Session-based Programming for Parallel Algorithms

Expressiveness and Performance

Andi Bejleri
Raymond Hu
Nobuko Yoshida

Imperial College London

Aim of this work

Investigate benchmark examples of session types to compare productivity, safety and performance with other communications programming languages.

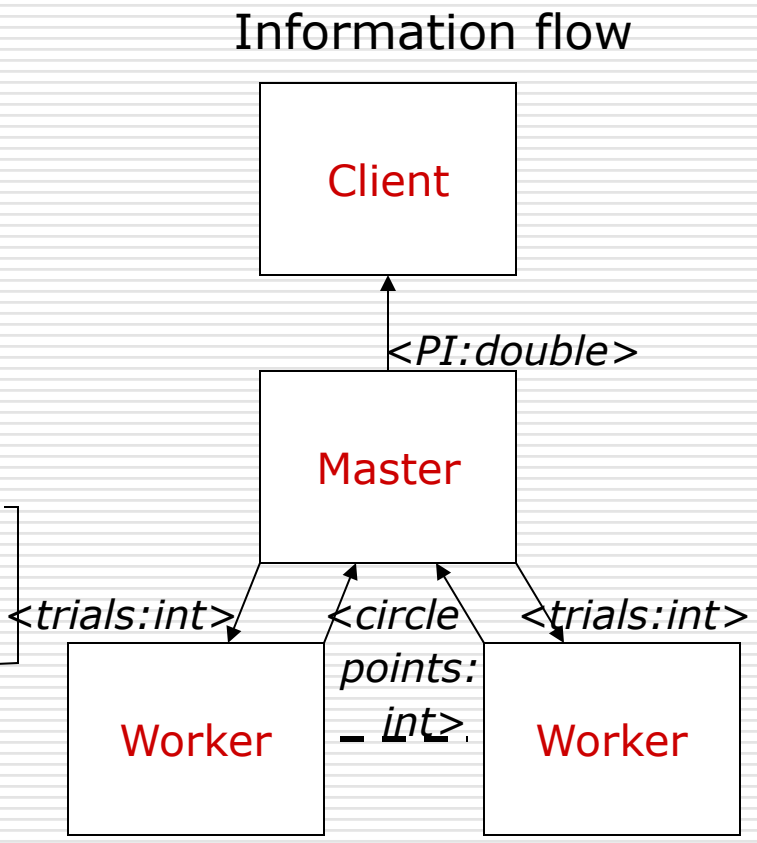
- Benchmark examples:
  - Monte Carlo Pi Approximation,
  - Jacobi solution of the Poisson Discrete Equation, and
  - a Simulation of the *n*-body Problem
- Session types: **SJ**, the first full object-oriented language to incorporate session types for type-safe concurrent and distributed programming.
- Other communications prog. lang. : **MPI** and **MPJ Express**

Monte Carlo π Approximation

```
protocol workerToMaster{
      sbegin.
            ?(int).
            !<int>
}

protocol masterToWorker{
      cbegin.
            !<int>.
            ?(int)
}
```

Safety (freedom from communication errors)

Duality of types: For each `send`(!) with a message type A corresponds a `receive`(?) with the same type.



