



# 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
    - Similarity Transformation



# ACES 2

---

- ACES 1 was created 1980
  - G. Purvis, G. Trucks, A. Salter, B. Laidig
- ACES 2 was rewritten starting 1990
  - J. Stanton, J. Gauss, A. Perera, J. Watts, M. Nooijen, P. Szalay, S. Kucharski, M. Musial, W. Lauderdale, D. Bernholdt



# 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



# Dark Ages (1992-2002)

---

- 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

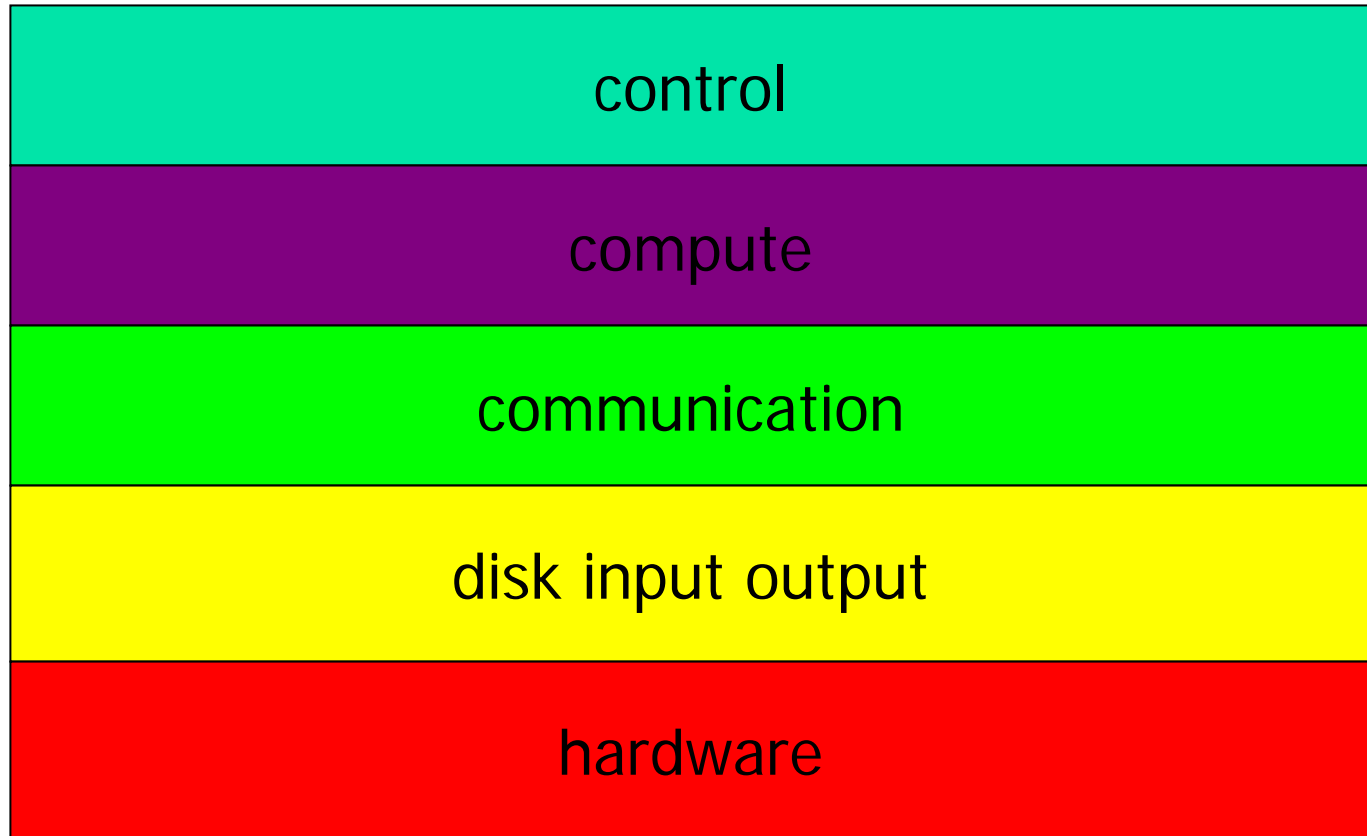
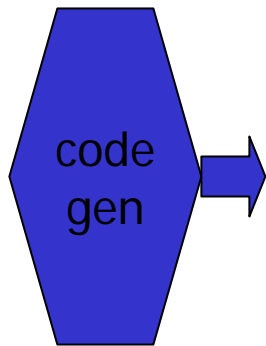


