SIAL Course Lectures 3 & 4

Victor Lotrich, Mark Ponton, Erik Deumens, Rod Bartlett, Beverly Sanders
AcesQC, LLC
QTP, University of Florida
Gainesville, Florida
Lecture 3: Workings
Serial computers

- CPU (core) is in control
  - Performs operations on 64 bit quantities
  - Uses 64 bit registers
- Coordinates data movement
  - RAM
  - DASD
  - Network
Parallel computers

• Multiple CPUs (cores) must coordinate
• Data management is more complex
  – Data sharing
  – Data locks
  – Data flow delays
Parallel programming

• Some design patterns
  – Master-worker
    • Master tells workers what to do
  – Client-server
    • Clients ask one or more servers what to do

• Data sharing
  – Between two or few CPUs
    • exchange messages or share a token or lock
  – Between all or most CPUs
    • Post barriers and send broadcast messages
SIA = super serial

- Super Instruction Architecture
- First guiding principle

**Parallel computer = “super serial” computer**
- **Number <-> super number = block**
  - 64 bit <-> 640,000 bit
- **CPU operation <-> compute super instruction = subroutine**
- **Data operation <-> data super instruction = block copy**
  - RAM, network, DASD <-> local or remote
SIA = simple code

• Second guiding principle

  Separate algorithm from execution
  - Define a simple language to express the algorithm
  - Leave details of execution to a lower level
  - Define a precise boundary

• The first principle helps to implement that
  - Every block operation takes finite time
  - No operation can be considered instantaneous
  - All operations count in scheduling
A computer with a single CPU

• Basic data item: 64 bit number
• High level language: Fortran, C
  – \( c = a + b \)
• Assembly language
  – ADD dest,src
  – ADD is an operation code
  – dest and src are registers
A parallel computer running SIAL

- Super Instruction Assembly Language SIAL
  - \( R(I,J,K,L) += V(I,J,C,D) \times T(C,D,K,L) \)

- Bytecode

- Super Instruction Processor SIP
  - Fortran/C/MPI code

- Hardware execution
  - x86_64 PowerPC

- Java
  - \( R(i,j,k,l) += V(i,j,c,d) \times T(c,d,k,l) \)

- Bytecode

- JavaVM
  - C code

- Hardware execution
  - x86_64, PowerPC
A parallel computer running SIAL

• Super Instruction Assembly Language SIAL
  – \( R(I,J,K,L) += V(I,J,C,D) \times T(C,D,K,L) \)

• Bytecode

• Super Instruction Processor SIP
  – Fortran/C/MPI code

• Hardware execution

• Java
  – \( R(i,j,k,l) += V(i,j,c,d) \times T(c,d,k,l) \)

• Bytecode

• JavaVM
  – C code

• Hardware execution
A parallel computer running SIAL

- Super Instruction Assembly Language SIAL
  - $R(I,J,K,L) += V(I,J,C,D) \times T(C,D,K,L)$

- **Bytecode**

- Super Instruction Processor SIP
  - Fortran/C/MPI code

- Hardware execution

- **Java**

  compile

  - $R(i,j,k,l) += V(i,j,c,d) \times T(c,d,k,l)$

- **Bytecode**

- JavaVM

  - C code

- Hardware execution
A parallel computer running SIAL

- Super Instruction Assembly Language SIAL
  - $R(I,J,K,L) += V(I,J,C,D) \times T(C,D,K,L)$
- Bytecode
- Super Instruction Processor **SIP**
  - Fortran/C/MPI code
- Hardware execution

- **Java**
- **execute**
  - $R(i,j,k,l) += V(i,j,c,d) \times T(c,d,k,l)$
- Bytecode
- **JavaVM**
  - C code
- Hardware execution
SIAL design

High level
- concepts
- Data structures
- algorithms
- Super instruction
- Assembly language
  SIAL (scf.sial)

Problem

Performance

Low level
- communication
- Input/output
- Super instruction
  Processor
  SIP (xaces3)

input
output
User level execution flow

- Input
- Bytecode
  - algo.sio
- SIAL
  - algo.sial
- SIP
  - xaces3
- SIAL
  - compiler
  - assembler
Coarse grain parallelism

• While executing super instructions in SIAL program

• Example: data super instruction
  – GET block
  – Can be from
    • Local node RAM
    • Other node RAM
  – Time for data to become available differs
Fine grain parallelism

• While executing individual super instructions

• Example: contractions and integrals
  • * (contractions)
  • compute_integrals
  – Can use multiple cores
  – Can use accelerators
    • GPGPUs and Cell processors
    • FPGAs (field programmable gate arrays)
## Super instruction flow

