Programming of Supercomputers
1st Assignment

Anca Berariu
berariu@in.tum.de

18.10.2013

Outline

1. About this lab course

2. Fire benchmark

3. SuperMUC at LRZ

4. 1st assignment

Outline

1. About this lab course
2. Fire benchmark
3. SuperMUC at LRZ
4. 1st assignment

Programming of Supercomputers (PoS13) Lab

- Every other week on Friday
  - Time: 13:30 – 15:00
  - Dates: 18 Oct. 2013, **25 Oct. 2013**, 15 Nov. 2013, 29 Nov. 2013 & 13 Dec. 2013
  - Room: MI 01.06.020
  - Final Presentations (20 min.): 31 Jan. 2014, 10:00 o'clock
  - Office Hours: Tuesday, 10:30 – 12:00
- Registration in TUMonline
- News, source code and assignments
  www.lrr.in.tum.de/~berariu/teaching/superprog1314.php

Programming of Supercomputers (PoS13) Lab

- Last semester: Introduction to Parallel Programming
  - Theoretical background about OpenMP and MPI
  - Tutorials using small exercises covering the basic usage
  - Team work allowed
  - Guided problem solving

- This semester: Programming of Supercomputers
  - Application of the gained knowledge on a single simulation code
  - Project-based format – more involved and autonomous work
  - Teams of 2 students: code together, tune and report individually
  - „Inter-teams“ submissions lead to course failure

PoS13: Assignments

- 1st assignment– Sequential optimization (30%)
  - Getting to know the application & the **coding guidelines**
  - Single-core compiler-based optimization
  - IO effects on performance
  - Visualization of results with ParaView
- 2nd assignment – MPI Parallelization (65%)
  - Milestone 1: Data Distribution
  - Milestone 2: Communication Model
  - Milestone 3: Parallelization using MPI
  - Milestone 4: Performance analysis and tuning
- Final report and presentation (5%)
  - Report on modeling, implementation and performance tuning results
  - 15 min. presentation + 10 min. Q&A session

PoS13: Submission

- Deadlines: 2nd Friday after each presentation @ 08:00 CET
- Plan for unscheduled maintenances & overbooked job queues
  http://www.lrz.de/services/compute/supermuc/
- Commit all required files to the Git repository
  - separate repository for each team with restricted access for every student
  - about Git:
    http://git-scm.com/book/en/Git-Basics-Getting-a-Git-Repository
  - address:
    http://periscope.in.tum.de/lectures/IN2190/teamXX
  - username/pass – per email after registration
- Check-in as often as you need and use meaningful commit messages

PoS13: Grading

- Each team member receives an **individual grade**
- Maximum points for each assignment: 100
- Contribution of the separate assignments:
  - Assignment 1: 30%
  - Assignment 2: 65%
    - Milestone 1 - Data distribution: 15 points
    - Milestone 2 - Communication model: 15 points
    - Milestone 3 - MPI Implementation: 15 points
    - Milestone 4 - Performance measurements and tuning: 25 points
    - Final Report: 30 points
  - Final Presentation: 5%
- Minimum points to pass: 50
- Both assignment 1 and 2 are required to pass!

PoS13: Coding Guidelines

- Special guidelines for the coding style
  - One uniform format and look
  - Higher quality code & less misunderstandings
- Current version based on industry standards
  - Google C++ Style Guide
  - id Software Coding Style
- Not following these guidelines leads to negative points and could contribute to failing the lab course

- Full version of the Coding Guidelines:
  http://www.lrr.in.tum.de/~berariu/teaching/res/pos1314/PoS1314_CodingStyle.pdf

PoS13: Coding Guidelines (1)

- Indentation
  - 4 spaces per level
  - never use tabs (configure your editor to emit 4 spaces on a Tab)
- Each code line should be at most 100 characters long
  - break up longer lines.
- Use whitespace liberally within expressions, but only one space at a time
  - 1 space before and after all operators and after every comma
- Non-ASCII characters should be rare and must use UTF-8 formatting

PoS13: Coding Guidelines (2)

- Function names
  - should be descriptive
  - use a “command” verb, or a verb followed by noun
  - avoid starting function names with nouns
  - start with a lower case:

```
void function( int arg1, double r );
```

