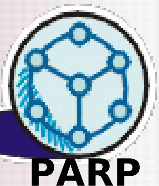
Dpto. Ingeniería y Tecnología de Computadores[1]
Dpto. Informática y Sistemas[2]

University of Murcia (Spain)
Artificial Perception and Pattern Recognition Research Group (PARP)

A design pattern for component oriented development of agent-based multithreaded applications

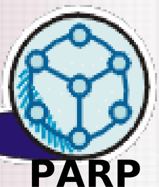
A case study in computer vision

Pedro E. López-de-Teruel[1], A.L. Rodriguez[1],
A. Ruiz[2], G. Garcia-Mateos[2], L. Fernández[1]
pedroe@ditec.um.es, alrl1@alum.es,
aruiz@um.es, ginesgm@um.es, lfmaimo@ditec.um.es



OUTLINE

- Introduction
  - Motivation
  - QVision
  - Outline of the worker pattern
  - Example application
  - Communication between workers
- Detailed pattern description
  - Coding example
- Implementation
- Performance
- Discussion
- References



Introduction

Objectives:

- Design pattern (for recurrent programming problems)
- Extended pipeline pattern:
  - Includes asynchronous communications, and
  - Event driven responses (i.e. GUI)
- Reusability
- Multithreaded (MT) programming *without expertise*
  - Hides efficient data sharing and synchronization issues

Application domain

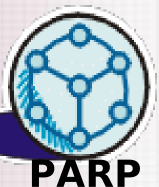
- Coarse grain solution for data flow processing based applications
  - Ideal in (maybe GUI guided) signal processing (i.e. Computer Vision)
- Simple and perhaps too restricted, but…
  - Compatible with more specific MT techniques

QVision (I)

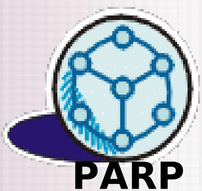
What is it?:

- Fast prototyping library for *real time* computer vision research
- Object oriented framework
- C++, built on Trolltech Qt 4.2
- Easy and homogeneous programming interface to:
  - Powerful and dedicated GUI
  - Support libs & tools: BLAS, LAPACK, GSL, IPP, MPlayer, ...
  - Multicore targeted ← ¡Must be easy to use!

↑ CV researchers are not expert parallel programmers!



QVision (II)



QVision (III)

QVImageCanvas for Canny

QVImageCanvas for MSER Regions

QVGUI

Control: MSER Worker
Iteration: 1135

Control: Contours Extractor W
Iteration: 7019

Control: Corners Worker
Iteration: 7699

Control: Canny Operator Wor
Iteration: 6870

Video

Workers input

MSER Worker

int **Delta** (1,128)
8

double **diffAreaThreshold** (0,1)
0.3

int **maxAreaMSER** (1,1000000)
10000

int **minAreaMSER** (1,10000)
10

Contours Extractor Worker

Corners Worker

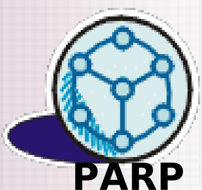
Canny Operator Worker

QVImageCanvas for Contours

Camera control

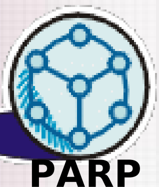
Madrid-Leverkusen.avi

fps: 25 size: 448x608
speed: x 1.00 secs: 73.7/6712.36
Speed 0,00 frames grabbed/read: 1845/1845



