

Optimization and Parallelization on the Altix

Victor Travieso
COSMOS parallel programmer
University of Cambridge

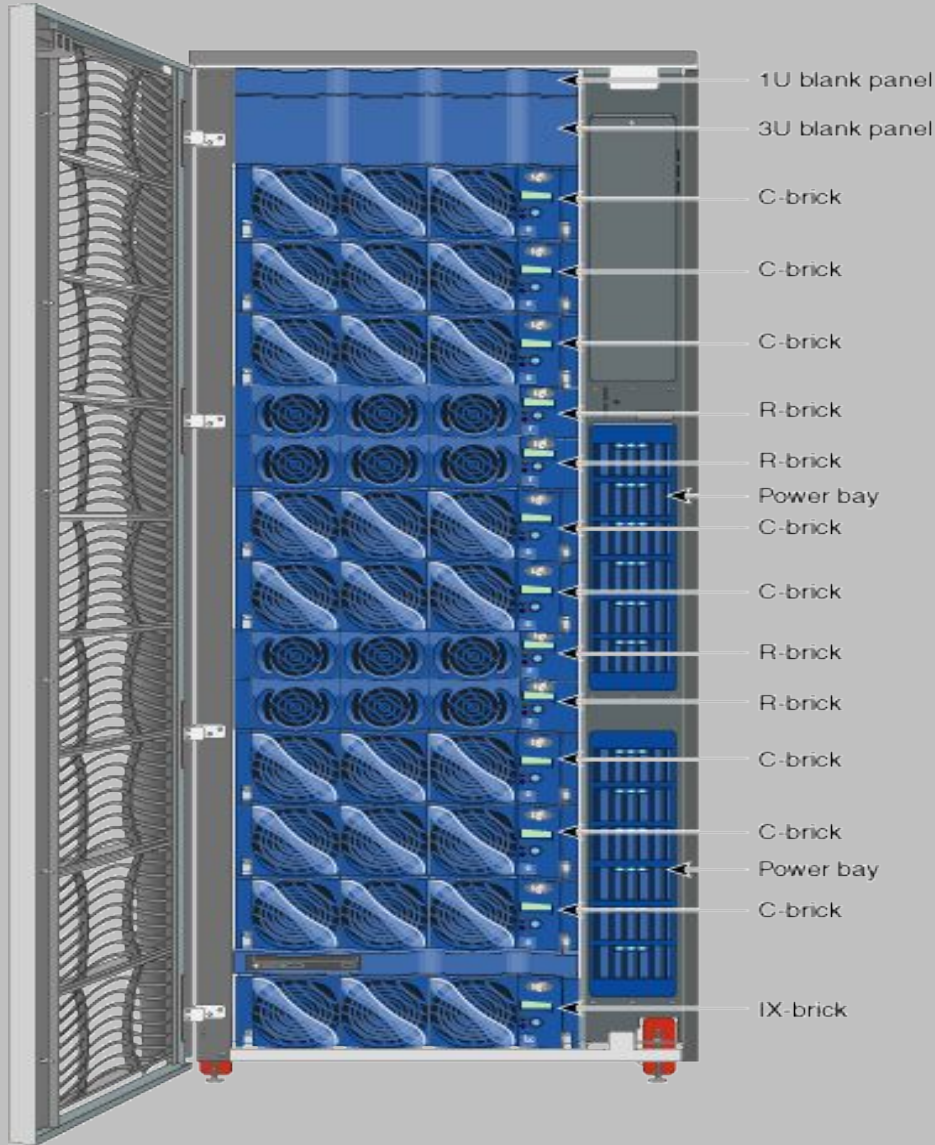
Overview

- Altix architecture
- The Itanium-II
- Optimization concepts
- Techniques
- Parallel processing

