Session-Based Programming for Parallel Algorithms
Expressiveness and Performance

Andi Bejleri Raymond Hu Nobuko Yoshida
Imperial College London, UK
`ab406@doc.ic.ac.uk` `rhu@doc.ic.ac.uk` `yoshida@doc.ic.ac.uk`

This paper investigates session programming and typing of benchmark examples to compare productivity, safety and performance with other communications programming languages. Parallel algorithms are used to examine the above aspects due to their extensive use of message passing for interaction, and their increasing prominence in algorithmic research with the rising availability of hardware resources such as multicore machines and clusters. We contribute new benchmark results for SJ, an extension of Java for type-safe, binary session programming, against MPJ Express, a Java messaging system based on the MPI standard. In conclusion, we observe that *(1)* despite rich libraries and functionality, MPI remains a low-level API, and can suffer from commonly perceived disadvantages of explicit message passing such as deadlocks and unexpected message types, and *(2)* the benefits of high-level session abstraction, which has significant impact on program structure to improve readability and reliability, and session type-safety can greatly facilitate the task of communications programming whilst retaining competitive performance.

1 Introduction

At PLACES'08, we discussed the need to investigate benchmark examples of session types [10, 6] to compare productivity, safety and performance with other communications programming languages. As a starting point into the investigation of these issues, we examine SJ [3], the first full object-oriented language to incorporate session types for type-safe concurrent and distributed programming. The SJ language extends Java with syntax for declaring session types (protocols), and a set of core operations (session initiation, send/receive) and high-level constructs (branching, iteration, recursion) for implementing the interactions that comprise the sessions. The SJ compiler statically verifies session implementations against their declared types. Together with runtime compatibility validation between peers at session initiation, SJ guarantees communication safety in terms of message types and the structure of interaction. SJ has been shown to perform competitively with widely-used communication APIs such as network sockets, in certain cases out-performing RMI [8].

This paper reports our on-going work on implementing parallel algorithms in SJ, with focus on the aforementioned aspects: ***productivity*** (including code readability and writability), ***safety*** (freedom from type and communication errors [10, 6]), and ***performance*** (optimisations enabled by SJ, and comparison against other communication systems). Parallel algorithms is a prominent topic in algorithmic research due to the increase of hardware resources such as multicore machines and clusters. The session-based programming methodology and expressiveness of SJ are demonstrated through implementations of: (1) a Monte Carlo approximation of π, (2) the Jacobi solution of the Discrete Poisson Equation, and (3) a simulation of the n-Body problem. These algorithms were selected to evaluate the SJ representation of, amongst other features, typical *task and data decomposition* patterns [9] (as featured in 1 and 2), a technique for exchanging *ghost points* [5] (in 2), and an intricate communication pattern over a circular pipeline structure (3). SJ is an evolving framework, and recent extensions to the SJ language (e.g. multicast output operations and advanced iteration structures) and the SJ Runtime (e.g. improved extensibility through the Abstract Transport) play an important part in the implementation of these algorithms.

Submitted to:
PLACES 2009

Using these programs, which feature complex and representative interaction structures, we contribute new benchmark results for analysis to supplement the existing benchmarks for SJ. In particular, benchmark comparisons between SJ and MPJ Express [1], a reference Java messaging system based on the MPI [4] standard, for (1) and (2) yield further promising peformance results for SJ. We also show how SJ *noalias* types can greatly optimise performance, such as for the shared memory communication of the ghost points in (2).

We then compare the SJ implementations of the above algorithms with their MPI counterparts from programming perspectives. Despite rich libraries and functionality, MPI remains a low-level API, and can suffer from such commonly perceived disadvantages of explicit message passing as unexpected message structures and deadlocks due to incorrect protocol implementations. From our experiences implementing the above algorithms, we found high-level session programming to be easier than the basic MPI functions, which often require manipulating numerical process identifiers and array indexes (e.g. for message lengths in (3)) in tricky ways. SJ is able to exploit session types to compensate for, or eliminate, many of the MPI problems: session types themselves are inherently deadlock free, for example.

