The Super Instruction Architecture

A Block-Oriented Language and Runtime System for Tensor Algebra with Very Large Arrays

Beverly A Sanders, Erik Deumens, Victor Lotrich, and Nakul Jindal

UF UNIVERSITY of FLORIDA

Motivating Domain: Computational Chemistry

- Electronic structure calculations (coupled cluster)
  - Dominated by tensor algebra using very large, dense multi-dimensional arrays
  - Irregular access patterns
  - Complex algorithms--need abstraction level that supports experimentation with algorithms
- ACES III
  - www.qtp.ufl.edu/ACES

Problem characteristics

- Data Requirements for CCSD
  - N = number of electrons
  - T amplitudes array: 4-index array of size n^2N^2
    - Need 2-10 copies
    - typical values N = 100, n=1000: 80GB
    - 3 need rapid access and are usually stored in RAM, others on disk
  - Additional arrays for integrals, up to 800GB

Architecture

- Domain specific programming language
  - Super instruction assembly language (SIAL)
  - scripting language to orchestrate parallelism and data movement
- Runtime system
  - Super instruction Processor (SIP)
  - interprets SIAL bytecode
  - manages parallelism
  - distributed data structures
  - I/O
- Super instructions
  - single node computational kernels
  - written in general purpose programming language

Super Instructions and Super Numbers

- Traditional programming languages
  - unit of data: floating point number
  - operations: combine floating point numbers
  - but operations and data must be aggregated for good performance
- SIA
  - unit of data: super number (block) of floating point numbers
  - operations: super instructions combine blocks
  - algorithms in SIAL are expressed in terms of blocks and super instructions

Why a new language?

- Domain specific language
  - expressiveness
    - describing algorithms in terms of super instructions and blocks
      - A(I,J) = B(I,K) * C(K,J)
      - AT(I,J) = A(J,I)
  - enforces abstractions

- "Scripting" language
  - simple compiler
  - language can be (and has been) easily extended
  - exploit programming language technology
    - eclipse-based IDE
    - static analyses and refactoring support
    - generation of performance models

- SIA architecture still takes advantage of highly optimizing compilers for super instruction implementation

Example: tensor contraction

$$R_{ij}^{\mu\nu} = \sum_{\lambda\sigma} V_{\lambda\sigma}^{\mu\nu} T_{ij}^{\lambda\sigma}$$

$\boldsymbol{F}$

Example: blocked version

$$R_{ij}^{\mu\nu} = \sum_{\lambda\sigma} V_{ij}^{\mu\nu} T_{ij}^{\lambda\sigma}$$

$$R(M,N,I,J)_{ij}^{\mu\nu} = \sum_{LS} \sum_{\lambda \in L} \sum_{\sigma \in S} V(M,N,L,S)_{\lambda\sigma}^{\mu\nu} T(L,S,I,J)_{ij}^{\lambda\sigma}$$

- *M,N,L,S,I,J* index segments of size *seg*
- Each block *R(M,N,I,J)* is a 4-index array of *seg*^{4} elements

Example: contraction super instruction

$$R_{ij}^{\mu\nu} = \sum_{\lambda\sigma} V_{ij}^{\mu\nu} T_{ij}^{\lambda\sigma}$$

$$R(M,N,I,J)_{ij}^{\mu\nu} = \sum_{LS} \sum_{\lambda\in L} \sum_{\sigma\in S} V(M,N,L,S)_{\lambda\sigma}^{\mu\nu} T(L,S,I,J)_{ij}^{\lambda\sigma}$$

$$R(M,N,I,J)_{ij}^{\mu\nu} = \sum_{LS} V(M,N,L,S)*T(L,S,I,J)$$

built-in super instruction

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
        do S
            get T(L,S,I,J)
            execute compute_integrals V(M,N,L,S)
            tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
            tmpsum(M,N,I,J) += tmp(M,N,I,J)
        enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Variable declarations and instantiation not shown

T and R are distributed arrays

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
        do S
            get T(L,S,I,J)
            execute compute_integrals V(M,N,L,S)
            tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
            tmpsum(M,N,I,J) += tmp(M,N,I,J)
        enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Divide iteration space among available workers and execute in parallel.

M,N,I,J count segments

Only parallel construct

Both static and dynamic load balancing supported

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
        do S
            get T(L,S,I,J)
            execute compute_integrals V(M,N,L,S)
            tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
            tmpsum(M,N,I,J) += tmp(M,N,I,J)
        enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Initialize local block

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
        do S
            get T(L,S,I,J)
            execute compute_integrals V(M,N,L,S)
            tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
            tmpsum(M,N,I,J) += tmp(M,N,I,J)
        enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Serial loops over declared ranges of L,S.

