--- config_kmeans.py ---

```
"""
Configuration for K-Means clustering - ML System Optimization Assignment
"""

# Dataset
DATASET = "mnist"  # mnist or blobs (synthetic)
DATA_DIR = "./data"
N_SAMPLES = 1500   # Digits has 1797; use subset for faster runs

# K-Means
N_CLUSTERS = 10
MAX_ITERS = 100

# Parallel
N_WORKERS = -1  # -1 = use all CPU cores (for parallel script)
```

--- kmeans_baseline.py ---

```
"""
Baseline: Single-process K-Means clustering.
Use for correctness comparison and speedup benchmarking.
"""

import argparse
import time
import numpy as np
from sklearn.datasets import load_digits, make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score, adjusted_rand_score

from config_kmeans import (
    DATASET,
    N_SAMPLES,
    N_CLUSTERS,
    MAX_ITERS,
)


def load_data(dataset: str, n_samples: int = None):
    """Load dataset for clustering. Returns (X, y) where y is true labels (for ARI)."""
    if dataset == "mnist":
        data = load_digits()
        X, y = data.data, data.target
    elif dataset == "blobs":
        X, y = make_blobs(n_samples=10000, n_features=64, centers=10, random_state=42)
    else:
        raise ValueError(f"Unknown dataset: {dataset}")

    if n_samples and n_samples < len(X):
        rng = np.random.default_rng(42)
        n_samples = min(n_samples, len(X))
        idx = rng.choice(len(X), n_samples, replace=False)
        X, y = X[idx], y[idx]
    X = StandardScaler().fit_transform(X)
    return X, y


def kmeans_baseline(X: np.ndarray, k: int, max_iters: int, seed: int = 42):
    """
    Standard K-Means: single process, sequential assignment and update.
    """
    rng = np.random.default_rng(seed)
    n_samples, n_features = X.shape

    # Initialize centroids: k-means++
    centroids = np.zeros((k, n_features))
    centroids[0] = X[rng.integers(n_samples)]
    for i in range(1, k):
        dist_sq = np.min(np.sum((X[:, None] - centroids[:i]) ** 2, axis=2), axis=1)
```

```
        probs = dist_sq / dist_sq.sum()
        centroids[i] = X[rng.choice(n_samples, p=probs)]

    for _ in range(max_iters):
        # Assignment: each point -> nearest centroid
        dists = np.sum((X[:, None] - centroids) ** 2, axis=2)
        labels = np.argmin(dists, axis=1)

        # Update: new centroid = mean of assigned points
        new_centroids = np.zeros_like(centroids)
        for j in range(k):
            mask = labels == j
            if mask.sum() > 0:
                new_centroids[j] = X[mask].mean(axis=0)
            else:
                new_centroids[j] = centroids[j]

        if np.allclose(centroids, new_centroids):
            break
        centroids = new_centroids

    return centroids, labels


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--dataset", type=str, default=DATASET)
    parser.add_argument("--n-samples", type=int, default=N_SAMPLES)
    parser.add_argument("--k", type=int, default=N_CLUSTERS)
    parser.add_argument("--max-iters", type=int, default=MAX_ITERS)
    args = parser.parse_args()

    print("Loading data...")
    X, y_true = load_data(args.dataset, args.n_samples)
    print(f"Data shape: {X.shape}")

    print("\n--- Baseline (Single Process) K-Means ---\n")
    start = time.perf_counter()
    centroids, labels = kmeans_baseline(X, args.k, args.max_iters)
    elapsed = time.perf_counter() - start

    # Metrics: inertia, silhouette, ARI (when ground truth available)
    inertia = sum(np.sum((X[labels == j] - centroids[j]) ** 2) for j in range(args.k))
    silhouette = silhouette_score(X, labels)
    ari = adjusted_rand_score(y_true, labels) if y_true is not None else None

    print(f"Converged in {len(np.unique(labels))} clusters")
    print(f"Training time: {elapsed:.2f}s")
    print(f"Inertia: {inertia:.2f}")
    print(f"Silhouette score: {silhouette:.4f}")
    if ari is not None:
        print(f"Adjusted Rand Index: {ari:.4f}")


if __name__ == "__main__":
    main()
```

--- kmeans_parallel.py ---

```
"""
Parallel K-Means: multiprocessing for assignment step.
