Supplementary Methods

Error correction. The error corrector Quake was run on all data sets, as was the Allpaths-LG error corrector. For Quake, we used a k-mer size of 18, and the recipe to run it was as follows:

```
echo frag_1.fastq      frag_2.fastq >  genome.ls
echo shortjump_1.fastq shortjump_2.fastq >> genome.ls
quake.py -f genome.ls  -k 18 –p 20
```

Allpaths-LG does not include a standalone error corrector; however, we ran the assembler only through the error correction steps and then extracted the corrected reads. The default k-mer sizes in Allpaths-LG are 24 for the fragment reads and 96 for the jump libraries. The error corrected reads are written by the assembly pipeline to disk . It creates two files called "frag_reads_corr.fastb" and "jump_reads_ec.fastb." We converted these files from FASTB to FASTA format and identified mate-pairs based on their adjacent positions in the files. These corrected reads were then used as input to other assemblers in the evaluation.

Software versions. Most of the assemblers in this study are under continual development, and are likely to change and improve over time. The versions used for the results described here are as follows:

- Abyss, version 1.2.7
- Allpaths-LG, release 3-35218
- Bambus, release 3.0.1
- CABOG, release 6.1
- MSR-CA, release 1.0
- SGA, release 0.9.8
- SOAPdenovo, release 1.0.5
- Velvet, release 1.0.13

SOAPdenovo contains a separate module called SOAPGapCloser. This improves the contiguity of assemblies dramatically: for example, on *Rhodobacter*, the contig N50 size increased from ~8 Kb to ~131 Kb, a 16-fold improvement. We ran this module on all assemblies used in these comparisons. Velvet contains a separate module called VelvetOptimizer. We found that this significantly improved assemblies using uncorrected reads. However, when we used corrected reads as input to Velvet, its assemblies were the same or better than the results obtained from VelvetOptimizer.

Recipes for genome assemblies

Here we describe how we ran each assembler on all the genomes described in this study. Note that these recipes may not be optimal for new genomes, but they should provide a good starting point. Better results may be obtained by running an assembler multiple times with different parameter settings, which is the strategy we would recommend.

The data is available at the GAGE website, gage.cbcb.umd.edu. For each genome, we have placed all the assemblies described in this study, the original data, the Quake-corrected reads, and the Allpaths-corrected reads at:

http://gage.cbcb.umd.edu/data/Staphylococcus_aureus/
http://gage.cbcb.umd.edu/data/Rhodobacter_sphaeroides/
http://gage.cbcb.umd.edu/data/Hg_chr14/
http://gage.cbcb.umd.edu/data/Bombus_impatiens/

The original raw data for all genomes is available at NCBI, http://www.ncbi.nih.gov. These recipes should allow others to replicate our results. Note that some algorithms include random components, which may produce some variations when re-running the code at different times. E.g., CABOG breaks ties randomly in its unitig module, and if the order of the input reads is changed even slightly, the resulting assembly will be different.

To run **AllPaths-LG**, we used this command:

Staphylococcus aureus:

```
RunAllPaths3G PRE=. REFERENCE_NAME=. DATA_SUBDIR=. RUN=allpaths
SUBDIR=run ERROR_CORRECTION=True FIX_SOME_INDELS=False
```

Rhodobacter sphaeroides:

```
RunAllPaths3G PRE=. REFERENCE_NAME=. DATA_SUBDIR=. RUN=allpaths
SUBDIR=run
```

Human Chromosome 14:

```
RunAllPaths3G PRE=. REFERENCE_NAME=. DATA_SUBDIR=. RUN=allpaths
SUBDIR=run ERROR_CORRECTION=True FIX_SOME_INDELS=True
```

For **CABOG**, we used the following command:

```
runCA -d . -p asm –s runCA.spec *.frg
```

`runCA.spec` contains multiple parameters used by CA or CABOG during assembly. The following parameters, most of which are defaults, were used in the best assembly for each organism:

Rhodobacter sphaeroides:

```
doOverlapBasedTrimming = 0, unittiger = bog,
bogBreakAtIntersections = 0, bogBadMateDepth = 1000,
merylThreads = 20, merOverlapperThreads = 1,
merOverlapperExtendConcurrency = 20,
merOverlapperSeedConcurrency = 20, ovlThreads = 1,
ovlConcurrency = 20, ovlCorrConcurrency = 20 ,
frgCorrThreads = 1, frgCorrConcurrency = 20, merylMemory =
12800, ovlStoreMemory = 24000, doExtendClearRanges = 0,
cnsConcurrency = 20
```