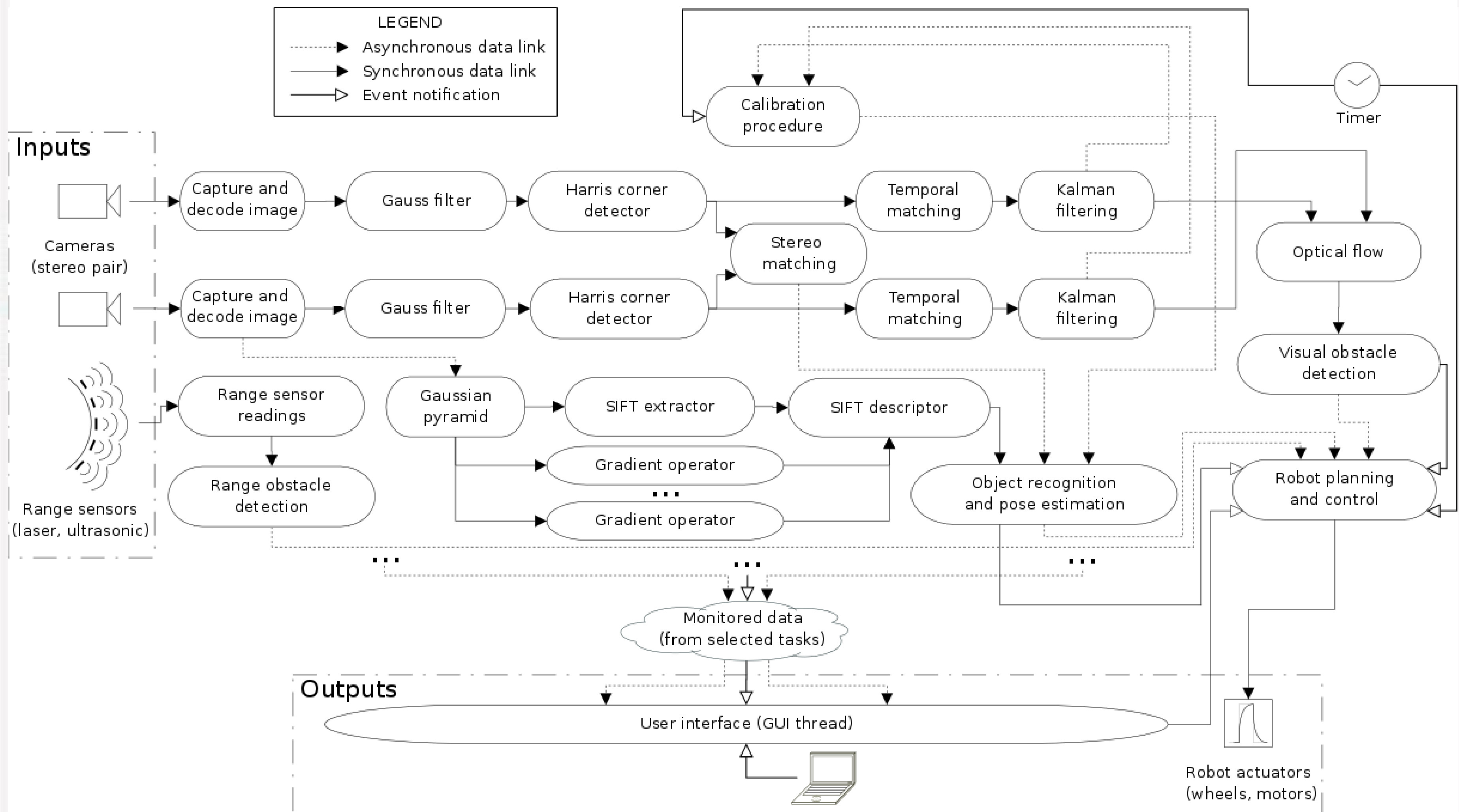
The *Worker* pattern: outline

- **Design pattern**: reusable solution to a commonly occurring problem in software design
  - *Template* to solve a problem that can be used in many different situations
- We extend Mattson/Sanders/Massingill *pipeline* and *event based* patterns
- Task oriented parallelism
  - Semi-independent, encapsulated agents...
  - ... who communicate through well defined I/O interfaces
  - Communication can be synchronous (classic pipeline), asynchronous (at any time) or event based (on demand).