Each worker processes a chunk of data points; centroids updated via reduction.
"""

import argparse
import os
import time
import numpy as np
from joblib import Parallel, delayed
from sklearn.datasets import load_digits, make_blobs
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score, adjusted_rand_score

from config_kmeans import (
    DATASET,
    N_SAMPLES,
    N_CLUSTERS,
    MAX_ITERS,
    N_WORKERS,
)


def load_data(dataset: str, n_samples: int = None):
    """Load dataset for clustering. Returns (X, y) where y is true labels (for ARI)."""
    if dataset == "mnist":
        data = load_digits()
        X, y = data.data, data.target
    elif dataset == "blobs":
        X, y = make_blobs(n_samples=10000, n_features=64, centers=10, random_state=42)
    else:
        raise ValueError(f"Unknown dataset: {dataset}")

    if n_samples and n_samples < len(X):
        rng = np.random.default_rng(42)
        n_samples = min(n_samples, len(X))
        idx = rng.choice(len(X), n_samples, replace=False)
        X, y = X[idx], y[idx]
    X = StandardScaler().fit_transform(X)
    return X, y


def _assign_chunk(X_chunk: np.ndarray, centroids: np.ndarray):
    """
    Worker: assign chunk of points to nearest centroid.
    Returns (labels_chunk, sum_per_cluster, count_per_cluster).
    """
    k = centroids.shape[0]
    dists = np.sum((X_chunk[:, None] - centroids) ** 2, axis=2)
    labels = np.argmin(dists, axis=1)

    sums = np.zeros_like(centroids)
    counts = np.zeros(k, dtype=np.int64)
    for j in range(k):
        mask = labels == j
        counts[j] = mask.sum()
        if counts[j] > 0:
            sums[j] = X_chunk[mask].sum(axis=0)
    return sums, counts


def kmeans_parallel(
    X: np.ndarray,
    k: int,
    max_iters: int,
    n_workers: int = -1,
    seed: int = 42,
):
    """
    Parallel K-Means: assignment step distributed across workers via joblib.
    """
    rng = np.random.default_rng(seed)
    n_samples, n_features = X.shape

    # Initialize centroids: k-means++
    centroids = np.zeros((k, n_features))
    centroids[0] = X[rng.integers(n_samples)]
    for i in range(1, k):
        dist_sq = np.min(np.sum((X[:, None] - centroids[:i]) ** 2, axis=2), axis=1)
        probs = dist_sq / dist_sq.sum()
        centroids[i] = X[rng.choice(n_samples, p=probs)]
```

```
    n_chunks = os.cpu_count() if n_workers == -1 else n_workers
    n_chunks = max(1, min(n_chunks, n_samples))

    for _ in range(max_iters):
        chunks = np.array_split(X, n_chunks)
        results = Parallel(n_jobs=n_workers)(
            delayed(_assign_chunk)(chunk, centroids) for chunk in chunks
        )

        # Aggregate: sum and count per cluster
        total_sums = np.zeros_like(centroids)
        total_counts = np.zeros(k, dtype=np.int64)
        for sums, counts in results:
            total_sums += sums
            total_counts += counts

        # Update centroids
        new_centroids = np.zeros_like(centroids)
        for j in range(k):
            if total_counts[j] > 0:
                new_centroids[j] = total_sums[j] / total_counts[j]
            else:
                new_centroids[j] = centroids[j]

        if np.allclose(centroids, new_centroids):
            break
        centroids = new_centroids

    # Final assignment (single pass or parallel) for labels
    dists = np.sum((X[:, None] - centroids) ** 2, axis=2)
    labels = np.argmin(dists, axis=1)
    return centroids, labels


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--dataset", type=str, default=DATASET)
    parser.add_argument("--n-samples", type=int, default=N_SAMPLES)
    parser.add_argument("--k", type=int, default=N_CLUSTERS)
    parser.add_argument("--max-iters", type=int, default=MAX_ITERS)
    parser.add_argument("--n-workers", type=int, default=N_WORKERS)
    args = parser.parse_args()

    print("Loading data...")