<table>
<thead>
<tr>
<th>Worker i</th>
<th>Worker j</th>
</tr>
</thead>
<tbody>
<tr>
<td>• GET a -&gt; ask j</td>
<td>• …</td>
</tr>
<tr>
<td>• …</td>
<td>• …</td>
</tr>
<tr>
<td>• d=b*c</td>
<td>• &lt;- send a</td>
</tr>
<tr>
<td>• … wait for a</td>
<td>• …</td>
</tr>
<tr>
<td>• a arrives &lt;-</td>
<td>• …</td>
</tr>
<tr>
<td>• e=a*d</td>
<td>• …</td>
</tr>
<tr>
<td>• …</td>
<td>• …</td>
</tr>
</tbody>
</table>
Distributed data

• N worker tasks
  – each worker has local RAM

• Data distributed in RAM of workers
  – AO-based: direct use of integrals
  – MO-based: use transformed integrals

• Array blocks are spread over all workers
  – simple, deterministic mapping
Served (disk resident) data

• N worker tasks
  – each worker has local RAM

• M server tasks
  – have access to local or global disk storage
  – each server has local RAM for caching
  – accept, store and retrieve blocks

• Data served to and from disk
  – workers only have indirect access to disk
  – through servers
A SIAL program

- Two-electron integral transformation
  - Generate integrals in AO basis
  - \( V(a,i,b,j) = AO(m,l,n,s) \times c(m,a) \times c(n,b) \times c(l,i) \times c(s,j) \)
  - \( m,n,l,s \) : AO indices
  - \( i, j \) : occupied (alpha) MO indices
  - \( a, b \) : virtual (alpha) MO indices
  - Sum over \( m, n, l, s \) is implied
  - Integrals \( V(a,i,b,j) \) are written to disk
A SIAL program

SIAL 2EL_TRANS

aoindex m = 1, norb
aoindex n = 1, norb
aoindex r = 1, norb
aoindex s = 1, norb

moaindex i = baocc, eaocc
moaindex j = baocc, eaocc
moaindex a = bavirt, eavirt
moaindex b = bavirt, eavirt

algorithm for two-electron integral transformation.
temp AO(m,r,n,s)
temp txxxi(m,n,r,i)
temp txixi(m,i,n,j)
temp taixi(a,i,n,j)
temp taibj(a,i,b,j)

local Lxixi(m,i,n,j)
local Laixi(a,i,n,j)
local Laibj(a,i,b,j)

served Vaibj(a,i,b,j)
served Vxixi(m,i,n,j)
served Vaixi(a,i,n,j)

one block only – super registers
all blocks on local node
disk resident large arrays
PARDO m, n
   allocate Lxixi(m,*,n,*)
   DO r
      DO s
         compute_integrals AO(m,r,n,s)
         DO j
            txxxi(m,r,n,j) = AO(m,r,n,s)*c(s,j)
         DO i
            txixi(m,i,n,j) = txxxi(m,r,n,i)*c(r,i)
            Lxixi(m,i,n,j) += txixi(m,i,n,j)
         ENDDO i
         ENDDO j
      ENDDO s
   ENDDO r
   DO i
      DO j
         PREPARE Vxixi(m,i,n,j) = Lxixi(m,i,n,j)
      ENDDO j
      ENDDO i
   deallocate Lxixi(m,*,n,*)
ENDPARDO m, n
execute server_barrier
PARDO m, n

allocate Lxixi(m,*,n,*)
DO r
DO s
    compute_integrals AO(m,r,n,s)
    DO j
        txxxi(m,r,n,j) = AO(m,r,n,s)*c(s,j)
    DO i
        txixi(m,i,n,j) = txxxi(m,r,n,i)*c(r,i)
        Lxixi(m,i,n,j) += txixi(m,i,n,j)
    ENDDO i
    ENDDO j
ENDDO s
ENDDO r
DO i
DO j
    PREPARE Vxixi(m,i,n,j) = Lxixi(m,i,n,j)
ENDDO j
ENDDO i
deallocate Lxixi(m,*,n,*).