Example application

Visually guided robotic platform:



PARP

DITEC DIS Universidad de Murcia

Communication among workers

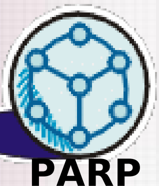
Three kind of links between *workers A* and *B*:

Data

- **Synchronous links**: Output data from iteration *i* of a *worker A* must be always read before starting iteration *i* in *worker B* (serial dependence)
  - Both safe shared data access and strict sequencing must be assured
  - Much like a hardware pipeline
- **Asynchronous links**: Output data from iteration *i* of a *worker A* can be read at any moment by *worker B* (weak dependence)
  - Only safe shared data access needed

Control

- **Event links**: Some condition on worker A *triggers* an iteration of worker B.
  - For example, time controlled, periodic actions...
  - ... or even user guided, GUI triggered tasks



Pattern description (I)

UML schema of the *worker* pattern:

destinationTaskLink
1
+sourceTaskLink
1

DataLink
+source: DataID
+destination: DataID
+link_semantics: LinkType
...
...()

Thread
...
+run()
...()

SharedDataContainer
... structures to store I/O data variables ...: list<Variant>
+getValue(name:DataID): Variant
+setValue(name:DataID,value:Variant)
+addVariable<Type>(name:DataId,IOtype:bool)
#read_input_data()
#write_output_data()
+...()

BaseWorker class reimplements virtual method run() from parent Thread class

+destinationTask
0..*
+sourceTask
0..*

BaseWorker
...
+setupDataVariables()
+linkData(data:DataID,source:Worker,sourceData:DataID)
+run()
+iterate()
...()

Example worker subclasses. They must reimplement iterate() method, and can reimplement initialize() from BaseWorker class

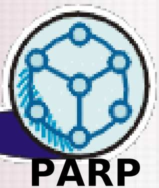
```
method run()
 initialize()
 while(not finished)
  read_input_data()
  iterate()
  write_output_data()
 end while
end method
```

Worker1
...
+iterate()
+setupDataVariables()
+eventTriggeredMethod()
+...()

Worker2
...
+iterate()
+setupDataVariables()
...()

Worker3
...
+iterate()
+setupDataVariables()
+...()

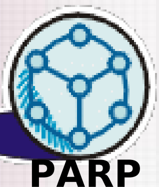
...



Pattern description (II): components

Main class of the pattern: `Worker` class

- A worker is an encapsulated task that iterates forever, computing a well defined set of outputs from inputs
  - Each iteration can be triggered:
    - Continuously, or...
    - ...by an external event (signaled by other worker, or the GUI)
  - Each new `Worker` = component oriented, reusable thread
- All programmer defined workers inherit from `BaseWorker`, and just redefine the `iterate()` method
  - This method justs defines how output is computed from input in each iteration
- No synchronization or safe shared access primitive must be explicitly used by programmer
  - `BaseWorker run()` method (reimplemented from base library `Thread` class) does all the job



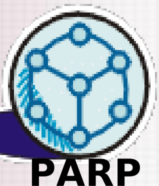
Pattern description (III): components

Input/output: SharedDataContainer class

- Generic class:
  - Internally holds lists of named Variant (=union) objects
  - Programmed using templated methods
  - It allows the final programmer to use any kind of input and output data types...
  - ...while allowing the designers of the framework to work with them without knowing specific types in advance

Usage:

- Programmer just adds I/O parameters of desired types in the constructor of each new Worker class, using the addVariable<T>(...) method, ...
- ... accesses in iterate() with get/readData<T>(...),
- ... and links them to other workers with linkData<T>(...)
- Completely **hides synchronization** to final programmer



Coding example (I)

- Defining a new worker:

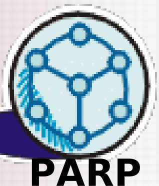
```
class CannyWorker: public QVWorker {
    public:
        CannyWorker(QString name): QVWorker(name) {
            addProperty< QVImage<uChar,1> >("Input image", inputFlag);
            addProperty<double>("Threshold high", inputFlag,150,50,1000);
            addProperty<double>("Threshold low", inputFlag,50,10,500);
            addProperty< QVImage<uChar,1> >("Canny image", outputFlag);
        }

        void iterate() {
            // Read input parameters:
            QVImage<uChar,1> image = getPropertyValue< QVImage<uChar,1> >("Input image")
             [...] // Some needed preprocessing code (type conversions, image gradients, and so on...)
            // Apply Canny operator:
            Canny(dX, dY, canny, buffer, getPropertyValue<double>("Threshold low"),
                                         getPropertyValue<double>("Threshold high"));
            // Publish output images
            setPropertyValue< QVImage<uChar,1> >("Canny image",canny);
        }
}
```

Defining I/O

Reading inputs

Writing outputs



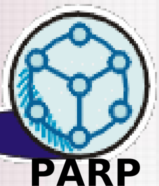
Coding example (II)

- Linking properties among workers:

```
int main(int argc, char *argv[]) {
  // Application object:
  QVApplication app(argc, argv,"Example program for QVision library");
  // Workers:
  ComponentTreeWorker componentTreeWorker("Component Tree");
  CannyWorker cannyWorker("Canny operator");
  ContourPainter contourPainter("Contour painter");
  // Video source(s):
  QVMPlayerCamera camera("Video");
  // GUI elements:
  QVImageCanvas imageCanvas("Rotoscoped image");
  // Links among workers, cameras, and GUI:
  camera.link(&componentTreeWorker,"Input image");
  componentTreeWorker.linkProperty("tree image", &cannyWorker,"Input image",SynchronousLink);
  cannyWorker.linkProperty("Canny image", &contourPainter, "Borders image", SynchronousLink);
  imageCanvas.linkProperty(contourPainter,"Output image",AsynchronousLink);
  [...] // Some more links...
  // Application launch (main event loop execution):
  return app.exec();
}
```