    X, y_true = load_data(args.dataset, args.n_samples)
    print(f"Data shape: {X.shape}")

    n_workers = args.n_workers if args.n_workers > 0 else -1
    print(f"\n--- Parallel K-Means ({n_workers} workers) ---\n")
    start = time.perf_counter()
    centroids, labels = kmeans_parallel(
        X, args.k, args.max_iters, n_workers=n_workers
    )
    elapsed = time.perf_counter() - start

    inertia = sum(np.sum((X[labels == j] - centroids[j]) ** 2) for j in range(args.k))
    silhouette = silhouette_score(X, labels)
    ari = adjusted_rand_score(y_true, labels) if y_true is not None else None

    print(f"Converged in {len(np.unique(labels))} clusters")
    print(f"Training time: {elapsed:.2f}s")
    print(f"Inertia: {inertia:.2f}")
    print(f"Silhouette score: {silhouette:.4f}")
    if ari is not None:
        print(f"Adjusted Rand Index: {ari:.4f}")


if __name__ == "__main__":
    main()
```

--- run_benchmark_kmeans.py ---

```
"""
Run baseline and parallel K-Means for benchmarking.
Captures: time, inertia, silhouette score, ARI, speedup.
Outputs results for report and PDF generation.
"""

import json
import os
import sys
import time

# Add project root for imports
sys.path.insert(0, os.path.dirname(os.path.abspath(__file__)))

import numpy as np
from sklearn.metrics import silhouette_score, adjusted_rand_score

from kmeans_baseline import kmeans_baseline, load_data
from kmeans_parallel import kmeans_parallel


def run_benchmark(n_samples=10000, k=10, max_iters=50, dataset="mnist", n_workers=-1):
    """Run both baseline and parallel, return metrics dict."""
    print("Loading data...")
    X, y_true = load_data(dataset, n_samples)
    print(f"Data shape: {X.shape}, Dataset: {dataset}\n")

    results = {"dataset": dataset, "n_samples": n_samples, "k": k, "max_iters": max_iters}

    # Baseline
    print("--- Baseline (Single Process) ---")
    start = time.perf_counter()
    centroids_b, labels_b = kmeans_baseline(X, k, max_iters)
    t_baseline = time.perf_counter() - start

    inertia_b = sum(np.sum((X[labels_b == j] - centroids_b[j]) ** 2) for j in range(k))
    sil_b = silhouette_score(X, labels_b)
    ari_b = adjusted_rand_score(y_true, labels_b)

    results["baseline"] = {
        "time_s": round(t_baseline, 2),
        "inertia": round(float(inertia_b), 2),
        "silhouette": round(float(sil_b), 4),
        "ari": round(float(ari_b), 4),
    }
    print(f"Time: {t_baseline:.2f}s | Inertia: {inertia_b:.2f} | Silhouette: {sil_b:.4f} | ARI: {ari_b:.4f}\n")

    # Parallel
    n_w = n_workers if n_workers > 0 else os.cpu_count()
    print(f"--- Parallel ({n_w} workers) ---")
    start = time.perf_counter()
    centroids_p, labels_p = kmeans_parallel(X, k, max_iters, n_workers=n_workers)
    t_parallel = time.perf_counter() - start

    inertia_p = sum(np.sum((X[labels_p == j] - centroids_p[j]) ** 2) for j in range(k))
    sil_p = silhouette_score(X, labels_p)
    ari_p = adjusted_rand_score(y_true, labels_p)

    results["parallel"] = {
        "time_s": round(t_parallel, 2),
        "inertia": round(float(inertia_p), 2),
        "silhouette": round(float(sil_p), 4),