In conclusion, we observe that high-level session abstraction has significant impact on program structure, improving readability and reliability, and session type-safety can greatly facilitate the task of communications programming whilst retaining competitive performance. We also argue that extending SJ with full multiparty session types would allow richer topologies such as the ring and 2D-mesh to be expressed more naturally, and enable performance improvements through massive parallelism.

2 Monte Carlo π Approximation

A simple Monte Carlo simulation for approximating the value of π is amenable to parallelisation. We use this example to (1) introduce basic and some new SJ constructs; (2) show their use in the description of a simple task decomposition pattern [9]; and (3) demonstrate the effect of parallelisation for performance gain in SJ (§ 5).

A unit square inscribes a circle of area $\pi/4$; hence, $\pi = 4t$, where t is the ratio of the circle area to the square. t can be determined by selecting a random set of points within the square ((x,y) where $x,y \in [-1,1]$), and checking how many fall inside the inscribed circle ($x^2 + y^2 <= 1$). A Master process (or thread) can instruct Workers to independently generate and check multiple sets of points in parallel, calculating the final value by combining the results from each Worker. The simple session type, from the Worker side, for the communications involved is:

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

Each Worker service (`sbegin`) is told how many points to test by the Master (`?(int)`) and sends back the number that fall inside the circle (`!<int>`). The code for a basic SJ implementation looks like

```
// 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 = // Collect the results.
    s_mw1.receive()
      + s_mw2.receive() + ...;
```

where `s_mw1` is the Master's session socket to Worker1, etc.; `s_wm` a Worker's session with the Master; and `hit` returns the boolean from testing a generated point. The Master can then calculate t by `totalHits / (trials * n)`, where `n` is the number of Workers. The SJ compiler statically verifies correctness by checking each session implementation against its declared type (e.g. `s_wm` against `workerToMaster`). At session initiation time, the runtime checks on each peer that the session type of the other end-point is reciprocal to its (peer) session type. If successful, the session is established, otherwise both parties raise an exception (`SJIncompatibleSessionEception`) and the session is aborted.

During the execution of a session, if an exception is raised, due to communication failure, at one or both sides in an enclosing session-try scope, a protocol is responsible for propagating a failure signal to all other active sessions within the same scope, maintaining consistency across such dependent sessions.

3 Jacobi Solution of the Discrete Poisson Equation

The implementation of this algorithm demonstrates (1) expressiveness of SJ due to multicast session-iteration operation; (2) guaranteed type and communication safety in SJ; (3) a type-directed optimisation (for exchanging ghost points) through the new SJ *noalias* type; and (4) the *transport-independence* of SJ programs, due to the design of the SJ language-runtime Framework. Poisson's Equation is a partial differential equation with applications in, for example, heat flow, electrostatics, gravity and climate computations. The discrete two-dimensional Poisson equation $(\nabla^2 u)_{ij}$ for a $m \times n$ grid can be written as in (a),

$$\text{(a)}\ \ u_{ij} = \tfrac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - dx^2 g_{i,j}) \qquad \text{(b)}\ \ u_{ij}^{k+1} = \tfrac{1}{4}(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k)$$

where $2 \leq i \leq m-1$, $2 \leq j \leq n-1$, 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 neighbouring values and $dx^2 g_{i,j}$; for this example, we set g to 0. Then from the k-th approximation of u, the next iteration performs the calculation in (b) above. Termination may be on reaching a target convergence threshold or completing a certain number of iterations. Parallelization exploits the fact that each element can be updated independently (within one step): the grid can be divided up and the algorithm performed on each subgrid in separate processes or threads. The key is that neighbouring processes must exchange their subgrid boundary values as they are updated.

