1a. What are the target applications and …

The same applications STL is targeting. We have targeted in the past regular and irregular scientific codes (particle transport), graph processing workloads, parallel robotic motion planning.

1b. … what is the expected sophistication of the target users (naive, knowledgeable, expert)?

Naive users need to know the STL and default scheduling and data placement policies will be chosen for them. A knowledgeable user knows about skeletons and how to use/compose them and is aware of the underlying data-flow representation (due to possible pressure on resources). Knowledgeable users are also able to specify data distributions from a library of distributions. An expert user can write new data distributions, write new schedulers, write new skeletons, write new compositional skeletons, write new containers. Developers can modify the data-flow engine and the runtime/communication system.

2. List the set of features/concepts of your programming model and divide them into two sets: What is the minimum set of things a new user needs to learn to become productive? What are the more advanced features a user can potentially take advantage of?

Algorithms, skeletons, containers, paragraph, views. Minimum set for new users: containers, skeletons, views. Advanced: data distributions, container composition, new schedulers, etc.

3. How do you decide whether a new application is a good fit? What metrics do you use to evaluate whether an application is implemented well in your model?

As a generic library, STAPL will equally fit any application to the best of our knowledge. For metrics of evaluation, we target minimization of some resource: time, memory usage, power, extreme scale scalability. Performance comparison against a theoretical model or known state-of-the-art reference implementations.

4. What is the plan for interoperability with MPI, OpenMP, Kokkos, etc.? If you could add requirements to MPI, what would those be?

We use MPI and OpenMP (and C++11 thread back-end) in mixed-mode configuration. We use BLAS routines from STAPL. The Kokkos multidimensional array data structure can be used as a base container (underlying storage) for our distributed multidimensional pContainer.

5. What is the plan for performance portability?
We use virtualization to write once and perform reasonably well everywhere. The model doesn't change, but is specialized. For example, we apply shared-memory optimizations while still retaining the fundamentally distributed model.

We obtain performance through adaptivity at all levels of abstraction. At the algorithmic level, we use k-level-asynchronous, hierarchical aggregation of communication for power-law graphs, and algorithmic substitution. At the runtime level, we perform RMI aggregation.

6. What is the plan for fault tolerance?

No current fault tolerance facilities, but plans for the future. A task can be restarted if machines fail by using its non-retired inputs. Our task graph based model lends itself very well to fault tolerance.

7. What static analyses and transformations could you do? What do you do today?

We can statically transform skeletons. For example, we can apply skeleton fusion. We can perform skeleton interchange. We can also use static analysis and transformation to specialize from distributed to shared-memory and perform general type manipulation.

8. Questions about task graphs:
*** When is the task graph generated (compile-time, load-time, run-time)?**
*** How do you manage task graph generation vs. task graph execution?**
*** What is the value of non-ready tasks in the DAG?**
*** Do you exploit the repetitiveness of iterative applications that repeatedly execute the same task graph?**

The task graph is created dynamically from a static specification (skeleton) and executed at runtime. For dynamic problems (where future task creation is decided during task graph execution) tasks are dynamically adding more tasks to the task graph.

We can interleave graph generation with graph execution, or pre-generate the entire graph and subsequently execute.

We can exploit the pre-generated task graph by reusing it in iterative applications.

Non-ready tasks are used as a measure for load balancing using our work-stealing scheduler.

9. Questions about tasks:
*** How is task granularity managed?**

*** What is the life-cycle of a task?**

The algorithm is specified at the finest granularity and automatically coarsened to create larger tasks.

The task is created by its specifying the predecessors.The task becomes non-ready. When the dependencies have been satisfied, the task is fired and executed and the output is fed into successor tasks. A task is retired when all of its successors have received its output.

10. What is the relationship between task and data parallelism --- can one be invoked from the other arbitrarily or are there restrictions?

Data parallelism is obtained through skeleton expansion proportional to its input data. Task parallelism is obtained through skeleton composition. Using nested parallelism, we can invoke data parallelism through task parallel operations and vice versa.

11. Where exactly is concurrency (meaning the ability to have races and deadlocks) exposed to the programmer, if at all?

STAPL follows a distributed-memory model with no shared references among processing elements. Thus, data races and deadlocks do not occur.