        "ari": round(float(ari_p), 4),
    }
    results["speedup"] = round(t_baseline / t_parallel, 2) if t_parallel > 0 else 0

    print(f"Time: {t_parallel:.2f}s | Inertia: {inertia_p:.2f} | Silhouette: {sil_p:.4f} | ARI: {ari_p:.4f}\n")

    print("=" * 60)
```

```
    print("BENCHMARK SUMMARY")
    print("=" * 60)
    print(f"Baseline time: {t_baseline:.2f}s")
    print(f"Parallel time: {t_parallel:.2f}s")
    print(f"Speedup: {results['speedup']:.2f}x")
    print(f"Correctness: Inertia diff={abs(inertia_b-inertia_p):.2f}, ARI diff={abs(ari_b-ari_p):.4f}")

    return results


def main():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--n-samples", type=int, default=10000)
    parser.add_argument("--k", type=int, default=10)
    parser.add_argument("--max-iters", type=int, default=50)
    parser.add_argument("--dataset", type=str, default="mnist")
    parser.add_argument("--n-workers", type=int, default=-1)
    parser.add_argument("--output", type=str, default="benchmark_results.json")
    args = parser.parse_args()

    results = run_benchmark(
        n_samples=args.n_samples,
        k=args.k,
        max_iters=args.max_iters,
        dataset=args.dataset,
        n_workers=args.n_workers,
    )
    with open(args.output, "w") as f:
        json.dump(results, f, indent=2)
    print(f"\nResults saved to {args.output}")


if __name__ == "__main__":
    main()
```

--- generate_pdfs.py ---

```
"""
Generate assignment deliverables: code.pdf, report.pdf (or .html fallback).
Run after: python run_benchmark_kmeans.py --output benchmark_results.json

If reportlab fails to install (e.g. Pillow build issues), generates HTML instead.
Open report.html/code.html in browser and use Print -> Save as PDF.
"""

import html
import json
import os
import sys
from pathlib import Path

# Try reportlab; fall back to HTML if unavailable
USE_REPORTLAB = False
try:
    from reportlab.lib.pagesizes import letter
    from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
    from reportlab.lib.units import inch
    from reportlab.platypus import SimpleDocTemplate, Paragraph, Preformatted, Spacer, Table, TableStyle
    from reportlab.lib import colors
    USE_REPORTLAB = True
except ImportError:
    pass


# Code files to include in code.pdf (K-means assignment)
CODE_FILES = [
    "config_kmeans.py",
    "kmeans_baseline.py",
    "kmeans_parallel.py",
```

```
    "run_benchmark_kmeans.py",
    "generate_pdfs.py",
]

# Team contribution table (edit as needed; corresponding author in bold)
# Format: (name, roll_number, contribution_phrase, is_corresponding_author)
TEAM_CONTRIBUTION = [
    ("DEBASHIS KUMAR SERAOGI", "2024ad05321", "Problem formulation, design", False),
    ("BANDI DEEPIKA", "2024ad05231", "Literature survey, report drafting", False),
    ("DURGARAJU SIVA SAKET", "2024ad05485", "Implementation, benchmarking", False),
    ("GOKUL PRASANTH A.", "2024ad05345", "Testing, results analysis", False),
    ("KADMLLU AMIT SUNIL", "2024ad05153", "Implementation, integration, submission", False),
]


def generate_code_pdf(output_path="code.pdf"):
    """Create PDF or HTML of source code."""