# History of Design Principles

---

- 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





# 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

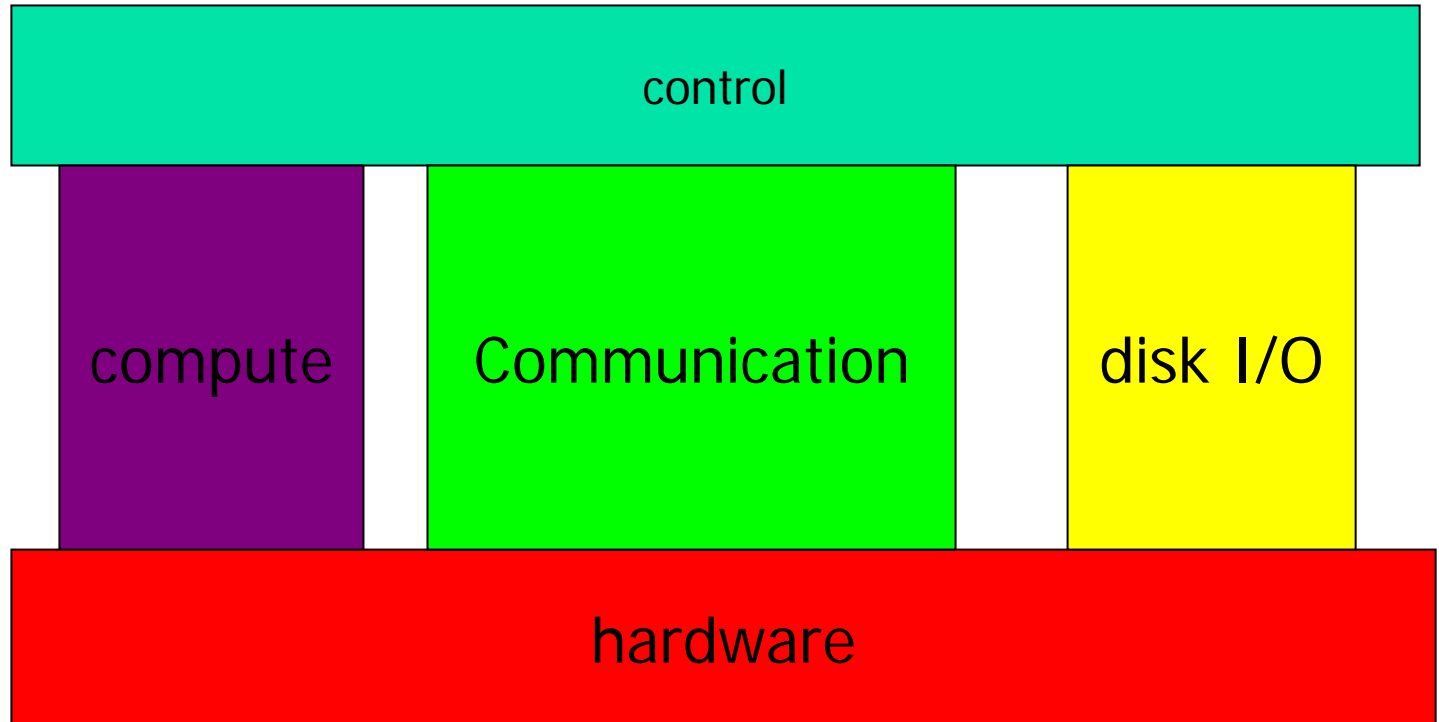
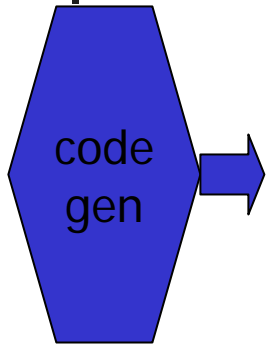