We illustrate a one-dimensional decomposition of a square grid into three non-overlapping subgrids for three separate processes. Two Workers are allocated the end subgrids; the Master has the central subgrid, and controls the termination condition for all three processes. In addition to their allocated subgrid, each process maintains a copy of the boundary values (*ghost points*) of its neighbours; the new values are communicated after each iteration. This scheme allows the original grid to be divided in subgrids of any size. The session type between the Master and two Workers from the side of the former is:

```
protocol masterToWorker {
  cbegin.                       // Request the Worker service.
  !<int>.                       // Send the size of the matrix.
  ![                            /* Enter scope of main algorithm iteration (check
                                   termination condition). */
    !<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.
}
```

To control all the Workers simultaneously, the implementation of Master (see Appendix for full implementation) uses the SJ session constructs for multicasting output operations such as message-send and session-iteration. For example,

```
// Master controls iteration condition.
<mw1, mw2>.outwhile( // ![...
    !accurateEnough(...) && iters < MAX_ITERS) {
  ... // Main body of the algorithm.
}                      // ...]*
```

```
// Workers obey the Master.
<wm>.inwhile() { // ?[...
  ... /* Main body
       of the algorithm. */
}                 // ...]*
```

Like the standard while-statement, the outwhile operation evaluates the boolean condition for iteration (!accurateEnough(...) && iters < MAX_ITERS) to determine whether the loop continues or terminates. The key difference is that this decision is implicitly communicated to the session peer (in this case, from Master to the two Worker), synchronising the control flow between two parties. Worker is programmed with the dual behaviour: `inwhile` does not specify a loop-condition because this decision is made by Master and communicated to Worker at each iteration.

Inter-thread communication of large messages, such as arrays, can be optimised using SJ `noalias` types. A `noalias` variable on the RHS of an assignment or as a method argument — such as to the `send` operation — becomes `null`. Combined with static type checking that precludes any potential assignment of aliased values to `noalias` targets, a `noalias` variable is guaranteed the sole reference to the pointed object at all times, permitting zero-copy message passing of `noalias` messages over compatible shared memory transports. In the present example, the `noalias` optimisation can be used to communicate the ghost point data; for example, the Worker implementations contain the following code extract.

```
/* 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.
```

Transports that do not support this feature (e.g. TCP) can fall back to copy-on-send; the overall semantics of the program remains unchanged. This illustrates the *transport-independent* nature of SJ programs: the virtualisation of communication due to the SJ Runtime allows programs to make the best use of the whichever transports are available, *without* requiring any modification to the programs themselves. If the Master and Worker processes are run on separate machines, then the SJ Runtime can arrange, e.g. a TCP-based session; for the same programs, run as co-located threads, shared memory will be used. This SJ feature is further demonstrated for the next algorithm.

4 The *n*-Body Problem

The *n*-Body Problem involves finding the motion, according to classical mechanics, of a system of bodies given their masses and initial position and velocities. This advanced example demonstrates (1) the expressiveness of SJ and the extensions for complex iteration structures, by implementing an intricate circular communication pipeline; (2) SJ transport-independence (see § 5); and (3) the benefits of high-level message types (see § 6). Parallelism is achieved by dividing the particle set, and hence the calculations to determine the resultant force exerted on each body, amongst a collection of parallel processes. We use the approach where the processes, maintaining only the current state of their individual particle sets, are deployed to form a circular pipeline (ring topology). Firstly, the number of processes in the pipeline, p, is dynamically determined by sending a token around the ring. Then each step of the simulation involves $p-1$ iterations. In the first iteration, each process sends their particle data to their neighbour on the right and calculates the partial resultant forces exerted within their own particle set. In the n-th iteration, each process forwards on the particle data received in the previous iteration (line (*i*) in the listing on the next page), adds this data to the running force calculation (*ii*), and receives the next data set (*iii*). The particle data from the right neighbour is received by the end of the final iteration: each data set has now been seen by all processors in the pipeline, allowing the final results for the current simulation step to be calculated.

