum( ip , MPI_COMM_WORLD , hwc , krp_rcy , krp_rus , krp_ucy , krp_uus
, t_b , ins_b , fp_b , dcm_b )

    enddo !end iterations

    ! when we are using the aggregation / in-iteration routine _rpt_init_sum() then ...

    if ( ip == 0 ) then ! ... report findings
        print * , 'PREDICTION P1( TOTAL FP_OPS ) = PEs * nits * (8.m.n.l + 13.m.n)'
        print * , '== ' , np * nits * ( 8 * m * n * l + 13 * m * n )
        print * , ' '
        print * , 'PREDICTION P1( FP_OPS ) = PEs * nits * (8.mm.nn.ll + 13.mm.nn)'
        print * , '== ' , np * nits * ( 8 * mm * nn * ll + 13 * mm * nn )
        print * , '
                                time
                                ins
                                fp
dm '
        print '( "P-1:", 4(2x,I20) )' , t_a , ins_a , fp_a , dcm_a
        print '( "P-2:", 4(2x,I20) )' , t_b , ins_b , fp_b , dcm_b
    endif

    call krp_rpt( ip , MPI_COMM_WORLD , hwc , krp_rcy , krp_rus , krp_ucy , krp_uus )

    deallocate( a , b , c )

    deallocate( x , y , z )

    call MPI_Barrier( MPI_COMM_WORLD , ierr )

    call MPI_Finalize( ierr )

end program f_usr_krp

```

### ***Compilation on Target Architecture :***

```

roche@jaguarpf-login1:/tmp/work/roche/GPRA-PMM-q4> module load xt-papi

roche@jaguarpf-login1:/tmp/work/roche/GPRA-PMM-q4> cat compile.f-usr-krp
date ;
cc -c ${PAPI_INCLUDE_OPTS} krp-init.c ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init.c ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init-sum.c ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt.c ;
cc -c get-crnd.c ;
ftn -c f-usr-krp.f90 ;
ftn -o xfusr-krp f-usr-krp.o get-crnd.o krp-init.o krp-rpt-init.o krp-rpt-
init-sum.o krp-rpt.o ${PAPI_POST_LINK_OPTS} -lsci -lpthread -lm

```

```

roche@jaguarpf-login1:/tmp/work/roche/GPRA-PMM-q4> rm -f *.o x* core* ;
source compile.f-usr-krp

```

```

Fri Jul 16 03:15:11 EDT 2010

```

```

/opt/cray/xt-asyncpe/3.7/bin/cc: INFO: linux target is being used
/opt/cray/xt-asyncpe/3.7/bin/ftn: INFO: linux target is being used
/opt/cray/xt-asyncpe/3.7/bin/ftn: INFO: linux target is being used

```

### ***Execution / Output of Example User Code on Target Architecture :***

```

roche@jaguarpf-login1:/tmp/work/roche/GPRA-PMM-q4> time aprun -n 10 ./xfusr-krp
m l n
32 32 32
m2 l2 n2
128 128 128

```

```

nits
3
  Initialize PAPI Hardware Event Profilers
    TotPES()[12]
    Mhz[2600]
    nCPU-SMPnode()[12]
    nSMPnodes()[1]
      vendor string cpu[AuthenticAMD]
      model string cpu[6-Core AMD Opteron(tm) Processor 23 (D0)]
      model number[16]

.... initialize the arrays
.... entering loop

  PAPI_TOT_INS : Tot[ 551908899299 ] Rt[ 6191071 ]
  PAPI_FP_INS : Tot[ 2088960 ] Rt[ 208896 ]
  PAPI_L2_DCM : Tot[ 446747839 ] Rt[ 21590 ]
  PAPI_real_cyc = 49135717298 PAPI_real_usec = 18898353
  PAPI_user_cyc = 49140000000 PAPI_user_usec = 18900000
.... entering phase 1 work

  PAPI_TOT_INS : Tot[ 11312331 ] Rt[ 1132323 ]
  PAPI_FP_INS : Tot[ 2764800 ] Rt[ 276480 ]
  PAPI_L2_DCM : Tot[ 5384 ] Rt[ 565 ]
  PAPI_real_cyc = 1874530 PAPI_real_usec = 721
  PAPI_user_cyc = 0 PAPI_user_usec = 0
.... entering phase 2 work

  PAPI_TOT_INS : Tot[ 699941124 ] Rt[ 69995205 ]
  PAPI_FP_INS : Tot[ 170065920 ] Rt[ 17006592 ]
  PAPI_L2_DCM : Tot[ 205503 ] Rt[ 21373 ]
  PAPI_real_cyc = 58440388 PAPI_real_usec = 22477
  PAPI_user_cyc = 52000000 PAPI_user_usec = 20000
.... entering phase 1 work

  PAPI_TOT_INS : Tot[ 11312315 ] Rt[ 1132325 ]
  PAPI_FP_INS : Tot[ 2764800 ] Rt[ 276480 ]
  PAPI_L2_DCM : Tot[ 5365 ] Rt[ 562 ]
  PAPI_real_cyc = 1931247 PAPI_real_usec = 742
  PAPI_user_cyc = 26000000 PAPI_user_usec = 10000
.... entering phase 2 work

  PAPI_TOT_INS : Tot[ 699941133 ] Rt[ 69995205 ]
  PAPI_FP_INS : Tot[ 170065920 ] Rt[ 17006592 ]
  PAPI_L2_DCM : Tot[ 205407 ] Rt[ 21287 ]
  PAPI_real_cyc = 58265279 PAPI_real_usec = 22410
  PAPI_user_cyc = 52000000 PAPI_user_usec = 20000
.... entering phase 1 work

  PAPI_TOT_INS : Tot[ 11312314 ] Rt[ 1132324 ]
  PAPI_FP_INS : Tot[ 2764800 ] Rt[ 276480 ]
  PAPI_L2_DCM : Tot[ 5365 ] Rt[ 566 ]
  PAPI_real_cyc = 1920881 PAPI_real_usec = 738
  PAPI_user_cyc = 0 PAPI_user_usec = 0
.... entering phase 2 work

  PAPI_TOT_INS : Tot[ 699941134 ] Rt[ 69995206 ]
  PAPI_FP_INS : Tot[ 170065920 ] Rt[ 17006592 ]
  PAPI_L2_DCM : Tot[ 205283 ] Rt[ 21287 ]
  PAPI_real_cyc = 58231930 PAPI_real_usec = 22397
  PAPI_user_cyc = 52000000 PAPI_user_usec = 20000
PREDICTION P1( TOTAL FP_OPS ) = PEs * nits * (8.m.n.l + 13.m.n)
== 8263680

PREDICTION P2( FP_OPS ) = PEs * nits * (8.mm.nn.ll + 13.mm.nn)
== 509706240

time ins fp dm
P-1: 2201 33936960 8294400 16114
P-2: 67371 2099823391 510197760 616193
PAPI_TOT_INS : Tot[ 26590 ] Rt[ 23854 ]
PAPI_FP_INS : Tot[ 0 ] Rt[ 0 ]
PAPI_L2_DCM : Tot[ 180 ] Rt[ 93 ]
PAPI_real_cyc = 1200544 PAPI_real_usec = 462

```

```
      PAPI_user_cyc = 0      PAPI_user_usec = 0
Application 2670781 resources: utime 0, stime 0

real    0m19.724s
user    0m0.148s
sys     0m0.076s
roche@jaguarpf-login1:/tmp/work/roche/GPRA-PMM-q4>
```

Note that the predicted number of floating point instructions executed is based upon knowledge of the kernel complexity multiplied by the number of iterations for the kernel multiplied by the number of processes that are locally computing the kernel. The agreement of the predictions to the measurements obtained by calling the KRP wrappers is outstanding. The code was executed on 10 PEs of the target architecture. In the first phase of execution, each PE first locally multiplied the 32x32 complex double precision matrices followed by a second phase where 128x128 complex double precision matrices were multiplied. There were three iterations of the loop over these phases of work per PE. For phase 1 and the example we predicted 8263680 and measured 8294400. For phase 2 of the example we predicted 509706240 and measured 510197760.



**APPENDIX B. TD\_SLDA**

## B.1 INPUT SETTINGS

The code was run with the same input for both quarters.

### *Nuclear Code*

```
40 40 64 1.25 1.25 1.25
92 146
1 1 1 10 2 2 1
1 0 1 1
1.d-8 3000. .4d0 0.d0 100.d0 0.d0 0
330 330 40 40

!

