# Tuning single node and network performance on PC-clusters

C. Urbach

Division of Theoretical Physics
The University of Liverpool

Zeuthen, 27.11.2006

# Motivation

- almost everybody has a PC under his desk
- PC clusters are quite common nowadays
- some even have a fast network

So why not get the most out of them...?

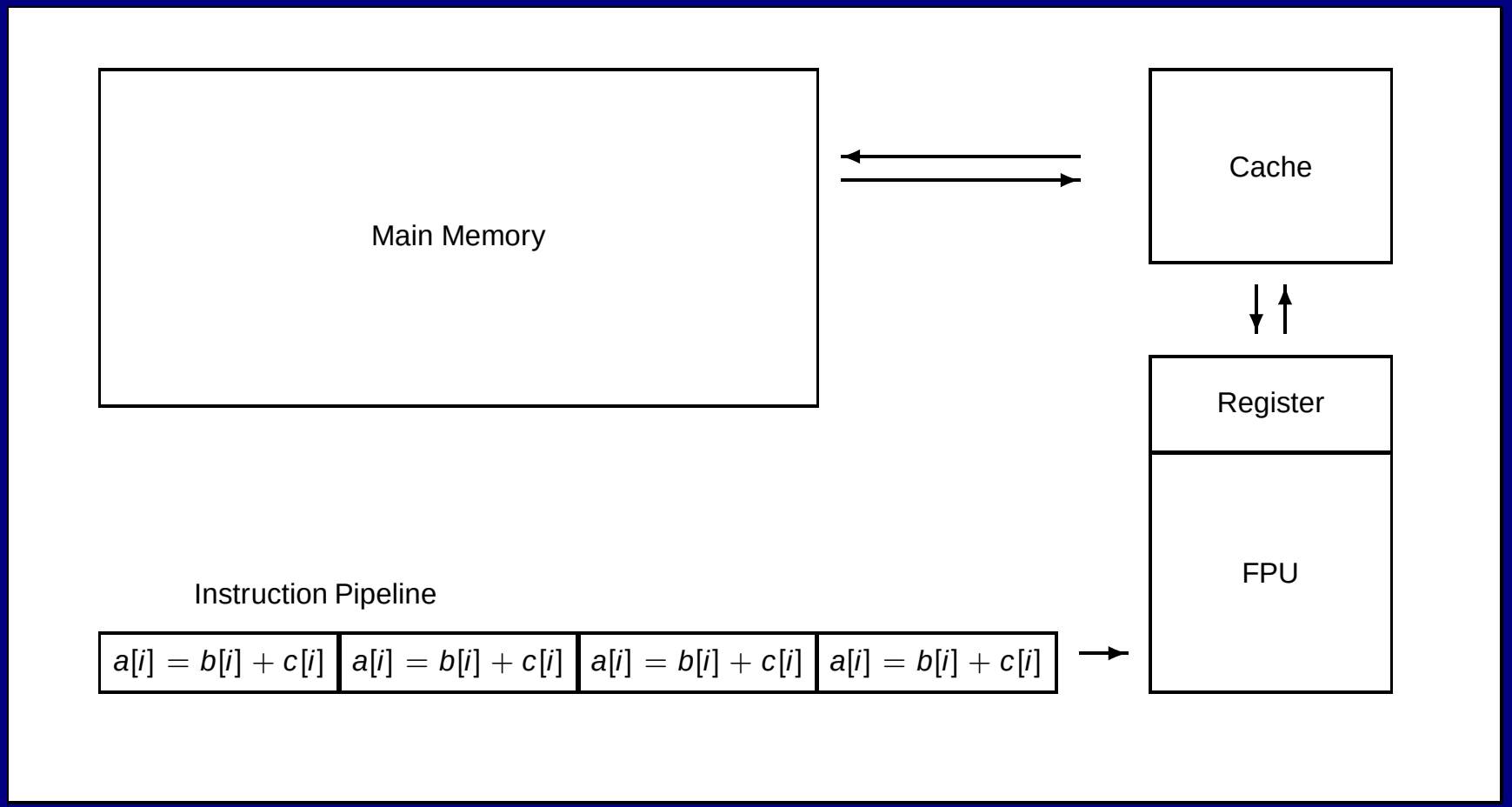
# Outline

# PC-Clusters

In general

- nodes with $1 - 4$ CPU's each sharing memory
- Intel or AMD CPU's:
  P3, P4, Athlon, opteron, itanium
- $\mathcal{O}(1 - 10)$ GB memory per node
- nework:
  - Gigabit ethernet
  - some fast/low latency network like myrinet or infiniband
- GNU compilers and maybe more (intel, PGM,...)
- some MPI implementation: MPICH or LAM
  with special drivers for the fast network
- some queueing system like OpenPBS