Reusing workers

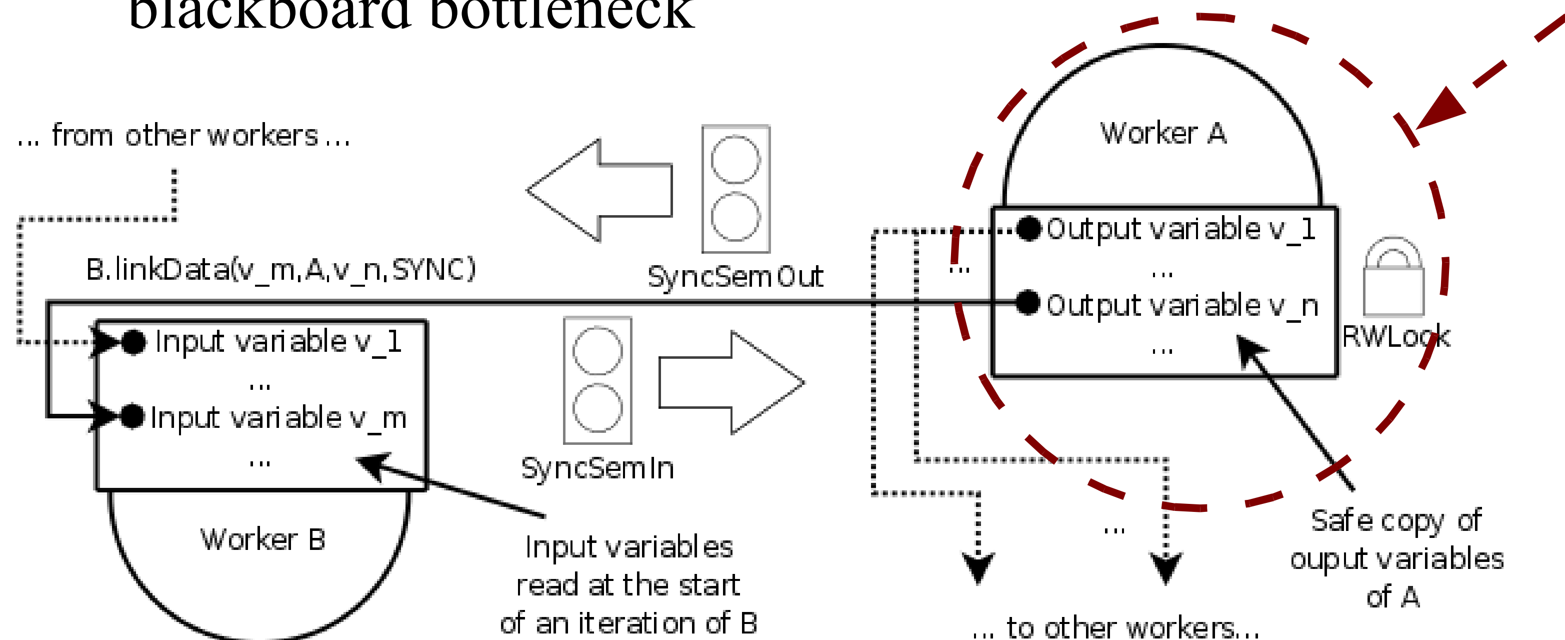
Linking properties among workers



Note that cameras and GUI elements are just like workers...

Implementation (I)

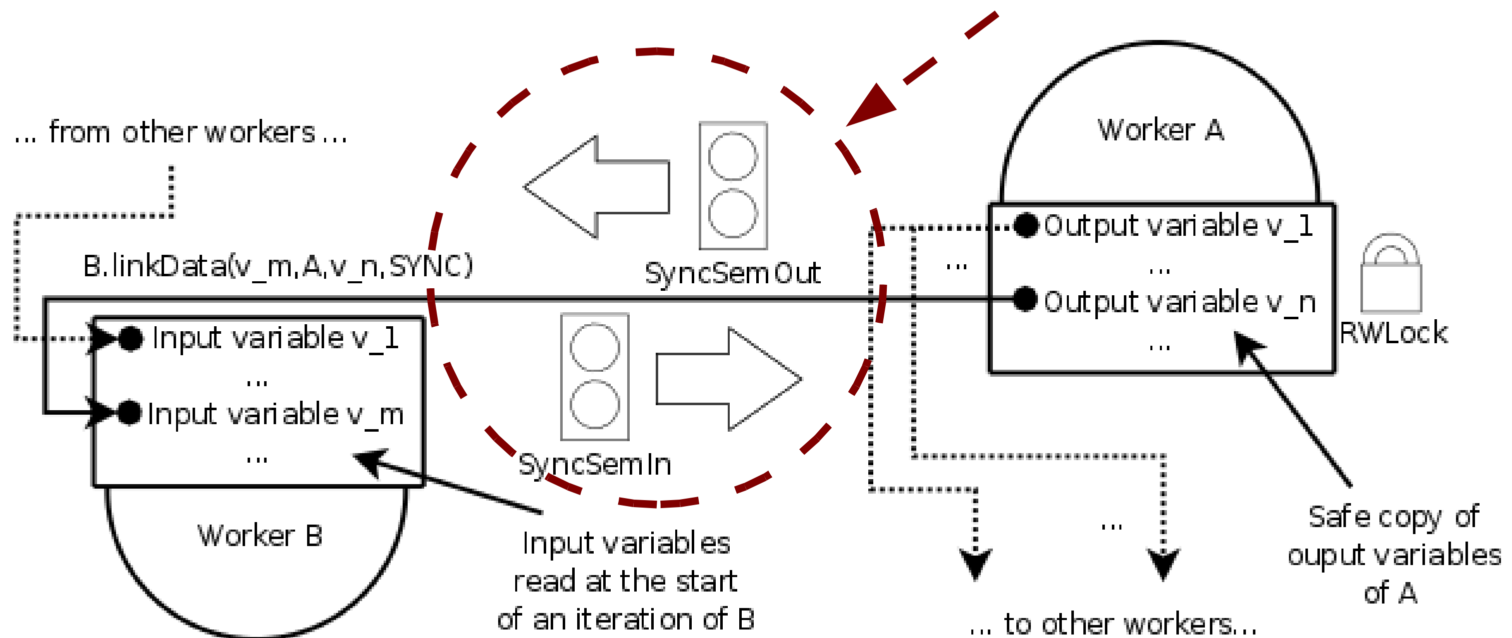
- Every worker has a copy of the last set of computed outputs (=*coherent state*)
- Every read access (sync. or async.) is protected by a standard *R/W lock* in each worker:
  - Several simultaneous reads possible...
  - ... but writing must wait, and when served, blocks readers.
  - Distributed among workers → avoids centralized blackboard bottleneck