/tmp/work/istet/td-run/U238_gs/data-404064/

      read(31,*)Nx,Ny,Nz,Lx,Ly,Lz
      read(31,*)Nprotons,Nneutrons
      read(31,*)iter,iread,nsave,nprint,itext_inp,itext,iopt_der
      read(31,*)icoul,iext,iforce,ihfb
      read(31,*)eps,beta,hbo,e_f,e_w,alpha
      read(31,*)p_proc , q_proc , m_block , n_block
```

### *Unitary Code*

```
3 , 0 , 0
/tmp/work/istet/td-run/U238_gs/data-404064/
/tmp/work/istet/td-run/U238_gs/data-td/
/tmp/work/istet/td-run/U238_gs/data-td/
28000.d0 600.d0
0.d0 0.d0 6.d0
0.d0 0.d0 -21.d0
25. 75. 9.d0
0 0 0

open( 438 , file = 'td_slda.inp' , status = 'old' )

read( 438 , * ) icprs , i_run , i_read_wf

read( 438 , '(a80)')file_work_dir_wf

read( 438 , '(a80)')file_work_dir_in

read( 438 , '(a80)')file_work_dir_out

read( 438 , * ) E_inc , theta_inc , phi_inc

read( 438 , * ) x_n , y_n , z_n
```

## B.2 COMPILATION

TD\_SLDA was compiled using the default Portland Group Fortran compiler, version 9.0.4. Below the makefile and build scripts are reproduced for the nuclear code and the unitary code.

### B.2.1 Q2 COMPILATION

#### *Nuclear Code*

```
compile.slda-nuclear-GPRA-PMM-bm :
  clear ;
  date ;
  echo "...compilation of nuclear-solver" ;
  time source compile.nuclear-slda ;
  echo "...compilation of time dependent nuclear code" ;
  time source compile.td-slda-nuclear ;

compile.nuclear-slda :
  cc -c ${PAPI_INCLUDE_OPTS} krp-init.c ;
  cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init.c ;
  cc -c ${PAPI_INCLUDE_OPTS} krp-rpt.c ;
  cc -c get-blcs-dscr.c ;
  cc -c get-mem-req-blk-cyc.c ;
  cc -c bc-fil-wr.c ;
  cc -c bc-fil-wr-ftn.c
  ftn -c -C -Kieeee ${FFTW_INCLUDE_OPTS} slda-nuclear-solver_v03.f90 ;
  ftn -c -fastsse -Mr8 liberf.f90 ;
  ftn -o xslda-test slda-nuclear-solver_v03.o liberf.o get-blcs-dscr.o get-
mem-req-blk-cyc.o bc-fil-wr.o bc-fil-wr-ftn.o krp-init.o krp-rpt-init.o krp-
rpt.o ${FFTW_POST_LINK_OPTS} ${PAPI_INCLUDE_OPTS} -lLUT -lsci -lpthread -lm ;

compile.td-slda-nuclear :
  cc -c ${PAPI_INCLUDE_OPTS} krp-init.c ;
  cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init.c ;
  cc -c ${PAPI_INCLUDE_OPTS} krp-rpt.c ;
  ftn -c ${FFTW_INCLUDE_OPTS} td-slda-nuclear-solver-read.f90 ;
  ftn -c -r8 liberf.f90 ;
  ftn -o xtldslda krp-init.o krp-rpt-init.o krp-rpt.o td-slda-nuclear-solver-
read.o liberf.o ${PAPI_POST_LINK_OPTS} ${FFTW_POST_LINK_OPTS} -lLUT -lsci -
lpthread -lm ;
```

#### *Unitary Code*

```
compile.slda-unitary-GPRA-PMM-bm :
  clear ;
  date ;
  echo "...compilation of kz-solver" ;
  time source compile.3d-kz-io ;
  echo "...compilation of time dependent unitary code" ;
  time source compile.td-slda-ug ;
  echo "...compilation of time dependent unitary restart code" ;
  time source compile.td-slda-ug-rs
```

**compile.3d-kz-io :**

```
echo "rm -f *.o xfort-usr *.mod" ;
rm -f *.o x* ;
echo "cc -c get_mem_req_blk_cyc.c" ;
cc -c get_mem_req_blk_cyc.c ;
echo "cc -c get_pwrk_spc.c";
cc -c get_pwrk_spc.c;
echo "cc -c init_wrk_bf.c ";
cc -c init_wrk_bf.c ;
echo "cc -c ${PAPI_INCLUDE_OPTS} krp-init.c" ;
cc -c ${PAPI_INCLUDE_OPTS} krp-init.c ;
echo "cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init.c" ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init.c ;
echo "cc -c ${PAPI_INCLUDE_OPTS} krp-rpt.c" ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt.c ;
echo "cc -c get_full_hfb.c" ;
cc -c get_full_hfb.c ;
echo "cc -c rsyeig.c" ;
cc -c rsyeig.c ;
echo "cc -c mk_nrm.c" ;
cc -c mk_nrm.c ;
echo "cc -c pmatrix-infnorm.c" ;
cc -c pmatrix-infnorm.c ;
echo "cc -c chk_err.c" ;
cc -c chk_err.c ;
echo "cc -c fvv.c" ;
cc -c fvv.c ;
echo "cc -c fuv.c" ;
cc -c fuv.c ;
echo "cc -c write_wf.c";
cc -c write_wf.c;
echo "cc -c write_ctrl_pot_c.c";
cc -c write_ctrl_pot_c.c;
echo "ftn -c paslda_init.f90" ;
ftn -c paslda_init.f90 ;
echo "...linking for build of binary : xkz-io-ug-GPRA-PMM" ;
echo "ftn -o xkz-io-ug-GPRA-PMM paslda_init.o krp-init.o krp-rpt-init.o
krp-rpt.o get_pwrk_spc.o init_wrk_bf.o get_mem_req_blk_cyc.o get_full_hfb.o
rsyeig.o mk_nrm.o chk_err.o fvv.o fuv.o pmatrix-infnorm.o write_wf.o
write_ctrl_pot_c.o ${PAPI_POST_LINK_OPTS} -lsci -lLUT -lpthread -lm" ;
ftn -o xkz-io-ug-GPRA-PMM paslda_init.o krp-init.o krp-rpt-init.o krp-
rpt.o get_pwrk_spc.o init_wrk_bf.o get_mem_req_blk_cyc.o get_full_hfb.o
rsyeig.o mk_nrm.o chk_err.o fvv.o fuv.o pmatrix-infnorm.o write_wf.o
write_ctrl_pot_c.o ${PAPI_POST_LINK_OPTS} -lsci -lLUT -lpthread -lm ;
```

**compile.td-slida-ug :**

```
echo "cc -c -DKMPT -DKRP -I/opt/xt-tools/papi/3.6.2.2/include -
I/opt/fftw/3.2.2.1/include 3d-td-slida-unitary-io.c" ;
cc -c -DKMPT -DKRP -I/opt/xt-tools/papi/3.6.2.2/include -
I/opt/fftw/3.2.2.1/include 3d-td-slida-unitary-io.c ;
echo "...linking for build of binary : xtddlida-ug-GPRA-PMM" ;
echo "cc -o xtddlida-ug-GPRA-PMM 3d-td-slida-unitary-io.o
${PAPI_POSTLINK_OPTS} ${FFTW_POST_LINK_OPTS} -lLUT -lpthread -lm";
cc -o xtddlida-ug-GPRA-PMM 3d-td-slida-unitary-io.o ${PAPI_POSTLINK_OPTS}
${FFTW_POST_LINK_OPTS} -lLUT -lpthread -lm ;
```

**compile.td-slida-ug-rs :**

```

    echo "cc -c -DKMPT -DRSTRT -DKRP -I/opt/xt-tools/papi/3.6.2.2/include -
I/opt/fftw/3.2.2.1/include 3d-td-sl-da-unitary-io.c" ;
    cc -c -DKMPT -DRSTRT -DKRP -I/opt/xt-tools/papi/3.6.2.2/include -
I/opt/fftw/3.2.2.1/include 3d-td-sl-da-unitary-io.c ;
    echo "...linking for build of restart binary : xtdsl-da-ug-GPRA-PMM-rs" ;
    echo "cc -o xtdsl-da-ug-GPRA-PMM-rs 3d-td-sl-da-unitary-io.o
${PAPI_POSTLINK_OPTS} ${FFTW_POST_LINK_OPTS} -lLUT -lpthread -lm";
    cc -o xtdsl-da-ug-GPRA-PMM-rs 3d-td-sl-da-unitary-io.o
${PAPI_POSTLINK_OPTS} ${FFTW_POST_LINK_OPTS} -lLUT -lpthread -lm

```

## B.2.2 Q4 COMPILATION

### *Nuclear code*