# CPU and Memory (schematic)



# Memory Hierarchy

| | speed | size |
|---|---|---|
| Register | very fast | very small |
| Cache | fast | small - medium |
| Main Memory | slow | large - very large |

Floating Point Unit (FPU) operates only on registers

# Memory Hierarchy

- Loading data:
  main memory $\rightsquigarrow$ cache $\rightsquigarrow$ registers.
- Loading into the Cache can be done in advance (prefetching)
- Prefetching can be done by the hardware, if your data structure allows for it (prefetch streams)
  - prefetch streams can be enabled if you access data in sequential order
- Prefetching can also be done by hand.

# Dirac Operator

- Dirac Operator:

$$D\ \psi(x) \equiv \sum_x \Big\{ (4 + m_0)\psi(x) + \sum_\mu \Big[ U_{x,\mu}(1+\gamma_\mu)\psi(x+\hat{\mu}) + U^\dagger_{x,-\mu}(1-\gamma_\mu)\psi(x-\hat{\mu}) \Big] \Big\}$$

- Optimise the operation:

$$U(1 \pm \gamma_\mu)\psi$$

# $U(1 \pm \gamma_\mu)$

- $(1 \pm \gamma_\mu)\psi$ has only two independent components
- example:

$$(1+\gamma_0)\psi = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \psi_3 \end{pmatrix} = \begin{pmatrix} \psi_0 + \psi_2 \\ \psi_1 + \psi_3 \\ \psi_0 + \psi_2 \\ \psi_1 + \psi_3 \end{pmatrix}$$

- multiply only two components with $U_{x,\mu}$
- allows for further improvement, see later

# $U(1 \pm \gamma_\mu)$

- define (Projector to a half spinor):

$$P_2 \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

and

$$P_{\pm\mu}^{4\to 2} = P_2 \left(1 \pm \gamma_\mu\right)$$

- define similar: $P_{\pm\mu}^{2\to 4}$
- Therefore implement:

$$\phi \;=\; U_{x,\mu} \left(1 \pm \gamma_\mu\right) \psi \;=\; P_{\pm\mu}^{2\to 4}\, U_{x,\mu}\, P_{\pm\mu}^{4\to 2}\, \psi$$

# Lookup Tables

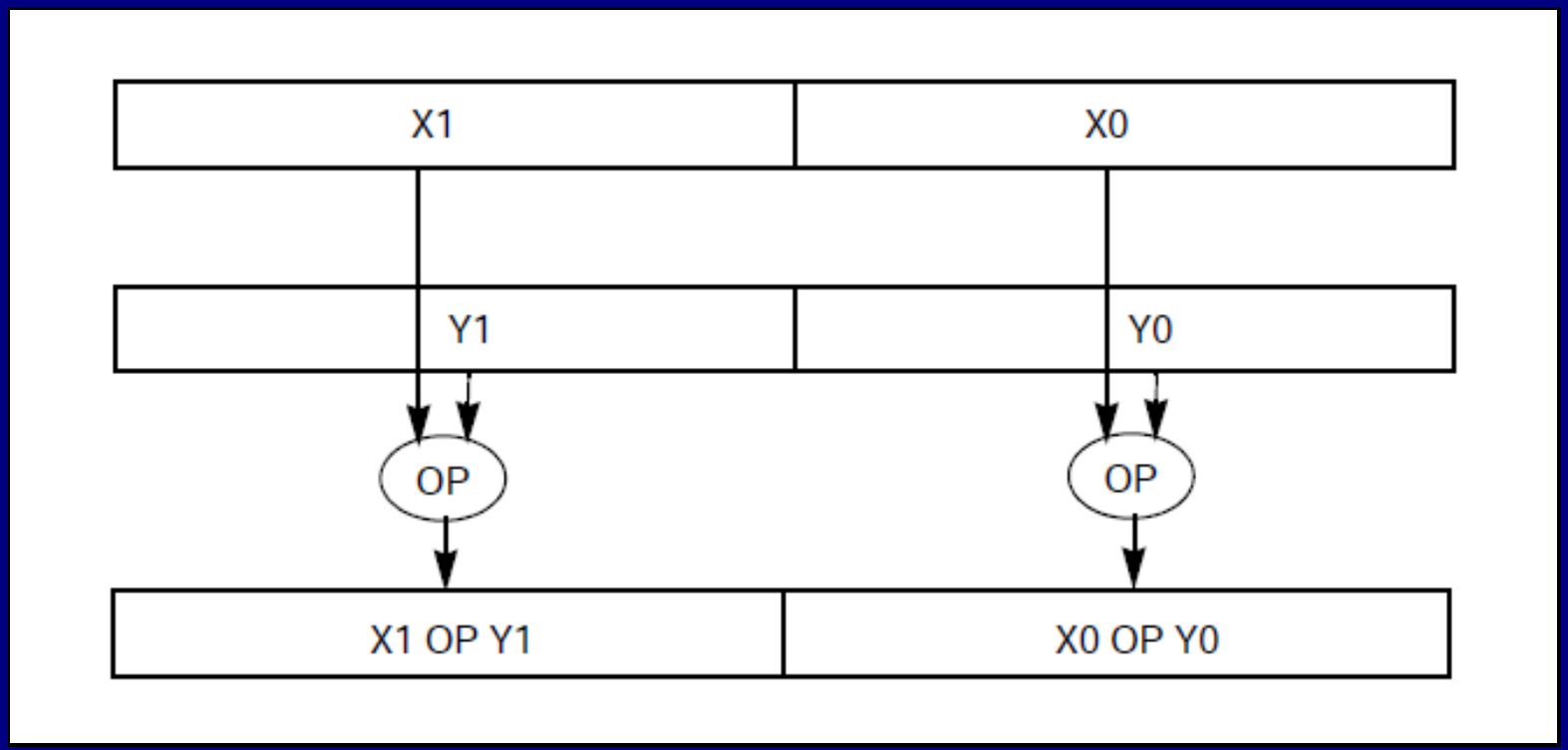
- when performing the loop over $x$, we need next neighbours of $x$
  - can be computed on the fly
  - can be computed once at the beginning and then stored in lookup table `nn[x][mu]`:
- my experience: lookup tables work faster
- but don’t have too many of them...