ENDPARDO m, n

execute server_barrier

parallel over block indices m and n
PARDO m, n
allocate Lxixi(m,*,n,*)
DO r
  DO s
    compute_integrals AO(m,r,n,s)
    DO j
      txxxi(m,r,n,j) = AO(m,r,n,s)*c(s,j)
    DO i
      txixi(m,i,n,j) = txxxi(m,r,n,i)*c(r,i)
      Lxixi(m,i,n,j) += txixi(m,i,n,j)
    ENDDO i
  ENDDO j
ENDDO s
ENDDO r
DO i
  DO j
    PREPARE Vxixi(m,i,n,j) = Lxixi(m,i.n,j)
  ENDDO j
ENDDO i
ENDPARDO m, n
execute server_barrier

deallocate Lxixi(m,*,n,*)
PARDO m, n
allocate Lxixi(m,*,n,*).
DO r
DO s
  compute_integrals AO(m,r,n,s)
  DO j
    txxxi(m,r,n,j) = AO(m,r,n,s)*c(s,j)
  DO i
    txixi(m,i,n,j) = txxxi(m,r,n,i)*c(r,i)
    Lxixi(m,i,n,j) += txixi(m,i,n,j)
  ENDDO i
  ENDDO j
ENDDO s
ENDDO r
DO i
DO j
  PREPARE Vxixi(m,i,n,j) = Lxixi(m,i,n,j)
  ENDDO j
  ENDDO i
deallocate Lxixi(m,*,n,*).
ENDPARDO m, n
execute server_barrier
PARDO m, n
allocate Lxixi(m,*,n,*)
DO r
  DO s
    compute_integrals AO(m,r,n,s)
    DO j
      txxxi(m,r,n,j) = AO(m,r,n,s)*c(s,j)
    DO i
      txixi(m,i,n,j) = txxxi(m,r,n,i)*c(r,i)
      Lxixi(m,i,n,j) += txixi(m,i,n,j)
  ENDDO i
  ENDDO j
ENDDO s
ENDDO r
DO i
  DO j
    PREPARE Vxixi(m,i,n,j) = Lxixi(m,i,n,j)
  ENDDO j
ENDDO i
deallocate Lxixi(m,*,n,*)
ENDPARDO m, n
execute server_barrier

transform two indices into local array using same integrals
PARDO m, n
allocate Lxixi(m,*,n,*)
DO r
  DO s
    compute_integrals AO(m,r,n,s)
    DO j
      txxxxi(m,r,n,j) = AO(m,r,n,s)*c(s,j)
    ENDDO j
    DO i
      txixi(m,i,n,j) = txxxxi(m,r,n,i)*c(r,i)
      Lxixi(m,i,n,j) += txixi(m,i,n,j)
    ENDDO i
  ENDDO s
ENDDO r
DO i
  DO j
    PREPARE Vxixi(m,i,n,j) = Lxixi(m,i,n,j)
  ENDDO j
ENDDO i
deallocate Lxixi(m,*,n,*)
ENDPARDO m, n
execute server_barrier

store in served array
PARDO m, n
allocate Lxixi(m,*,n,*)
DO r
  DO s
    compute_integrals AO(m,r,n,s)
    DO j
      txxxi(m,r,n,j) = AO(m,r,n,s)*c(s,j)
    DO i
      txixi(m,i,n,j)  = txxxi(m,r,n,i)*c(r,i)
    Lxixi(m,i,n,j) += txixi(m,i,n,j)
  ENDDO i
  ENDDO j
ENDDO s
ENDDO r
DO i
  DO j
    PREPARE Vxixi(m,i,n,j) = Lxixi(m,i,n,j)
  ENDDO j
ENDDO i
deallocate Lxixi(m,*,n,*)
ENDPARDO m, n
execute server_barrier
PARDO n, i, j
    allocate Laixi(*,i,n,j)
    DO m
        REQUEST Vxixi(m,i,n,j) m
        DO a
            taixi(a,i,n,j) = Vxixi(m,i,n,j) * c(m,a)
            Laixi(a,i,n,j) += taixi(a,i,n,j)
        ENDDO a
    ENDDO m
    DO a
        PREPARE Vaixi(a,i,n,j) = Laixi(a,i,n,j)
    ENDDO a
    deallocate Laixi(*,i,n,j)
ENDPARDO n, i, j
execute server_barrier
PARDO a, i, j
allocate Laibj(a,i,*,j)
DO n
   REQUEST Vaixi(a,i,n,j) n
   DO b
      taibj(a,i,b,j) = Vaixi(a,i,n,j)\*c(n,b)
      Laibj(a,i,b,j) += taibj(a,i,b,j)
   ENDDO b
ENDDO n
DO b
   PREPARE Vaibj(a,i,b,j) = Laibj(a,i,b,j)
ENDDO b
deallocate Laibj(a,i,*,j)
ENDPARDO a, i, j
execute server_barrier
ENDSIAL 2EL_TRANS
Lecture 4: Performance
SIAL performance