```

istet@jaguarpf-login2:/tmp/work/istet/td-run/U238_gs> cat
~/sl-da/v09/compile9
cc -c ${PAPI_INCLUDE_OPTS} krp-init.c ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init-sum.c ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init.c ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt.c ;
cc -c get-blcs-dscr.c ;
cc -c get-mem-req-blk-cyc.c ;
cc -c bc-fil-wr.c ;
cc -c bc-fil-wr-ftn.c
cc -c kr-cwr.c ;
cc -c bc-wr-lstr-scl.c ;
ftn -c -fastsse -Kieee -pc 64 -r8 ${FFTW_INCLUDE_OPTS} sl-da-nuclear-
solver_v095.f90 ;
ftn -c -fastsse -Kieee -pc 64 -r8 liberf.f90 ;
ftn -o xsl-da-gs-q4 -Kieee -r8 -pc 64 sl-da-nuclear-solver_v095.o bc-
wr-lstr-scl.o liberf.o kr-cwr.o get-blcs-dscr.o get-mem-req-blk-cyc.o
bc-fil-wr.o bc-fil-wr-ftn.o krp-init.o krp-rpt-init.o krp-rpt-init-
sum.o krp-rpt.o ${FFTW_POST_LINK_OPTS} ${PAPI_INCLUDE_OPTS} -lLUT -
lsci -lpthread -lm

```

### *Unitary code*

```

istet@jaguarpf-login2:/tmp/work/istet/td-run/U238_gs> cat
~/td/cmpl.td-der
cc -c f-betim.c ;
cc -c cprs-cases.c ;
cc -c fposio-wf.c ;
cc -c get-rnd-wf.c ;
cc -c ${PAPI_INCLUDE_OPTS} krp-init.c ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init.c ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init-sum.c ;
cc -c ${PAPI_INCLUDE_OPTS} krp-rpt.c ;
ftn -c liberf.f90 ;
ftn -c -C -I/opt/xt-tools/papi/3.7.2/v23/include -
I/opt/fftw/3.2.2.1/include td-sl-da-nuclear-solver-read-gp.f90 ;
ftn -o xtdsl-da td-sl-da-nuclear-solver-read-gp.o liberf.o krp-init.o
krp-rpt-init.o krp-rpt-init-sum.o krp-rpt.o f-betim.o get-rnd-wf.o

```

```
fposio-wf.o cprs-cases.o ${FFTW_POST_LINK_OPTS} -lsci -L/opt/xt-  
tools/papi/3.7.2/v23/lib -lpapi -lpfm -lLUT -lpthread -lm
```

## B.3 BATCH SCRIPTS

### B.3.1 Q2 BATCH SCRIPTS

#### qscr-csc053sld-nuc :

```
#!/bin/bash

#PBS -V
#PBS -l walltime=01:00:00,size=73728
#PBS -A csc053sld
#PBS -N slda-nuc-slv-GPRA-PMM
#PBS -j oe

cd ${PBS_O_WORKDIR}
time aprun -n 73728 ./xsllda-test
```

#### qscr-csc053sld-td-nuc :

```
#!/bin/bash

#PBS -V
#PBS -l walltime=06:00:00,size=16416
#PBS -A csc053sld
#PBS -N tdslda_nuc-GPRA-PMM
#PBS -j oe

cd ${PBS_O_WORKDIR}
time aprun -n 16414 ./xtdslda
```

#### qscr-csc053sld :

```
#PBS -V
#PBS -l walltime=04:00:00,size=7344
#PBS -A csc053sld
#PBS -N slda-ug-slv-GPRA-PMM-q2
#PBS -j oe
cd ${PBS_O_WORKDIR}
time aprun -n 7344 ./xkz-io-ug-GPRA-PMM
```

#### qscr-csc053sld-td :

```
#PBS -V
#PBS -l walltime=15:00:00,size=103920
#PBS -A csc053sld
#PBS -N slda-ug-td-GPRA-PMM-q2
#PBS -j oe
cd ${PBS_O_WORKDIR}
time aprun -n 103917 ./xtdslda-ug-GPRA-PMM-rs 50 50 100 103917
0.2338143545868483 4.5060815728113202 /tmp/work/roche/slda-io/run
/tmp/work/roche/slda-io/run/td-slda-dir 52000.
```

#### qscr-csc053sld-td-rs :

```

#PBS -V
#PBS -l walltime=8:00:00,size=103920
#PBS -A csc053sld
#PBS -N slda-ug-td-GPRA-PMM-q2
#PBS -j oe
cd ${PBS_O_WORKDIR}
time aprun -n 103917 ./xtdslda-ug-GPRA-PMM-rs 50 50 100 103917
0.2338143545868483 4.5060815728113202 /tmp/work/roche/slda-io/run
/tmp/work/roche/slda-io/run/td-slda-dir 26100.

```

### **B.3.2 Q4 BATCH SCRIPTS**

#### *Nuclear Code*

```

#!/bin/bash

#PBS -V
#PBS -l walltime=07:30:00,size=217800
#PBS -A csc053sld
#PBS -N slda-404064-U238_gs
#PBS -j oe

cd ${PBS_O_WORKDIR}
time aprun -n 217800 ./xslda-gs-q4

```

#### *Unitary Code*

```

istet@jaguarpf-login2:/tmp/work/istet/td-run/U238_gs> cat run-td.sc
#!/bin/bash

#PBS -V
#PBS -l walltime=2:30:00,size=136632
#PBS -A csc053sld
#PBS -N td-404064-GPRA-PMM
#PBS -j oe

cd ${PBS_O_WORKDIR}
time aprun -n 136628 ./xtdslda

```

## B.4 RUNTIME ENVIRONMENT

### B.4.1 Q2 RUNTIME ENVIRONMENT

```
roche@jaguarpf-login1:/tmp/work/roche/slida-io> module load xt-papi
liblut fftw
roche@jaguarpf-login1:/tmp/work/roche/slida-io> module list
Currently Loaded Modulefiles:
 1) modules/3.1.6
 2) DefApps
 3) torque/2.4.1b1-snap.200905191614
 4) moab/5.3.6
 5) /opt/cray/xt-asyncpe/default/modulefiles/xtpe-istanbul
 6) cray/MySQL/5.0.64-1.0000.2342.16.1
 7) xtpe-target-cn1
 8) xt-service/2.2.41A
 9) xt-os/2.2.41A
10) xt-boot/2.2.41A
11) xt-lustre-ss/2.2.41_1.6.5
12) cray/job/1.5.5-0.1_2.0202.18632.46.1
13) cray/csa/3.0.0-1_2.0202.18623.63.1
14) cray/account/1.0.0-2.0202.18612.42.3
15) cray/projdb/1.0.0-1.0202.18638.45.1
16) Base-opts/2.2.41A
17) pgi/9.0.4
18) xt-libsci/10.4.1
19) xt-mpt/3.5.1
20) xt-pe/2.2.41A
21) xt-asyncpe/3.5
22) PrgEnv-pgi/2.2.41A
23) xt-papi/3.6.2.2
24) liblut/0.9.9
25) fftw/3.2.2.1
roche@jaguarpf-login3:/tmp/work/roche/slida-io> lfs setstripe
/tmp/work/roche/slida-io/run -c 32 -s 2m -i -1
roche@jaguarpf-login3:/tmp/work/roche/slida-io> lfs setstripe
/tmp/work/roche/slida-io/run/td-slida-dir -c 64 -s 2m -i -1
roche@jaguarpf-login3:/tmp/work/roche/slida-io>
```

### B.4.2 Q4 RUNTIME ENVIRONMENT