  - for multi-word function names, all words should be lower case with underscores (_) between words

```
void do_something_function( void );
```

- Variable names – follow the function name rules

PoS13: Coding Guidelines (3)

- Function calls
  - on one line if it fits
  - otherwise, wrap arguments at the parenthesis

- Function length and contents
  - use single empty lines to add clarity where necessary
  - never have more than one consecutive blank line
  - each function should be a coherent unit of work
  - no longer than 50 lines / 1 page of code - factor out common code
  - avoid having more than 5 nesting levels

PoS13: Coding Guidelines (4)

- Use a Doxygen compliant header for all functions

```
/**
* Calculate the distribution factor of a dataset.
*
* @param nintci start element
* @param nintcf end element
* @param cgup data array for which a factor is calculated
*
* @return distribution factor
*/
double calc_distribution_factor( int nintci,
                                 int nintcf,
                                 double* cgup );
```

Doxygen Manual: www.doxygen.org

PoS13: Coding Guidelines (5)

- Use trailing braces everywhere: if, else, functions, structures, typedefs, class definitions, etc.
- Else statement starts on the same line as the last closing brace

```
if ( x ) {
    // ...
} else {
    // ...
}
```

- Pad parenthesized expressions with spaces

```
x = ( y * z );
```

- Always indent the following
  - Statements within function body
  - Statements within blocks
  - Statements within ‘case’ body of switch operators

PoS13: Coding Guidelines (6)

- Comments
  - should be brief
  - should explain WHY instead of HOW
    - do not restate code in the comments
    - give reasons
    - why a particular algorithm was chosen
    - why a particular data structure is used
    - why a certain action must be taken

Bad Example

```
// set product to "base"
product = base;
// loop from 2 to "num"
for ( int i = 2; i <= num; i++ ) {
    // multiply "base" by "product"
    product = product * base;
}
```

Good Example

```
// compute the square root of num using
// Newton-Raphson approximation
r = num / 2;
while ( abs( r - (num/r) ) > EPSILON ) {
    r = 0.5 * ( r + (num/r) );
}
```

