# Gossamer

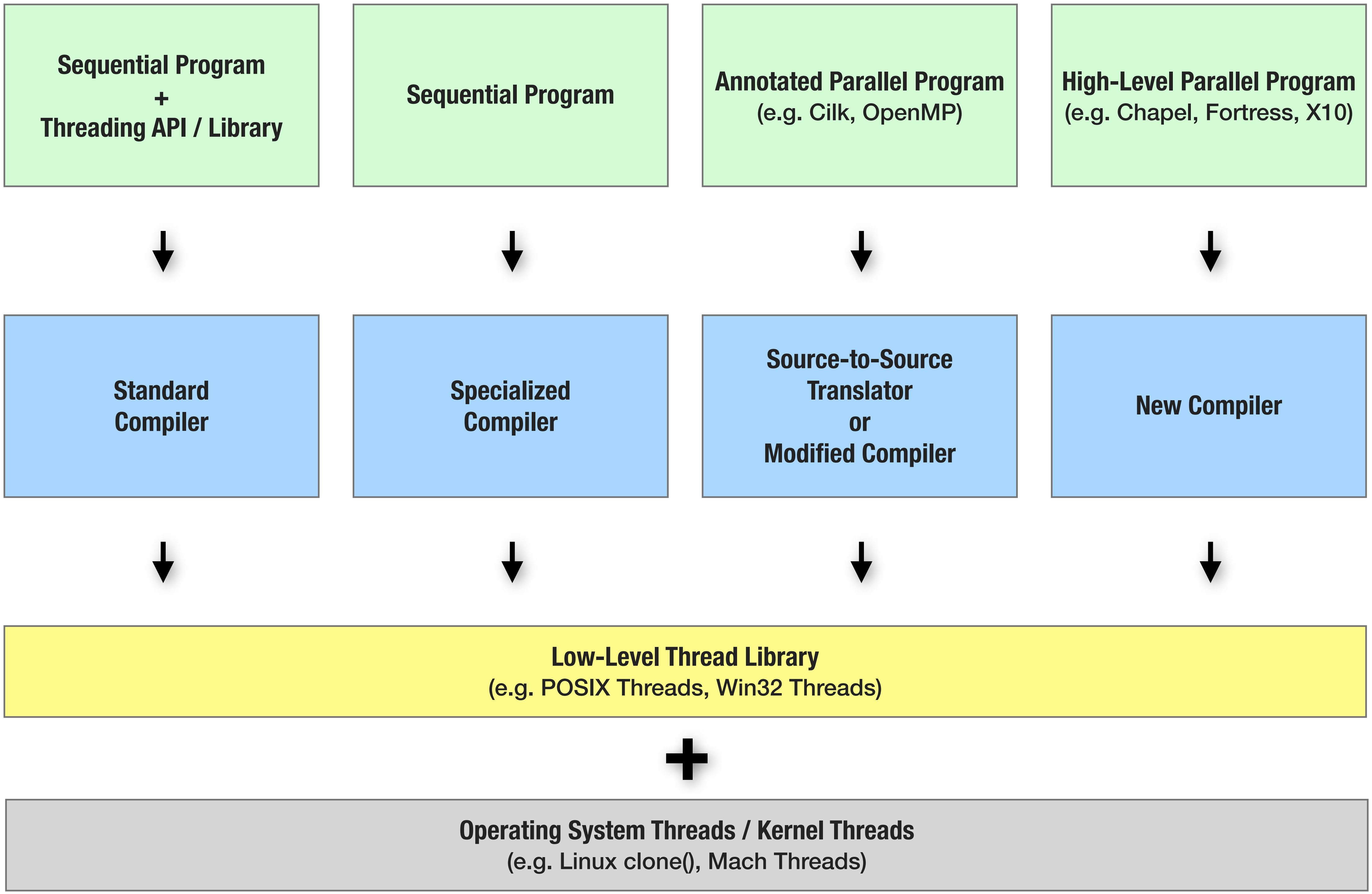
## A Lightweight Programming Framework for Multicore Machines

Joseph A. Roback and Gregory R. Andrews
Department of Computer Science
The University of Arizona