L and S count segments

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
        do S
            get T(L,S,I,J)
            execute  compute_integrals V(M,N,L,S)
            tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
            tmpsum(M,N,I,J) += tmp(M,N,I,J)
        enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Request block of distributed array

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
        do S
            get T(L,S,I,J)
            execute compute_integrals V(M,N,L,S)
            tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
            tmpsum(M,N,I,J) += tmp(M,N,I,J)
        enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
```

Compute block of V on demand.

Overlaps with communication of T

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
        do S
            get T(L,S,I,J)
            execute compute_integrals V(M,N,L,S)
            tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
            tmpsum(M,N,I,J) += tmp(M,N,I,J)
        enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Block contraction.

Wait for T(L,S,I,J) if necessary

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
        do S
            get T(L,S,I,J)

            execute  compute_integrals V(M,N,L,S)
            tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
            tmpsum(M,N,I,J) += tmp(M,N,I,J)
        enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Accumulate sum.

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
        do S
            get T(L,S,I,J)
            execute  compute_integrals V(M,N,L,S)
            tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
            tmpsum(M,N,I,J) += tmp(M,N,I,J)
        enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Store block to distributed array

Implementation in SIAL

```
pardo M,N,I,J
    tmpsum(M,N,I,J) = 0.0
    do L
        do S
            get T(L,S,I,J)
            execute  compute_integrals V(M,N,L,S)
            tmp(M,N,I,J) = V(M,N,L,S) * T(L,S,I,J)
            tmpsum(M,N,I,J) += tmp(M,N,I,J)
        enddo S
    enddo L
    put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
sip_barrier
```

Synchronize one-sided communication

Key idea: “Programming with blocks"

- Algorithms are expressed in terms of blocks
    - Individual array elements not mentioned in SIAL program—only in the implementation of the super instruction.
    - Each super instruction performs a substantial amount of computation
    - Each communication transmits substantial amount of data

Consequences of “programming with blocks”

- Algorithms can be effectively parallelized
- Source programs are independent of
  - number of processors
  - segment sizes
  - data layout

Allows tuning on different systems without changing SIAL code

Super Instructions

- Built-in
  - contraction in example
- Provided by programmer
  - compute_integrals in example
  - reusable, but most programmers will need to write some
- Efficient implementation for each platform
  - written in Fortran and/or C to take advantage of highly optimizing compilers
  - operates on local blocks, no communication
- Unconstrained, can escape abstraction

Language elements: Array types

- static
  - small, replicated
- local
  - individual blocks for intermediate results
- temp
  - local partial array, at least one dimension fully formed
- distributed
- served (disk-backed)

Language elements: Index types

- Three kinds
  - simple: counts interations
  - segment : counts segments
  - subindex : counts subsegments
- Finite range given in declaration
  - Uses symbolic constants given a value at runtime
    - Depends on size of problem
    - Size of segments
  - Used in array declarations

```
index kiter    = 1, cc_iter

aoindex mu      = 1, norb
aoindex nu      = 1, norb
aoindex lambda = 1, norb

moaindex i = baocc, eaocc
moaindex i1= baocc, eaocc

mobindex j = bbocc, ebocc
mobindex j1= bbocc, ebocc
```

- Index declarations
  - Use symbolic constants
  - Domain specific type names
  - Different segment types may be segmented differently.

```
distributed Vxixi(mu,i1,lambda,i)
distributed Vxxii(mu,nu,i1,i)
distributed Vxjxj(mu,j1,lambda,j)
distributed Vxxjj(mu,nu,j1,j)
```

- Array declarations
  - Size determined by index
  - Type system ensures consistent use

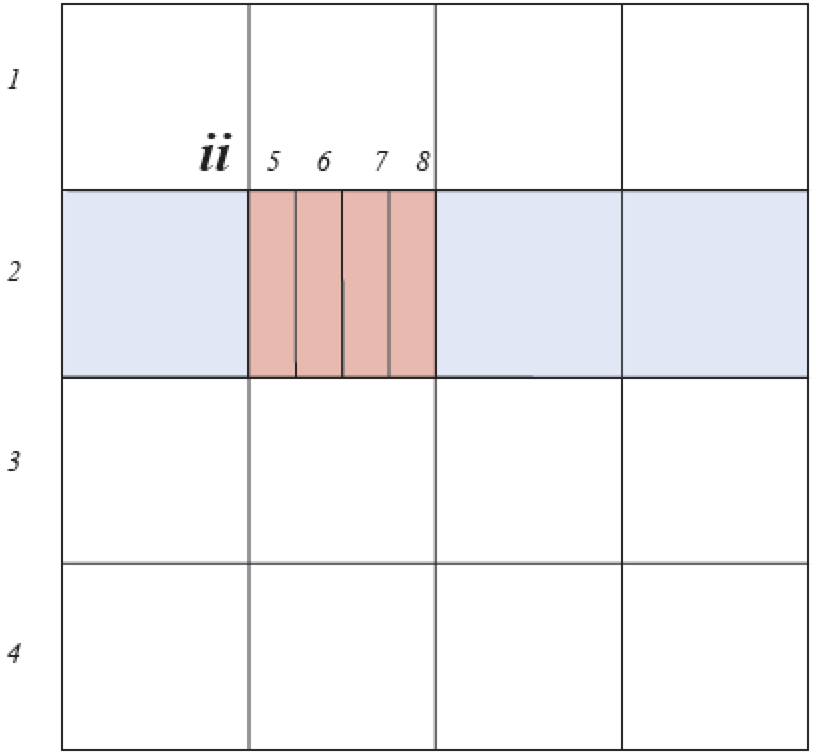
Subindices

- Problem
  - $C(a,b,c,l,m,n) = A(a,b,c,k)*B(k,l,m,n)$
  - Each block of *A* and *B* has seg^4 element
  - Each block of *C* has seg^6 element—not feasible
  - Reducing *seg* makes rest of computation perform poorly

- Subindices allow dealing with subblocks in a way that is consistent with the way blocks are handled in SIAL

Subindices, continued