# Vector Processing Units

- go under the names `MMX, SSE, SSE2, SSE3, 3DNow!, Altivec, DoubleHummer, ...`
- originally to speed up multimedia applications
- parallel operation on data vectors
  (SIMD: Single instruction, multiple data)
- require in general one or two cycles to complete
- will concentrate here on SSE2
  supported by P4, opteron and later athlon revisions

# SSE2 schematically

Operation OP acting on vector X and Y
both containing two doubles.



# SSE2 Instructions

- there are 8 `MMX` 128 Bit registers available: `xmm0-xmm7`
- Note that opteron has 16 `MMX` registers
- special load instructions for packed doubles:
  load from memory to `MMX` registers
- special store instructions:
  store a `MMX` register to memory
- special instruction set for arithmetic operations on `MMX` registers
- the special instructions can be used with inline assembly (gcc) or with compiler specific functions (Intel compiler)

# SSE2 example

- Suppose you want to use SSE2 for:

```
for(i=0; i<20; i++) {
  c[i] = a[i] + b[i];
}
```

- Using inline assembly with gcc
- `a`, `b`, `c` are double arrays of length 20

# SSE2 example

```
for(i=0; i<20; i+=2) {
__asm__ __volatile__ (
  "movapd %1, %%xmm0 \n\t" \    //load a[i]
  "movapd %2, %%xmm1 \n\t" \    //load b[i]
  "addpd %%xmm0, %%xmm1 \n\t" \ //add
  "movapd %%xmm1, %0" \   //store in c[i]
  : \
  "=m" (c[i]) \  //output
  : \
  "m" (a[i]), \  //input
  "m" (b[i]));
}
```

# SSE2 example 2

Complex multiply $a * b$ ($b$ already in xmm3)

```
"movsd %0, %%xmm6 \n\t" \          \\(a.re,-)
"movsd %1, %%xmm7 \n\t" \          \\(a.im,-)
"unpcklpd %%xmm6, %%xmm6 \n\t" \ \\(a.re,a.re)
"unpcklpd %%xmm7, %%xmm7 \n\t" \ \\(a.im,a.im)
"movapd %%xmm3, %%xmm0 \n\t" \    \\(b.re,b.im)
"mulpd %%xmm6, %%xmm3 \n\t" \ \\(a.re*b.re,a.re*b.im)
"mulpd %%xmm7, %%xmm0 \n\t" \ \\(a.im*b.re,a.im*b.im)
"shufpd $0x1, %%xmm0, %%xmm0 \n\t" \
"xorpd %2, %%xmm0 \n\t" \     \\ (-a.im*b.im,a.im*b.re)
"addpd %%xmm0, %%xmm3 \n\t" \
: \
: \
"m" (Real(a)), \
"m" (Imag(a)), \
"m" (sign)
```