Outline

1. About this lab course

2. Fire benchmark

3. SuperMUC at LRZ

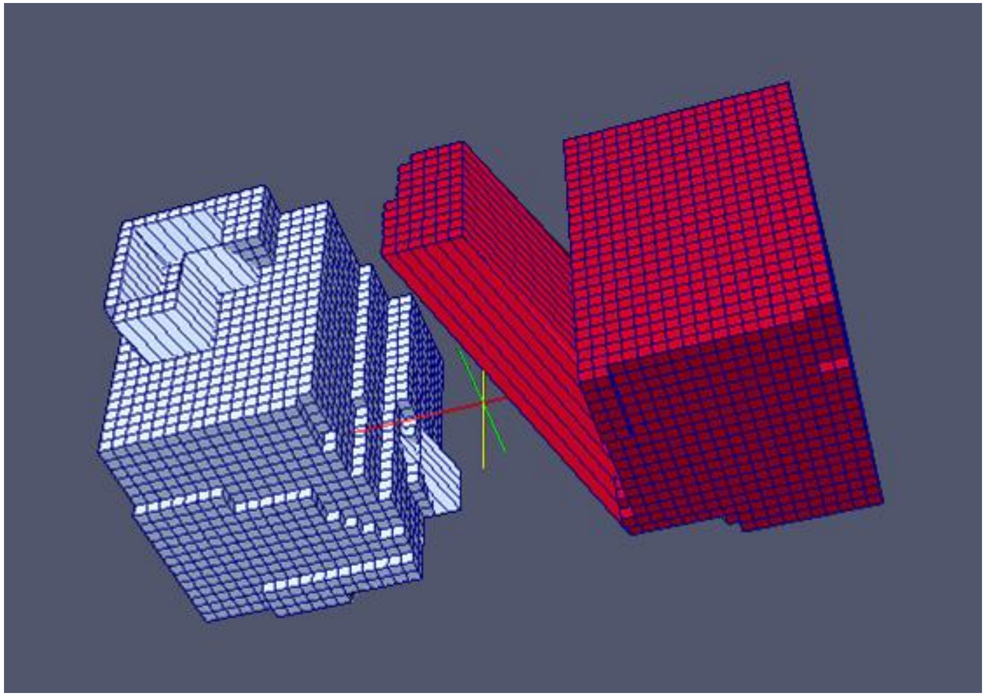
4. 1st assignment

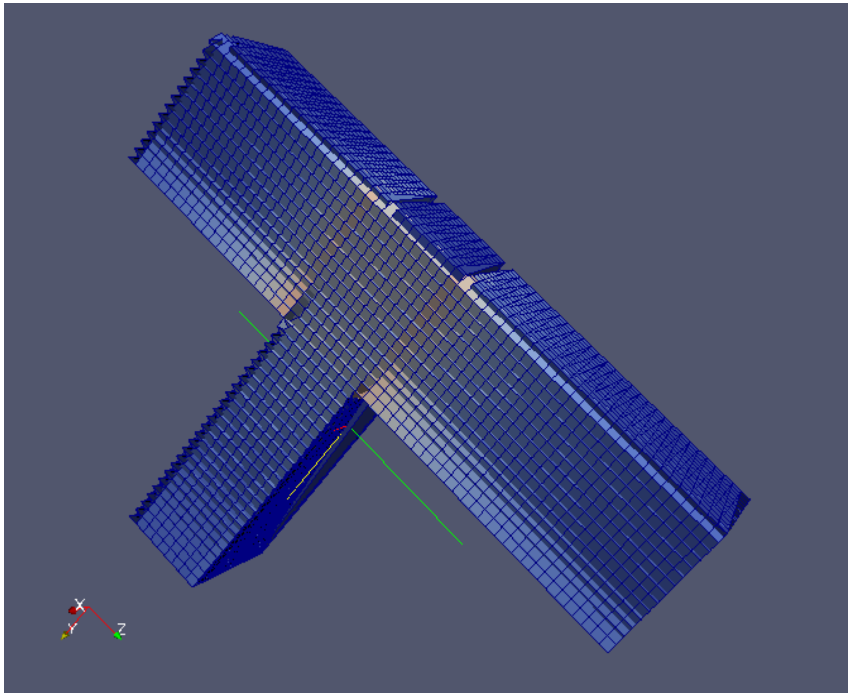
Fire Benchmark

Two- or three-dimensional (un-)steady simulations of flow and heat transfer within arbitrarily complex geometries with moving or fixed boundaries

- Computational Fluid Dynamics (CFD) solver framework for arbitrary geometries
- Developed by AVL LIST GmbH, Graz, Austria
- Written in C
  - main computational function is only 350 lines
  - two extra files for data import and export
- Black-box approach
  - do not spend time on understanding the physics behind
  - concentrate on the performance and optimization and not the theory!!

Fire Benchmark - Geometries





Fire Benchmark - GCCG

- GCCG – generalized orthomin solver with diagonal scaling

- Linearized Continuity Equation

$$A_p \varphi_p = \sum_{c=E,S,N,\ldots} A_c \varphi_c + S_\varphi$$

given

– source value $S_\varphi \rightarrow SU$

– boundary cell coefficients $A_c \rightarrow BE, BS, \ldots$

– boundary pole coefficients $A_p \rightarrow BP$

wanted

– variation vector/flow to be transported $\varphi_p \rightarrow VAR$

Fire Benchmark - GCCG

- Domain discretisation in volume cells
- Unstructured grid with neighboring information (LCC) and indirect addressing
- Internal and external (ghost) cells
- Iterate until acceptable residual achieved
  - Phase 1: compute the new *directi*onal values from the old ones
  - Phase 2:
    - normalize and update values
    - compute new residual