2nd USENIX Workshop on Hot Topics in Parallelism
June 15th, 2010

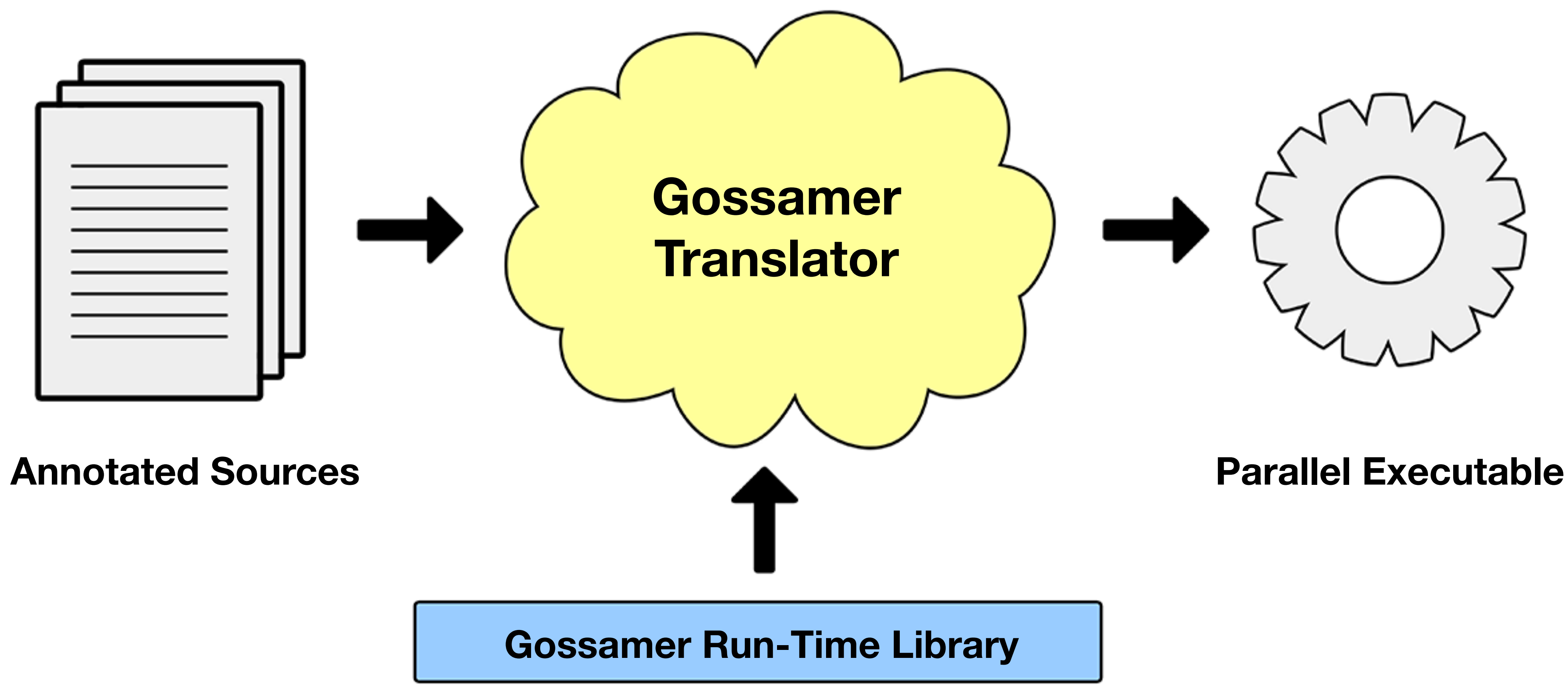
THE UNIVERSITY OF ARIZONA®

# Current Approaches



# Gossamer Approach

- ***General***: Covers broad domain of parallel computations
- ***Simple***: 15 annotations in total, no bookkeeping
- ***Efficient***: Lightweight, scalable run-time



# Gossamer Annotations

| | |
|---|---|
| Concurrency | fork, parallel, divide/replicate |
| Synchronization | join, barrier, atomic, buffered, copy, ordered, shared |
| Associative Memory | mrspace, mrlist<br>mr_put<br>mr_getkey, mr_getvalue |