# SSE2 example 2

Complex multiply $a * b$

- `shufpd` swaps upper and lower double
- `xorpd` performs bitwise xor operation with mask: `int sign[4] = {0x0,0x80000000,0x0,0x0};`

# Hiding Latency

- Every operation needs a certain number of cycles to finish
  - `movpd`: 2 cycles
  - `addpd`: 5 cycles
  - ...
- Perform several independent operations on several, different `MMX` registers

```
...
"mulpd %%xmm6, %%xmm3 \n\t" \
"mulpd %%xmm6, %%xmm4 \n\t" \
"mulpd %%xmm6, %%xmm5 \n\t" \
"mulpd %%xmm7, %%xmm0 \n\t" \
...
```

# SSE2: Remarks

- all data must be aligned, see processor specific documentation
- avoid too many load and store operations
- hand-tuned SSE2 code gives up to a factor 2 better performance
- single precision instructions can give another factor of 2
- SSE3 makes complex arithmetics easier
  - gives another $\approx 30\%$ improvement

# Documentation can be found...

Intel:
`http://www.intel.com/products/processor/manuals/`

AMD:
`http://www.amd.com/us-en/Processors/TechnicalResources/`
and then `Microprocessor Tech Docs`

GNU GCC info pages and manuals

See also one of the various SSE2 implementations of the Dirac operator

# Memory and Cache Management

Cache is significantly faster than main memory

- data should be in the cache when needed for computations
- if possible data in the cache should be reused as often as possible
- load and store operations should be reduced as much as possible
  - to reduce memory traffic

# Manual Prefetching

- data can be prefetched manually into the cache
- prefetching is done in parallel to computations
- cache is organised in cache lines
  - prefetch operation will always prefetch a whole cache line

# Prefetch Example

Prefetch for one spinor with cache line 128 Byte long (P4)

```
__asm__ __volatile__ (
  "prefetcht0 %0 \n\t" \
  "prefetcht0 %1" \
  : \
  : \
  "m" (*(((char*)(((ULI)(addr))&~0x7f)))), \
  "m" (*(((char*)(((ULI)(addr))&~0x7f))+128))
)
```

- cache line length depends on CPU type
- `ULI` is here a shortcut for `unsigned long int`

# Prefetch Distance

- prefetch distance: how far ahead to prefetch?
- prefetch distance must be long enough for data being in cache when needed
- in fact: SSE2 performance depends strongly on prefetch distance
- optimal distance must be tried out
  - gauge field about one $\mu$ value in advance
  - spinor field also about one $\mu$ value in advance, after gauge

# Increase Data Re-usage

- e.g.: one application of Dirac operator:
  - every gauge field is used twice
  - every space-time element of source spinor is used 9 times
- possible solution: divide lattice in small blocks and process the blocks one after the other
  - so called strip-mining
  - can significantly improve cache hit rate.

# Prefetch Streams

- if memory is read in successive order the processor can enable hardware prefetching
- usually each processor supports several independent prefetch streams
- example:
  - Dirac operator is often applied on the same gauge background many times (e.g. in iterative solver)
  - make a (double) copy of gauge background in the same order as used in the Dirac operator
  - very easy to implement
  - down side: more memory needed

# Single Precision Acceleration

- single precision arithmetic is about a factor of 2 faster on 32 Bit architectures
- single precision can be used in various places
  - in certain stages of iterative solvers
  - in the HMC for preconditioning
  - in the HMC force computation
  - care must be taken in the HMC: reversibility violations!
- on 64-Bit architectures other tricks possible, see later