    base = Path(__file__).parent
    if USE_REPORTLAB:
        doc = SimpleDocTemplate(output_path, pagesize=letter, topMargin=0.5*inch, bottomMargin=0.5*inch)
        styles = getSampleStyleSheet()
        story = []
        code_style = ParagraphStyle(name="Code", fontName="Courier", fontSize=8, leading=10)
        for filename in CODE_FILES:
            filepath = base / filename
            if not filepath.exists():
                continue
            story.append(Paragraph(f"<b>--- {filename} ---</b>", styles["Normal"]))
            story.append(Spacer(1, 6))
            with open(filepath, "r", encoding="utf-8") as f:
                code = f.read()
            story.append(Preformatted(code, code_style))
            story.append(Spacer(1, 12))
        doc.build(story)
    else:
        out = output_path.replace(".pdf", ".html")
        parts = ['<!DOCTYPE html><html><head><meta charset="utf-8"><title>Code</title>']
        parts.append('<style>pre{font-family:monospace;font-size:11px;}</style></head><body>')
        for filename in CODE_FILES:
            filepath = base / filename
            if not filepath.exists():
                continue
            with open(filepath, "r", encoding="utf-8") as f:
                code = html.escape(f.read())
            parts.append(f'<h3>{html.escape(filename)}</h3><pre>{code}</pre>')
        parts.append('</body></html>')
        with open(out, "w", encoding="utf-8") as f:
            f.write("\n".join(parts))
        print(f"Generated: {out} (open in browser, Print -> Save as PDF)")
        return
    print(f"Generated: {output_path}")


def _write_report_html(out_path, sections_text, results=None, github_url=""):
    """Write report as HTML when reportlab unavailable."""
    html_parts = [
        '<!DOCTYPE html><html><head><meta charset="utf-8"><title>Report</title>',
        '<style>',
        'body{font-family:Helvetica,sans-serif;max-width:700px;margin:40px auto;line-height:1.5;}',
        'h1,h2{color:#333;} table{border-collapse:collapse;} th,td{border:1px solid #ccc;padding:8px;}',
        '.report-title{font-size:1.15em;font-weight:bold;margin-bottom:0.25em;}',
        '.report-subtitle{font-size:1em;margin-bottom:0.15em;}',
        '.report-parts{font-size:0.9em;color:#444;margin-bottom:1em;}',
        '@media print{body{max-width:100%;}.page-break{page-break-before:always;}}',
        '</style></head><body>',
        '<p class="report-title">ML System Optimization - Assignment 2</p>',
        '<p class="report-subtitle">Parallel K-Means Clustering</p>',
        '<p class="report-parts">[P0] Problem Formulation, [P1] Design, [P2] Implementation, [P3] Results</p>',
    ]
    for title, body in sections_text:
```

```
            if title == "ML System Optimization - Assignment 2":
                continue  # already in title block above
            html_parts.append(f'<h2>{html.escape(title)}</h2>')
            if title == "Facing Sheet":
                gh = github_url or "https://github.com/akadmllu/Assignment-2"
                html_parts.append('<p><b>GitHub (link to code):</b> ' + html.escape(gh) + '</p>')
                html_parts.append('<p><b>Team Contribution:</b></p>')
                html_parts.append('<table><tr><th>Name</th><th>Roll Number</th><th>Contribution</th></tr>')
                for name, roll, contrib, is_corr in TEAM_CONTRIBUTION:
                    name_cell = f"<b>{html.escape(name)}</b>" if is_corr else html.escape(name)
                    html_parts.append(f'<tr><td>{name_cell}</td><td>{html.escape(roll)}</td><td>{html.escape(contri
                html_parts.append('</table>')
            else:
                for p in body.split("\n\n"):
                    html_parts.append(f'<p>{html.escape(p).replace(chr(10), "<br/>")}</p>')
        if results:
            html_parts.append('<h2>Benchmark Results</h2><table>')
            for row in [["Metric", "Baseline", "Parallel"], ["Time (s)", results.get("baseline", {}).get("time_s"),
                html_parts.append(f'<tr><td>{row[0]}</td><td>{row[1]}</td><td>{row[2]}</td></tr>')
            html_parts.append(f'<tr><td>Speedup</td><td colspan="2">{results.get("speedup")}x</td></tr></table>')
        html_parts.append('</body></html>')
        with open(out_path, "w", encoding="utf-8") as f:
            f.write("\n".join(html_parts))
        print(f"Generated: {out_path} (open in browser, Print -> Save as PDF)")


    def generate_report_pdf(output_path="report.pdf", results_path="benchmark_results.json", github_url=""):
        """Create report PDF or HTML with P0-P3 structure and benchmark results."""
