ACES 3 Tutorial: Efficient Parallel Implementation design

Erik Deumens
AcesQC and University of Florida
Jun 26, 2006
A story

- No new chemistry or physics:
  - build better tool to do new chemistry

- New computer science and software engineering:
  - a design pattern

- New paradigm of scientific activity:
  - work of a team of specialists
ACES 2

- Mature and complex code
- Theoretical foundation
  - Coupled Cluster theory
  - Many developments from there:
    - Higher orders
    - Multi Reference
    - Equation of Motion
    - Simularity Transformation
ACES 2

- ACES 1 was created 1980
  - G. Purvis, G. Trucks, A. Salter, B. Laidig
- ACES 2 was rewritten starting 1990
Roman times (1990-1991)

- Two heroes, exquisite gladiators:
  - Jürgen Gauss
  - John Stanton
- Living in a flat (Cray) world: fast CPU, fast RAM, fast disk (SSD)
- Created ACES 2

- Still vision of a flat world:
  - Vectorization was made automatic
  - Parallelization remained hard
  - MPI emerged as standard
- Several attempts to make parallel CCSD were partially successful
- Heroes are skilled knights, like
  - Wojtek Cencek
Renaissance (2003-)

- Recognition that the world is not flat
- Three heroes
  - Norbert Flocke
  - Victor Lotrich
  - Mark Ponton
- With support team
  - Ajith Perera
  - Anthony Yau
  - Marshall Cory
The problem to solve: CCSD

- Coupled Cluster singles and doubles
- Quantum mechanical description for electrons in molecules
- Diagrammatic techniques for math
- Example term: $R_{ij}^{ab} = \sum_{cd} V_{cd}^{ab} T_{ij}^{cd}$
- And many more...
Why is this problem hard?

- CCSD calculations are compute and data intensive
  - Large number of T amplitudes
  - Large numbers of integrals
    - to be kept in RAM, or on disk: stored method
    - to be computed multiple times: direct method
History of Design Principles

- Roman and Medieval design
  - Inflexible
  - Uniform architecture for flat world
- Renaissance design
  - Dynamic
  - Component or object architecture can adapt to mountains
Roman and Medieval Design

code gen

- control
- compute
- communication
- disk input output
- hardware
Roman and Medieval Design

- Assumptions
  - Data access latency and bandwidth
  - Computation intertwined with communication
  - Size for data that can be replicated
  - Hardware characteristics must fall in certain ranges to reach performance goals
Roman and Medieval Design

- Consequences
  - Detailed analysis by programmer
  - Match data flow with work flow
  - Manage communication deep in code
Renaissance Design

code gen

code

gen

compute

control

Communication

disk I/O

hardware
Renaissance Design

- **Requirement**
  - Allow flexibility to control separately at run-time:
    1. Computation
    2. Communication
    3. Disk input and output
Renaissance Design

- Principles
  - Define units of data
    - For movement and computation
  - Define basic operations on data units
    - All movement is asynchronous
  - Schedule operations and movement
    - Optimize hiding communication behind computation for every machine
    - Optimize data size to make its computation longer than its transportation
VAX 11/780 analogy

- Define data element: **super number**
  - A block of T or V is 10 KByte
- Define set of basic **super operations**
  - Get block from and put on disk
  - Get block from and put on remote RAM
  - Contract block of T with block of V in one of a few ways
VAX 11/780 analogy

- Reserve space in local RAM for holding blocks (super stack)
- Schedule all operations asynchronously
  - Issue get of data using registers for compute instruction after next
  - Issue put of result from previous compute instruction
  - Issue compute instructions on ready data
VAX 11/780 analogy

- Every operation takes some time
- **Super program** controls computation
  - schedule to keep all CPUs busy
  - manage outstanding communication and IO requests
- **Super instruction processor** executes instructions, is MIMD MPI program
Benefits

- Each **super instruction** can optimize use of
  - superscalar microprocessor architecture
  - multi-level caches
  - vector processors
  - SMP nodes
  - message passing mechanisms
  - disk input and output scheduling
Renaissance coding

- Object oriented to the extreme
- Write code in low level language for super instruction processor to obtain optimal performance
  - Fortran, C, C++
  - Non blocking MPI
  - Asynchronous I/O
Renaissance coding

- Write algorithm in high level **super instruction assembly language**
  - Declare (block) arrays, (block) indices
  - DO - END DO construct
  - PARDO – END PARDO construct
  - Basic operations: add and multiply and contract
  - Each line maps to a few **super instructions**
Data organization: numbers

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{11}$</td>
<td>$T_{12}$</td>
<td>$T_{13}$</td>
<td>$T_{14}$</td>
</tr>
<tr>
<td>$T_{21}$</td>
<td>$T_{22}$</td>
<td>$T_{23}$</td>
<td>$T_{24}$</td>
</tr>
<tr>
<td>$T_{31}$</td>
<td>$T_{32}$</td>
<td>$T_{33}$</td>
<td>$T_{34}$</td>
</tr>
<tr>
<td>$T_{41}$</td>
<td>$T_{42}$</td>
<td>$T_{43}$</td>
<td>$T_{44}$</td>
</tr>
</tbody>
</table>
Data organization: blocks