The SJ implementation of the above algorithm has each process, i.e. each Worker unit in the pipeline, open a session server socket to accept a connection from its left neighbour, and create the connection to its right neighbour using a session client socket. The session type for the interaction in this algorithm, from the server side of each unit, is:

```
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 session type for the corresponding client side of each unit is simply the direct dual of `serverSide`: `protocol clientSide { cbegin.?(int).![![!<Particle[]>]*]* }`, given by inverting the input (`?`) and output (`!`) symbols. For this client-server architecture, the ring topology is bootstrapped by designating two neighbouring processes to be the “first” and “last” pipeline units.

The SJ code and a comparison with the MPI implementation is given in Section 6.

5 Performance Benchmarks

This section presents performance measurements for the three parallel algorithms described above. The first two benchmarks show that the SJ Runtime, although still at an early implementation version with much scope for further optimisation, can perform competitively with MPJ Express [1]. Unlike Java MPI implementations built around JNI wrappers to C functions, MPJ Express adopts a pure Java approach which makes for an interesting comparison with SJ.

The same machines in the same network environment were used for all the following benchmark experiments. Each machine is a dual-core Intel Core 2 Duo (Conroe B2) at 2.13GHz with 2MB cache, 2GB main memory, running Ubuntu Linux 4.2.3 (kernel 2.6.24); the machines were connected via gigabit Ethernet, and the latency between two machines was measured using ping (64 Bytes) to be on average 0.10ms. The benchmark applications were compiled and executed using the standard Sun Java SE compiler and runtime versions 1.6.0. For each experiment, the results from 100 executions for each parameter configuration were recorded; here, we give the mean values. The full source code for the benchmark applications and the complete results can be found at [2].

Monte Carlo π approximation. The first benchmark uses the SJ implementation of this algorithm to (1) verify the performance gain from increased parallelism, and (2) to compare the performance of the SJ Runtime against MPJ Express. Each process (Master, Workers and Client) was run on a separate machine, communicating via TCP. The results (Figure 1), comparing both sequential and parallel versions of the algorithm, show that for a constant sample size (total number of test points), increasing the

Configuration	SJ (ms)	MPJ (ms)
Sequential (1 Worker)	6717	
1 Master & 1 Worker	3764	3846
1 Master & 2 Workers	2466	2606
1 Master & 3 Workers	1885	1966
1 Master & 4 Workers	1487	1579

Figure 1: Monte Carlo π for a varying number of Workers.

number of Workers indeed reduces the time to complete the algorithm proportionally. The results for the SJ implementation are around 5–6% faster than the MPJ Express implementation.

Jacobi Poisson solution. The second benchmark, through the SJ implementation of the Jacobi iteration algorithm, demonstrates (1) the effectiveness of `noalias` types for zero-copy message transfer in a shared memory environment, and (2) again compares SJ performance to MPJ Express. Firstly, “Ordinary” (i.e. without `noalias`) and `noalias` versions of the Master and two Workers were run as co-VM threads on a single machine; the Client is connected to the Master from a separate machine via a TCP-session. We measured the time to complete the algorithm for square matrices of size (i.e. the length of one side of the matrix) 100 and 300. In both cases, the `noalias` version is approximately 20% faster than the ordinary one (Figure 2(a)a). For sizes greater than 300, we have observed that the local computation costs dominates the communication costs for this fixed number of Workers, therefore there is no difference between the execution time of the “Ordinary” and `noalias` versions for matrix size 1000. Secondly, the distributed SJ implementation of Jacobi (the Client, Master and Workers run on separate machines connected via TCP) performs better than the MPJ Express implementation by 6% on average (Figure 2(b)b).