Human Chromosome 14:

```
doOverlapTrimming = 0, unittiger = bog, merylThreads = 20,
merOverlapperThreads = 1, merOverlapperExtendConcurrency =
20, merOverlapperSeedConcurrency = 20, ovlThreads = 1,
ovlConcurrency = 20, ovlCorrConcurrency = 20,
frgCorrThreads = 4, frgCorrConcurrency = 8, cnsConcurrency
= 20, merylMemory = 12800, obtMerThreshold = 1000,
ovlMerThreshold = 1000, ovlStoreMemory = 12800,
frgCorrBatchSize = 2000000, merOverlapperExtendBatchSize =
2000000, merOverlapperSeedBatchSize = 2000000,
ovlCorrBatchSize = 2000000, doExtendClearRanges = 1,
ovlMemory = 8GM --hashload 0.8, ovlHashBlockSize = 2000000,
ovlRefBlockSize = 32000000
```

Bombus impatiens:

```
doOverlapTrimming = 0, unittiger = bog, merylThreads = 24,
merOverlapperThreads = 1, merOverlapperExtendConcurrency =
24, merOverlapperSeedConcurrency = 24, ovlThreads = 1,
ovlConcurrency = 24, ovlCorrConcurrency = 24,
frgCorrThreads = 4, frgCorrConcurrency = 8, cnsConcurrency
= 24, merylMemory = 8192, obtMerThreshold = 1000,
ovlMerThreshold = 1000, ovlStoreMemory = 8192,
frgCorrBatchSize = 2000000, merOverlapperExtendBatchSize =
2000000, merOverlapperSeedBatchSize = 2000000,
ovlCorrBatchSize = 2000000, doExtendClearRanges = 0,
doFragmentCorrection = 0, ovlMemory = 8GM --hashload 0.8 --
hashstrings 4000000, ovlHashBlockSize = 2000000,
ovlRefBlockSize = 2000000
```

For **Velvet**, we used:

```
velveth . 31 -fastq \
    -shortPaired  frag_12.fastq \
    -shortPaired2 shortjump_12.rev.fastq \
    -shortPaired3 longjump_12.fastq

velvetg . -exp_cov auto \
    -ins_length  $MEA_FRAG      -ins_length_sd  $STD_FRAG \
    -ins_length2 $MEA_SHORTJUMP -ins_length2_sd $STD_SHORTJUMP \
    -ins_length3 $MEA_LONGJUMP  -ins_length3_sd $STD_LONGJUMP \
    -scaffolding yes -exportFiltered yes -unused_reads yes
```

where the `longjump` library was used only for the assembly of human chromosome 14, and $MEA_* and $STD_* are mean and standard deviations of the corresponding libraries.

For **SOAPdenovo**, we created a `SOAPdenovo.config` file and ran the `SOAPdenovo` and `GapCloser` commands as follows:

Staphylococcus aureus:

```
echo "[LIB]\n avg_ins=180\n reverse_seq=0\n asm_flags=1\n
rank=1\n f1=frag_1.cor.fasta\n f2=frag_2.cor.fasta\n" >
SOAPdenovo.config
echo "[LIB]\n avg_ins=3500\n reverse_seq=0\n asm_flags=2\n
rank=2\n f1=shortjump_1.cor.fasta\n f2=shortjump_2.cor.fasta\n"
>> SOAPdenovo.config

SOAPdenovo all -K 31 -p 24 -s SOAPdenovo.config -o asm

GapCloser -b SOAPdenovo.config -a asm.scafSeq -o asm2.scafSeq -t
8 -p 31
```

Rhodobacter sphaeroides:

```
echo "[LIB]\n avg_ins=180\n reverse_seq=0\n asm_flags=1\n
rank=1\n f1=frag_1.cor.fasta\n f2=frag_2.cor.fasta\n" >
SOAPdenovo.config
echo "[LIB]\n avg_ins=3500\n reverse_seq=0\n asm_flags=2\n
rank=2\n f1=shortjump_1.cor.fasta\n f2=shortjump_2.cor.fasta\n"
>> SOAPdenovo.config

SOAPdenovo all -K 31 -p 16 -s SOAPdenovo.config -o asm

GapCloser -b SOAPdenovo.config -a asm.scafSeq -o asm2.scafSeq -t
8 -p 31
```