PARP

Implementation (II)

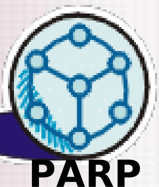
- Two semaphores enforce temporal constraints among *synchronously linked* threads:
  - *SyncSemOut* blocks consumers until new data available
  - *SyncSemIn* prevents producers from overwriting an output state until every consumer has read it
  - Maximizes computation overlap, while preserving sequential (pipelined) execution:



PARP

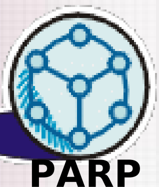
Implementation (III)

- Implicit data sharing technique:
  - Isn't the pattern **data copying intensive?** (ressembles more message passing than shared memory...)
  - A naive approach to data communication could be a bottleneck (specially when copying large data structures)
  - *Copy-on-write* (well known to OS implementers!):
    - Every shared data class is in fact just a pointer to a structure which contains (1) a reference count and (2) the real, possibly large sized data
    - The counter is incremented whenever a new object references data, and decremented when dereferenced
    - Shared data is deleted when counter becomes 0
    - More importantly, making a copy of an object involves only setting a pointer and incrementing the counter
    - Real copying only occurs if we need to modify shared data



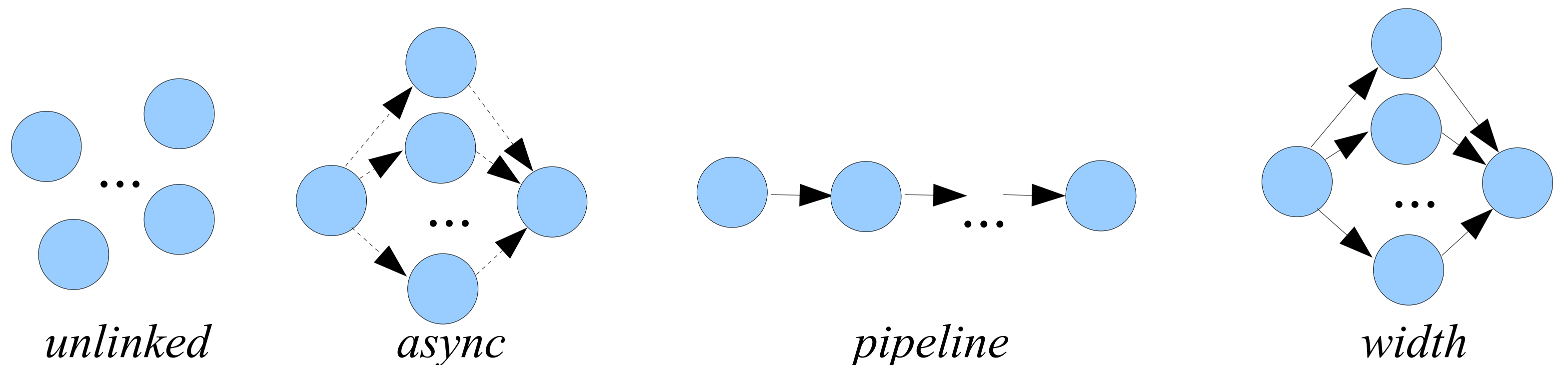
Performance (I)