Matrix Size	“Ordinary” (ms)	`noalias` (ms)
100	1270	992
300	24436	19448
1000	288532	299279

(a)

Matrix Size	SJ (ms)	MPJ (ms)
100	3713	4460
300	19501	19834

(b)

Figure 2: (a) Jacobi: “ordinary” vs. `noalias` versions; (b) Jacobi: SJ vs. MPJ Express.

***n*-Body simulation.** The third benchmark uses the *n*-Body simulation to demonstrate the important improvement in productivity enabled by SJ transport-independence: this single SJ implementation was run in the different communication environments (locally concurrent, distributed), making the best use of the available transports (TCP, shared memory, etc.), without *any* changes to the source code for the Workers (although the shared memory version required a few lines of external code to bootstrap the Workers as Java threads). The benchmark was executed using two pipeline Worker units (not using `noalias`) in three different configurations: the two Workers on separate machines using TCP (Distributed), as separate processes on the same machine using TCP (Localhost), and as co-VM threads using shared memory

(Threads). We recorded the results for simulations involving 100, 300 and 1000 particles, distributed equally between the Workers.

As expected, the results (Figure 3) show the Threads version is faster than Localhost: around 27% for 100 particles, 24% for 300, and 10% for 1000. The Distributed version is in turn slightly slower (latency is very low) than Localhost: 10% for 100 particles, 4% for 300, and 3% for 1000. The relative performance gain between each version decreases for larger particle sets because the local computation costs begin to dominate the communication costs for this fixed number of Workers. Naturally, performance can be improved for simulations involving many particles by increasing the degree of parallelism, i.e. using more Workers.

Particles	Distrib. (ms)	Localhost (ms)	Threads (ms)
100	496	452	326
300	1194	1144	865
1000	7702	7497	6785

Figure 3: *n*-Body simulation: Distributed vs. Localhost vs. Threads versions.

6 SJ and MPI Comparison

This section compares SJ against MPI in terms of language support for communications programming, with reference to MPI implementations of the above algorithms [5]. Since MPI has an extensive library of functions developed over 15 years, many of these are not yet directly supported in SJ, e.g. MPI Jacobi makes use of a virtual topology (`MPI_Cart_Create`) and collective data movement operations (`MPI_Bcast` and `MPI_Allreduce`, for broadcasting the matrix size and distributing the termination condition in (2)). However, many of these features can be encoded into a session type, as shown above. Furthermore, we observed the following benefits of SJ against MPI.

Type and communication safety from session types. MPI is designed as a portable API specification to be implemented for varying host languages. Coupled to the low-level nature of many MPI functions, the design of accompanying MPI program verification techniques for a host language can be difficult. Common MPI errors recognized by the community include:

- **Doing things before** `MPI_Init` **and after** `MPI_Finalize`. The execution of such MPI operations can lead to runtime errors such as broken invariants, messages not broadcasted, and incorrect collective operations. In SJ, if an action is performed before the socket or the session has been initialised then a runtime exception will be raised. Also, the static type system of SJ does not allow to perform actions after the `finally` clause. Following is the code in MPI and SJ for setting up the ring topology in the n-body algorithm.