# N-Queens: Recursive Parallelism

```
static int solutions = 0;

void putqueen(char **board, int row) {
   int j;

   if (row == n) {
      solutions++;
      return;
   }

   for (j = 0; j < n; j++) {
      if (OK(board, row, j)) {
         board[row][j] = 'Q';
         putqueen(board, row + 1);
         board[row][j] = '-';
      }
   }
}
```

# N-Queens: Recursive Parallelism

```
static int solutions = 0;

void putqueen(char **board, int row) {
  int j;

  if (row == n) {
    solutions++;
    return;
  }

  for (j = 0; j < n; j++) {
    if (OK(board, row, j)) {
      board[row][j] = 'Q';
      fork putqueen(copy board[n][n], row + 1);
      board[row][j] = '-';
    }
  }
  join;
}
```

# N-Queens: Recursive Parallelism

```
static int solutions = 0;

void putqueen(char **board, int row) {
   int j;

   if (row == n) {
      atomic { solutions++; }
      return;
   }

   for (j = 0; j < n; j++) {
      if (OK(board, row, j)) {
         board[row][j] = 'Q';
         fork putqueen(copy board[n][n], row + 1);
         board[row][j] = '-';
      }
   }
   join;
}
```

# Bzip2: Task Parallelism

```
int main(int argc, char **argv) {
   ...
   while (!feof(infp)) {
      insize = fread(in, 1, BLKSIZE, infp);
      compressBlk(in, insize);
   }
   ...
}

void compressBlk(char *in, int insize) {
   ...
   BZ2_bzCompress(in, insize, out, &outsize);
   fwrite(out, 1, outsize, outfp);
   ...
}
```

# Bzip2: Task Parallelism

```
int main(int argc, char **argv) {
    ...
    while (!feof(infp)) {
        insize = fread(in, 1, BLKSIZE, infp);
        fork compressBlk(copy in[insize], insize);
    }
    join;
    ...
}

void compressBlk(char *in, int insize) {
    ...
    BZ2_bzCompress(in, insize, out, &outsize);
    fwrite(out, 1, outsize, outfp);
    ...
}
```

# Bzip2: Task Parallelism

```
int main(int argc, char **argv) {
   ...
   while (!feof(infp)) {
      insize = fread(in, 1, BLKSIZE, infp);
      fork compressBlk(copy in[insize], insize);
   }
   join;
   ...
}

void compressBlk(char *in, int insize) {
   ...
   BZ2_bzCompress(in, insize, out, &outsize);
   ordered {
      fwrite(out, 1, outsize, outfp);
   }
   ...
}
```

# Matrix Multiplication: Iterative Parallelism

```
double **A, double **B, double **C;
int n;

void mm(void) {
   int i, j, k;

   for (i = 0; i < n; i++) {
      for (k = 0; k < n; k++) {
         for (j = 0; j < n; j++) {
            C[i*n + j] += A[i*n + k] * B[k*n + j];
         }
      }
   }
}
```

# Matrix Multiplication: Iterative Parallelism

```
double **A, double **B, double **C;
int n;

void mm(void) {
   int i, j, k;

   parallel for (i = 0; i < n; i++) {
      for (k = 0; k < n; k++) {
         for (j = 0; j < n; j++) {
            C[i*n + j] += A[i*n + k] * B[k*n + j];
         }
      }
   }
}
```

# Jacobi Iteration: Domain Decomposition

```
double **old, **new;
int i, j, n, m, it;

void jacobi(void) {
  old++; new++; // skip top+bottom grid borders
  n-=2;
  for (it = 0; it < MAXITERS; it += 2) {
    for (i = 0; i < n; i++)
      for (j = 1; j < m-1; j++)
        new[i][j] = (old[i-1][j] + old[i+1][j] +
          old[i][j-1] + old[i][j+1]) * 0.25;

    for (i = 0; i < n; i++)
      for (j = 1; j < m-1; j++)
        old[i][j] = (new[i-1][j] + new[i+1][j] +
          new[i][j-1] + new[i][j+1]) * 0.25;
  }
}
```