- More details with the 2nd assignment

Outline

1. About this lab course
2. Fire benchmark
3. SuperMUC at LRZ
4. 1st assignment

SuperMUC @ Leibniz Supercomputer Centre

Movie on YouTube

SuperMUC – Peak Performance

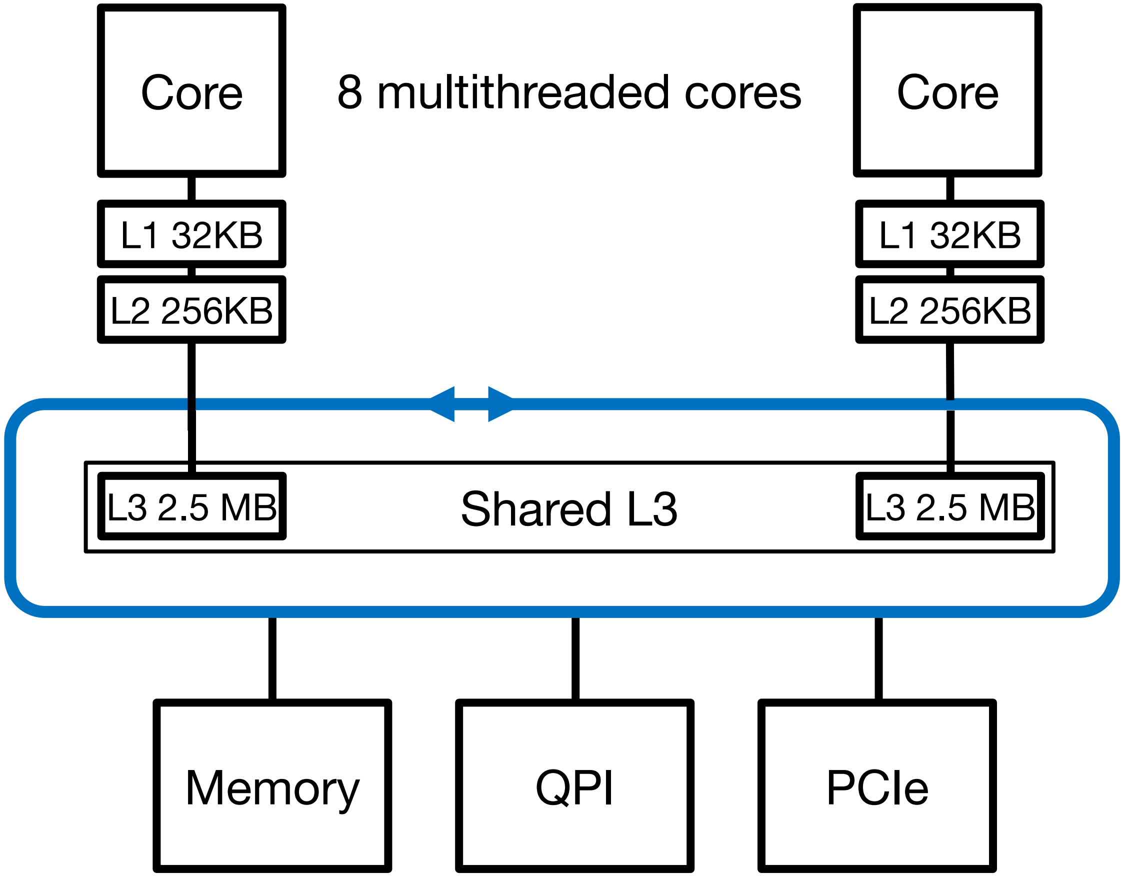
- Peak performance: 3 Peta Flops = $3*10^{15}$ Flops
  - Mega 10^6 million
  - Giga 10^9 billion
  - Tera 10^{12} trillion
  - Peta 10^{15} quadrillion
  - Exa 10^{18} quintillion
  - Zetta 10^{21} sextillion
- Flops: Floating Point Operations per Second