T on node 1

- T(1,1)
- T(1,2)
- T(2,1)

T on node 2

- T(2,2)
- T(1,3)
- T(2,3)
ACES 3 = Parallel ACES 2

- Distributed data in RAM of workers
  - AO direct use of integrals
  - MO use transformed integrals
- N worker tasks each with 1 GB RAM
- Array blocks are spread over all workers
- Workers compute integrals when integral instruction is called
ACES 3 = Parallel ACES 2

- Served data to and from disk
  - AO no transformation of integrals
  - MO use transformed integrals
- N worker tasks and M server tasks
  - Workers are as before
  - Servers have disk cache and disk
  - Servers take and give blocks
  - Servers compute integrals when asked
Build Code

- Writing SI AL is simpler than writing MPI in Dark Ages: focus on algorithm
  - Hero: Victor

- Writing SIP is simpler too: forget about algorithm, focus on basic operations
  - Heroes: Mark, Norbert
Optimize SIP

- Optimize with traditional techniques:
  - Anthony optimized the basic contraction operations by mapping them to DGEMM calls
  - Norbert created fast integral block code
  - Mark optimized memory allocation by using multiple block stacks
  - Mark optimized execution and data movement
Optimize SIAL

Victor quickly wrote very different implementation of basic algorithms using different strategies:

- Which intermediate blocks to compute?
- Store intermediate blocks or compute them repeatedly? How many times?
- No intermediates are computed as distributed arrays -> less synchronization
Some tests

- Do SCF and CCSD
  - $\text{H}_2\text{O}$ 115 functions 5 occupied
  - $\text{CH}_2\text{F}_2$ 116 functions 13 occupied
  - DMS 127 functions 17 occupied
  - $\text{C}_6\text{H}_4\text{F}_2$ 140 functions 29 occupied
  - $\text{Ar}_4$ 200 functions 36 occupied
  - $\text{Ar}_6$ 300 functions 54 occupied
  - $\text{Ar}_{10}$ 500 functions 90 occupied
<table>
<thead>
<tr>
<th></th>
<th>Distrib AO</th>
<th>Distrib MO</th>
<th>Served AO</th>
<th>Served MO</th>
<th>Serial MO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integral</td>
<td>159</td>
<td>1,977</td>
<td>158</td>
<td>2,307</td>
<td>829</td>
</tr>
<tr>
<td>transform</td>
<td>1</td>
<td>6/2</td>
<td>1</td>
<td>2/2</td>
<td></td>
</tr>
<tr>
<td>Total w/o SCF</td>
<td>3,022</td>
<td>3,500</td>
<td>3,330</td>
<td>12,258</td>
<td>3,257</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>2/2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Distr AO</td>
<td>Distr MO</td>
<td>Served AO</td>
<td>Served MO</td>
<td>serial</td>
</tr>
<tr>
<td>----------------</td>
<td>----------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------</td>
<td>--------</td>
</tr>
<tr>
<td>Integral</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transform</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total w/o SCF</td>
<td>6,341</td>
<td>6,049</td>
<td>26,575</td>
<td>16,259</td>
<td>36,566</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>15</td>
<td>2/2</td>
<td>2/2</td>
<td></td>
</tr>
</tbody>
</table>
$C_6H_4F_2$

<table>
<thead>
<tr>
<th>Method</th>
<th>CCSD</th>
<th>MO</th>
<th>15,856</th>
<th>7,743</th>
<th>4,848</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>32</td>
<td>.76</td>
<td>.61</td>
<td>.61</td>
</tr>
<tr>
<td>CCSD</td>
<td>35,278</td>
<td>10,687</td>
<td>6,294</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>AO</td>
<td>8</td>
<td>32</td>
<td>.82</td>
<td>.70</td>
<td>.70</td>
</tr>
<tr>
<td>CCSD</td>
<td>255,976</td>
<td>211,564</td>
<td>140,424</td>
<td>32</td>
<td>.68</td>
</tr>
<tr>
<td>Geom</td>
<td>12</td>
<td>16</td>
<td>.91</td>
<td>.68</td>
<td>.68</td>
</tr>
<tr>
<td>3 steps</td>
<td>1.</td>
<td>.91</td>
<td>.68</td>
<td>.68</td>
<td>.68</td>
</tr>
</tbody>
</table>
\[ \text{Ar}_4 \ 36 + 164 = 200 \text{ bf on 64 processors} \]

<table>
<thead>
<tr>
<th>Machine</th>
<th>SCF</th>
<th>trans</th>
<th>CCSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM P4 shelton</td>
<td>82 s</td>
<td>776 s</td>
<td>1,431 s</td>
</tr>
<tr>
<td>Compaq emerald</td>
<td>53 s</td>
<td>2,957 s</td>
<td>6,997 s</td>
</tr>
<tr>
<td>Cray X1 diamond</td>
<td>4,535 s</td>
<td>X1 busy</td>
<td>26,871 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>30 h</td>
</tr>
</tbody>
</table>

