CELERA ASSEMBLER LOGICAL ARCHITECTURE

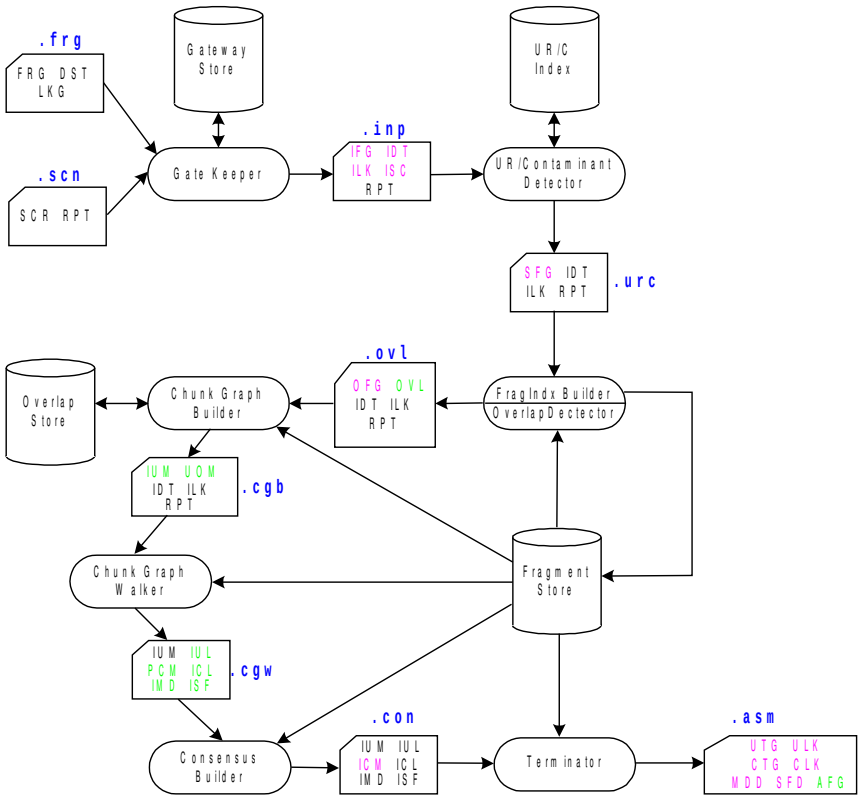
The Celera assembler consists of a pipeline of processing modules as described in the document "Celera Assembler''. This document specifies the data flow between these modules in detail, and futher refines our "organic" development approach of building a series of successively more powerful "prototypes". The input/output requirements for modules will pass through three stages as follows:

Proto: Initially all modules will only be required to operate in batch mode, where they receive all their input in a single phase, and run to completion, placing their output in another file. Initially these files will contain our ASCII-based 3-code encoding of the information to facilitate debugging and human monitoring.

Binary: As we proceed toward later milestones the information in the files will be encoded in binary.

Incremental: Eventually the codes will be able to accept a series of files over time whose concatenation constitutes the entire problem. Conceptually, the design could migrate down to a fine-grained resolution where one might envision passing single records via a remote-procedure-call protocol. In the current design only the stages through the Chunk Graph Builder are engineered to work incrementally. All subsequent stages are not so designed: the primary addition that will be required will be stores to record their intermediate states.

The Celera assembler also requires several disk stores for some data components that are too large to fit in memory and these stores along with the intermediates of the pipeline can further be used as check-points for the purposes of fault tolerance. Initially the stores should support sequential and random access via simple file-based implementations on binary objects. This will be considered the **Prototype** milestone for these stores. Unlike the pipeline files, the stores may partition the information in any way that facilitates its operation. Later the stores may be further optimized to promote optimal disk streaming and performance under the anticipated access patterns. This will be called the **Optimal** milestone



The diagram above gives a preliminary data flow diagram. The ".xxx" suffixes at the side of each file box indicate the file extension for the pipeline data in the prototype and binary versions. Inside each file box is the set of prototype objects being communicated. An object that is passed through unchanged retains it prototype name, every object that does change has a different prototype name to avoid an ambiguity (e.g. FRG, IFG, SFG, and OFG for fragments). At each type prototypes inked in purple are modified form of an input message and those inked in green represent new message types. Every pipeline file starts with an audit record (see ADT prototype record).