SuperMUC – Distributed Memory Architecture

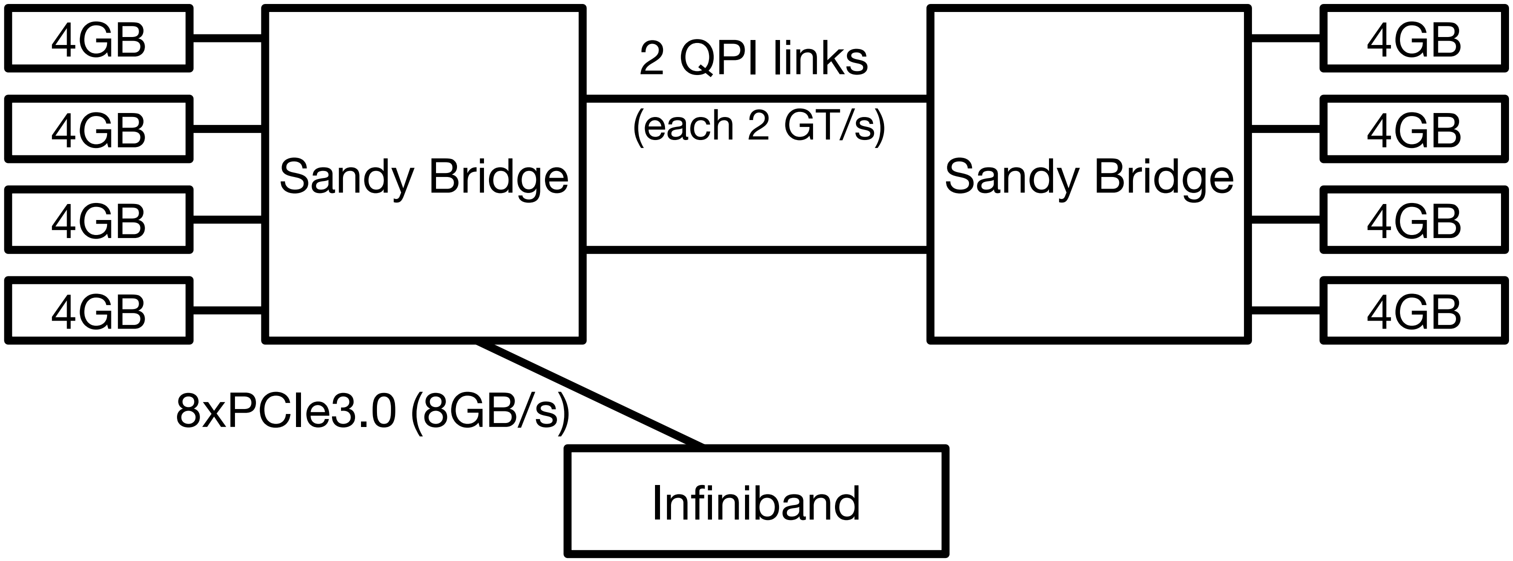
- 18 partitions (*islands*) with 512 nodes each
- One node is a shared memory system with 2 processors
  - Sandy Bridge-EP Intel Xeon E5-2680 8C
    - 2.7 GHz (Turbo 3.5 GHz)
  - 32 GByte memory
  - Inifiniband network interface
- Each processor has 8 cores
  - 2-way hyperthreading
  - 21.6 GFlops @ 2.7 GHz per core
  - 172.8 GFlops per processor

Sandy Bridge Processor

Latency:
- 4 cycles
- 12 cycles
- 31 cycles



SuperMUC – NUMA Node



- 2 processors with 32 GB of memory
- Aggregate memory bandwidth per node 102.4 GB/s
- Latency
  - local ~50ns (~135 cycles @2.7 GHz)
  - remote ~90ns (~240 cycles)

SuperMUC – Access

- Accounts per Email after TUMOnline registration

  first of all, change your password by visiting the ID-Portal of LRZ:
  http://idportal.lrz.de/r/entry.pl?Sprache=en

- SSH-only access (login / data transfer):

  connection only allowed from trusted DNS (e.g. lxhalle)

```
ssh -Y <username>@supermuc.lrz.de
```

- Details and info:

  http://www.lrz.de/services/supermuc/access_and_login/

SuperMUC – Job scheduling

- LoadLeveler batch system
  http://www.lrz.de/services/compute/supermuc/loadleveler/

  - build a job command file – plain text file
  - submit with `llsubmit`
  - check status with `llq`

- Interactive jobs
  - used in general for testing
  - have limited resources

- Never run measurements on the login node

Outline

1. About this lab course
2. Fire benchmark
3. SuperMUC at LRZ
4. 1st assignment

1st assignment

- General facts
  - Get to know the machine you are using
  - Reproducible results – at least 3 runs for each configuration
  - Code instrumentation using the PAPI hw counters library
- Different runtime behavior in different application phases
  - Initialization: read input data files
  - Computation: efficient usage of resources
  - Finalization: output the results
- Carry out performance experiments using different compiler optimization flags
- Metrics: execution time, MFlops, L2/L3 cache miss rate