```
Currently Loaded Modulefiles:
 1) modules/3.1.6
 2) DefApps
 3) torque/2.4.1b1-snap.200905191614
 4) moab/5.3.6
 5) /opt/cray/xt-asyncpe/default/modulefiles/xtpe-istanbul
 6) cray/MySQL/5.0.64-1.0000.2342.16.1
 7) xtpe-target-cn1
 8) xt-service/2.2.41A
 9) xt-os/2.2.41A
10) xt-boot/2.2.41A
```

- 11) xt-lustre-ss/2.2.41\_1.6.5
- 12) cray/job/1.5.5-0.1\_2.0202.18632.46.1
- 13) cray/csa/3.0.0-1\_2.0202.18623.63.1
- 14) cray/account/1.0.0-2.0202.18612.42.3
- 15) cray/projdb/1.0.0-1.0202.18638.45.1
- 16) Base-opts/2.2.41A
- 17) pgi/10.3.0
- 18) xt-libsci/10.4.4
- 19) pmi/1.0-1.0000.7628.10.2.ss
- 20) xt-mpt/4.0.0
- 21) xt-pe/2.2.41A
- 22) xt-asyncpe/3.7
- 23) PrgEnv-pgi/2.2.41A
- 24) fftw/3.1.1
- 25) xt-papi/3.7.2
- 26) liblut/0.9.9

**APPENDIX C. POP**

**C.1          INPUT SETTINGS**

## **C.2            COMPILATION**

POP was compiled using the default Portland Group compiler, version 9.0.4.

**C.3**

**BATCH SCRIPT**

## C.4 RUNTIME ENVIRONMENT

Currently Loaded Modulefiles:

- 1) modules/3.1.6
  - 2) DefApps
  - 3) torque/2.4.1b1-snap.200905191614
  - 4) moab/5.3.6
  - 5) /opt/cray/xt-asyncpe/default/modulefiles/xtpe-istanbul
  - 6) cray/MySQL/5.0.64-1.0000.2342.16.1
  - 7) xtpe-target-cn1
  - 8) xt-service/2.2.41A
  - 9) xt-os/2.2.41A
  - 10) xt-boot/2.2.41A
  - 11) xt-lustre-ss/2.2.41\_1.6.5
  - 12) cray/job/1.5.5-0.1\_2.0202.18632.46.1
  - 13) cray/csa/3.0.0-1\_2.0202.18623.63.1
  - 14) cray/account/1.0.0-2.0202.18612.42.3
  - 15) cray/projdb/1.0.0-1.0202.18638.45.1
  - 16) Base-opts/2.2.41A
  - 17) pgi/9.0.4
  - 18) xt-libsci/10.4.1
  - 19) xt-mpt/3.5.1
  - 20) xt-pe/2.2.41A
  - 21) xt-asyncpe/3.5
  - 22) PrgEnv-pgi/2.2.41A
  - 23) subversion/1.5.0
  - 24) nco/3.9.4
  - 25) ferret/6.1
  - 26) xt-papi/3.6.2.2
- setenv MPICH\_RANK\_REORDER\_METHOD 0

**APPENDIX D. LS3DF**

## D.1 INPUT SETTINGS

There are a few input files, including the atomic configuration file for the system (e.g., `xatom.config`), the atomic pseudopotential files `vwr.H`, `vwr.Zn.d`, `vwr.O`, `vwr.H0.5`, `vwr.H1.5`. There are also a few special files needed for fragment generation: `gen_ZnO_rod.input`, `H_O_O.group`, `H_O_Zn.group`, `H_O_Zn_Zn.group`. All these files are available on jaguar in the directory

```
/ccs/proj/nti009/linwang/GPRA-PMM_CODE/WORK.
```

Besides the above files, the main LS3DF input and control file is `LS3DF.input`. The Q2 and Q4 input files are included in `/ccs/proj/nti009/linwang/GPRA-PMM_CODE/WORK` as `LS3DF.input.Q2` and `LS3DF.input.Q4` respectively. They need to be copied to `LS3DF.input` before the run.

The input files for LS3DF are all nearly identical, except for the line that inputs the number of groups and the group sizes, so we reproduce one representative input file below, which uses 43,200 cores.

```
1  xatom.config, 1, 0, 1          ! the total f_xatom file,
iflag_mem_wave,iflag_mem_loc,iflag_report
2  108, 80 , 720                !num_group,num_proc_pergroup, nnodes_G(global
GENPOT)
3  18,6,6,18,6,6,1             !m1,m2,m3, mn1,mn2,mn3 (mn1,2,3 divide m1,2,3
for patching parallelization),ngroup(always 1)
4  40,50,50                    !nd1,nd2,nd3 (n1=nd1*m1,etc)
5  0.5 0.5 0.5                 ! add_vac1,2,3 (frag length in dir1=
(L1+2*add_vac1)*nd1, L1=1,2)
6  18 40                       ! nslice, multi (MUST ***nnodes_G=nslice*multi***)
(for gen_dvr,gen_vrF,get_denstot)
7  10, 0., 1                   ! ido_dVr (whether to calc dVr at the begin
of the calc)
8  2 10                        ! iter_init,niter:(iter=iter_init,iter_init+niter-
1),init pot: vr.in_tot."iter_init"
9  3                            ! Number of atoms for bond information (used to
assign passivation atoms)
10 30 4.570 4                  ! iatom, bond_length, num_neighbor(not used)
11 8 3.50 4                    ! Zn-O bond (3.7588)
12 1 1.257 1                   ! H-O bond (1.9029)
-----
1  1, 0                        ! islda, igga
2  50, 100, 100, 1.           ! Ecut,Ecut2,Ecut2L,Smth
3  0, 0., 0., 0.             ! icoul,xcoul(1),(2),(3), this is for fragment
potential only (if to be calculated)
4  1.D-6, 1.D-10             ! tolug, tolE
5  20, 4, 2, 0, 0.0, 0, 0    ! niter,nline,mCGbad (for the
first frag_PEtot iteration, iter=1)
3  0 0.2 1                    ! iCGmth,iscfmth,FermidE,itYPEFermi
3  10 0.2 1                   ! iCGmth,iscfmth,FermidE,itYPEFermi
3  10 0.2 1                   ! iCGmth,iscfmth,FermidE,itYPEFermi
3  10 0.2 1                   ! iCGmth,iscfmth,FermidE,itYPEFermi
3  10 0.2 1                   ! iCGmth,iscfmth,FermidE,itYPEFermi
3  10 0.2 1                   ! iCGmth,iscfmth,FermidE,itYPEFermi
```

```

3 10 0.2 1 ! iCGmth,iscfmth,FermidE,itYPEFermi
6 1, 4, 2, 1, -2.8 ! niter,nline,mCGbad (for subsequent
frag_PEtot iterations, iter > 1)
3 1 0.2 1 ! iCGmth,iscfmth,FermidE,itYPEFermi (always use
iscfmth=1 to have output dens)
7 3 ! ilocal
8 3.4 ! rcut
9 5 ! ntype
vwr.O 1
vwr.Zn.d 1
vwr.H0.5 1
vwr.H1.5 1
vwr.H 1
10 720 300 300 720 300 300 ! n1,n2,n3;n1L,n2L,n3L (n1 must be
nd1*m1, etc), for GENPOT
11 1, 0.0, 0.0, 0.0 ! icoul,xcoul(1),(2),(3), this is for
the whole system, for GENPOT
12 0, vext_file ! ivext, f_vext (external potential), for
GENPOT
13 0, symm.file ! isymm, symm.file, if isymm=1, read symm
info from symm.file, for GENPOT
14 24432. ! totNel (for the total system), for
GENPOT
*****
*****
** END OF INPUT
*****

```

```

cccccccccccccccccccccccccccccccccccccccc
cccc The total num of proc for calculatio: nnodes_all
cccc nnodes_all can be smaller or equal to num_group*num_proc_pergroup.
cccc But nnodes_all must be muple of num_proc_pergroup
cccc nnodes_all must be >= mn1*mn2*mn3 (that is why mn1,mn2,mn3 could be
smaller than m1*m2*m3)
cccc nnodes_all could be smaller than num_group
cccc nslice: smaller nslice (e.g,1), more efficient for
gen_dvr,gen_vrF,get_denstot, but more memory.
cccc multi is used to format the write out in the total density and
potential. nproc=nslice*multi in dens.file
cccc for iter_init=1, there is no input wavefunction
cccc for iter_init>1, must have input wavefunction in d_wg.odd or d_wg.even
(no iter index for wg name)
cccc for iter=odd(1,3,5..), input from d_wg.even, output d_wg.odd
cccc for iter=even(0,2,4..), input from d_wg.odd, output d_wg.even
cccc The output total potential and density are stored for all the
iterations.
cccc For each iter, it will generate fragment pot in d_vrF from total input
pot: vr.in_tot."iter".

```



