Motivation

Why are we looking into these model?

- Concurrent programming is the future (no choice)
- Parallel programming was not mainstream in the last decades
  - There is no (good) support for parallel programming in current programming languages
  - Only in domain specific areas (e.g. HPC) research for parallel programming
  - Concepts and approaches of domain specific areas are most of the time not suitable for general purpose programming

Glimpse at the future

"HPC systems are about 20 year ahead of mainstream systems"
Motivation

http://www.top500.org
Motivation
Motivation

Motivation

http://www.top500.org
Motivation

What are we looking for?
- What are the current/proposed solutions out there?
  - Are the solutions domain specific or general purpose?
  - What are the short comings and limitations?
  - Is there a consensus of features that seems to solve the issues?

How to look at it?
From current standards to proposed solutions
MPI - Overview

What is MPI?

- API for writing parallel applications
- de-facto (industry) standard in HPC (High Performance Computing)
- API defined for C, C++ and Fortran
- widely deployed, support for basically all type of machines (dual-core → Cray → ”Roadrunner”)
- http://www.mpi-forum.org
MPI - Features

MPI Feature List

- Message Passing
  - P2P communication
    - blocking
    - non-blocking
  - Collective communication (broadcast, scatter, reduce)
- Parallel I/O
- Remote Memory Access
- build-in data conversion for heterogeneous setups
MPI - Example

```c
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank = 0, size = 0;
    short int recv = 0;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    switch (rank) {
    case 0:
        MPI_Send(&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&recv, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
        break;
    case 1:
        MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Recv(&recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        break;
    default:
        break;
    }

    MPI_Finalize();
    return 0;
}
```
### MPI - Example

**[BROKEN]**

<table>
<thead>
<tr>
<th>BROKEN 1</th>
<th>Broker</th>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MPI_Send(…)</td>
<td>MPI_Send(…)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_Recv(…)</td>
<td>MPI_Recv(…)</td>
</tr>
</tbody>
</table>

**[BROKEN 2]**

What happens if the program is started with just 1 process?

**[BROKEN 3]**

The `recv` variable is of type `short` → **buffer overflow**.
MPI - Example

Will this program run or not?

- **It runs:**
  If the implementation uses an *eager protocol* then the send call will act non-blocking (common case for small messages)

- **It hangs:**
  The implementation uses a *rendezvous protocol* to and each process is waiting for the corresponding receive (common case for bigger messages)

- **It crash:**
  A restrictive compiler/system could detect the stack overflow and cause the application to segfault.
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank = 0, size = 0, recv;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (size >= 2) {
        switch (rank) {
        case 0:
            MPI_Send(&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
            MPI_Recv(&recv, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
            break;
        case 1:
            MPI_Recv(&recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
            break;
        default:
            break;
        }
    }

    MPI_Finalize();
    return 0;
}
**MPI - Issues**

**Pro**
- standard API
- support for basically almost every system out there

**Contra**
- very low level primitives
- no real resource management
- user responsible for synchronisation and data distribution
- too complicated (> 200 methods are defined → 6 function MPI common case)
- wrong programs may work to some degree (e.g. different system, different problem size)
- basically all bigger MPI programs have bugs
OpenMP Overview

What is OpenMP?

- industry standard for shared memory parallelism
- defined for C, C++ and Fortran
- 'annotations' for the compiler (e.g. gcc >= 4.2)
- http://www.openmp.org
OpenMP - Features

- transparently implemented via `#pragmas` or comments
- specification of parallel tasks
  - sections
  - for
  - tasks (new in version 3.0)
- synchronisation
  - master
  - barrier
  - taskwait
  - flush (consistent memory view)
  - atomics
for (init-expr; test-expr; incr-expr) structured-block

init-expr  One of the following:
            var = lb
            integer-type var = lb
            random-access-iterator-type var = lb
            pointer-type var = lb

test-expr  One of the following:
            var relational-op b
            b relational-op var

incr-expr  One of the following:
            ++var
            var++
            --var
            var--
            var += incr
            var -= incr
            var = var + incr
            var = incr + var
            var = var - incr
Example

Parallel Sections

```c
#pragma omp parallel sections ...
{
    #pragma omp section ...
        {
            ... 
        }
    #pragma omp section ...
        {
            ... 
        }
    ... 
}
```

Parallel For Loop

```c
#pragma omp parallel for ...
for ( i = 0 ; i < N ; i++ ) {
    ...
}
```