```
...//Variables
main(int argc, char *argv[])
{
 //Set up of the topology
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
...//Variables
public void run(...)//Args. to create sockets
{
  //Set up the sockets of the topology
  final noalias SJService c_r =
      SJService.create(pc_nbody, host_r, port_r);
  final noalias SJSocket s_l;
```

```
  /* Get the best ring in the topology */
  periodic = 1;
  MPI_Cart_create(MPI_COMM_WORLD, 1, &size,
            &periodic, 1, &commring);
  MPI_Cart_shift(commring, 0, 1, &left,
            &right);
  ...//algorithm
MPI_Finalize();
  return 0;
}
```

```
  try (ss_l)
  {
    ss_l = SJServerSocketImpl.create(ps_nbody,
                                       port_l);
    try (left, right)
    {
      left = ss_l.accept();
      right = c_r.request();
      //size of topology
      left.send(right.receiveInt() + 1);
      ...//algorithm

}finally{ }
  }catch(SJIncompatibleSessionException ise){ }
  ...//Capture other exceptions
  finally{ }
}
```

- **Unmatched `MPI_Send` and `MPI_Recv`.** Such errors can lead to a mismatch between the sent and expected message type/structure, or a variety of deadlock situations depending on the communication mode. For example, two processes deadlock if each is waiting for a message before sending the message expected by the other. In the standard (buffer-blocking) mode, the converse situation (both processes attempting to send before receiving) can also deadlock: if both message sizes are bigger than the available space in the medium and opposing receive buffers, then the processes cannot complete their write operations. A related problem is matching a `MPI_Bcast` output with `MPI_Recv`. Standard usage is to receive a broadcast message using the complementary `MPI_Bcast` input. `MPI_Recv` consumes the message; hence, the receiver must be able to determine which processes have not yet seen the message and manually re-broadcast it.
- **Concurrency issues.** Incorrect access of a shared communicator by separate threads can violate the intended message causalities between the sender(s) and the receivers. In addition, race conditions can arise due to modifying, or even just by accessing, messages that are in transit.

As illustrated in the previous sections, ***SJ programs are guaranteed free from all of the above errors*** by the semantics of session communication and static session type checking. The first two points are directly prevented by the properties of session types. For the third point, the SJ compiler disallows sharing of session socket objects (implicitly `noalias`), and message copying/linear transfer can be safely and explicitly controlled via `noalias` types.

High-level message types. In many parallel algorithms, messages are mainly communicated via arrays. For MPI, effort is required to manually track and communicate array indices, e.g. for message length or the number of messages. In contrast, the high-level type-abstraction for messages allows SJ programmers to treat both object and primitive array messages as regular Java array objects. For instance, the MPI version of the *n*-Body simulation broadcasts the number of particles managed by each process so the amount of data to be read from each particle set message can be determined; in SJ, the particle data is simply received as discrete array messages, avoiding manual handling of message sizes. Following is the MPI[1] and SJ code for the main simulation loop.

---

[1] The code of n-body in MPI has been taken from Using MPI website[4].

```
/* Get the sizes and displacements */
MPI_Allgather(&npart, 1, MPI_INT, counts,
      1, MPI_INT, commring);
displs[0] = 0;
for (i=1; i<size; i++)
  displs[i] = displs[i-1] + counts[i-1];
totpart = displs[size-1] + counts[size-1];

InitParticles(particles, pv, npart);


while (cnt--)
{
  double max_f, max_f_seg;

  /* Load the initial sendbuffer */
  memcpy(sendbuf, particles,
      npart * sizeof(Particle));
  for (pipe=0; pipe<size; pipe++)
  {
    if (pipe != size-1)
    {
      MPI_Isend(sendbuf, npart, particletype,
          right, pipe, commring, &request[0]);

      MPI_Irecv(recvbuf, npart, particletype,
          left, pipe, commring, &request[1]);
    }
    /* Compute forces */
    max_f_seg = ComputeForces(particles, sendbuf,
                              pv, npart);
    /* Wait for non-blocking receives to return */
    if (pipe != size-1)
      MPI_Waitall(2, request, statuses);
    memcpy(sendbuf, recvbuf,
             counts[pipe] * sizeof(Particle));
  }
  /* Update our own particle data. */
  sim_t += ComputeNewPos(particles, pv, npart,
                         max_f, commring);
}
```

```
initParticles(particles, pvs);

    /* Synchronise with our two neighbours
    for each simulation step. s_l: ?[.. */
   right.outwhile(left.inwhile())
   {
     /* Load the initial sendbuffer */
     Particle[] current =
              new Particle[numParticles];
     System.arraycopy(particles, 0, current,
                      0, numParticles);
     /* Inner iterations within each
     simulation step. left: ?[.. */
     right.outwhile(left.inwhile())
     {
       /* (i) Forward the current data set.
       right: !<Particle[]>. */
       right.send(current);
        /* (ii) Add the current data
       to the running calculation. */
       computeForces(particles, current, pvs);
        /* (iii) Receive next data set.
        left: ?(Particle[]). */
       current = (Particle[]) left.receive();
     } // left: ..]*
     /* Calculate the final results for
     this sim. step and update our own
     particle data. */
     computeForces(particles, current, pvs);
     computeNewPos(particles, pvs, i);