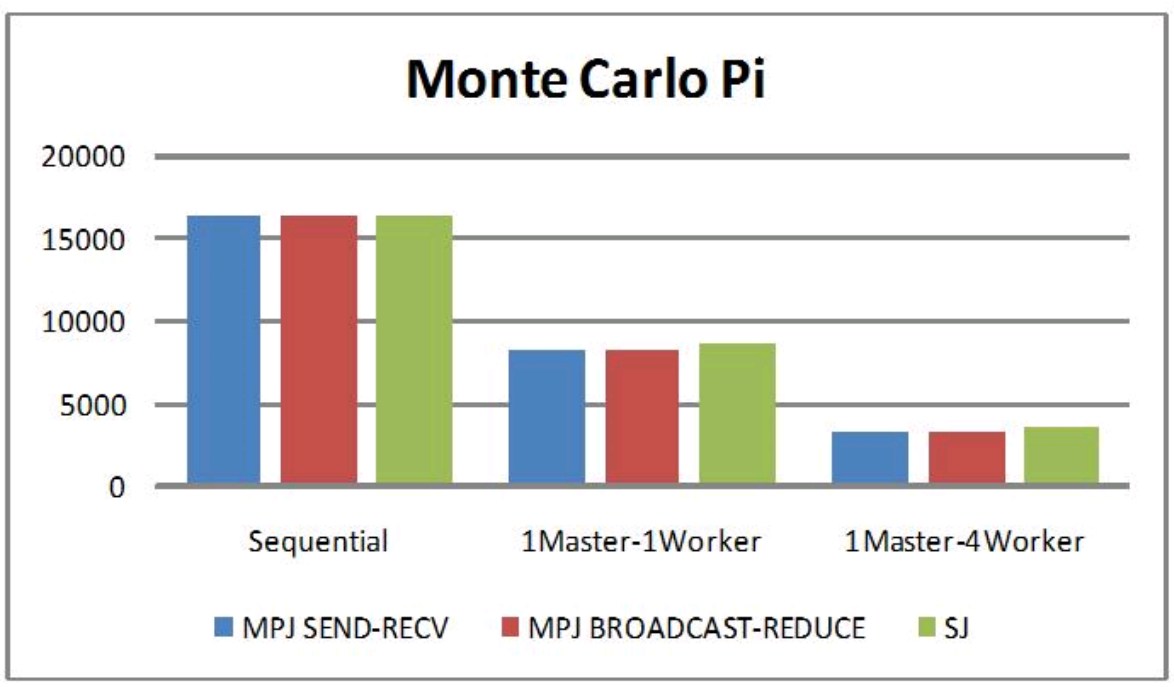
Monte Carlo π Approximation

Productivity

```
// Workers run the simulation.
int trials = s_wm.receive();
                                    //?(int)
for (int i = 0; i < trials; i++)
    if (hit()) hits++;
s_wm.send(hits); // !<int>
```

```
// Master controls the Workers.
<s_mw1, s_mw2>.send(trials);
                                    // Multicast.
int totalHits =
    s_mw1.receive() +
    s_mw2.receive(); // Collect the
                        results.
```