Task

```c
#pragma omp parallel task ...
```
OpenMP - Example

```c
static const int N = 100;

int main(int argc, char *argv[]) {
    int x, y;
    double * restrict a = NULL;
    double * restrict B = NULL;
    double * restrict c = NULL;

    a = (double*) malloc(N * sizeof(double));
    B = (double*) malloc(N*N * sizeof(double));
    c = (double*) malloc(N * sizeof(double));

    // ... init data

    /* compute */
    #pragma omp parallel for default(none) shared(a,B,c) private(x,y)
    for (y = 0; y < N; y++) {
        for (x = 0; x < N; x++) {
            c[y] += a[y] * B[x+y*N];
        }
    }

    for (y = 0; y < N; y++) {
        printf("%f\n", c[y]);
    }

    free(a); free(B); free(c);
    return 0;
}
```
## OpenMP - Issues

### Pro
- incremental parallelization approach
- single code base (debugging of sequential version)
- easy to use for structured problems (e.g. loops)
- allows fine tuning off concurrent execution

### Contra
- mainly focused on loop parallelism
- user has still main responsibility for synchronisation
- some features are implementation specific (e.g. nested parallelism)
Overview

- LINQ = Language Integrated Query
- developed by Microsoft for the .NET platform (C#, VB, ...)
- 'SQL'ish style for manipulating data sets (e.g. collections, XML, databases)
- PLINQ is a parallel implementation of LINQ based on TPL
LINQ - Features

Features

- everything that implements `IEnumerable<T>` can be used inside a query
- automatic type inference

Supported Commands

`from ... in ...` selecting object from "collection"
- `where ...` specifies selection condition
- `select ...` specifies which what to return from the query
- `orderby ...` specifies by which property the result should be sorted
- `join ... on ...` merge several sequences
- `group ... by ...` apply operation only to group of objects
LINQ - Example

LINQ before transformation

```csharp
var Found = from o in Orders
            where o.CustomerID == 84
            select o.Cost;
```

LINQ after transformation

```csharp
var Found = Orders.Where(o => o.CustomerID == 84).Select(o => o.Cost);
```
LINQ - Example

using System.Collections.Generic;
using System.Linq;

class Data {
    public int id;
    public string name;
    public int age;
};

public class Simple {
    public static int Main(string[] args) {
        var objs = new List<Data> {
            new Data {id=1, name="Hans", age=12},
            ...
        };

        var result = from o in objs
                      where o.name == args[0] && o.age >= 21
                      orderby o.age ascending
                      select new {o.name, o.age};

        foreach (var o in result) {
            System.Console.WriteLine(o);
        }

        return 0;
    }
};
LINQ - Issues

Pro
- declarative versus imperative
- backend independent
- allows automatically parallelization
- async. and 'on-the-fly' implementations

Contra
- are automatic/anonymous types good or bad?
Erlang - Overview

Overview

- functional programming language
- message passing based concurrency model
- originally developed by Ericson for developing telecom software (basically 24/7 requirement)
- has been released as open source
- some usage: Wings 3D, the chat system of Facebook
- http://www.erlang.org

^http://www.facebook.com/notes.php?id=9445547199
Erlang - Features

Features

- functional programming language
- 'hot code’ swapping (replace code in running program)
- designed for concurrency
  - message passing is 1st class citizen
  - support for SMP and distributed computing
- FT support (if one 'erlang’ process dies simply restart this one)
Erlang - Features

**Process creation**

New process are created via the spawn function:

\[
\langle \text{PID} \rangle = \text{spawn} (\langle \text{MODULE} \rangle, \langle \text{FUNC} \rangle, \langle \text{PARAMETERS} \rangle)
\]

**Message sending**

For sending message the object that is 'piped' into the PID via the '!' operator:

\[
\text{Pong}_{-}\text{PID} \; ! \; \{ \text{ping} \; , \; \text{N} \}
\]
Erlang - Features

Message reception

Message as handled by a receive block ("switch" statement + pattern matching)

```
receive
    pattern1 ->
        actions1;
    pattern2 ->
        actions2;
    ....
    patternN
        actionsN
end.
```
Erlang - Example

```erlang
-module(tut15).
-export([start/0, ping/2, pong/0]).

ping (0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("Ping_Finished\n", []).

ping (N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping_received_pong\n", [])
            end,
    ping(N - 1, Pong_PID).

pong () ->
    receive
        finished ->
            io:format("Pong_Finished\n", []);
        {ping, Pong_PID} ->
            io:format("Pong_received_ping\n", []),
            Pong_PID ! pong,
            pong()
            end.

start () ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [10, Pong_PID]).
```

Erlang - Issues

**Pro**
- functional + message passing avoids 'shared state'
- lightweight process model
- supports running distributed across multiple machines
- transparent transmission of any Erlang expression