PAPI Instrumentation

- Library for accessing the performance counter hardware on microprocessors
  - main website: http://icl.cs.utk.edu/papi/
  - User's Guide: http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_USER_GUIDE_23.htm
- Requires user instrumentation of applications
- Available on SuperMUC: `module load papi`
- Supported events and counters: `papi_avail`
  - check which counters you can use on SuperMUC
- High-Level API vs. Low-Level API

PAPI Instrumentation – High-Level API HW Counters

```
#include <papi.h>
#define NUM_EVENTS 2

void main( ) {
    int Events[NUM_EVENTS] = { PAPI_TOT_INS, PAPI_TOT_CYC };
    long_long values[NUM_EVENTS];

    // Start counting events
    if ( PAPI_start_counters( Events, NUM_EVENTS ) != PAPI_OK ) handle_error( 1 );

    // Do some computation here

    // Read the counters
    if ( PAPI_read_counters( values, NUM_EVENTS ) != PAPI_OK ) handle_error( 1 );

    // Do some more computation here

    // Read again the counters and stop counting events
    if ( PAPI_stop_counters( values, NUM_EVENTS ) != PAPI_OK ) handle_error( 1 );
}
```

PAPI Instrumentation – Low-Level API HW Counters

```
int EventSet = PAPI_NULL;
if ( PAPI_library_init( PAPI_VER_CURRENT ) != PAPI_VER_CURRENT ) exit(1);

// Create an EventSet
if ( PAPI_create_eventset( &EventSet ) != PAPI_OK ) handle_error( 1 );

// Add Total Instructions Executed to the EventSet
if ( PAPI_add_event( &EventSet, PAPI_TOT_INS ) != PAPI_OK) handle_error(1);

// Start counting
if ( PAPI_start( EventSet ) != PAPI_OK) handle_error(1);

// Do some computation here

// Read the counters
if ( PAPI_read( values ) != PAPI_OK ) handle_error( 1 );

// Read again the counters and stop counting events
if ( PAPI_stop( EventSet, values ) != PAPI_OK ) handle_error( 1 );
```

PAPI Instrumentation – Timers

```
long_long start_cycles, end_cycles, start_usec, end_usec;

if ( PAPI_library_init( PAPI_VER_CURRENT ) != PAPI_VER_CURRENT ) exit(1);

start_cycles = PAPI_get_real_cyc(); // Gets the starting time in clock cycles
start_usec = PAPI_get_real_usec(); // Gets the starting time in microseconds

// Do some computation here

end_cycles = PAPI_get_real_cyc(); // Gets the ending time in clock cycles
end_usec = PAPI_get_real_usec(); // Gets the ending time in microseconds

printf ( "Wall clock time in usecs: %lld\n", end_usec - start_usec );
```

I/O – ASCII vs. Binary Data Files

- Change initial data format: ASCII → binary
- Compare execution time in both cases
- Analyze storage space
- Discuss the differences

Visualization with ParaView

- ParaView visualization software
  - open-source product: www.paraview.org
  - load the module on SuperMUC

    ```
    module load paraview
    ```
  - you can download & install it locally on your computer

- Uses VTK file format
  - use the supplied functions to convert the data prior to export
  - export the vector values using the provided function

- Visualize the resulting VTK files for tjunc.dat for the VAR, CGUP and SU arrays

- Store the images in jpeg format

Submission

- Deadline: **1. Nov 2012 @ 08:00 CET**
- Plan for unscheduled maintenances & overbooked job queues!!!
- Choose a team-mate until Monday, 21st Oct. and announce your group at `berariu@in.tum.de`
- Git repositories are prepared for each group and access details will be sent via email.
- Repository structure:
  - Folder A1/code/ : *.c, *.h, Makefile
  - Folder A1/data/ : Data.ods /.xlsx
  - Folder A1/report/ : Report.pdf
  - Folder A1/plots/ :
    - SU.jpeg & SU.vtk (input: tjunc.dat)
    - VAR.jpeg & VAR.vtk (input: tjunc.dat)
    - CGUP.jpeg & CGUP.vtk (input: tjunc.dat)

Thank You

and good luck with your first assignment!