• All super instructions are asynchronous
• Thus execution is very elastic
• Helps maintain consistent performance on many parallel architectures
ACES III software

- Developed since 2003
- Parallel for shared and distributed memory
- Capabilities
  - Hartree-Fock (RHF, UHF)
  - MBPT(2) energy, gradient, hessian
  - CCSD(T) energy and gradient (DROPMO)
  - EOM-CC excited state energies
- Runs on 60,000 cores, and going...
ACES III performance

- Ar$_6$ RHF CCSD(T)
  - 300 bf
    - Jan 2008 code on midnight (ARSC)
    - May 2009 code on kraken XT5 (NICS)
- Luciferin RHF CCSD
  - 494 bf 12 iterations on midnight (ARSC)
- $N_{\text{procs}} = 2^X$
- $T \ (\text{min}) = 10^Y$
ACES III scaling

- RHF CCSD
- Sucrose 546 bf
  - Kraken XT4 (NICS)
  - Pingo XT5 (ARSC)
- \((\text{H}_2\text{O})_{21}\text{H}^+\) 657 bf
  - Kraken XT4 (NICS)
- \(N_\text{procs} = 2^X\)
  - 256 -> 4096
- \(T \text{ (min)} = 10^Y\)
ACES III Performance comparisons

• Invitation to compare performance on some problems

• Downloads and more at
  – [http://www.qtp.ufl.edu/ACES](http://www.qtp.ufl.edu/ACES)

• Reliable and predictable performance and scaling
  – ACES III runs often when others are constrained by design limitations
Running ACES III

• Input file “ZMAT”
  – Title line
  – Molecule specification
    • One line per atom: label and coordinates
• *ACES2(…) name list record
• *SIP name list record
Sucrose input for ACES III

SUCROSE
C 0.000 0.000 0.000
C 1.542 0.000 0.000
C 2.041 1.457 0.000
...
H 1.428 3.987 0.394

*ACES2(CALC=SCF,BASIS=6-311G**,MEMORY=10000000,
REF=RHF,SPHERICAL=ON,UNITS=ANGSTROM,
DROPMO=1-23
COORDINATES=CARTESIAN,
DIRECT=ON,
MULT=1
SYMMETRY=OFF)
Sucrose input for ACES III (2)

*SIP
MAXMEM = 900
TIMERS = YES
COMPANY = 1 1 384 0
IOCOMPANY = 2 1 128 0
SIP_MX_SEGSIZE = 26
SIP_MX_OCC_SEGSIZE = 17
SIP_MX_VIRT_SEGSIZE = 25
SIAL_PROGRAM = scf_rhf_isymm_diis10.sio
SIAL_PROGRAM = tran_rhf_ao_sv1.sio
SIAL_PROGRAM = ccsd_rhf_ao_sv1_diis5.sio
Sucrose run with ACES III

- 512 processors: 384 workers / 128 servers
- AO segment size = 26
- Occupied segment size = 17
- Virtual segment size = 25
- 0.9 Gbyte of memory per processor
- Timing data will be printed
SIP optimization and tuning

• Optimize with traditional techniques
  – optimize the basic contraction operations
    • map to DGEMM calls
  – create fast code to generate integrals
  – optimize memory allocation
    • use multiple block-stacks for different block sizes
  – optimize execution instructions
  – optimize data placement and movement
SIAL optimization and tuning

• Optimization
  – Algorithms
  – Understand data flow
  – Understand computational demand
  – Understand dependencies
  – Match them up
  – Simplicity of SIAL helps develop quickly
SIAL programmer productivity

• SIAL has simple syntax
  – Experience shows it is very expressive
• Exact data layout is done by SIP
  – Allows runtime tuning and optimization
• SIAL has rich set of data structures
  – temporary, local, distributed, and served arrays
• SIAL specific IDE in development
  – integrated in Eclipse
Productivity comparisons

• Other tools for parallel development
  – UPC (Universal Parallel C)
  – CAF (Co-Array Fortran)
  – GA (Global Array Tools)
  – DDI (Distributed Data Interface)

• Simple syntax

• Specify precise data layout
  – PGAS partitioned global address space
  – Rigorous array blocking
Extensions

- SIAL and SIP extensions
  - Create API
    - Define indices, segments, blocks
    - Define special super instructions
    - Specify hooks for input and output
  - support development in other problem domains
  - Retain the architecture