```
moaindex j = 1,4
moaindex i = 1,4
subindex ii of i
temp Xi(i,j)
temp Xii(ii,j)
..
pardo j
    do i
        do ii in i
            Xii(ii,j) = Xi(ii,j)
            …
        enddo ii
    endo i
endpardo j
```



Loop over subblocks and extract

Runtime System: SIP

- Organization
  - set of worker nodes with one master
    - distributed array blocks managed by workers
  - set of I/O nodes that handle served (disk-backed arrays)
- Single threaded implementation (currently)
  - loops over op table containing SIAL byte code
  - periodically checks for MPI messages

Data Management

- Handles distributed data layout
  - data access very irregular
  - currently no attempts to exploit locality or block ownership
- Memory at individual nodes
  - partitioned into “stacks” of blocks of fixed sizes that match the segment sizes of the run
  - workers responsible for holding blocks of distributed arrays
  - caches blocks of distributed and served arrays

Dry Run

- Performed as part of SIAL program initialization
- Estimates memory usage
  - Determines feasibility of computation on system
  - Used to set up memory configuration
    - local memory (block stacks)
    - distributed data layout

- Typical SIA application:
  - Initialization
  - Several consecutive SIAL programs
    - Dry run and initialization of memory configuration between each one
    - Data may be saved on disk

One-sided Communication

- Distributed arrays: put, get, +=
  - workers cache blocks
- Served arrays: prepare, request, +=
  - I/O servers cache blocks and write to disk lazily
- SIP manages data descriptors used to locate blocks of distributed and served arrays
- Uses asynchronous message passing

Experience

- Used to implement ACES III
  - www.qtp.ufl.edu/ACES

- Capabilities
  - Hartree-Fock(RHF, UHF)
  - MBPT(2) energy, gradient, hessian
  - CCSD(T) energy and gradient (DROPMO)
  - EOM-CC excited state energies

Ports

- SGI Altix SMP
- Cray XT3
- Cray XT4/XT5
- IBM Cluster 1600 with Power 5+
- Linux Networx Advanced Technology Cluster
- Sun Opteron cluster
- BlueGene/P
- Power7s running Linux and AIX (Blue Drop, Blue Waters)

Tuning

- Tuning the SIP runtime
  - Easy with similar systems
  - BlueGene has been the most problematic port
- Tuning the super instructions that implement computational kernels
  - Can proceed independently from tuning the SIP
  - Can be done incrementally

Support for Tuning

- Low overhead but useful profiling info
  - Blocking time per pardo loop
  - Time for each superinstruction
  - …
- Ongoing work:
  - Generate performance model from SIAL code
  - Instantiate with measured data from network benchmarks and 2-node SIAL run

Programmer Productivity

- Anecdotal experience:
  - weeks with SIA vs months with straight MPI
- Not a silver bullet: It is still possible to write poorly performing programs.
- Each run provides timing information for each super instruction
  - low overhead but useful profiling info
- Programs need adjustment when used for significantly different problem sizes

Ongoing and future work

- Open system
  - Python interface
  - Re-architect and define interfaced for subsystems
- Enhance runtime
  - Petascale (Blue Waters)
  - Multicore
  - GPU
- Enhance expressiveness of SIAL
  - High rank arrays
  - Parallel regions
  - Support better software engineering
- Generalize
  - Other domains (types, symbolic constants)
- Performance modeling
  - Understand performance on very large systems without extensive experimentation