Human Chromosome 14:

```
echo "[LIB]\n avg_ins=180\n reverse_seq=0\n asm_flags=1\n
rank=1\n f1=chr14_fragment_1.cor.fasta\n
f2=chr14_fragment_2.cor.fasta\n" > SOAPdenovo.config
echo "[LIB]\n avg_ins=3000\n reverse_seq=0\n asm_flags=2\n
rank=2\n f1=chr14_shortjump_1.cor.fasta\n
f2=chr14_shortjump_2.cor.fasta\n" >> SOAPdenovo.config
echo "[LIB]\n avg_ins=35000\n reverse_seq=0\n asm_flags=2\n
rank=3\n f1=chr14_longjump_1.cor.fasta\n
f2=chr14_longjump_2.cor.fasta\n" >> SOAPdenovo.config

SOAPdenovo all -K 47 -p 16 -s SOAPdenovo.config -o asm

GapCloser -b SOAPdenovo.config -a asm.scafSeq -o asm2.scafSeq -t
8 -p 31
```

Bombus impatiens:

```
echo "[LIB]\n avg_ins=400\n reverse_seq=0\n asm_flags=2\n
rank=2\n q1=s_1.1_sequence.cor.rev.fastq\n
q2=s_1.2_sequence.cor.rev.fastq\n" > SOAPdenovo.config
```

```
echo "[LIB]\n avg_ins=8000\n reverse_seq=0\n asm_flags=2\n
rank=3\n q1=s_2.1_sequence.cor.rev.fastq\n
q2=s_2.2_sequence.cor.rev.fastq\n" >> SOAPdenovo.config
echo "[LIB]\n avg_ins=400\n reverse_seq=0\n asm_flags=3\n
rank=1\n q1=s_3.1_sequence.cor.rev.fastq\n
q2=s_3.2_sequence.cor.rev.fastq\n" >> SOAPdenovo.config
echo "[LIB]\n avg_ins=400\n reverse_seq=0\n asm_flags=3\n
rank=1\n q1=s_5.1_sequence.cor.rev.fastq\n
q2=s_5.2_sequence.cor.rev.fastq\n" >> SOAPdenovo.config
echo "[LIB]\n avg_ins=400\n reverse_seq=0\n asm_flags=3\n
rank=1\n q1=s_6.1_sequence.cor.rev.fastq\n
q2=s_6.2_sequence.cor.rev.fastq\n" >> SOAPdenovo.config
echo "[LIB]\n avg_ins=400\n reverse_seq=0\n asm_flags=3\n
rank=1\n q1=s_7.1_sequence.cor.rev.fastq\n
q2=s_7.2_sequence.cor.rev.fastq\n" >> SOAPdenovo.config
echo "[LIB]\n avg_ins=400\n reverse_seq=0\n asm_flags=3\n
rank=1\n q1=s_8.1_sequence.cor.rev.fastq\n
q2=s_8.2_sequence.cor.rev.fastq\n" >> SOAPdenovo.config
echo "[LIB]\n avg_ins=3000\n reverse_seq=0\n asm_flags=2\n
rank=2\n q1=s_9.1_sequence.cor.rev.fastq\n
q2=s_9.2_sequence.cor.rev.fastq\n" >> SOAPdenovo.config

SOAPdenovo all -K 47 -p 16 -s SOAPdenovo.config -o asm

GapCloser -b SOAPdenovo.config -a asm.scafSeq -o asm2.scafSeq -t
8 -p 31
```

For **ABySS**, we used:

Staphylococcus aureus:
```
abyss-pe  \
    k=31 n=5 name=asm lib='frag short' frag=frag_12.cor.fastq
short=short_12.cor.fastq aligner=bowtie
```

Rhodobacter sphaeroides:
```
abyss-pe  \
    k=31 n=5 name=asm lib='frag short' frag=frag_12.cor.fastq
short=short_12.cor.fastq aligner=bowtie
```

Human Chromosome 14:
```
abyss-pe  \
    k=31 n=5 j=6 name=asm lib='frag short long'
frag=chr14_frag_12.cor.fastq short=chr14_shortjump_12.cor.fastq
long=chr14_longjump_12.cor.fastq aligner=bowtie
```

For **SGA**, we used the following commands:

Staphylococcus aureus:

```
sga preprocess -p 1 frag_?.fastq > frag.pp.fa
sga index -t 20 frag.pp.fa
sga correct -k 31 -t 20 frag.pp.fa -o frag.pp.ec.fa
sga index -t 31 frag.pp.ec.fa
sga filter frag.pp.ec.fa
sga overlap -t 20 –m 45 frag.pp.ec.filter.pass.fa
sga assemble frag.pp.ec.filter.pass.asqg.gz
ln -s default-contigs.fa genome.ctg.fasta
sga-align --name frag  genome.ctg.fasta  frag_1.fastq
frag_2.fastq
sga-align --name shortjump genome.ctg.fasta  shortjump_1.fastq
shortjump_2.fastq
sga-bam2de.pl -n 5 --prefix frag frag.bam
sga-bam2de.pl -n 5 --prefix shortjump shortjump.bam
sga-astat.py  -m 200 frag.refsort.bam > genome.ctg.astat
sga scaffold -m 200 --pe frag.de  --mate-pair shortjump.de  -a
genome.ctg.astat -o genome.scaf genome.ctg.fasta
sga scaffold2fasta -m 200 -f genome.ctg.fasta -o genome.scf.fasta
genome.scaf --write-unplaced --use-overlap
```

Rhodobacter sphaeroides:

```
sga preprocess -p 1 frag_?.fastq > frag.pp.fa
sga preprocess -p 1 short_?.fastq > short.pp.fa
sga index -t 20 frag.pp.fa
sga index -t 20 short.pp.fa
sga correct -k 31 -t 20 frag.pp.fa -o frag.pp.ec.fa
sga correct -k 31 -t 20 short.pp.fa -o short.pp.ec.fa
cat frag.pp.ec.fa short.pp.ec.fa > allReads.pp.ec.fa
sga index -t 31 allReads.pp.ec.fa
sga filter allReads.pp.ec.fa
sga overlap -t 20 allReads.pp.ec.filter.pass.fa
sga assemble allReads.pp.ec.filter.pass.asqg.gz
ln -s default-contigs.fa genome.ctg.fasta
sga-align --name frag  genome.ctg.fasta  frag_1.fastq
frag_2.fastq
sga-align --name shortjump genome.ctg.fasta  shortjump_1.fastq
shortjump_2.fastq
sga-bam2de.pl -n 5 --prefix frag frag.bam
sga-bam2de.pl -n 5 --prefix shortjump shortjump.bam
sga-astat.py  -m 200 frag.refsort.bam > genome.ctg.astat
sga scaffold -m 200 --pe frag.de  --mate-pair shortjump.de  -a
genome.ctg.astat -o genome.scaf genome.ctg.fasta
sga scaffold2fasta -m 200 -f genome.ctg.fasta -o genome.scf.fasta
genome.scaf --write-unplaced --use-overlap
```

Human Chromosome 14:

```
sga preprocess -p 1  frag_?.fastq > frag.pp.fa
sga index -t 20 frag.pp.fa
sga correct -k $K -i 10 -t 20 frag.pp.fa -e 0.04 -m 45 -o
frag.pp.ec.fa
sga index -t 20 frag.pp.ec.fa
```

```
sga filter -t 20 frag.pp.ec.fa
sga overlap -m 45 -t 20 frag.pp.ec.filter.pass.fa
sga assemble frag.pp.ec.filter.pass.asqg.gz
ln -s default-contigs.fa genome.ctg.fasta
sga-align --name frag  genome.ctg.fasta  frag_1.fastq
frag_2.fastq
sga-align --name shortjump genome.ctg.fasta  shortjump_1.fastq
shortjump_2.fastq
sga-bam2de.pl -n 5 --prefix frag frag.bam
sga-bam2de.pl -n 5 --prefix shortjump shortjump.bam
sga-astat.py  -m 200 frag.refsort.bam > genome.ctg.astat
sga scaffold -m 200 --pe frag.de  --mate-pair shortjump.de  -a
genome.ctg.astat -o genome.scaf genome.ctg.fasta
sga scaffold2fasta -m 200 -f genome.ctg.fasta -o genome.scf.fasta
genome.scaf --write-unplaced --use-overlap
```

For **MSR-CA**, our recipes were as follows. Running the assembler simply amounts to specifying the locations of input files for various input data types, such as short paired end Illumina reads and jump library mate pairs, in the configuration file. (A brief manual for MSR-CA assembler is available at http://www.genome.umd.edu/SR_CA_MANUAL.htm). Running runSRCA.pl from the assembler bin directory, with the configuration file specified as the only command line parameter, produces the assemble.sh script which contains a complete set of commands to run the assembly. The core assembly engine for MSR-CA is Celera (CABOG) Assembler (http://sourceforge.net/apps/mediawiki/wgs-assembler/index.php?title=Main_Page). The CABOG version runs under the CA folder in the assembly directory. The final products of the assembly such as contig and scaffold fasta files along with additional assembly information can be found under CA/9-terminator/.