# Roman and Medieval Design

---

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

# Renaissance Design





# 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

---

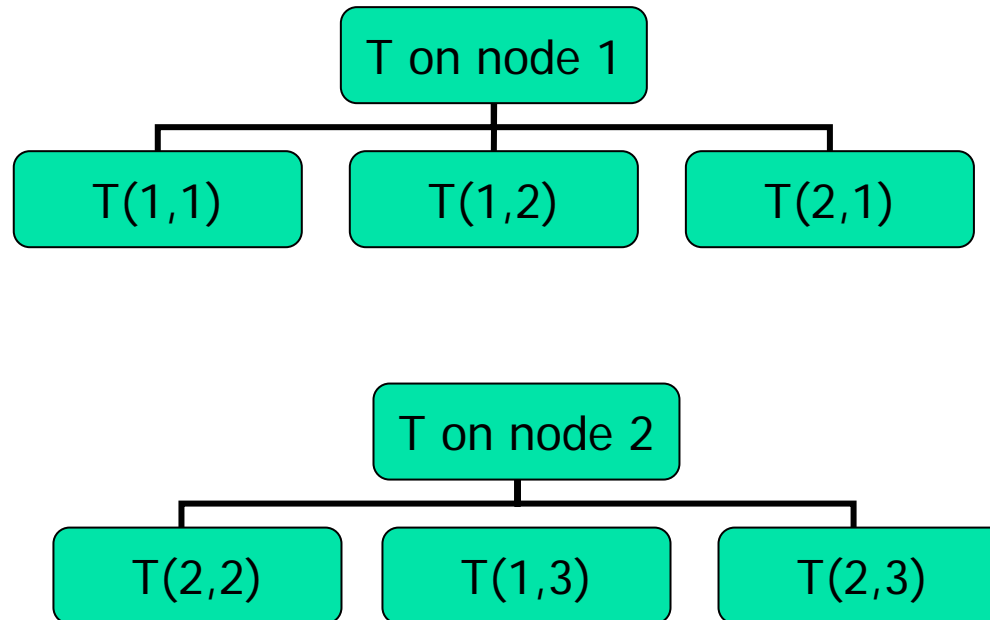
- Write algorithm in high level super instruction assembly language
  - Declare (block) arrays, (block) indices
  - DO - END DO construct
  - PARDO – END PARDO construct
  - Basic operations: add and multiply and contract
  - Each line maps to a few super instructions



# Data organization: numbers

$T_{11}$	$T_{12}$	$T_{13}$	$T_{14}$
$T_{21}$	$T_{22}$	$T_{23}$	$T_{24}$
$T_{31}$	$T_{32}$	$T_{33}$	$T_{34}$
$T_{41}$	$T_{42}$	$T_{43}$	$T_{44}$

# Data organization: blocks





# 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 SIAL 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
  - H<sub>2</sub>O 115 functions 5 occupied
  - CH<sub>2</sub>F<sub>2</sub> 116 functions 13 occupied
  - DMS 127 functions 17 occupied
  - C<sub>6</sub>H<sub>4</sub>F<sub>2</sub> 140 functions 29 occupied
  - Ar<sub>4</sub> 200 functions 36 occupied
  - Ar<sub>6</sub> 300 functions 54 occupied
  - Ar<sub>10</sub> 500 functions 90 occupied



# Water

	Distrib AO	Distrib MO	Served AO	Served MO	Serial MO
<b>Integral transform</b>	159 1	1,977 6/2	158 1	2,307 2/2	829
<b>Total w/o SCF</b>	3,022 1	3,500 8	3,330 1	12,258 2/2	3,257



# DMS

**Integral  
transform**

**Total  
w/o  
SCF**

Distr AO	Distr MO	Served AO	Served MO	serial
309 4/2	1,889 12/4	659 2/2	2,771 2/2	4,080
6,341 6	6,049 15	26,575 2/2	16,259 2/2	36,566



CCSD	15,856	7,743	4,848
MO	12	32	64
	1.	.76	.61
CCSD	35,278	10,687	6,294
AO	8	32	64
	1.	.82	.70
CCSD	255,976	211,564	140,424
Geom	12	16	32
3 steps	1.	.91	.68



Ar<sub>4</sub> 36 + 164 = 200 bf on 64 processors

Machine	SCF	trans	CCSD 1 iteration
IBM P4 shelton	82 s	776 s	1,431 s .4 h
Compaq emerald	53 s	2,957 s	6,997 s 1.9 h
Cray X1 diamond	4,535 s <b>X1 busy</b>	26,871 s <b>X1 busy</b>	30 h <b>X1 busy</b>



Ar<sub>6</sub> 54 + 246 = 300 bf on 64 processors

Machine	SCF	trans	CCSD 1 iteration
IBM P4 shelton	313 s	4,242 s	16,363 s 4.5 h
Cray X1 diamond	582 s	6,452 s	19,601 s 5.4 h
Compaq emerald	132 s	4,180 s	29,188 s 8.1 h

# Cray X1 on 64 processors

Basis functions	SCF	trans	CCSD 1 iteration
Ar <sub>4</sub> 200 36+164	4,535 s <b>X1 busy</b>	26,871 s <b>X1 busy</b>	30 h <b>X1 busy</b>
Ar <sub>6</sub> 300 54+247	582 s	6,452 s	5.4 h
Ar <sub>10</sub> 500 90+410	2,810 s	32,855 s	77 h

# 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  $n^4 o^2$  .



# 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:  $3 v^2 o^2$
  - 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



Segment 25  
Seqment 22

Integral  
transform

Total  
w/o  
SCF

Distr AO	Distr MO	Served AO	Served MO	serial
323 1	5,204 <b>4,879</b> 3/1	298 1	1,745 <b>1,904</b> 1/1	1,201
12,303 1	11,777 <b>10,430</b> 3/1	13,719 1	23,813 <b>14,540</b> 1/1	17,657