```

3 10 0.2 1 ! iCGmth,iscfmth,FermiE,itypeFermi
6 1, 4, 2, 1, -2.8 ! niter,nline,mCGbad (for subsequent frag_PEtot
iterations, iter > 1)
3 1 0.2 1 ! iCGmth,iscfmth,FermiE,itypeFermi (always use iscfmth=1
to have output dens)
7 3 ! ilocal
8 3.4 ! rcut
9 5 ! ntype
vwr.O 1
vwr.Zn.d 1
vwr.H0.5 1
vwr.H1.5 1
vwr.H 1
10 720 300 300 720 300 300 ! n1,n2,n3;n1L,n2L,n3L (n1 must be nd1*m1, etc),
for GENPOT
11 1, 0.0, 0.0, 0.0 ! icoul,xcoul(1),(2),(3), this is for the whole
system, for GENPOT
12 0, vext_file ! ivext, f_vext (external potential), for GENPOT
13 0, symm.file ! isymm, symm.file, if isymm=1, read symm info from
symm.file, for GENPOT
14 24432. ! totNel (for the total system), for GENPOT
*****
****
** END OF INPUT
*****

cccccccccccccccccccccccccccccccccccc
cccc The total num of proc for calculatio: nnodes_all
cccc nnodes_all can be smaller or equal to num_group*num_proc_pergroup.
cccc But nnodes_all must be muple of num_proc_pergroup
cccc nnodes_all must be >= mn1*mn2*mn3 (that is why mn1,mn2,mn3 could be smaller than
m1*m2*m3)
cccc nnodes_all could be smaller than num_group
cccc nslice: smaller nslice (e.g,1), more efficient for gen_dvr,gen_vrF,get_denstot,
but more memory.
cccc multi is used to format the write out in the total density and potential.
nproc=nslice*multi in dens.file
cccc for iter_init=1, there is no input wavefunction
cccc for iter_init>1, must have input wavefunction in d_wg.odd or d_wg.even (no iter
index for wg name)
cccc for iter=odd(1,3,5..), input from d_wg.even, output d_wg.odd
cccc for iter=even(0,2,4..), input from d_wg.odd, output d_wg.even
cccc The output total potential and density are stored for all the iterations.
cccc For each iter, it will generate fragment pot in d_vrF from total input pot:
vr.in_tot."iter".
cccc To start the iteration, the first input total potential is vr.in_tot."iter_init"
(make sure it exists).
cccc If ido_dVr=1, it will generates (or over write) vr.in_tot.001 (= vr.atom_tot from
dens.atom_tot)
cccc For iter=1, it will calculate and store the noloc reference states (from
mainMV2_S) and vion_n.GENPOT.
cccc for iter=1, it will start from random wavefunc, and use the first set of
(niter,nline,mCGbad) in above

```

## D.2 COMPILATION

LS3DF uses the default Portland Group compiler, version 9.0.4. It uses the modules acml and xt-papi. The compiler option is `-fast`. The source codes are stored in (on the NCCS file system):

```
/ccs/proj/nti009/linwang/GPRA-PMM_CODE
```

The Q2 source code is in:

```
/ccs/proj/nti009/linwang/GPRA-PMM_CODE/LS3DF_Q2_SOURCE
```

To compile, follow the steps:

```
>module load acml
>module load xt-papi
>cc -c ${PAPI-INCLUDE_OPTS} krp-init.c
>cc -c ${PAPI-INCLUDE_OPTS} krp-rpt-init.c
>cc -c ${PAPI-INCLUDE_OPTS} krp-rpt.c
>make LS3DF
```

The final link will look like:

```
/opt/cray/xt-asyncpe/3.7/bin/ftn: INFO: linux target is being used
ftn -o LS3DF krp-init.o krp-rpt-init.o krp-rpt.o -L/opt/xt-
tools/papi/3.7.2/v23/lib -lpapi -lpfm LS3DF.o load_data.o fft_data.o
data.o MParallel.o IOdata.o IOmch.o input_head.o mainMV2_S2.o
GENPOT_V3_S.o Hpsi_comp.o Hpsi_comp_AllBand.o CG_comp.o CG_new.o
CG_AllBand.o getpot2.o getpot3.o getpot4.o getpot2L.o getpot3L.o
getpot4L.o getpot4_force.o getpot5.o getpot5L.o getpot5_force.o
GGAPBE.o d3fft_comp.o d3fft_comp_block.o cfft.o cfftd.o diag_comp.o
diag_comp_allband.o djacobi.o convert_SLvr.o d3fft_real2.o
d3fft_real2L.o d3fft_real2L2.o fftprep_comp.o fftprep_real2.o
fftprep_real2L.o fftprep_real2L2.o fwdcpfft_comp.o
fwdcpfft_comp_block.o fwdcpfft2.o fwdcpfft2L.o fwdcpfft2L2.o
invcpfft_comp.o invcpfft_comp_block.o invcpfft2.o invcpfft2L.o
invcpfft2L2.o gen_G_comp.o gen_G2_real.o gen_G2L_real.o
gen_G2L2_real.o global_maxi.o global_sumr.o global_sumc.o fmin.o
heapsort.o inputSP.o input_GENPOT.o init_ugSP.o init_ugSP00.o
init_ugSP001.o gaussj.o UxcCA.o UxcCA2.o w_line.o getewald.o
getwmask.o getwmaskX.o getwmask_q.o add_rho_beta.o getVrho.o
mch_pulayF.o mch_kerkF.o mch_pulay_GENPOT.o mch_kerk_GENPOT.o
Thomas3.o getNLsign.o getwq.o atomMV.o Etotcalc_V.o rhoIO_dir.o
rhoIO.o ugIO.o wqIO.o beta_psiIO.o get_ALI.o occup.o gen_Gstar_ind.o
symmop.o symmopf.o symmcheck.o forcLC.o forcNLq.o forcNLr.o ran1.o
dens_out.o densWr_out.o readusp_head.o w_line_vvr.o w_line_usp.o
LegendreSP.o clebsch_gordan.o get_Dij.o system_orth_comp.o
system_ccfft.o system_csfft.o system_scfft.o system_czheev.o
system_flush.o getwmask_dq.o get_VdqR.o getvcoul.o convert_2LtoL.o
getV_Hartree0.o getV_Hartree.o getewald3D.o getewald2D.o forcLC2.o
getEextV.o getrho_only.o write_wgSP.o gen_vrF_fmPN_NEW.o
get_denstot_fmPN_NEW.o gen_dvr_fmPN_S.o frag2cubicN_S.o
ave_dvr_cubicN_S.o getVrhoL_GENPOT.o system_time.o
orthogonal_cholesky.o orthogonal_cholesky2.o projection_shift.o
orthogonal_projection.o
/opt/cray/xt-asyncpe/3.7/bin/ftn: INFO: linux target is being used
```

One can rename the executable LS3DF as LS3DF.Q2

```
>mv LS3DF LS3DF.Q2
```

The Q4 source code is in:

```
/ccs/proj/nti009/linwang/GPRA-PMM_CODE/LS3DF_Q4_SOURCE
```

To compile, follow the steps:

```
>module load acml
>module load xt-papi
>cc -c ${PAPI_INCLUDE_OPTS} krp-init.c
>cc -c ${PAPI_INCLUDE_OPTS} krp-rpt-init.c
>cc -c ${PAPI_INCLUDE_OPTS} krp-rpt.c
>make LS3DF_BP
```

The final link step will look like:

```
/opt/cray/xt-asyncpe/3.7/bin/ftn: INFO: linux target is being used
ftn -o LS3DF_BP krp-init.o krp-rpt-init.o krp-rpt.o -L/opt/xt-
tools/papi/3.7.2/v23/lib -lpapi -lpfm LS3DF.o load_data.o fft_data.o
data.o MParallel.o IOdata.o IOmch.o input_head.o mainMV2_S2.o
GENPOT_V3_S.o Hpsi_comp.o Hpsi_comp_AllBand.o Hpsi_comp_AllBandBP.o
CG_comp.o CG_AllBand.o DIIS_comp.o orth_comp_DIIS.o getpot2.o
getpot3.o getpot4.o getpot2L.o getpot3L.o getpot4L.o getpot4_force.o
getpot5.o getpot5L.o getpot5_force.o GGAPBE.o d3fft_comp.o
d3fft_comp_block.o diag_comp.o djacobi.o convert_SLvr.o d3fft_real2.o
d3fft_real2L.o d3fft_real2L2.o fftprep_comp.o fftprep_real2.o
fftprep_real2L.o fftprep_real2L2.o fwdcpfft_comp.o
fwdcpfft_comp_block.o fwdcpfft2.o fwdcpfft2L.o fwdcpfft2L2.o
invcpfft_comp.o invcpfft_comp_block.o invcpfft2.o invcpfft2L.o
invcpfft2L2.o gen_G_comp.o gen_G2_real.o gen_G2L_real.o
gen_G2L2_real.o global_maxi.o global_sumr.o global_sumc.o fmin.o
heapsort.o inputSP.o input_GENPOT.o init_ugSP.o init_ugSP001.o
init_ugSP00.o gaussj.o UxcCA.o UxcCA2.o w_line.o getewald.o getwmask.o
getwmaskX.o getwmask_q.o add_rho_beta.o getVrho.o mch_pulayF.o
mch_kerk_GENPOT.o mch_kerkF.o Thomas3.o getNLsign.o getwq.o atomMV.o
Etotcalc_V.o rhoIO_dir.o rhoIO.o ugIO.o ugIOBP.o wqIO.o beta_psiIO.o
get_ALI.o occup.o gen_Gstar_ind.o symmopf.o symmopf.o symmcheck.o
forcLC.o forcNLq.o forcNLr.o ranl.o dens_out.o densWr_out.o
readusp_head.o w_line_vwr.o w_line_usp.o LegendreSP.o clebsch_gordan.o
get_Dij.o getwmask_dq.o get_Vdqdr.o getvcoul.o convert_2LtoL.o
getV_Hartree0.o getV_Hartree.o getewald3D.o getewald2D.o forcLC2.o
getEextV.o getrho_only.o write_wgSP.o gen_vrF_fmPN_NEW.o
gen_vrF_fmPN_fix.o get_denstot_fmPN_NEW.o get_denstot_fmPN_fix.o
gen_dvr_fmPN_S.o frag2cubicN_S.o ave_dvr_cubicN_S.o getVrhoL_GENPOT.o
mch_pulay_GENPOT.o system_ccfft.o system_csfft.o
system_czheev.scalapack.o system_flush.o system_orth_comp.o
system_scfft.o system_time.o orthogonal_cholesky.o
orthogonal_cholesky2.o orthogonal_choleskyBP.o orthogonal_projection.o
```