- Of course, it **strongly depends on load balancing**, but...
- How much will a perfectly balanced application move away from ideal speedup, due to
  1) synchronization (locks and semaphores) overhead?
  2) memory copying overhead (when needed)?
- In the first case, it depends on the synchronization pattern:
  - Synchronous links tend to slow performance, due to temporal constraints among workers
- In the second case, it depends on the size of the (copied) data

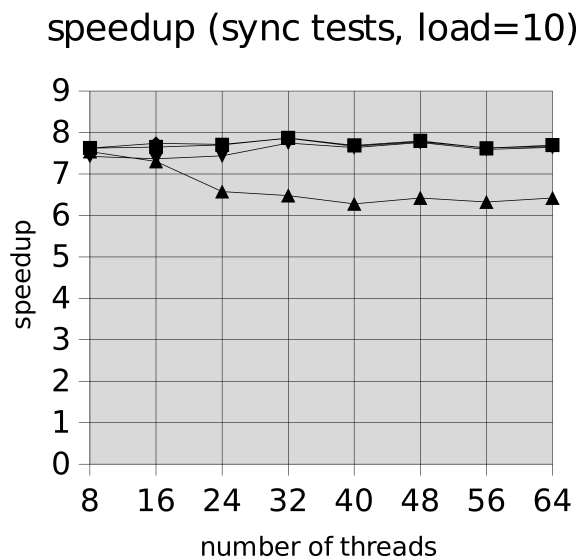


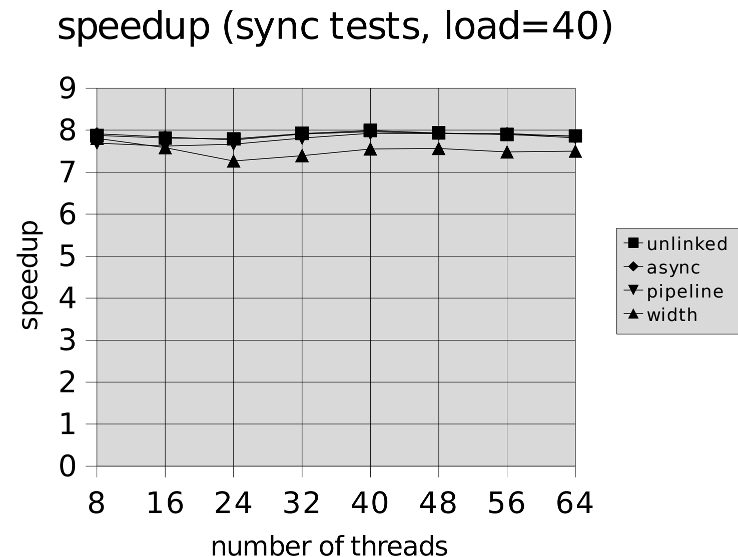
Performance (II): Synchronization overhead test

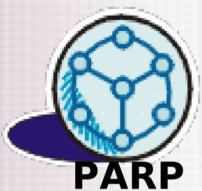
- Four case studies:



- Tests on Intel Xeon, two 64 bits 2GHz CPUs, 4 cores each (8 cores total):