In the following we list the notes for the individual assemblies:

1. *R. sphaeroides*: we specified the following parameters to CA in the config file:
   CA_PARAMETERS= ovlMerSize=30 cgwErrorRate=0.25 merylMemory=8192 ovlMemory=4GB
   We kept all other parameters at their default values. We also used only a part of the jump library reads -- the first 400,000 reads, because CA is not designed to handle data sets with over 100x clone coverage.
2. *S. aureus*: we specified the following parameters in the config file:
   CA_PARAMETERS= cgwErrorRate=0.25 merylMemory=8192 ovlMemory=4GB
   EXTEND_JUMP_READS=1
   The EXTEND_JUMP_READS parameter triggered an additional step of extending the jump library reads on the 3' ends to make them compatible with CA. Originally jump library reads were 37 bases long, whereas CA requires reads to be at least 64 bases long. We also used only part of the jump library reads – the first 400,000 reads.

3. *H. sapiens*: we specified the following parameters in the config file: CA_PARAMETERS= cgwErrorRate=0.25 merylMemory=8192 ovlMemory=4GB utgErrorRate=0.03
   We also manually reverse complemented the 35 Kb ("long jump") library before the assembly because MSR-CA assembler assumes that the jump libraries are "outties", that is the 3' ends of the mated reads are on the fragment ends. We used all reads.
4. *B. impatiens*: we specified the following parameters in the config file: CA_PARAMETERS= cgwErrorRate=0.15 merylMemory=8192 ovlMemory=4GB. We used all reads and kept the parameters at their default values.

MSR-CA's run times for the assemblies on a 48-core AMD Opteron 6134 2.2GHz computer with 256 GB of RAM were:

1. R. sphaeroides – 16 minutes
2. S. aureus – 17 minutes
3. H. sapiens – 20 hours
4. B. impatiens – 52 hours

Bambus2. The current version of Bambus 2 does not make use of sequencing data, making it fast but leading to short-range inaccuracies (5bp < indels < 65bp). It includes source and pre-built binaries for a Linux-amd64 to run the Bambus 2 pipeline with CA. The pipeline requires AMOS 3.0.1 and CA 6.1. The *Staphylococcus aureus* assembly also required Bowtie to be installed.
This recipe assumes you have installed these tools and they are available in your path. Mate-pair, "outtie", libraries were reverse-complemented. All datasets and packages are available from the GAGE Bambus 2 recipe page at http://gage.cbcb.umd.edu/recipes/bambus2.html

Staphylococcus aureus. Here are the step-by-step instructions to reproduce this assembly:

1. Step 1: Download tarball above and run `tar xvzf gageBambusPackage.tar.gz`
2. Step 2: Download data below and run `tar xvzf staph.tar.gz`
3. Step 3: Run `cd bambus2`
4. Step 4: Run `fastqToCA -insertsize 180 20 -libraryname frag -innie -type sanger -fastq 'pwd'/frag_1.cor.fastq,'pwd'/frag_2.cor.fastq > frag_12.cor.frg`
5. Step 5: Run `sh run.sh`
6. Step 6: Done!

Rhodobacter sphaeroides. Here are the step-by-step instructions to reproduce this

assembly:

1. Step 1: Download tarball above and run *tar xvzf gageBambusPackage.tar.gz*
2. Step 2: Download data below and run *tar xvzf rhodo.tar.gz*
3. Step 3: Run *cd bambus2*
4. Step 4: Run *fastqToCA -insertsize 180 20 -libraryname frag -innie -type sanger -fastq 'pwd'/frag_1.cor.fastq,'pwd'/frag_2.cor.fastq > frag_12.cor.frg*
5. Step 5: Run *fastqToCA -libraryname short -type sanger -fastq 'pwd'/short_12.cor.ignore > short_12.cor.frg*
6. Step 6: Run *sh run.sh*
7. Step 7: Done!

Human Chromosome 14. Here are the step-by-step instructions to reproduce this assembly:

1. Step 1: Download tarball above and run *tar xvzf gageBambusPackage.tar.gz*
2. Step 2: Download data below and run *tar xvzf human.tar.gz*
3. Step 3: Run *cd bambus2*
4. Step 4: Run *fastqToCA -insertsize 180 20 -libraryname frag -innie -type sanger -fastq 'pwd'/chr14_fragment_1.cor.fastq,'pwd'/chr14_fragment_2.cor.fastq > chr14_fragment_12.cor.fastq*
5. Step 5: Run *fastqToCA -type sanger -libraryname lj -fastq 'pwd'/chr14_longjump_12.cor.fastq > chr14_longjump_12.frg*
6. Step 6: Run *fastqToCA -type sanger -libraryname sj -fastq 'pwd'/chr14_shortjump_12.cor.fastq > chr14_shortjump_12.frg*
7. Step 7: Run *sh run.sh*
8. Step 8: Done!