```
orthogonal_projectionBP.o projection_shift.o rotate_wfBP.o  
dot_product_BP.o  
/opt/cray/xt-asyncpe/3.7/bin/ftn: INFO: linux target is being used
```

One can rename the executable LS3DF\_BP as LS3DF.Q4  
>mv LS3DF\_BP LS3DF.Q4

### D.3 BATCH SCRIPT

The working directory in which to run the job is  
`/ccs/proj/nti009/linwang/GPRA-PMM_CODE/WORK`

There is a `README_TO_RUN` file that provides brief instructions for running the Q2 and Q4 tests. One first needs to copy the directory to a scratch directory, such as `/tmp/work/linwang/WORK`. One also needs to copy the above compiled executables `LS3DF.Q2` and `LS3DF.Q4` to this directory.

Then to run Q2 job, one can do:

```
>cp LS3DF.input.Q2 LS3DF.input
>mkdir_all
>gen_ZnO_rod.r
>qsub job.LS3DF.Q2
```

The batch scripts for the Q2 runs differ only in number of processors and walltime requested, so we reproduce one representative script below.

```
#!/bin/csh
#PBS -V
#PBS -A csc0533df
#PBS -N ZnO
#PBS -j oe
#PBS -l walltime=06:00:00,size=43200
```

```
cd $PBS_O_WORKDIR
time aprun -n43200 ./LS3DF.Q2
```

For Q4 runs, in the same directory, one can do the following:

```
>cp LS3DF.input.Q4 LS3DF.input
>mkdir_all
>gen_ZnO_rodNEW.r
>qsub job.LS3DF.Q4
```

Likewise, the batch scripts for the Q4 runs differ only in number of processors and walltime requested. The batch script `job.LS3DF.Q4` looks like:

```
#!/bin/csh
#PBS -V
#PBS -A csc0533df
#PBS -N ZnO
#PBS -j oe
#PBS -l walltime=06:00:00,size=86400
```

```
cd $PBS_O_WORKDIR
time aprun -n86400 ./LS3DF.Q4
```

## D.4 RUNTIME ENVIRONMENT

No special run time environment is needed; the default runtime environment on Jaguarpf is used. As described above, one can submit the job simply by invoking: `>qsub job.LS3DF.Q2`, or `>qsub job.LS3DF.Q4`.

Results from one Q2 and one Q4 run are stored in `/ccs/proj/nti009/linwang/GPRA-PMM_CODE/Q2_Q4_results`.

The Q2 run uses 43200 cores on Jaguarpf. The Q2 results are in the following files:

`EtotN.report.Q2.43200` (the energy and potential converged result);

`LS3DF.report.Q2.43200` (the LS3DF report file for times on each steps); and

`LS3DF.count.Q2.43200` (the krp standard output result for floating point operation counts). For both Q2 and Q4 runs, there are 20 initial iterations for self-consistent calculations on each fragment, followed by 40 iterations of self-consistent runs for the global system. As reported in `LS3DF.report.Q2.43200`, the total Q2 run time is 3.87 hours.

The reported results for the final two self-consistent iterations in `EtotN.report.Q2.43200` are:

```
-----
NSCF                =                40
Ewald               = 0.80444822217138E+07
Ealphat            = 0.000000000000000E+00
E_kin+E_noloc      = -.11790709236651E+04
E_Hxc               = 0.81169415878854E+07
E_ion              = -.16256258085613E+08
E_dvrho, |E_dvrho| = -.23302191424857E+02      0.557E+05
E_tot, dE_tot      = -.96036649129028E+05      -0.652E-08
-----
V_error =0.263731E-06
-----
-----
NSCF                =                41
Ewald               = 0.80444822217138E+07
Ealphat            = 0.000000000000000E+00
E_kin+E_noloc      = -.11790709382839E+04
E_Hxc               = 0.81169415877122E+07
E_ion              = -.16256258085426E+08
E_dvrho, |E_dvrho| = -.23302190753184E+02      0.557E+05
E_tot, dE_tot      = -.96036649129033E+05      -0.559E-08
-----
V_error =0.262371E-06
```

The reported results for one of the self-consistent iterations for the global system in `LS3DF.count.Q2.43200` are:

```
-----
PAPI_TOT_INS : Tot[ 10972641 ] Rt[ 86483 ]
PAPI_FP_INS  : Tot[ 6 ]      Rt[ 6 ]
PAPI_L2_DCM  : Tot[ 321593 ] Rt[ 107 ]
PAPI_real_cyc = 1458307 PAPI_real_usec = 561
PAPI_user_cyc = 0       PAPI_user_usec = 0
```

```
(below, for GEN_VF)
PAPI_TOT_INS : Tot[ 927621961982903 ] Rt[ 20667955727 ]
PAPI_FP_INS : Tot[ 1885996802 ] Rt[ 43291 ]
PAPI_L2_DCM : Tot[ 17272255230 ] Rt[ 864136 ]
PAPI_real_cyc = 14376838436 PAPI_real_usec = 5529554
PAPI_user_cyc = 14300000000 PAPI_user_usec = 5500000
```

```
(below, for PEtot_F)
PAPI_TOT_INS : Tot[ 24551708285257680 ] Rt[
558017490394 ]
PAPI_FP_INS : Tot[ 4416612394295743 ] Rt[ 118340026316 ]
PAPI_L2_DCM : Tot[ 10307760119460 ] Rt[ 257186392 ]
PAPI_real_cyc = 464872764009 PAPI_real_usec = 178797217
PAPI_user_cyc = 410306000000 PAPI_user_usec = 157810000
```

```
(below, for GEN_DENS)
PAPI_TOT_INS : Tot[ 917321716685209 ] Rt[ 17152778599 ]
PAPI_FP_INS : Tot[ 2806878555950 ] Rt[ 258106169 ]
PAPI_L2_DCM : Tot[ 245129321585 ] Rt[ 13304738 ]
PAPI_real_cyc = 21597013149 PAPI_real_usec = 8306543
PAPI_user_cyc = 20644000000 PAPI_user_usec = 7940000
```

```
(below, for POISSON)
PAPI_TOT_INS : Tot[ 3027974496647585 ] Rt[ 64918997726 ]
PAPI_FP_INS : Tot[ 60595536148 ] Rt[ 84135313 ]
PAPI_L2_DCM : Tot[ 62322321277 ] Rt[ 3197937 ]
PAPI_real_cyc = 45267776644 PAPI_real_usec = 17410683
PAPI_user_cyc = 44226000000 PAPI_user_usec = 17010000
```

-----

The reported results for a representative self-consistent iteration (the 30<sup>th</sup>) for the global system in LS3DF.report.Q2.43200 are (the numbers are in seconds):

```
-----
30          enter gen_vrF_fmPN_S
30      5.54  11498.24      out gen_vrF_fmPN_S
30          enter mainMV2_S
30     177.72  11676.05      out mainMV2_S
30          enter get_denstot_fmPN_S
30      8.34  11684.44      out get_denstot_fmPN_S
30          enter GENPOT_V3_S
30     17.88  11702.52      out GENPOT_V3_S
-----
```

The following are for the Q4 run results on 86,400 cores. The results are stored in /ccs/proj/nti009/linwang/GPRA-PMM\_CODE/Q2\_Q4\_results as: EtotN.report.Q4.86400, LS3DF.report.Q4.86400, and LS3DF.count.Q4.86400. As reported in LS3DF.report.Q4.86400, the total run time is 1.48 hours.

The reported results for the final two self-consistent iterations in EtotN.report.Q4.86400 are:

```

-----
NSCF                =                40
Ewald               = 0.80444822217138E+07
Ealphat            = 0.000000000000000E+00
E_kin+E_noloc      = -.11794352468133E+04
E_Hxc              = 0.81169315470978E+07
E_ion              = -.16256247983043E+08
E_dvrho, |E_dvrho| = -.88141925787295E+00      0.550E+05
E_tot, dE_tot      = -.96014530896990E+05      0.866E-07
-----

```

V\_error =0.801700E-07

```

-----
NSCF                =                41
Ewald               = 0.80444822217138E+07
Ealphat            = 0.000000000000000E+00
E_kin+E_noloc      = -.11794352449335E+04
E_Hxc              = 0.81169315471252E+07
E_ion              = -.16256247983072E+08
E_dvrho, |E_dvrho| = -.88141867397644E+00      0.550E+05
E_tot, dE_tot      = -.96014530896978E+05      0.112E-07
-----

```

V\_error =0.788565E-07

The reported results for one of the self-consistent iterations for the global system in LS3DF.count.Q4.86400 are:

```

-----
PAPI_TOT_INS : Tot[ 21857399 ] Rt[ 84838 ]
PAPI_FP_INS  : Tot[ 6 ]      Rt[ 6 ]
PAPI_L2_DCM  : Tot[ 737020 ] Rt[ 107 ]
PAPI_real_cyc = 1416477 PAPI_real_usec = 545
PAPI_user_cyc = 0        PAPI_user_usec = 0

```

(below, for GEN\_VF)

```

PAPI_TOT_INS : Tot[ 2070404346252707 ] Rt[ 23760166563 ]
PAPI_FP_INS  : Tot[ 3769464002 ]      Rt[ 13887 ]
PAPI_L2_DCM  : Tot[ 307003900006 ]     Rt[ 2477030 ]
PAPI_real_cyc = 16184107996      PAPI_real_usec = 6224657
PAPI_user_cyc = 16198000000      PAPI_user_usec = 6230000

```

(below, for PEtot\_F)

```

PAPI_TOT_INS : Tot[ 18238909877999718 ]      Rt[
243590522988 ]
PAPI_FP_INS  : Tot[ 3306262880906291 ] Rt[ 78876429185 ]
PAPI_L2_DCM  : Tot[ 8418141926934 ]      Rt[ 129904503 ]
PAPI_real_cyc = 157781836853      PAPI_real_usec = 60685322
PAPI_user_cyc = 149162000000      PAPI_user_usec = 57370000

```

```

(below, for GEN_DENS)
PAPI_TOT_INS : Tot[ 2022716253054937 ] Rt[ 17685550182 ]
PAPI_FP_INS : Tot[ 5585681681678 ] Rt[ 257986509 ]
PAPI_L2_DCM : Tot[ 720479294394 ] Rt[ 13145287 ]
PAPI_real_cyc = 21407074107 PAPI_real_usec = 8233490
PAPI_user_cyc = 20410000000 PAPI_user_usec = 7850000

```

```

(below, for POISSON)
PAPI_TOT_INS : Tot[ 5644192881686333 ] Rt[ 57832352899 ]
PAPI_FP_INS : Tot[ 61633253666 ] Rt[ 85576359 ]
PAPI_L2_DCM : Tot[ 789419086428 ] Rt[ 3273336 ]
PAPI_real_cyc = 41531081185 PAPI_real_usec = 15973493
PAPI_user_cyc = 39000000000 PAPI_user_usec = 15000000

```

-----

The reported results for a representative self-consistent iteration (the 30<sup>th</sup>) for the global system in LS3DF.report.Q4.86400 are (the numbers are in seconds):

```

-----
30          enter gen_vrF_fmPN_S
30      6.14  4227.59      out gen_vrF_fmPN_S
30          enter mainMV2_S
30      62.36  4289.95      out mainMV2_S
30          enter get_denstot_fmPN_S
30      8.17  4298.13      out get_denstot_fmPN_S
30          enter GENPOT_V3_S
30      15.50  4313.64      out GENPOT_V3_S
-----

```

**APPENDIX E.      DENOVO**

## E.1 INPUT SETTINGS

The mesh and problem input for the Denovo GPRA-PMM PWR-900 benchmark problem was generated on orthanc.ornl.gov (LINUX x86\_64 cluster) using Denovo's python tools. The process for generating input for Denovo on Jaguar can be summarized as follows

1. Generate mesh and cross sections using ADVANTG or MAVRIC (SCALE)
2. Use the pykba toolset to generate binary input files for Jaguar.
3. Port the binary inputs to Jaguar using sftp through dtn01.ccs.ornl.gov.
4. Use Denovo's hpckba HPC executable front-end to run the problem on Jaguar.

The use of SCALE is well documented elsewhere (for example, see *SCALE: A Modular Code System for Performing Standardized Computer Analyses for Licensing Evaluation*, ORNL/TM-2005/39, Version 6, Vols I-III, January 2009).

For the PWR-900 problem, we generated mesh and cross-sections from a KENO IV model of the PWR core using the SCALE MAVRIC sequence. The following python script was used to develop binary inputs for the hpckba front-end:

**setup\_ms.py:**

```
## setup.py
## 9te [orthanc]
## Wed Jan 13 22:59:09 2010
#####
##
## Copyright (C) 2008 Oak Ridge National Laboratory, UT-Battelle, LLC.
##-----
##
## generated by /home/9te/work/build/opt/bin/pygen built on 20100113
#####
##

import os, sys, math, string

# pykba equation type
from sc import *

##-----
##
## MAIN
##-----
##

initialize(sys.argv)

if node() == 0:
    print "Denovo - pykba Python Front-End"
    print "-----"
    print "Release      : %16s" % (release())
    print "Release Date : %16s" % (release_date())
    print "Build Date   : %16s" % (build_date())
    print

timer = Timer()
timer.start()

##-----
##
## DB
```

```

##-----
##

reader = Mavric_Parser("edf_578x700.dnv")
#reader = Mavric_Parser("edf_289x14.dnv")

db = DB("pykba")

reader.read_db(db)

# problem type
db.insert("problem_type", "EIGENVALUE")
db.insert("problem_name", "edf_44grp")

# high-level solver tolerance
db.insert("tolerance", 1.0e-3)

# solver descriptions
db.insert("eigen_solver", "power_iteration")
db.insert("mg_solver", "krylov")

# within-group solver
db.insert("within_group_solver", "GMRES_R")

# upscatter options
db.add_db("upscatter_db", "upscatter")
db.insert("upscatter_db", "tolerance", 1.0e-3)
db.insert("upscatter_db", "aztec_output", 1)

# eigenvalue tolerance
db.add_db("eigenvalue_db", "eigenvalue")
db.insert("eigenvalue_db", "k_tolerance", 1.0e-3)
db.insert("eigenvalue_db", "max_tolerance", 0.1)
db.insert("eigenvalue_db", "diagnostic_level", 2)

# data
db.insert("downscatter", 0, 1)
db.insert("Pn_order", 0)
db.insert("num_groups", 44)

# boundary conditions
db.insert("boundary", "vacuum")

# quadrature
db.add_db("quadrature_db", "quadrature")
db.insert("quadrature_db", "Sn_order", 12)

# add partitioning info
db.insert("num_sets", 1)
db.insert("partition_upscatter", 0, 1)

# decomposition (only for setup purposes)
db.insert("num_z_blocks", 1)
db.insert("num_blocks_i", 1)
db.insert("num_blocks_j", 1)

##-----
##

```

```

## MANAGER
##-----
##

# make manager, material, and angles
manager = Manager()
mat      = Mat()
angles   = Angles()

# partition the problem
manager.partition(db, mat, angles)

# get mapping and mesh objects
mapp     = manager.get_map()
indexer  = manager.get_indexer()
mesh     = manager.get_mesh()

# global and local cell numbers
Gx = indexer.num_global(X)
Gy = indexer.num_global(Y)
Gz = mesh.num_cells(Z)
Nx = mesh.num_cells(X)
Ny = mesh.num_cells(Y)
Nz = mesh.num_cells(Z)

if node() == 0:
    print ">>> Partitioned global mesh with %i x %i x %i cells" \
          % (Gx, Gy, Gz)

# print out the input database
# db.output()

##-----
##
## PARALLEL I/O
##-----
##

out = HPC_Problem_Output(500, Gx, Gy, Gz)
out.open("pwr900")

out.write_db(db)