Performance Results

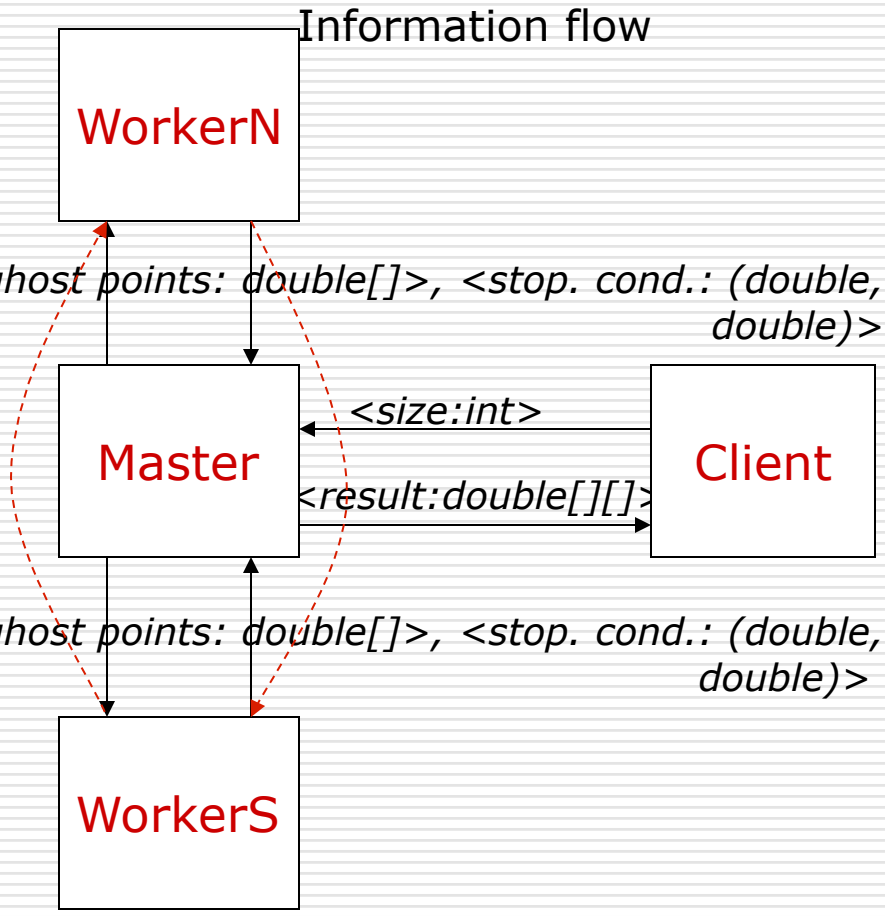


Client, Master and Workers run on different machines.

- Increasing the number of Workers reduces the time to complete the algorithm proportionally.
- MPJ Express implementation is 5-6% faster than the SJ implementation.

Jacobi Solution of the Discrete Poisson Equation

```
protocol masterToWorker {
  cbegin. // Request the Worker service.

    !<int>. // Send the size of the matrix.

    ![ // Enter scope of main algorithm
         iteration (check term. cond.).

      !<double[]>.?(double[]). // Send
           our boundary values; get Worker's
           updated ghost points.

      ?(double).?(double) // Receive the
       convergence data for Worker's subgrid.

    ]*. // After the last iteration...

    ?(double[][]) // ...get the final results.
}
```