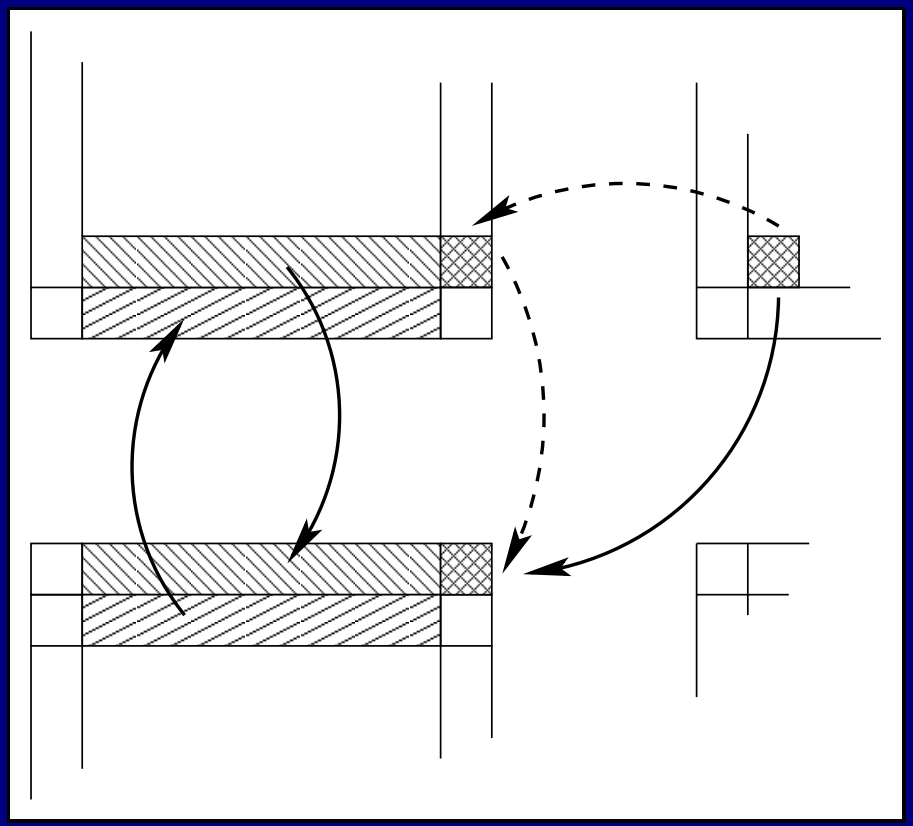
# Outline

# Parallelisation with MPI

- Divide global lattice into equally sized sub-lattices
- every sub-lattice is assigned to one MPI process
- sub-lattices do overlap (boundary fields)
- the local Dirac operator acts as follows:

$$\begin{pmatrix} I_{0,0} & \dots & I_{0,V-1} & c_0 \\ I_{1,0} & \dots & I_{1,V-1} & c_1 \\ . & \dots & . & . \\ I_{V-1,0} & \dots & I_{V-1,V-1} & c_{V-1} \\ 0 & \dots & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} I_0 \\ I_1 \\ . \\ I_{V-1} \\ b \end{pmatrix}$$

# Parallelisation



- Every proc. holds local lattice and boundary
- Every proc. sends inner boundary and receives outer boundary
- most can be done with next-neighbour communication

# Parallelisation

- PC cluster usually have two network types:
  - inter-node: shared memory
  - extra-node: some (hopefully fast) network
- shared memory communication is (should be) faster
- the MPI driver should know about this
- therefore: let MPI set-up the processor grid
  - Use `MPI_Cart_create`
  - neighbouring processes can be obtained with `MPI_Cart_shift`

# Parallelisation: Standard Approach

- add boundary fields at “end” of local gauge and spinor fields
- modify lookup tables to deal with boundaries (instead of periodic boundary conditions)
- Dirac operator essentially unchanged as compared to the serial code
- one Dirac operator application:
  1. exchange gauge and spinor fields
  2. apply “serial” Dirac operator

# MPI Performance

- Communication time depends on:
  1. latency of the network
  2. bandwidth of the network
- latency is paid per transfer once
  it is important in small sized data is communicated often
- time for data transfer depends on size of transfered data
  is limiting if much data is communicated at once

# Hiding Communication

- MPI supports non-blocking communication
- therefore: communication can overlap with computation
- you can reorganise data such that e.g.
  1. inner boundary is processed
  2. communication is initialised
  3. bulk lattice is processed
  4. finish communication
- problem: some MPI implementations do not implement this

# Half Spinor Communication

- Dirac operator times vector $\chi = D\,\psi$ in three steps
  1. project to a half Spinor $\phi$:

$$\phi = P^{4\to 2}\psi$$

  2. exchange $\phi$
  3. expand to full Spinor:

$$\chi = P^{2\to 4}\phi$$

- Multiplication with $U$'s can be done before or after communication (not shown)

# Half Spinor Communication

- amount of data to be communicated is halfed
- latency of network stays the same (of course)
- requires more load and store operations
- will pay off if boundary to bulk ratio becomes too large

# Single Precision $\phi$

- reduced precision: $\phi$ only single precision. All the rest in double.
  - reduces memory traffic significantly
  - halves network traffic
- works also for serial code
- again: take care in the HMC

# Exploiting Bi-directional Bandwidth

- fast networks usually provide bi-directional network bandwidth
- if two processors share the same network card
  - one processor can send while the other one receives data
  - requires knowledge about processor grid
    - you might need to influence this manually

# Summary

- Tuning serial code
  - SSE/SSE2 improvement for PC's
  - prefetching
  - data re-usage
  - single precision acceleration
- Tuning parallel performance
  - overlap communication and computation
  - "half spinor" construction
  - single precision

Happy implementing ...!