X1 busy
Ar₆ 54+246=300 bf on 64 processors

<table>
<thead>
<tr>
<th>Machine</th>
<th>SCF</th>
<th>trans</th>
<th>CCSD 1 iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM P4 shelton</td>
<td>313 s</td>
<td>4,242 s</td>
<td>16,363 s</td>
</tr>
<tr>
<td>Cray X1 diamond</td>
<td>582 s</td>
<td>6,452 s</td>
<td>19,601 s</td>
</tr>
<tr>
<td>Compaq emerald</td>
<td>132 s</td>
<td>4,180 s</td>
<td>29,188 s</td>
</tr>
</tbody>
</table>
## Cray X1 on 64 processors

<table>
<thead>
<tr>
<th>Basis functions</th>
<th>SCF</th>
<th>trans</th>
<th>CCSD</th>
<th>1 iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ar$_4$ 200</td>
<td>4,535 s</td>
<td>26,871 s</td>
<td>30 h</td>
<td>X1 busy</td>
</tr>
<tr>
<td>36+164</td>
<td>X1 busy</td>
<td>X1 busy</td>
<td></td>
<td>X1 busy</td>
</tr>
<tr>
<td>Ar$_6$ 300</td>
<td>582 s</td>
<td>6,452 s</td>
<td>5.4 h</td>
<td></td>
</tr>
<tr>
<td>54+247</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ar$_{10}$ 500</td>
<td>2,810 s</td>
<td>32,855 s</td>
<td>77 h</td>
<td></td>
</tr>
<tr>
<td>90+410</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comments

- IBM and Compaq are distributed memory systems with a fast switch; the Compaq CPUs are a bit faster; the IBM switch is a bit faster.
- Cray has a shared memory architecture and uses vector processors and has slow scalar performance, activity of other jobs, especially I/O can severely impact wall clock time.
- CCSD scaling $n^4o^2$. 
Computer Science: Design Pattern

- Identify:
  - atomic data item, big enough
  - atomic instructions to operate on these data items as a whole
- Reading, receiving, writing, sending data items becomes clear
- Optimal scheduling of operations becomes possible
Computer Science: Design Pattern

- Programmer
  - Can operate on entire data item
  - Work on parts of a data item is a bit super operation
- Too many bit operations, means the data item concept is not chosen well
Computer Science: Design Pattern

- Optimization of SIP:
  - SIAL programmer cannot break the rules
  - SIP programmer can optimize large SIAL programs (30,000 lines) with simple changes inside a few instructions of algorithms or data structures
  - SIP optimization introduces no errors
- Good performance obtained
Understanding ACES 3 runs

- Given: molecule and computer
- Make estimate of space needed
- Choose algorithm
- Choose segment size
Rule 1

- Every run needs servers for
  - DIIS
  - Integral transformation
Rule 2

- Run distributed or served?
  - It is always better, if you can, to run distributed
    - CCSD
    - Lambda
    - One-grad
  - It is always better to run served
    - Two-grad
Rule 3

- Big molecules on small computers need served version of
  - CCSD
  - Lambda
  - One-grad
Estimating space need

- Distributed CCSD needs several versions of T
  - 3 in RAM: $3 \times 2 \times 2$
  - DIIS histories on disk (served)

- That is all
Estimating space need

- Integral transformation needs most space on disk (served), if that step passes the memory test, everything will pass
ACES 3 space test

- ACES 3 does memory estimate at beginning and will end immediately on error
Estimating the work balance

- The ratio of worker tasks and server tasks
  - Too few server makes everybody wait
  - Too many servers wastes CPUs that could be workers
- Good ratio 7:1
  - 128 CPUs = 112 workers + 16 servers
Estimating the segment size

- Segment: piece of basis set that determines the basic block
  - AO segments
    - must fall on shell boundaries or integral computation wastes effort
  - MO segments
    - can be whatever you want
    - Make nr of occupied and unoccupied segments the same for better load balancing
Estimating the segment size

- Choice can strongly impact run time
- Choice depends on hardware
  - Ratio of CPU speed and communication speed can affect the choice
<table>
<thead>
<tr>
<th>Integral Transform</th>
<th>Segment 25</th>
<th>Segment 22</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total w/o SCF</td>
<td>12,303 12,303</td>
<td>11,777 11,777</td>
</tr>
<tr>
<td>Distr AO</td>
<td>323 1</td>
<td>298 1</td>
</tr>
<tr>
<td>Distr MO</td>
<td>5,204 4,879 3/1</td>
<td>1,745 1,904 1/1</td>
</tr>
<tr>
<td>Served AO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Served MO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serial</td>
<td>1,201</td>
<td>17,657</td>
</tr>
</tbody>
</table>