##-----
##
## READ GIP FOR FORMALITY

gip = GIP(44)
reader.read_gip(gip)
del gip

##-----
##
## MATERIAL SETUP
##-----
##
## Material map:
## 1-45 --> Low Enrichment

```

```

## 46-90 --> Med Enrichment
## 47-135 --> High Enrichment

# read AMPX files
low = AMPX()
med = AMPX()
high = AMPX()

low.read_AMPX("fulllib/hmogmacrolib_low_44.bin")
med.read_AMPX("fulllib/hmogmacrolib_mid_44.bin")
high.read_AMPX("fulllib/hmogmacrolib_high_44.bin")

# set number of materials
mat.set_num(136)

# set material 0 to zero
mat.assign_zero(0)

# assign AMPX materials
for m in xrange(1, 46):
    mat.assign_ampx(m, m - 1, low, 1)
    mat.assign_ampx(m + 45, m - 1, med, 1)
    mat.assign_ampx(m + 90, m - 1, high, 1)

## write cross sections
out.write_xs(mat)

block = Upscatter_Matrix(mat)

print "Number of Upscatter Groups = %i " % (block.num_groups())

# material ids
matids = Vec_Int(out.chunk())
out.start_field_loop()
while not out.finished_field_loop():
    k = out.current_chunk()
    if k < Nz:
        reader.read_chunk_matids(matids)
        out.write_matids(matids)
        out.advance_loop()

reader.close()

##-----
##
## SOURCE SETUP
##-----
##

# allocate problem state (use a zero source)
shapes = Vec_Dbl()
out.write_src_info(ZERO_SOURCE, 0, shapes)

out.close()

##-----
##
## TIMING

```

```

##-----
##

timer.stop()
time = timer.wall_clock()

keys = timer_keys()
if len(keys) > 0 and node() == 0:
    print "\n"
    print "TIMING : Problem ran in %16.6e seconds." % (time)
    print "-----"
    for key in keys:
        print "%30s : %16.6e" % (key, timer_value(key) / time)
    print "-----"

##-----
##

```

```

manager.close()
finalize()

#####
##
## end of setup.py
#####
##

```

This script used the pykba Denovo-python front-end script that automatically loads the Denovo bindings into Python. It is run using

```
> pykba setup_ms.py
```

The files

```
hmogmacrolib_low_44.bin
hmogmacrolib_low_44.bin
hmogmacrolib_low_44.bin
```

and

```
edf_578x700.dnv
```

are the cross sections and mesh/material input generated by MAVRIC, respectively.

Denovo's HPC front-end executable, hpckba, requires two sets of inputs: the binary input files described above and a text input file that allows the setting of certain runtime parameters and I/O instructions. The basic input file for all of the GPRA-PMM runs had the following format:

```

pwr900.in:
eq_set: sc
hpc_input: pwr900
num_blocks_i: 102
num_blocks_j: 100
num_z_blocks: 10
num_sets: 11
Pn_order: 0
silo_off: 1

```

With these files in place, the executable is run as follows:

```
> aprun -n 112200 ./hpckba -I pwr900.in
```

Other than changing parallel decompositions and solver options that have been documented in Section 3.4.9, this process was used to run all of the GPRA-PMM problems.

## E.2 COMPILATION

Denovo uses a standard autoconf/make system for compiling/installing. On Jaguar the following modules were loaded at compile time:

Currently Loaded Modulefiles:

- 1) modules/3.1.6
- 2) DefApps
- 3) torque/2.4.1b1-snap.200905191614
- 4) moab/5.3.6
- 5) /opt/cray/xt-asyncpe/default/modulefiles/xtpe-istanbul
- 6) cray/MySQL/5.0.64-1.0000.2342.16.1
- 7) xtpe-target-cn1
- 8) xt-service/2.2.41A
- 9) xt-os/2.2.41A
- 10) xt-boot/2.2.41A
- 11) xt-lustre-ss/2.2.41\_1.6.5
- 12) cray/job/1.5.5-0.1\_2.0202.18632.46.1
- 13) cray/csa/3.0.0-1\_2.0202.18623.63.1
- 14) cray/account/1.0.0-2.0202.18612.42.3
- 15) cray/projdb/1.0.0-1.0202.18638.45.1
- 16) Base-opts/2.2.41A
- 17) pgi/10.3.0
- 18) xt-libsci/10.4.4
- 19) pmi/1.0-1.0000.7628.10.2.ss
- 20) xt-mpt/4.0.0
- 21) xt-pe/2.2.41A
- 22) xt-asyncpe/3.7
- 23) PrgEnv-pgi/2.2.41A

Denovo used the PGI compiler (10.3.0) through Jaguar's CC, cc, and ftn compiler scripts. The build script was as follows:

**build Denovo:**

```
#!/bin/sh
```

```
target='denovo-20100728'
```

```
VENDORS=/ccs/proj/nfi004/vendors
```

```
GSL=$VENDORS/gsl
```

```
TRILINOS=$VENDORS/trilinos
```

```
HDF5=$VENDORS/hdf5
```

```
SILO=$VENDORS/silo
```

```
SPRNG=$VENDORS/sprng
```

```
PAPI=/opt/xt-tools/papi/3.7.2/v23
```

```
LAPACK=/opt/xt-libsci/10.4.2/pgi
```

```
cd /ccs/proj/nfi004/denovo/$target/build
```

```
/ccs/proj/nfi004/denovo/$target/denovo/configure --
```

```
prefix=/ccs/proj/nfi004/denovo/$target --with-lapack="-lsci_istanbul" -
```

```
--with-lapack-lib=$LAPACK/lib --with-gsl-dir=$GSL --with-trilinos-
```

```
dir=$TRILINOS --with-mpi=mpich --with-mpi-dir=$MPICH_DIR --with-hdf5-
```

```
dir=$HDF5 --with-silo-dir=$SILO --with-cxx=pgi --with-comm=mpi --with-
```

```
opt=3 --with-dbc=0 --enable-krp-papi --with-papi-dir=$PAPI --enable-mc
```

```
--with-sprng-dir=$SPRNG --disable-strict-ansi --with-cxxflags=--  
diag_suppress=1396  
make nj=12 CXX=CC CC=cc F90=ftn
```

We used versions of GSL, Trilinos, HDF5/SILO, and SPRNG that we built and maintained. The versions were:

- GSL-1.9
- HDF5-1.8.2
- SILO-4.7
- Trilinos-10.2.0

### E.3 BATCH SCRIPT

The basic runtime script for our runs was:

**run\_pwr900:**

```
#!/bin/bash
```

```
#PBS -A csc053den
```

```
#PBS -l size=112200
```

```
#PBS -l walltime=1:00:00
```

```
#PBS -o pwr900.out
```

```
#PBS -e pwr900.error
```

```
#PBS -N pwr900
```

```
#PBS -m a
```

```
#PBS -m b
```

```
#PBS -m e
```

```
#PBS -M evanstm@ornl.gov
```

```
cd $PBS_O_WORKDIR
```

```
export MPICH_PTL_MATCH_OFF=1
```

```
date
```

```
time aprun -n 112200 ./hpckba -i pwr900.in
```

```
This script was queued using
```

```
> qsub run_pwr900
```

## E.4 RUNTIME ENVIRONMENT

The runtime environment was listed in Section E.2. For completeness, it is repeated here:  
Currently Loaded Modulefiles:

- 1) modules/3.1.6
- 2) DefApps
- 3) torque/2.4.1b1-snap.200905191614
- 4) moab/5.3.6
- 5) /opt/cray/xt-asyncpe/default/modulefiles/xtpe-istanbul
- 6) cray/MySQL/5.0.64-1.0000.2342.16.1
- 7) xtpe-target-cn1
- 8) xt-service/2.2.41A
- 9) xt-os/2.2.41A
- 10) xt-boot/2.2.41A
- 11) xt-lustre-ss/2.2.41\_1.6.5
- 12) cray/job/1.5.5-0.1\_2.0202.18632.46.1
- 13) cray/csa/3.0.0-1\_2.0202.18623.63.1
- 14) cray/account/1.0.0-2.0202.18612.42.3
- 15) cray/projdb/1.0.0-1.0202.18638.45.1
- 16) Base-opts/2.2.41A
- 17) pgi/10.3.0
- 18) xt-libsci/10.4.4
- 19) pmi/1.0-1.0000.7628.10.2.ss
- 20) xt-mpt/4.0.0
- 21) xt-pe/2.2.41A
- 22) xt-asyncpe/3.7
- 23) PrgEnv-pgi/2.2.41A