*local state is still possible*

**Contra**
- sometimes shared state is good to have
- functional programming can be awkward
Overview

- TBB - Threading Building Blocks
- C++ library for multi-core programming
- Originally developed by Intel and later open sourced (most interesting part is missing)
- [http://www.threadingbuildingblocks.org](http://www.threadingbuildingblocks.org)
TBB - Features

Features

- parallel_for
- parallel_reduce
- auto partitioner (not in OSE)
- pipeline support
- containers (e.g. concurrent hashmap)
- low level wrapper (e.g. atomic operations, locks)
- memory manager
/ ... skipped includes

using namespace tbb;

class HelloWorld {
    public:
        protected char *name;
    // shared between all clones

    HelloWorld(char * name) { this->name = name; }

    void operator() (const blocked_range<size_t>& r) const {
        std::cout << name << "["
            << r.begin() << ","  
            << r.end() << "]" << std::endl;

        // create some artificial load
        int i = 500000;
        while (i-- != 0) { continue; }
    }
};

int main(int argc, char* argv[]) {
    task_scheduler_init init;

    parallel_for(blocked_range<size_t>(0, 10000, 100), HelloWorld("foo"));
    return 0;
}
TBB - Issues

Pro

- fits into object oriented approach
- library approach is portable and don’t require changes of the programming language
- no inheritance from a specific base class required

Contra

- it’s up to the user to come up with an non-overlapping implementation → CHEATING
- can cause lot of object copies
- the user is still responsible for the sync. (or the avoidance of it)
- problems without a ”range” cannot easily be represented
Cilk - Overview

Overview

- research programming language
- task parallel language based on C
- a port for Java has been made
- http://supertech.csail.mit.edu/cilk/
- new startup company who created Cilk++
- http://www.cilk.com/
Cilk - Features

Features

- 3 additional keywords to C programming language
  - cilk
    - marks a function to be a cilk function
  - sync
    - used to wait for previously spawned functions
  - spawn
    - spawn a new function/task off
Cilk - Example

```c
#include <stdio.h>

cilk fib(int n) {
    int left = 0;
    int right = 0;

    if (n <= 1) return 1;

    left = spawn fib(n-1);
    right = fib(n-2);

    sync;

    return left + right;
}

void main(void) {
    int result = 0;
    result = spawn fib(10);
    sync;
    printf("fib(10)=\%i\n", result);
}
```
## Cilk - Issues

<table>
<thead>
<tr>
<th>Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>- allows incremental parallelization of source code</td>
</tr>
<tr>
<td>- single code base for sequential and parallel version</td>
</tr>
<tr>
<td>- it’s composable to some degree</td>
</tr>
<tr>
<td>- work stealing $\rightarrow$ work balancing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contra</th>
</tr>
</thead>
<tbody>
<tr>
<td>- wait is not selective $\rightarrow$ wait always for all previously spawned ’tasks’</td>
</tr>
<tr>
<td>- function call granularity might be limiting</td>
</tr>
</tbody>
</table>
Overview

- part of HPCS (High Productivity Computing Systems)
- developed by IBM
- designed for non-uniform memory machines
- http://x10.sourceforge.net/x10examples.shtml
X10 - Features

X10 almost like Java

**Missing Features**
- concurrency (threads)
- arrays
- built-in types

**Additional Features**
- places (partitioned global address space)
- distributed multi-dimensional arrays
- clocks (barriers)
- activities (tasks)
- atomic blocks (TM)
- sequential for but parallel foreach
Atomic Blocks aka TM

- act like a transaction
  - either the whole result of the block is visible
  - or nothing
- automatic conflict detecting and resolution

atomic {
  ...
}

Create of a 'remote' object

```java
final place Other = here.next();
final T t = new T();

finish async (Other){
    final T t1 = new T();
    async (t) {
        t.val = t1;
    }
}
```

async = async. task
finish async = bocking
X10 - Issues

**Pro**
- distributed global address space allows support optimizing programs for NUMA and distributed memory architectures
- higher level abstractions for

**Contra**
- cumbersome data fetching via multiple activities
- cannot have delayed wait for async. activities
## Conclusion

### Bad "Features"

- explicit representation
  - granularity is fixed
  - implementation fixed
- manual resource management/mapping
- user responsible for synchr.
- user responsible for data transfers
- lack of verification
- too many features (complicated and error prone)
- domain specific designs
- lack of tools
Promising Features

- implicit representation (like LINQ)
- notion of distributed system (X10, Erlang)
- sequential execution opportunity (OpenMP)
- composability (Cilk, OpenMP, LINQ, Erlang)
- automatic resource management (LINQ, OpenMP, Cilk)
- balance defaults $\iff$ configuration (OpenMP)
- incremental parallelization (LINQ, OpenMP, Cilk)
The End

Thanks for the attention! Questions ???