A pervasive issue for the assembler is the mapping of unique external UIDs to internal accession numbers more useful for efficient indexing and usage of objects. There are several external UIDs passed to the assembler: ScreenItem UIDs, Fragment UIDs, and Distance UIDs. Every class of external UIDs will be translated to consecutive internal IDs within the assembler. This translation is accomplished by the Gate Keeper whose other primary responsibility is to check the validity of the input and further split input buckets if it is deemed necessary. When reporting answers back to the DMS, we need to report objects in external UID terms. Therefore, the translation replaces external UIDs with `uint32` internal IDs, except for `accession` number fields where the external UID is augmented with the internal ID. All such `accession` references to ID's thus become (`uint64`,`uint32`) pairs where the first integer is the external UID and the second is our internal index.

The prototype document details the content of each prototype object. The flow through each pipeline stage is as follows:

Gate Keeper: All messages are checked for consistency, as is the temporal ordering of these messages. Internal IDs are established for all external UIDs except for Repeat_IDs which are already consecutive from 0. Thus the FRG, LKG, DST, and SCR messages become IFG, ILK, IDT, and ISC messages.

Screener: ISC items are consumed and stored in a disk-based image of the UR/C index for fault tolerance and restart purposes (usually the UR/C index is memory resident). IDT, RPT, and ILK items are passed through unaltered. IFG items are augmented with match records and output as SFG items.

Overlap Detector: IDT, ILK, and RPT messages are passed through unaltered. All information in the SFG records is stored in the Fragment store for both searching and check-pointing purposes. A stripped down version of the SFG not containing sequence and quality values are output as OFG messages followed in each case by a sequence of OVL records encoding the detected overlaps. ~~The overlapper also produces BRC messages containing updated and/or new branch point information.~~

Chunk Graph Builder: OVL messages are stored in the Overlap Store. OFG messages are consumed and their relevant information is codified in new IUM messages output for each chunk. In addition to chunking, this phase now also detects branchpoints at the end of chunks and outputs that information in the IUM messages. Moreover the essential edges between chunks are output as UOM messages. The IDT, ILK, and RPT messages are passed on to the next stage that will be maintaining a disk image of this information for check-pointing purposes.

Chunk Graph Walker: The chunk graph receives a collection of IUM and UOM messages that in total encode the chunk. In addition, it receives the IDT, ILK, and RPT messages it needs. Currently this stage does both mate linking and repeat resolution. It merges essential edges and links in its first stage to produce extended chunk graph edges that are ultimately output as IUL messages. It also outputs its final scaffold graph as a collection of PCM messages modeling the contigs, and ICL messages modeling the link edges between them. ILK and RPT messages are consumed here. The IDT message becomes an IMD message wherein the observed histogram of distances is recorded. Finally, the assembler's best guess at the final assembly is output in a set of ISF, or internal scaffold messages.

Consensus Builder: The consensus stage passes all message received through itself, with the exception of PCM messages which it outputs as ICM messages after it has computed the consensus for the contig and determined which chunks needed to be used as surrogates.

Terminator: The terminator produces AFG messages from the fragment store and further converts all

messages it recieves from internal to external form: thus IMD, IUM, IUL, ICM, ICL, and ISF messages are modified to be MDD, UTG, ULK, CTG, CLK, and SCF messages, respectively.

AUTHORS

Gene Myers

```
Created: November, 23 '98

Revised: Jan. 7, '99 by Gene Myers

         Added Gate Keeper and penciled in Chunk Walker.

Revised: Feb. 1, '99 by Gene Myers

         External ID elimination except for accession refs.
         Integration of other chunk builder state.

Revised: Feb. 22, '99 by Gene Myers

         Added dataflow for chunk builder and consensus modules.

Revised: May 5, 1999 by Clark Mobarry

            Removed the join messages and updated the figure.

Revised: June 6, '99 by Gene Myers

         Complete redo of process from chunk graph builder on to
         reflect final ProtoIO design pass.
```