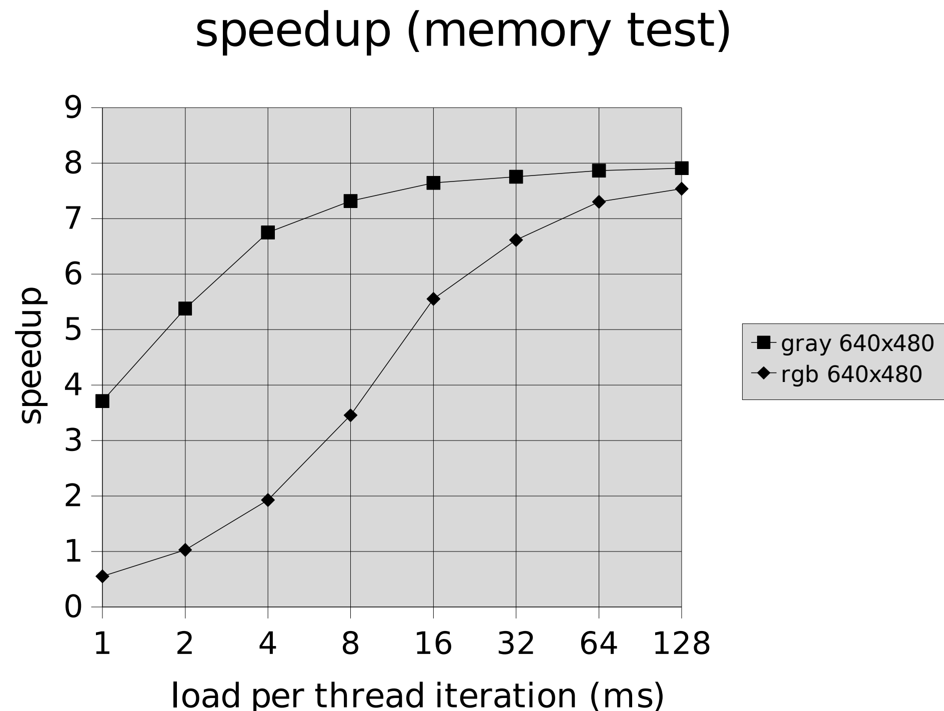






Performance (III): Memory copying overhead test

– Again, tests on Intel Xeon, two 64 bits 2GHz CPUs, 4 cores each:



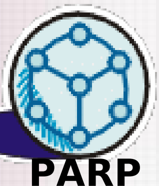
– Of course, the bigger the size of data to be copied, the worst the speedup (always), but...

– ...performance loss is much more appreciable for (excessively) light CPU load per iteration per thread

PARP

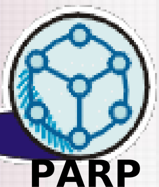
Discussion (I): advantages

- Reusability:
  - Component oriented threads
  - Library of (precompiled) workers
  - Allows replication of workers
- Composability:
  - Allows *nested* workers / other MT techniques
- Simple programming:
  - Declarative synchronism (vs. imperative synchronism)
  - Safe data sharing (deadlock / race conditions free) hidden to programmer
- Scalability:
  - Distributed state (vs. centralized *blackboard*) favors it
- Flexibility:
  - Asynchronous links allow for *reactive agents*, which execute only when signaled (adequate for GUI, sensors and actuators, etc.)



Discussion (II): drawbacks

- Structural restriction on target applications:
  - Must be clearly modular, ...
  - ... repetitive, ...
  - ... and task oriented
  - But adequate for signal processing, computer vision, etc.
- Load balancing falls on the programmer side:
  - Programmers must divide work adequately
  - Consider several agent organizations
  - Detect possible bottlenecks
  - Maybe using more specific MT techniques in *heavier agents*:
    - Data parallelism based (i.e., OpenMP)
    - More dynamically oriented nested task parallelism (i.e. Intel TBB)
  - We rely on OS thread scheduling (Linux 2.6)



References

Main paper:

- *A design pattern for component oriented development of agent based multithreaded applications.* A. Rodriguez, P.E. López-de-Teruel, A. Ruiz, G. García-Mateos and L. Fernández. Accepted in Euro-Par 2008.

Additional readings:

- *The free lunch is over.* H. Sutter. Dr. Dobbs Journal **3**(30), 2005.
- *Software and the concurrency revolution.* H. Sutter. Queue **3**(7), 2005.
- *Patterns for parallel programming.* T. Mattson, B. Sanders and B. Massingill. Addisson-Wesley, 2005.
- *C++ programming with QT 4.* J. Blanchette and M. Summerfield. Prentice Hall, 2006.