        results = None
        if os.path.exists(results_path):
            with open(results_path) as f:
                results = json.load(f)

        sections = []

        def add_section(title, body):
            sections.append((title, body))

        add_section("ML System Optimization - Assignment 2", "Parallel K-Means Clustering")
        github_text = github_url if github_url else "[INSERT LINK - to be included in facing sheet]"
        add_section("Facing Sheet",
            f"GitHub: {github_text}\n\n"
            "Team Contribution:\n"
            "Name | Roll Number | Contribution")
        add_section("Introduction",
            "This assignment addresses ML System Optimization through parallelization of the K-Means clustering alg
            "We implement and compare a baseline (single-process) and a parallel (multi-process) version.")
        add_section("Literature Survey",
            "K-Means clustering [MacQueen, 1967]. Parallelization: data parallelism (Spark MLlib, Dask-ML). "
            "k-means++ [Arthur & Vassilvitskii, 2007]. Joblib for single-machine parallelism.")
        add_section("Abstract",
            "Data-parallel K-Means using Python and joblib. Compare baseline vs parallel on Digits dataset. "
            "Metrics: training time, inertia, silhouette score, Adjusted Rand Index.")
        add_section("P0: Problem Formulation",
            "Algorithm: K-Means. Parallelization: data parallelism over assignment step. "
            "Expectations: Speedup ~linear with CPU cores; Communication O(k*d); Reduced response time.")
        add_section("P1: Design",
            "Architecture: Single-machine, multi-process with joblib. Chunk-based split, parallel assignment, k-mea
        add_section("P1 (Revised): Implementation Details",
            "Environment: Python 3.10+, CPU multi-core. Libraries: NumPy, scikit-learn, joblib.")
        add_section("P2: Implementation",
            "Files: kmeans_baseline.py, kmeans_parallel.py, run_benchmark_kmeans.py.")
        if results:
            b, p = results.get("baseline", {}), results.get("parallel", {})
            spd = results.get("speedup", 0)
            add_section("P3: Results and Discussion",
                f"Dataset: {results.get('dataset')}, n={results.get('n_samples')}. "
                f"Baseline: Time={b.get('time_s')}s, Inertia={b.get('inertia')}, Silhouette={b.get('silhouette')}, A
                f"Parallel: Time={p.get('time_s')}s. Speedup: {spd}x. Correctness: inertia and ARI comparable.")
            add_section("Deviation from Expectations",
```

```
        f"If speedup ({spd}x) is below expected (e.g. ~linear with cores): possible causes—overhead from pr
        "small dataset size, or I/O bottlenecks. If clustering quality (ARI) differs between baseline and pa
        "k-means is stochastic; small differences are normal due to floating-point order. "
        "Fill in specific reasons if your results deviated significantly.")
else:
    add_section("P3: Results and Discussion", "Run run_benchmark_kmeans.py then re-run this script.")
    add_section("Deviation from Expectations", "After running benchmark, add reasons if results deviated fr
add_section("Conclusion", "Successfully parallelized K-Means using joblib with measurable speedup.")
add_section("References",
    "[1] MacQueen (1967). [2] Arthur & Vassilvitskii (2007) k-means++. [3] scikit-learn, joblib docs.")