Altix architecture

- High performance 'cluster'
 - modular design – bricks and cables - NUMAflex
- SSI, global shared memory
 - ccNUMA – cache coherent non-uniform memory access
- 64-bit computing with Itanium2 CPUs
- Load latency, high bandwidth interconnect
 - NUMALink 3 (4) – Same bandwidth as the Origin but ~ 40% faster
- Fastest Linux I/O
 - CXFS, XFS filesystems > 2GB/s

Altix architecture



- C-Brick
 - CPU and memory
- R-Brick
 - Router interconnect
- IX-Brick
 - Base I/O Module
- PX, M, D
 - Expansions – PCI-X, Memory, Disk

Altix architecture

- Compute brick in Cosmos

- Memory

 - 2 GB per node

 - 6.4 GB/s

- Numalink4

 - Inside a C-Brick

 - 6.4 GB/s

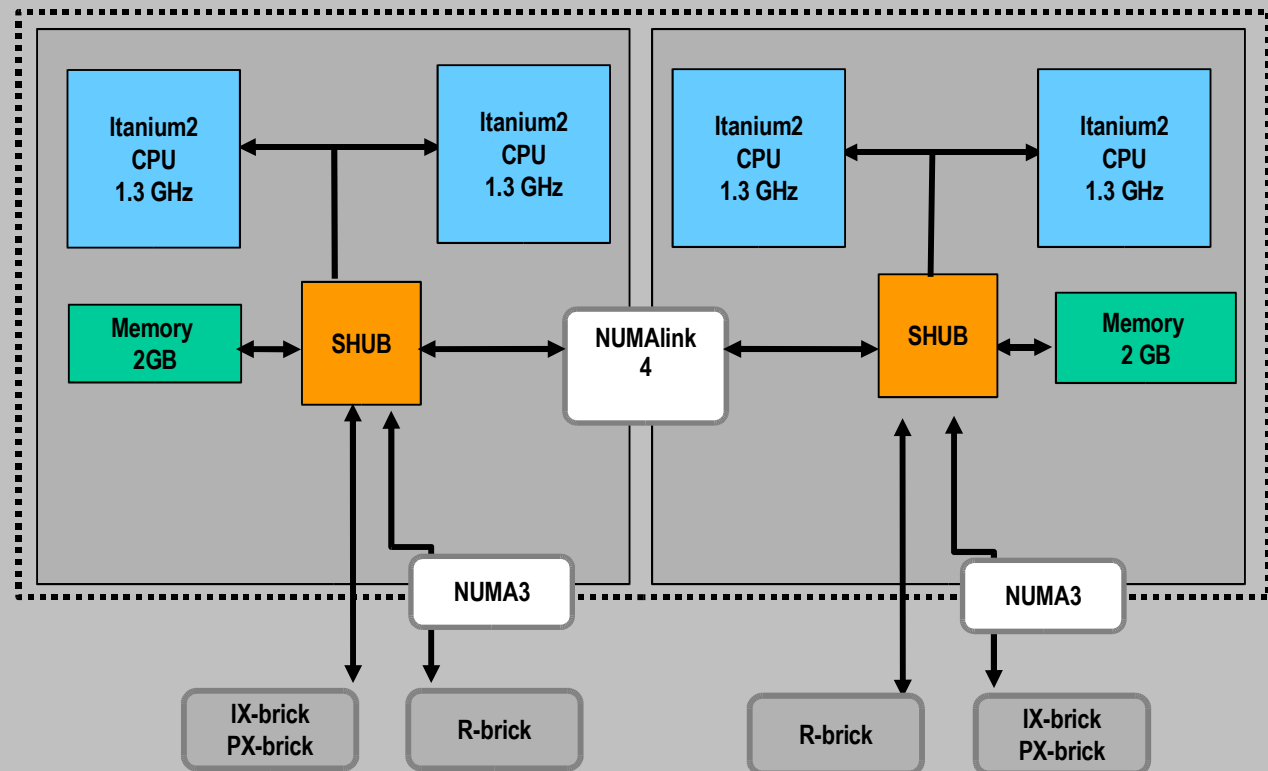
- Numalink 3

 - Inter brick

 - 3.2 GB/s

- Latencies

 - Local ~ 100 ns, nn ~ 240 ns + 50 ns extra per hop (fat tree topology, max hops = 6)