Jacobi Solution of the Discrete Poisson Equation

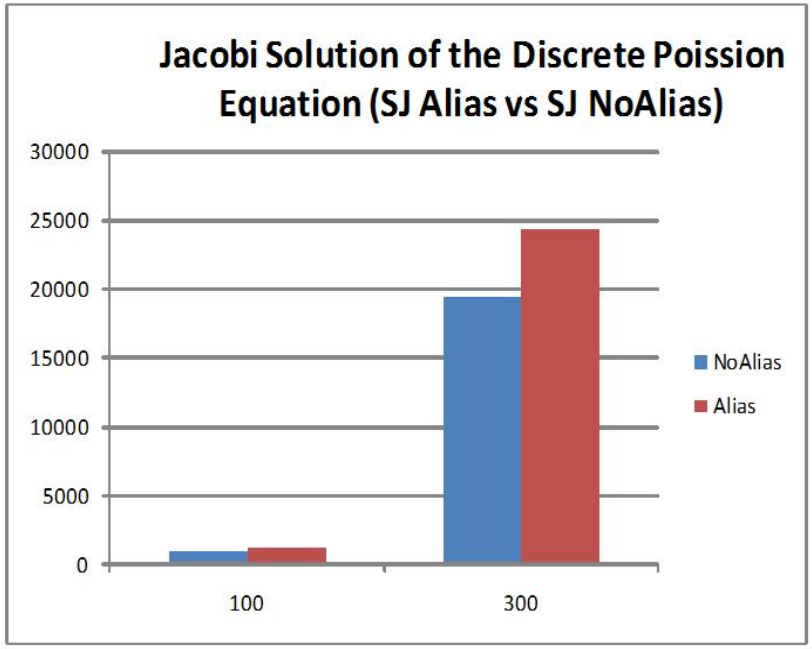
Productivity + Safety

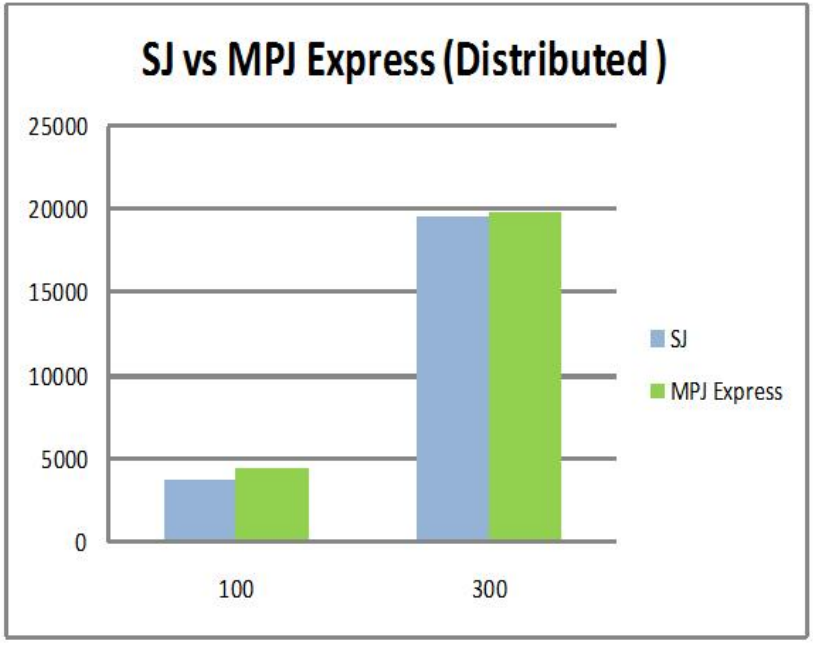
```
// Master controls iteration condition.                    // Workers obey the Master.

<mw1, mw2>.outwhile( // ![...                               <wm>.inwhile() { // ?[...
  !accurateEnough(...)
  && iters < MAX_ITERS){

    ... // Main                                  n.                ... /* Main body of the algorithm.*/

} // ...]*                                                    } // ...]*
```

Zero-copy message passing (when run on same VM)

```
// Array containing our boundary values (ghost points for the Master) declared as noalias.
noalias double[] ghostPoints = ...; // Update and prepare our boundary values for sending.

s_wm.send(ghostPoints); // Zero-copy send, as directed by types: !<noalias double[]>.

... // ghostPoints variable becomes null.
```

Performance Results

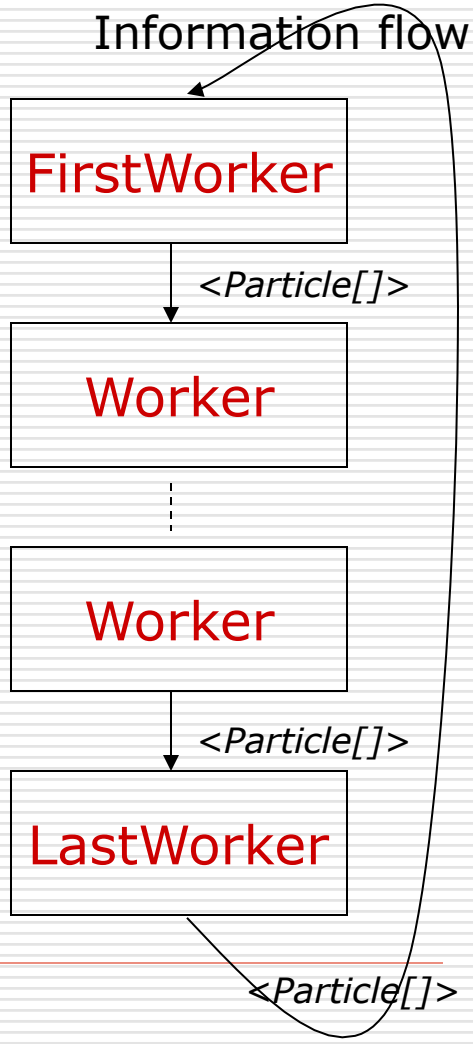




- The noalias version is approximately 20% faster than the ordinary one.
- The SJ implementation performs better than the MPJ Express one by 6% on average.