     i++;
   }// left: ..]*
```

In the SJ implementation of n-body, the assignment in (*iii*) is permitted because the received message is implicitly `noalias`.

Transparent zero-copy message passing. SJ provides direct language support for zero-copy transfer in shared memory contexts through `noalias` types. This feature can enable significant performance increases for multi-threaded programs (see § 5). Moreover, the communication of `noalias` types retains consistent semantics in all transport contexts (see *transport-independence* in § 3).

7 Future Work and Conclusion

We demonstrated expressiveness, productivity and performance benefits of session-based programming in SJ through the presented parallel algorithm implementations. Although we have seen that the above algorithms were readily implemented in the current SJ, immediate future work includes 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. For example, whilst the MPI *standard* mode (send and receive block on their respective buffers) corresponds to the session communication semantics in SJ, MPI has several additional modes: *synchronous* (send and receive operations synchronise), *ready* (programmer notifies the system that a receive has been posted), and *buffered* (user manually handles send buffers). We also wish to compare SJ to PGAS languages such as X10 [11] using parallel algorithm implementation as a basis.

We believe that extending SJ with full multiparty session types [7] would allow richer topologies such as the ring and 2D-mesh to be expressed more naturally in a type-safe manner. For example, the SJ n-Body implementation currently requires creating one intermediary session (for the final pipeline link) in each simulation step; with multiparty sessions, we would only need to open a single session for the complete simulation. Our prediction is that multiparty sessions will offer better support for massive parallelism than the current client-server based session sockets. We plan to identify design issues and possible overheads for global type-checking through further implementation of parallel algorithms with complex communication patterns.

SJ programs are guaranteed free from type and communication errors, and perform competitively against other Java communication runtimes. In certain cases, SJ programs can out-perform their counterparts implemented in communication-safe systems such as RMI [8] and also lower-level, non communication-safe message passing systems such as MPJ Express (§ 5).

8 Acknowledgments

We thank Kohei Honda and Vijay Saraswat for their comments on a first draft of this abstract. The work is partially supported by EPSRC GR/T03208 and GR/T03215.

References

[1] *MPJ Express homepage*. `http://mpj-express.org/`.

[2] *Session-based Programming for Parallel Algorithms*. `http://www.doc.ic.ac.uk/~ab406/parallel_algorithms.html`.

[3] *SJ Homepage*. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`.

[4] *Using MPI: Example Programs*. http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/.

[5] William Gropp, Ewing Lusk & Anthony Skjellkum (1999): *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press.

[6] Kohei Honda, Vasco Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP*, *LNCS* 1381. Springer, pp. 122–138.

[7] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: George C. Necula & Philip Wadler, editors: *POPL*. ACM, pp. 273–284.

[8] Raymond Hu, Nobuko Yoshida & Kohei Honda (2008): *Session-Based Distributed Programming in Java*. In: Jan Vitek, editor: *ECOOP*, *LNCS* 5142. Springer, pp. 516–541.

[9] Timothy G. Mattson, Beverly A. Sanders & Berna L. Massingill (2004): *Patterns for Parallel Programming*. Addison-Wesley Professional.

[10] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: *PARLE*, *LNCS* 817. Springer, pp. 398–413.

[11] *X10 homepage*. `http://x10.sf.net`.

A Appendix

The complete code of Master for Jacobi is given below. The process is implemented as a single monolithic unit in the file "master.sj". The reader interested in the implementation of the two workers can find them in [2].

```
package onedimjacobi.Noalias;

import java.io.*;
import java.util.*;

import sessionj.runtime.*;
import sessionj.runtime.net.*;

public class Master
{
  private final noalias protocol p_mc
  {
    sbegin.?(int).!<double[][]>
  }
  private final noalias protocol matrix_size { !<int> }
  private final noalias protocol stopping_condition
  {
    ?(Double).?(Double)
  }
  private final noalias protocol ghost_points
  {
    !<double[]>.?(double[])
  }
  private final noalias protocol partial_result { ?(double[][]) }
  private final noalias protocol p_mw
  {
    cbegin
    .@(matrix_size)
    .![
      @(ghost_points)
      .@(stopping_condition)
    ]*
    .@(partial_result)
  }

  private static final int MAX_ITERATIONS = 100000;

  public void run(int port_m, String host_n, int port_n, String host_s, int port_s)
  {
    /* Socket that communicates with Client. */
    final noalias SJServerSocket ss;

    /* Sockets that communicates with neighbours. */
    final noalias SJService c_n = SJService.create(p_mw, host_n, port_n);
```

```
final noalias SJService c_s = SJService.create(p_mw, host_s, port_s);

try (ss)
{
  /* Create and set up the server socket. */
  ss = SJServerSocketImpl.create(p_mc, port_m);

  while (true)
  {
    final noalias SJSocket cm;

    try (cm)
    {
      // Accept the connection from client.
      cm = ss.accept();