if not USE_REPORTLAB:
    _write_report_html(output_path.replace(".pdf", ".html"), sections, results, github_url)
    return

def add_page_footer(canv, _doc):
    """Draw '-- N --' at bottom center."""
    page_num = canv.getPageNumber()
    canv.saveState()
    canv.setFont("Helvetica", 9)
    canv.drawCentredString(letter[0] / 2, 0.5 * inch, f"-- {page_num} --")
    canv.restoreState()

doc = SimpleDocTemplate(
    output_path,
    pagesize=letter,
    topMargin=0.75 * inch,
    bottomMargin=0.75 * inch,
    onFirstPage=add_page_footer,
    onLaterPages=add_page_footer,
)
styles = getSampleStyleSheet()
story = []

def section(title, body):
    story.append(Paragraph(f"<b>{title}</b>", styles["Heading2"]))
    for para in body.split("\n\n"):
        story.append(Paragraph(para.replace("\n", "<br/>"), styles["Normal"]))
    story.append(Spacer(1, 12))

# Page 1: Title block
story.append(Paragraph(
    "<b>ML System Optimization - Assignment 2</b>",
    ParagraphStyle(name="ReportTitle", fontName="Helvetica-Bold", fontSize=14, spaceAfter=6),
))
story.append(Paragraph(
    "Parallel K-Means Clustering",
    ParagraphStyle(name="Subtitle", fontName="Helvetica", fontSize=12, spaceAfter=4),
))
story.append(Paragraph(
    "[P0] Problem Formulation, [P1] Design, [P2] Implementation, [P3] Results",
    ParagraphStyle(name="Parts", fontName="Helvetica", fontSize=10, spaceAfter=16),
))
# Facing sheet: GitHub + team table
gh_text = github_url if github_url else "[INSERT LINK - to be included in facing sheet]"
story.append(Paragraph(f"<b>GitHub (link to code):</b> {gh_text}", styles["Normal"]))
story.append(Spacer(1, 8))
story.append(Paragraph("<b>Team Contribution:</b>", styles["Normal"]))
# Table: header + 5 members
table_data = [["Name", "Roll Number", "Contribution"]]
small_style = ParagraphStyle(name="Small", fontName="Helvetica", fontSize=9)
for name, roll, contrib, is_corr in TEAM_CONTRIBUTION:
    name_para = Paragraph(f"<b>{name}</b>" if is_corr else name, small_style)
    table_data.append([name_para, roll, Paragraph(contrib.replace("\n", "<br/>"), small_style)])
tbl = Table(table_data, colWidths=[1.8 * inch, 1.1 * inch, 3.1 * inch])
tbl.setStyle(TableStyle([
    ("BACKGROUND", (0, 0), (-1, 0), colors.lightgrey),
    ("GRID", (0, 0), (-1, -1), 0.5, colors.black),
    ("VALIGN", (0, 0), (-1, -1), "TOP"),
]))
```

```
story.append(tbl)
story.append(Spacer(1, 16))

# Introduction
section(
    "Introduction",
    "This assignment addresses ML System Optimization through parallelization of the K-Means clustering alg
    "We implement and compare a baseline (single-process) and a parallel (multi-process) version, "
    "demonstrating correctness and performance improvement on CPU-based execution."
)

# Literature Survey
section(
    "Literature Survey",
    "K-Means clustering [MacQueen, 1967] is a widely used unsupervised algorithm. "
    "Parallelization strategies include data parallelism (splitting points across workers), "
    "as in Spark MLlib and Dask-ML; and model parallelism for streaming variants. "
    "k-means++ [Arthur & Vassilvitskii, 2007] improves initialization. "
    "Joblib and multiprocessing enable single-machine parallelism in Python."