The main package includes the pipeline to run CABOG + Bambus 2 on genome assemblies. The package has been tested with AMOS v3.0.1 and CA 6.1. Previous versions will not work, but later versions may work. The code is available under src/ and a build is available under bin/. All code is machine-independent, except for the modified CA terminator program. It is build for Linux 64 and can be built for other platforms using the provided source code + CA source. The modifications to terminator are now included in CA after release 6.1

Detailed use instructions: The basic outline to run an assembly is:

- Set up inputs
- Any paired-end (non-jumping) illumina libraries can be input as is using the *fastqToCA* program to generate a frg
- Any mate-pair (jumping) illumina libraries should be input as an interleaved but unmated fastq file to CA
- For these library, the pipeline looks for a file named PREFIX.libSizes which looks like:
  short 2450 4550
  to specify the mate min and max distance for each library.
- Run CA up to unitiggging
- The script relies on the *runCA.sh* script to specify any additional parameters to CA that you would like such as SGE settings or parallel settings. You may also include a CA spec file. A set of CA parameters for execution on a single machine in parallel is included in the recipes above.
- The pipeline automatically runs everything up through unitigging and uses the unitig output to select a threshold for bad mate breaking in CA
- Unitigging and consensus is re-run with the selected cutoff.
- CA output is converted to Bambus 2
- A user has two options to specify this:
  - First, they can take CA directly and add any mates for the library specified in the *PREFIX.libSizes* file. This is the approach taken in Ecoli and Rhodobacter as all the reads can be input to CA directly.
  - Secondly, the user can choose to map the original sequences to the unitigs (bowtie is used). This is the approach taken in *S. aureus* as the jumping libraries are only 35bp. CA 6.1 requires a minimum read length of 64bp and thus 35bp reads cannot be input to CA without making code modifications.
  - The Bambus 2 pipeline is run using the *goBambus2* executive
  - This pipeline generates the final output and fasta sequences for the scaffolds and contigs files.
  - Bambus 2 does not recall consensus so if two unitigs overlap, only of their sequences will be represented in the overlapping region.

The main script (*convertToCA.sh*) with example usage (for *S. aureus* and *R. sphaeroides respectively*):

convertToCA.sh 0 ./ genome ../original 1 utg.fasta
convertToCA.sh 1 ./ genome 0 1

has the following parameters:

- 0 if you want to map, 1 if you want to use the existing CA assembly read placements
- the location of the assembly where it will use the unitigs from, if the assembly doesn't exist in that directly, it will try to run the runCA.sh script there to generate an assembly

- the prefix for your input/output (so it assumes there will be a *PREFIX·libSizes*, *PREFIX·gkpStore*, etc)
- For mapping assemblies, the location of the fastq files to map to your data. These are assumed to have a <libName>.1.fastq and a <libName>.2.fastq. The prefix of the file name determines the library name to compute pairing. It is expected that the file *PREFIX·libSizes* will have an entry for <libName>. For non-mapping assemblies, this specifies whether you want to use unitigs or contigs (0 means unitigs, 1 means contigs).
- Whether you want to run Bambus 2 or only generate an AMOS bank. 1 = run, 0 = do not run, stop at bank generation.
- Only for the mapping assembly, the suffix for the file you want to use for the mapping. So *utg·fasta* means it will look for the file named PREFIX.utg.fasta to map the reads to.

Removal of adapter sequences

The Bombus impatiens reads contain parts of the adapters used for the 3kb and 8kb libraries. We removed these adaptors from the reads available on the GAGE site, http://gage.cbcb.umd.edu/data/index.html. The adapter sequences are:

```
3 CGTAATAACTTCGTATAGCATACATTATACGAAGTTATACGA
3 CGGCATTCCTGCTGAACCGAGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGT
5 GATCGGAAGAGCGGTTCAGCAGGAATGCCGAGATCGGAAGAGCGGTTCAGCAGGAATGCCGAGACCG
```

Data sets used for best results

For each of the four genomes, we tried using raw and corrected reads with each assembler, and we chose the best result to present in the tables in the main text. Space does not allow us to present all the results, but here we list which data set was used for each genome and each assembler.