Altix architecture

Benefits of shared memory:

- Supports both parallel programming models
 - threads (OpenMP, posix) and message passing (MPI)
 - OpenMP is easier to do – incremental parallelization
- No need to decompose data or swapping
 - large data sets fit into memory
- Efficient load balancing
 - no need to move data between processes
- More efficient MPI implementation
 - can use 'shortcuts' to speed up communication (single-copy)
- Easier to administer

Itanium II processor

Intel 64-bit offer for HPC:

- Second iteration of the Itanium - Madison
 - the first one gained the nick name 'Itanic'..
- Designed and optimized for scientific workloads
- Large and fast on-die cache (including L3)
 - Level 3 cache is 3 mb on Cosmos
- EPIC architecture – superscalar, up to 6 IPC
- Large amount of resources (registers, issue units, HPW)
- Logic simplified
 - So called 'logic transfer'... responsibility of good instruction selection in the compiler
 - No out of order execution

Itanium II processor

Cache system:

- Level 1 - Instructions (L1I)
 - 16 Kb, 1 clock cycle latency
- Level 1- Data (L1D)
 - 16 Kb, 1 clock cycle, integer only
- Level 2
 - 256 Kb, 6 clock cycles
- Level 3
 - 3 Mb (Cosmos), 12 cycles, on chip

Itanium II processor

EPIC Architecture:

- Clock speeds are relatively slow (<1.5 GHz)
 - 1.3 Ghz on Cosmos
- Superscalar – up to 6 instructions per cycle
 - up to 4 memory operations – 2 loads and 2 stores
 - up to 2 floating point operations (can sustain 2 fma = 5.2 Gflops peak on Cosmos)
 - up to 6 from – 2 integer operations, 1 shift, 6 address increments
 - up to 3 branches
- No out of order logic
 - if no instructions can be issued at the moment (eg. might be waiting for memory) the processor stalls
 - pressure on the compiler to find parallelism and produce appropriate ordering of operations

Itanium II processor

Other features:

- Very large amount of registers
 - 128 integer, 128 floats, 64 predicate (branching),..
- Reduced function calling overhead
 - Stacked registers – argument passing stays in registers and frame is restored on function return
- Improved support for software pipelining
 - so called rotating registers
- TLB supplemented by a Hardware Page Walker
 - the translation lookaside buffer handles virtual to physical memory mapping – the itanium 2 TLB has room for 128 entries, with 16k page size
 - TLB page miss is very costly ~ 1000 cycles
 - the Hardware Page Walker provides a second level of TLB to reduce misses

Optimization Concepts

Is the implementation making full use of the hardware?

- Assuming the algorithm is well designed...
 - the pipeline might not be delivering instructions every cycle
 - processor stalls waiting for memory
 - cycles wasted.. no-op
- Most optimizations have to do with hiding the memory latency
 - memory speed \ll processor speed
- Profiling helps identify the bottlenecks and hot spots
 - where is the program spending most of the time ?
 - why ?
- Compiler might need help to generate optimal code

Optimization Concepts

Memory hierarchy:

- Memory is organized as a pyramid with different layers of increasing size and latency holding a copy of a chunk of memory
 - Registers – no delay
 - L1 Cache – 1 cycle
 - L2 Cache – 6 cycles
 - L3 Cache – 12 cycles
 - Memory ~ 200 cycles
 - Disk > 10^6 cycles

Principle behind it is data reuse and locality of reference

Optimization Concepts

Cache:

- Organised in lines
 - loads are performed on adjacent memory locations – cache lines
 - L1 has 64 byte lines
 - L2 and L3 have 128 byte lines – 8 doubles
- Caches are n-way associative
 - n is the number of places where a cache line might reside – fast access
 - L1 is 4-way associative, L2 is 8-way, L3 is 12-way
- Loading a cache line will purge the least used one from the cache
- Good practice: organize data access so code reuses data and displays locality

Optimization Concepts

Software pipelining:

- Organize instructions efficiently to expose parallelism and hide latencies
 - ideally aims to keep all the functional units busy every cycle
 - there is a prologue and epilogue overhead before the pipeline is fully established
- One of the most important factors for good performance in loop intensive codes
 - the compiler should always swp large loops
 - failures to swp normally derive from loop carried dependencies – these can often be eliminated through iteration reordering

Optimization Concepts

Prefetching:

- Issue load instructions ahead of time so data is readily available when needed
- Very effective technique to hide latency in loops
- Can be counterproductive – waste of bandwidth if the data is not used or needs to be loaded again
 - Intel compiler v8 prefetchs a lot.. sometimes you need to tell it not to

Optimization Concepts

Measuring performance:

- When do we need to optimize ?
- Estimate operation count and theoretical performance
 - a program might be floating point or memory bound
 - theoretical performance works as a lower bound for the time that a code could take to execute assuming ideal memory performance (no latency)
- Measure the actual performance by timing the program
 - a quick check before going into details with the profiling tools
- Check optimization reports from the compiler
 - tells you which parts of the code might not have been fully optimised
 - is performance degraded when full optimizations are on ? - compiler needs help
 - use the flags '-opt_report_file *filename* -opt_report_level2 -opt_report_phase hlo'

Optimization Concepts

Measuring performance:

- Example: Matmul

- easy to estimate, short code

```
do i=1,n
  do j=1,n
    do k=1,n
      a(i,j)=a(i,j) + b(i,k)*c(k,j)
    enddo
  enddo
enddo
```

- $2*N^3$ flops

- $3*N^2$ data values

- flop/memop yields a volume to surface ratio – theoretical limit at 2 fma per cycle is 5.2 Gflops

Optimization Concepts

Measuring performance - matmul:

- Take a size of $n=1000$ ($>$ cache, latency counts)
- '-O2' gives $\sim 3\%$ of theoretical limit – clearly this is inefficient!
- '-O3' gives around 4500 Gflops $\sim 87\%$ of limit
- The compiler seems to be doing a good job with the full optimizations.. what exactly is it doing ?
 - reorganizing data for optimal access pattern
 - cache blocking
 - prefetching
 - software pipelining (unrolling helps here)

Optimization Concepts

Matmul – optimization report:

```
'ifort -O3 -o matmul matmul.f90 -opt_report -opt_report_file matmul.txt -opt_report_phase hlo'
```

```
=====
```

High Level Optimizer Report for: MAIN__

Total #of lines prefetched in MAIN__ for loop in line 9=2

#of Array Refs Scalar Replaced in MAIN__ at line 16=36

Total #of lines prefetched in MAIN__ for loop in line 16=8

#of Array Refs Scalar Replaced in MAIN__ at line 16=3

Total #of lines prefetched in MAIN__ for loop in line 16=5

Total #of lines prefetched in MAIN__ for loop in line 16=2

Optimization Concepts

Matmul – optimization report:

LOOP INTERCHANGE in loops at line: 8 9

Loopnest permutation (1 2) --> (2 1)

LOOP INTERCHANGE in loops at line: 14 15 16

Loopnest permutation (1 2 3) --> (2 3 1)

Block, Unroll, Jam Report:

(loop line numbers, unroll factors and type of transformation)

Loop at line 9 unrolled without remainder by 2

Loop at line 16 blocked by 125

Loop at line 15 blocked by 125

Loop at line 14 blocked by 125

Loop at line 14 unrolled and jammed by 4

Loop at line 15 unrolled and jammed by 4

Optimization Techniques

Prefetching:

- Data is being pulled from main memory, so latency counts.
We could insert manual prefetch instructions:
 - `!DIR$ prefetch reference`
 - `#pragma prefetch reference`
- Or enable prefetching at O2...
 - `'-mP2OPT_hlo_prefetch=T'`
 - with `'-O2'` it gives an additional 5% performance increase

Optimization Techniques

Reordering data:

- Our matmul kernel:

$$a(i,j)=a(i,j) + b(i,k)*c(k,j)$$

- Inner loop over k allows reuse of a in registers
- However, b isn't accessed with optimal pattern (row-major in Fortran)
- If b is transposed, both input matrices are accessed along cache lines

$$a(i,j)=a(i,j) + T[b(k,i)] *c(k,j)$$

Optimization Techniques

Cache blocking:

- Work on small, contiguous data sets
- By tiling the matrices we can push the data reuse further

```
do ii=1,n,blk
do jj =1,n,blk
do i=ii,ii+blk-1
    do j=jj,jj+blk-1
        do k=1,n
            a(i,j)=a(i,j) + b(i,k)*c(k,j)
        enddo
    enddo
enddo
enddo
enddo
```

Optimization Techniques

Software pipelining:

- Grouping and unrolling loops help expose more parallelism to the compiler
- If the swp is not optimal, the compiler will give you an idea of what is wrong
 - add the flag '-opt_report_phase ecg_swp
 - eg. Failures:
 - loop dependencies – deambiguate pointers with '!DIR\$ ivdep before the loop
 - use '-fno-alias' with C codes
 - trip count too low– override with '!DIR\$ loop count(n)' or group loops together

Optimization Techniques

Putting it all together:

- We can get ~ 80% of the theoretical peak by performing hand optimizations on the code..
- Or we can let the compiler do it.
 - but often it will need help or will get it wrong, so guiding it might be necessary
- Also, we can do better: use an optimized library
 - linking with the SCSL library – rewrite the call as a dgemm operation (Lapack)
 - achieves 93% of the theoretical peak.. and is multiprocessor
 - ifort -o matmul_dgemm matmul_dgemm.f90 -lscs_mp

```
Call Dgemm('n','n',mat_size,mat_size,mat_size,1.0_wp,b, &  
mat_size,c,mat_size,0.0_wp,a,mat_size)
```

Parallel programming

When to parallelize:

- There's enough work to compensate for parallel overheads
 - thread synchronization, communications, and time!
- A significant portion of the program can be parallelized
 - Ahmdal's law
- Things to consider:
 - adequate programming model : OpenMP or MPI
 - load balancing
 - OpenMP: locality and cache trashing
 - MPI: improve communications with MPT

Parallel programming

OpenMP:

- Very easy to program
 - code directives
 - incremental parallelization
 - don't need to worry about distributing data
- Memory placement
 - first touch applies, no migration! - poor placement leads to memory bottleneck and kills scalability
 - solution: ensure that you initialize data in the parallel region by the thread that will use it most
- Cache trashing
 - ccNUMA needs to ensure cache view is coherent
 - reading and writing to the same line from multiple processors results in continuous purging and reloading of cache lines – even if not accessing the same data
 - check with 'pfmon -e L3_LINES_REPLACED ./a.out'

Parallel programming

MPI:

- Requires data decomposition and often rewriting the algorithms
 - on the other hand it ensures good data locality
 - it is the most portable and widely used paradigm
- Communication is the bottleneck – also load balancing
 - use the MPT library on the Altix (default when linking with '-lmpi')
 - use -v and -stat when launching mpirun to get detailed information of environment variables and communication statistics
 - if there is a high number associated with “retries allocating mpi PER_PROC buffers”, increase the size: 'export \$MPI_BUFS_PER_PROC=n' – defaults to 32
 - enable single copy message passing 'export MPI_BUFFER_MAX=2048' – only certain types of data – static and private heap, must be contiguous - but double the bandwidth