)

# Abstract
section(
    "Abstract",
    "This report presents the parallelization of K-Means clustering for ML System Optimization. "
    "We implement data-parallel K-Means using Python and joblib, distributing the assignment step "
    "across CPU cores. We compare baseline (single-process) and parallel implementations on the "
    "Digits dataset, measuring training time, inertia, silhouette score, and Adjusted Rand Index."
)

# P0
section(
    "P0: Problem Formulation",
    "Algorithm: K-Means clustering. Parallelization: Data parallelism over the assignment step. "
    "Each worker processes a chunk of data points. Expectations: Speedup ~linear with CPU cores; "
    "Communication cost O(k*d) per iteration; Reduced response time."
)

# P1
section(
    "P1: Design",
    "Architecture: Single-machine, multi-process parallelism using joblib. "
    "Key choices: chunk-based data split, parallel assignment, sequential centroid update, k-means++ init."
)

# P1 Revised
section(
    "P1 (Revised): Implementation Details",
    "Environment: Python 3.10+, CPU multi-core. Libraries: NumPy, scikit-learn, joblib."
)

# P2
section(
    "P2: Implementation",
    "Files: kmeans_baseline.py, kmeans_parallel.py, run_benchmark_kmeans.py. "
    "Run: python kmeans_baseline.py | python kmeans_parallel.py | python run_benchmark_kmeans.py"
)

# P3
if results:
    b = results.get("baseline", {})
    p = results.get("parallel", {})
    speedup = results.get("speedup", 0)
    section(
        "P3: Results and Discussion",
        f"Dataset: {results.get('dataset')}, n={results.get('n_samples')}, k={results.get('k')}. "
        f"Baseline: Time={b.get('time_s')}s, Inertia={b.get('inertia')}, Silhouette={b.get('silhouette')}, A
        f"Parallel: Time={p.get('time_s')}s, Inertia={p.get('inertia')}, Silhouette={p.get('silhouette')}, A
        f"Speedup: {speedup}x. Correctness: inertia and ARI comparable between baseline and parallel."
    )
```

```
        section(
            "Deviation from Expectations",
            "If speedup is below expected (e.g. near-linear with CPU cores): possible causes include overhead fr
            "process spawning, small dataset size, or I/O bottlenecks. If clustering quality (ARI/silhouette) di
            "k-means is stochastic; small differences are normal. Fill in specific reasons if your results devia
            "significantly from expectations."
        )
    else:
        section("P3: Results and Discussion", "Run 'python run_benchmark_kmeans.py' then re-run this script.")
        section("Deviation from Expectations", "After running benchmark, add reasons if results deviated from ex

    # Conclusion
    section(
        "Conclusion",
        "We successfully parallelized K-Means using joblib, achieving measurable speedup on multi-core CPUs."
    )

    # References
    story.append(Paragraph("<b>References</b>", styles["Heading2"]))
    for r in [
        "[1] MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations.",
        "[2] Arthur, D. & Vassilvitskii, S. (2007). k-means++: The Advantages of Careful Seeding.",
        "[3] scikit-learn KMeans, joblib Parallel documentation.",
    ]:
        story.append(Paragraph(r, styles["Normal"]))

    doc.build(story)
    print(f"Generated: {output_path}")


def main():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--code", action="store_true", help="Generate code.pdf")
    parser.add_argument("--report", action="store_true", help="Generate report.pdf")
    parser.add_argument("--all", action="store_true", help="Generate both (default)")
    parser.add_argument("--results", type=str, default="benchmark_results.json")
    parser.add_argument("--github", type=str, default="https://github.com/akadmllu/Assignment-2", help="GitHub U
    args = parser.parse_args()

    if args.all or (not args.code and not args.report):
        generate_code_pdf()
        generate_report_pdf(results_path=args.results, github_url=args.github)
    else:
        if args.code:
            generate_code_pdf()
        if args.report:
            generate_report_pdf(results_path=args.results, github_url=args.github)


if __name__ == "__main__":
    main()
```