Staphylococcus aureus:
Uncorrected reads: Allpaths-LG, SGA, MSR-CA
Reads corrected by Quake: ABySS*
Reads corrected by Allpaths-LG: Bambus2, SOAPdenovo, Velvet

Rhodobacter sphaeroides:
Uncorrected reads: MSR-CA, Allpaths-LG, SGA
Reads corrected by Quake: ABySS*
Reads corrected by Allpaths-LG: Bambus2, CABOG, SOAPdenovo, Velvet

Bombus impatiens:
Uncorrected reads: none
Reads corrected by Quake: CABOG, MSR-CA, SOAPdenovo
Reads corrected by Allpaths-LG: none

Human chromosome 14:
Uncorrected reads: Allpaths-LG, MSR-CA, SGA

Reads corrected by Quake: ABySS*, CABOG, Velvet
Reads corrected by Allpaths-LG: Bambus2, SOAPdenovo

*Note added in revision: we discovered a bug in the paired-end version of ABySS, abyss-pe, that prevented it from using the Allpaths-LG corrections. After fixing this bug in our version of the code, we ran ABySS again using a variety of word sizes k. With k=63, abyss-pe was able to produce substantially larger contigs and scaffolds than with uncorrected reads or with the Quake-corrected reads. We did not evaluate these assemblies for correctness, but in an effort to present a more complete picture we include these results in Supplementary Table 3.

Assembly Correctness

Supplementary Table 1 details the full results of the correctness analysis. These figures are taken directly from either the ".report" or ".diff" output generated by dnadiff. The table entries are defined as follows. Unaligned references bases are reference sequences that did not align to any contig; these are most commonly sequencing gaps. Unaligned assembly bases are contig sequences that did not align to the reference genome; these could be low quality or contaminant sequence included in the assembly. Duplicated bases are sequences that occurred more times in the assembly than in the reference genome. (These can be caused by "bubbles" in the assembly graph, which represent either sequencing errors or haplotype polymorphisms. When haplotype differences occur, most assemblers will pick one haplotype to output, but sometimes both haplotypes get inserted in error.) Compressed bases are sequences that occurred more times in the reference than in the assembly: these are usually collapsed repeats. "Bad trim" tallies the bases at the beginning or end of contigs that did not align to the reference. Average identity and SNPs were computed on the one-to-one aligned segments, ignoring duplicated bases. Insertions and deletions (indels) occurring in stretches of less than 5 bp and greater than 5 bp were computed separately, in order to differentiate between point-like indels and larger-scale indels.

A "misjoin" is any event where two sequences are joined together in the assembly in a manner that is inconsistent with the reference. These were tallied for inversion events, relocations, and translocations. Inversions (*Inv*) are a switch between strands (and orientation). Relocations (*Reloc*) connect distant segments from the same chromosome. Translocations (*Transloc*) connect segments from different chromosomes; note that because the human assembly only involved 1 chromosome, translocations were not possible. The number and type of mis-joins are computed by inspecting the relative ordering of aligned segments on the reference and contig. For example, suppose three local alignments {*a,b,c*} have been identified between a contig and the reference. When ordered by contig start coordinate, from lowest to highest, they are ordered {*a,b,c*}. However, when ordered by reference start coordinate, they are ordered {*a,c,b*}. The fact that *b* and *c* have been interchanged in the reference ordering indicates a relocation event. By traversing the aligned

segments in sorted order of their contig start position and noting any inconsistencies in the reference ordering, dnadiff is able to detect and categorize mis-joined contigs. For simplicity, each junction point is tallied individually, so an internal inversion will be counted as two inversion misjoins (one for the 5' and one for the 3' end), while an inversion at the end of a contig will count as just one misjoin. Additionally, no weighting is applied to the misjoins, so a large segmental rearrangement is treated the same as a small one. An alternative scoring scheme might assign different penalties for different mis-assembly types, but we did not explore that here. Supplementary Figure 2 shows an example dotplot for many of the common alignment types used here to categorize mis-assemblies.

The indel profiles in Figure 1 are computed as follows: Let (a_1,a_2) and (b_1,b_2) be two adjacent sequence segments in the reference genome, where a_1 and a_2 are the start and end of segment a, respectively. Let (a_1',a_2') and (b_1',b_2') be the corresponding and collinear segments in the assembly. The distance between the two segments in the reference is measured as $D=(b_1-a_2+1)$ and in the query $D'=(b_1'-a_2'+1)$. It follows that the size of the indel is measured as the difference between these two distances $I=|D-D'|$. A negative distance D or D' indicates overlapping alignment segments caused by repetitive sequence on the ends of each alignment. A positive value for I indicates extra bases in the reference, and a negative value for I indicates extra bases in the assembly. See Supplementary Figure 2 for graphical depictions.