      // size of the problem.
      int size = cm.receiveInt();

      int rows = size / 3;

      final noalias SJSocket mn, ms;

      try (cm, mn, ms)
      {
          // Set up the connection with neighbours.
           mn = c_n.request();
           ms = c_s.request();

           // Send to the first and second neighbor the size of the problem.
        <mn, ms>.send(size);

        // Build its sub-grid.
        double[][] u = new double[rows + 2][size + 2];

        // Sub-grid next iterations with the same dimension as u.
        double[][] newu = new double[rows + 2][size + 2];

        // Initialise u, unew, f.
        init(u, newu, rows, size);

        double diff = 1.0;
        double valmx = 1.0;
        int iterations = 1;

        // Master controls iteration condition.
        <mn, ms>.outwhile( (diff / valmx) >= (1.0 * Math.pow(10, -5))
                            && iterations <= MAX_ITERATIONS)
        {
        // Main body of the algorithm.

          diff = 0.0;
```

```
valmx = 0.0;

// Jacobi iterations.
for(int i = 1; i < rows + 1; i++)
{
  for(int j = 1; j < size + 1; j++)
  {
    newu[i][j] = (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] +
                  u[i][j + 1]) / 4.0;

    diff = Math.max(diff, Math.abs(newu[i][j] - u[i][j]));
    valmx = Math.max(valmx, Math.abs(newu[i][j]));
  }
}

// Ghost zone of neighbors.
noalias double[] border_n = new double[size];
noalias double[] border_s = new double[size];

// Send the ghost zones of neighbours.
for(int k = 0; k < size; k++)
{
  border_n[k] = newu[1][k + 1];
}
for(int k = 0; k < size; k++)
{
  border_s[k] = newu[rows][k + 1];
}

mn.send(border_n);
ms.send(border_s);

// Receive ghost zones from neighbours.
noalias double[] ghost_n = (double[]) mn.receive();
noalias double[] ghost_s = (double[]) ms.receive();

// Copy ghost zones in newu
for (int k = 0; k < ghost_n.length; k++)
{
  newu[0][k + 1] = ghost_n[k];
}
for(int k = 0; k < ghost_s.length; k++)
{
  newu[rows + 1][k+1] = ghost_s[k];
}

// Update u with newu.
double[][] tmp = u;
u = newu;
newu = tmp;

// Computing newerror.
```

```
          diff = Math.max(diff, ((Double) mn.receive()).doubleValue());
          valmx = Math.max(valmx, ((Double) mn.receive()).doubleValue());

          diff = Math.max(diff, ((Double) ms.receive()).doubleValue());
          valmx = Math.max(valmx, ((Double) ms.receive()).doubleValue());

          if (iterations == 1)
          {
            diff = 1.0;
            valmx = 1.0;
          }
          iterations++;
        }

        double[][] w1 = (double[][]) mn.receive();
        double[][] w2 = (double[][]) ms.receive();

        double[][] result = new double[size][size];

        for (int i = 0; i < rows; i++)
        {
          for(int j = 0; j < size; j++)
          {
            result[i][j] = w1[i + 1][j + 1];
          }
        }

        for (int i = rows; i < 2 * rows; i++)
        {
          for(int j = 0; j < size; j++)
          {
              result[i][j] = u[i - rows + 1][j + 1];
          }
        }

        for (int i = 2 * rows; i < size; i++)
        {
          for(int j = 0; j < size; j++)
          {
               result[i][j] = w2[i - 2 * rows + 1][j + 1];
          }
        }
        cm.send(result);
      }
      finally{ }
    }
    finally{ }
  }
}
catch (SJIncompatibleSessionException ise)
{
  System.err.println("[Master] Non-dual behavior: "+ ise);
```

```
      }
      catch (SJIOException sioe)
      {
        System.err.println("[Master] Communication error: " + sioe);
      }
      catch (ClassNotFoundException cnfe)
      {
        System.err.println("[Master] Class error: " + cnfe);
      }
      finally{ }
    }
  }
```