The *n*-Body Problem

```
protocol serverSide { // Interaction with the left
                      neighbour.
    sbegin. // Accept connection from left neighbour.

        !<int>. // Forward on the ring initialisation token.

        ?[ // Main simulation loop (iteration flag received
           from the left).

            ?[  // Inner iterations within each simulation
                step.

                ?(Particle[]) // Particle data forwarded
                              through pipeline.
            ]*
        ]*
}
```



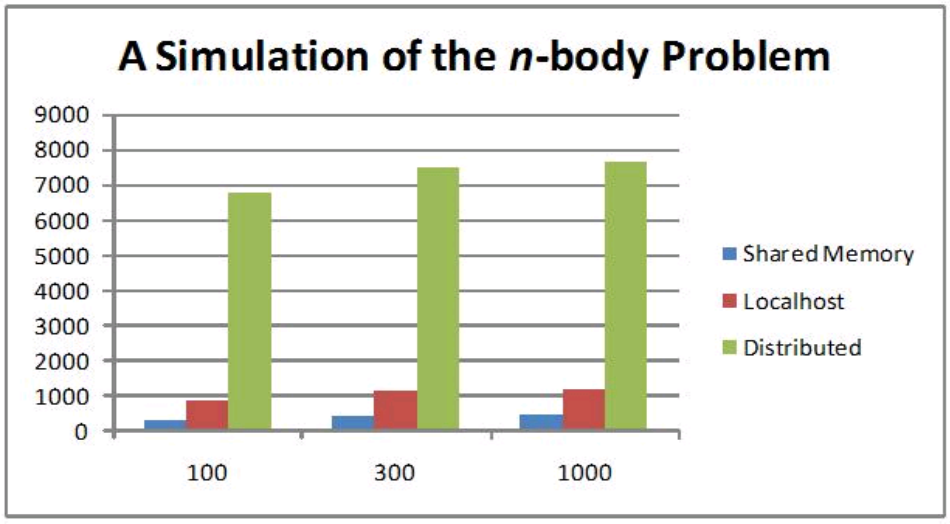
The *n*-Body Problem

Productivity + Safety

```
s_r.outwhile(s_l.inwhile()) { // Synch. with our two n'bours for each  sim. step. s_l: ?[..
    noalias Particle[] current = ...; // Prepare our own particle data for sending.
    s_r.outwhile(s_l.inwhile()) { // Inner iters. within each simulation step. s_l: ?[..
        s_r.send(current); // (i) Forward the current data set. s_r: !<Particle[]>.
        ...                // (ii) Add the current data to the running calculation.
        rrent = (Particle[]) s_l.receive(); // (iii) Receive next data set. s_l:
                                               ?(Particle[]).
    ..]*
    ... // Calculate the final results for this sim. step and update our own particle data.
} // s_l: ..]*
```

Zero-copy message passing (when run on same VM)

Performance Results



2 pipeline Workers

- Thread version faster than Localhost
  - 27% for size 100
  - 24% for size 300
  - 10% for size 1000
- Thread version faster than Distributed
  - 34% for size 100
  - 27% for size 300
  - 12% for size 1000

Comparing SJ with MPI

- Common MPI errors recognized by the community.
  - Doing things before `MPI_Init` and after `MPI_Finalize`
    - Runtime exceptions
  - Unmatched `MPI_Send` and `MPI_Recv` or matched `MPI_Broad` with `MPI_Recv`.
    - Deadlock
    - Broken invariants
    - Messages lost
  - Incorrect access of a shared communicator by separate threads.
    - Broken causalities
- ***SJ programs are guaranteed free from all of the above errors*** by the semantics of session communication and static session type checking.

Comparing SJ with MPI

- High-level message types.
  - In parallel algorithms, a common message structure is array.
  - In MPI, the programmer has to know the precise length of the array sent in order to read the data contained in it (e.g. in the MPI version of *n*-body).
  - In SJ, the length of the array is set by the runtime. At the application level arrays are considered as normal objects.
- Transparent zero-copy message passing.
  - Language support for zero-copy message passing in shared memory environment through `noalias` type.