To compute scaffold correctness statistics, we left contigs intact (because their errors were computed separately) and looked at breakpoints between them. Scaffolds were split whenever at least a single N was encountered and numbered sequentially in increasing order. That is, a scaffold X with two gaps (indicated by Ns) will generate scaffolds X_1, X_2, and X_3. The split scaffolds were mapped to the reference using nucmer (--maxmatch -l 30 -D 5 -banded). Matches were filtered and any match below 95% identity was removed. Any match that overlapped another match by over 95% was also removed. Finally, show-tiling (-c -l 1 -i 0 -V 0) was run to generate a tiling of split scaffolds on the reference. Three types of errors were tabulated and scaffolds broken at each. The first, an indel, occurs when a piece of a scaffold is missing from the mapping (X_1 followed by X_3 in the above example). Missing X_2 was not counted as an error if X_2's length was less than 200bp or if the gap between X_1 and X_3 was within 1000bp of X_2's length. The second type of errors is a translocation. This occurs when pieces of a single scaffold map to multiple reference chromosomes. The third error types is an inversion. The inversion occurs whenever a piece of a scaffold (one or more contigs) changes strands within a single scaffold. Finally, we counted the average absolute difference between the scaffold gap estimate (number of Ns) versus the true gap sizes within each assembly.

Supplementary Tables

[Supplementary Table 1 provided as a separate file]

Supplementary Table 2. Assemblies of *Rhodobacter sphaeroides* using different combinations of paired-end libraries as input to the assemblers. Shown are the number of contigs larger than 200 bp and the N50 size, in kilobases (Kb), for each assembly. Allpaths-LG could not be run on any combination except the 180bp+3Kb pairing, because it requires at least one library of overlapping reads and one library of longer-distance linking reads. Note that N50 values are uncorrected, and as shown in Table 3, the true N50 sizes are much lower in some instances; e.g., SOAPdenovo has a corrected N50 of 14.3 kb (rather than 131.7 kb) for assembly with the 180-bp and 3-kb libraries.

Assembler	Libraries used for assembly							
	1 library 180 bp		2 libraries 180 + 210 bp		2 libraries 180 + 3 kb		2 libraries 210 + 3 kb	
	Num	N50	Num	N50	Num	N50	Num	N50
ABySS	2,238	3.4	1,268	7.3	1915	5.9	4,222	1.2
Allpaths-LG	-	-	-	-	204	42.5	-	-
Bambus2	760	8.5	760	8.5	**177**	93.2	1,598	**12.0**
CABOG	612	9.0	518	9.9	322	20.2	**1,093**	2.1
MSR-CA	**550**	**13.9**	**190**	**43.8**	395	22.1	1,292	6.0
SGA	2,802	2.1	1,256	6.9	3067	4.5	3,411	1.3
SOAPdenovo	856	10.1	528	19.6	204	**131.7**	4,780	0.8
Velvet	873	9.6	701	12.6	583	15.7	1,242	5.9

Supplementary Table 3. After fixing a bug in the paired-end module, ABySS was able to use Allpaths-LG corrected reads. This fix combined with a much longer k-mer length for its de Bruijn graph produced much larger contigs and scaffolds, as shown here.

ABySS update	Original: Quake-corrected reads, k=31		Updated: Allpaths-corrected reads, k=63	
	Contig N50	Scaffold N50	Contig N50	Scaffold N50
S. aureus	29.2 kb	32.5 kb	129 kb	170 kb
R. sphaeroides	5.9 kb	8.9 kb	19.7 kb	50.8 kb
Human chr 14	2.0 kb	2.1 kb	14.7 kb	18.4 kb

Supplementary Figure legends

Supplementary Figure 1: Average depth of coverage for human chromosome 14 in the read data used in this study. Coverage is computed by calculating the coverage in 50 Kb non-overlapping intervals. As the plot illustrates, there were no large gaps in coverage across the chromosome. Note that this alignment is relative to the finished part of Hs14, which does not include the centromeric gap.

Supplementary Figure 2: A taxonomy of common dot plot motifs representing insertions, deletions, inversions, and relocations that can be identified through inspection of Nucmer alignments.