Future Work

- Expanding the set of SJ operations and constructs, e.g. with session typed equivalents of MPI functions and features that are not yet directly supported.
- Compare SJ to PGAS languages such as X10.
- Extending SJ with full multiparty session types.
  - To express richer topologies such as rings and 2D-mesh
  - To enable performance improvements through massive parallelism

Conclusion

- SJ is an evolving framework, recent extensions
  - multicast output operations (<s_mw1, s_mw2>.send(trials);)
  - advanced iteration structures (<mw1, mw2>.outwhile(…), s_r.outwhile(s_l.inwhile()))
  - the Abstract Transport (use the same program to make use of the best transport available)
- SJ programs are guaranteed free from type and communication errors.
- In certain cases, SJ programs can out-perform their counterparts implemented
  - in communication-safe systems such as RMI
  - lower-level, non communication-safe message passing systems such as MPJ Express

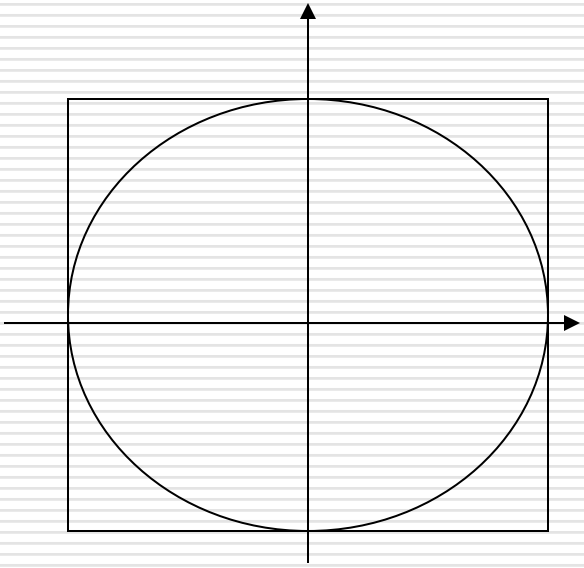
References

- Full implementation of the parallel algorithms in **SJ** and **MPJ Express**: http://www.doc.ic.ac.uk/~ab406/parallel_algorithms.htm
- **SJ** homepage: http://www.doc.ic.ac.uk/~rhu/sessionj.html
- **MPJ Express** homepage: http://mpj-express.org/
- William Gropp et al. *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (1999).
- Session types:
  - Kaku Takeuchi et al. *An interaction-based language and its typing system* (PARLE 94).
  - Kohei Honda et al. *Language primitives and type discipline for structured communication-based programming* (ESOP 98).
  - Raymond Hu et al. *Session-based distributed programming in Java* (ECOOP 08).
  - Kohei Honda et al. *Multiparty asynchronous session types* (POPL 08).

Monte Carlo π Approximation

- Square area = 4 unit^2
- Circle area = π unit^2
- t = Circle area/Square area, π = 4*t

- Generate points (x, y) in [-1, 1] and check if they fall inside the circle ($x^2 + y^2 <= 1$)



Jacobi Solution of the Discrete Poisson Equation

- A partial differential equation with applications in heat flow, electrostatics, gravity and climate computations.

- The discrete two-dimensional Poisson equation for a $m \times n$ grid:

  $u_{i\,j} = 1/4\ (u_{i-1,\,j} + u_{i+1,\,j} + u_{i,\,j-1} + u_{i,\,j+1} - dx^2 g_{i,\,j})$

  where $2 \le i \le m-1$, $2 \le j \le n-1$, and $dx = 1/(n+1)$.

- Jacobi's Method converges on a solution by repeatedly replacing each element of the matrix u by an average of its four neighbours and $dx^2 g_{i,\,j}$; for this example, we set g to 0.

  $u_{i\,j}^{k+1} = 1/4\ (u_{i-1,\,j}^{k} + u_{i+1,\,j}^{k} + u_{i,\,j-1}^{k} + u_{i,\,j+1}^{k})$

- Parallelism:
  - The grid is divided into three sub-grids
  - Jacobi iteration over each subgrid

The *n*-Body Problem

- Find the movements of *n* particles following the Newton Laws of Motion and law of universal gravitation when the force between particles is present.
- The force between particles is defined by adding the forces between all pairs of particles.
- Parallelism
  - The particles are distributed among different processes
  - The force of the local particles is calculated.
  - The particles from every other process are gathered and the force between the local ones and the gathered ones is calculated.
  - The solution in this work uses a ring topology, where several worker processes form a pipeline, to gather the particles.