# Jacobi Iteration: Domain Decomposition

```
double **old, **new;
int i, j, n, m, it;

void jacobi(void) {
  old++; new++; // skip top+bottom grid borders
  n-=2;
  divide old[n][], new[n][] replicate {
    for (it = 0; it < MAXITERS; it += 2) {
      for (i = 0; i < n; i++)
        for (j = 1; j < m-1; j++)
          new[i][j] = (old[i-1][j] + old[i+1][j] +
            old[i][j-1] + old[i][j+1]) * 0.25;

      for (i = 0; i < n; i++)
        for (j = 1; j < m-1; j++)
          old[i][j] = (new[i-1][j] + new[i+1][j] +
            new[i][j-1] + new[i][j+1]) * 0.25;
    }
  }
}
```

# Jacobi Iteration: Domain Decomposition

```
double **old, **new;
int i, j, n, m, it;

void jacobi(void) {
   old++; new++; // skip top+bottom grid borders
   n-=2;
   divide old[n][], new[n][] replicate {
      for (it = 0; it < MAXITERS; it += 2) {
         for (i = 0; i < n; i++)
            for (j = 1; j < m-1; j++)
               new[i][j] = (old[i-1][j] + old[i+1][j] +
                  old[i][j-1] + old[i][j+1]) * 0.25;
         barrier;

         for (i = 0; i < n; i++)
            for (j = 1; j < m-1; j++)
               old[i][j] = (new[i-1][j] + new[i+1][j] +
                  new[i][j-1] + new[i][j+1]) * 0.25;
         barrier;
      }
   }
}
```

# Run Length Encoding: Domain Decomposition

```
FILE *out_fp;
char *data;
int size, run, val;

void rle(void) {
  while (size > 0) {
    val = *data++;
    size--;
    run = 1;

    while (val == *data && size > 0) {
      run++; data++; size--;
      if (run == RUNMAX) { break; }
    }

    fwrite(&val, sizeof(int), 1, out_fp);
    fwrite(&run, sizeof(int), 1, out_fp);
  }
}
```

# Run Length Encoding: Domain Decomposition

```
FILE *out_fp;
char *data;
int size, run, val;

void rle(void) {
  divide data[size] replicate {
    while (size > 0) {
      val = *data++;
      size--;
      run = 1;

      while (val == *data && size > 0) {
        run++; data++; size--;
        if (run == RUNMAX) { break; }
      }
      fwrite(&val, sizeof(int), 1, out_fp);
      fwrite(&run, sizeof(int), 1, out_fp);
    }
  }
}
```

# Run Length Encoding: Domain Decomposition

```
FILE *out_fp;
char *data;
int size, run, val;

void rle(void) {
  divide data[size] replicate {
    while (size > 0) {
      val = *data++;
      size--;
      run = 1;

      while (val == *data && size > 0) {
        run++; data++; size--;
        if (run == RUNMAX) { break; }
      }
      buffered (ordered) {
        fwrite(&val, sizeof(int), 1, out_fp);
        fwrite(&run, sizeof(int), 1, out_fp);
      }
    }
  }
}
```

# Run Length Encoding: Domain Decomposition

```
FILE *out_fp;
char *data;
int size, run, val;

void rle(void) {
  divide data[size]
    where data[divide_left] != data[divide_right] replicate {
    while (size > 0) {
      val = *data++;
      size--;
      run = 1;

      while (val == *data && size > 0) {
        run++; data++; size--;
        if (run == RUNMAX) { break; }
      }
      buffered (ordered) {
        fwrite(&val, sizeof(int), 1, out_fp);
        fwrite(&run, sizeof(int), 1, out_fp);
      }
    }
  }
}
```

# Word Count: MapReduce

```
mr_space wordcount(char *, int);

main(void) {
  char *key;
  for (i = 0; i < n; i++)
    map(file[i]);

  mr_list values;
  while (mr_getkey(wordcount, &key, &values))
    reduce(key, values);
}

void map(char *file) {
  char *word;
  while ((word = getnextword(file)) != NULL)
    mr_put(wordcount, word, 1);
}

void reduce(char *key, mr_list values) {
  int val, count = 0;
  while (mr_getvalue(wordcount, values, &val))
    count += val;
  printf("word: %s, count: %d\n", key, count);
}
```

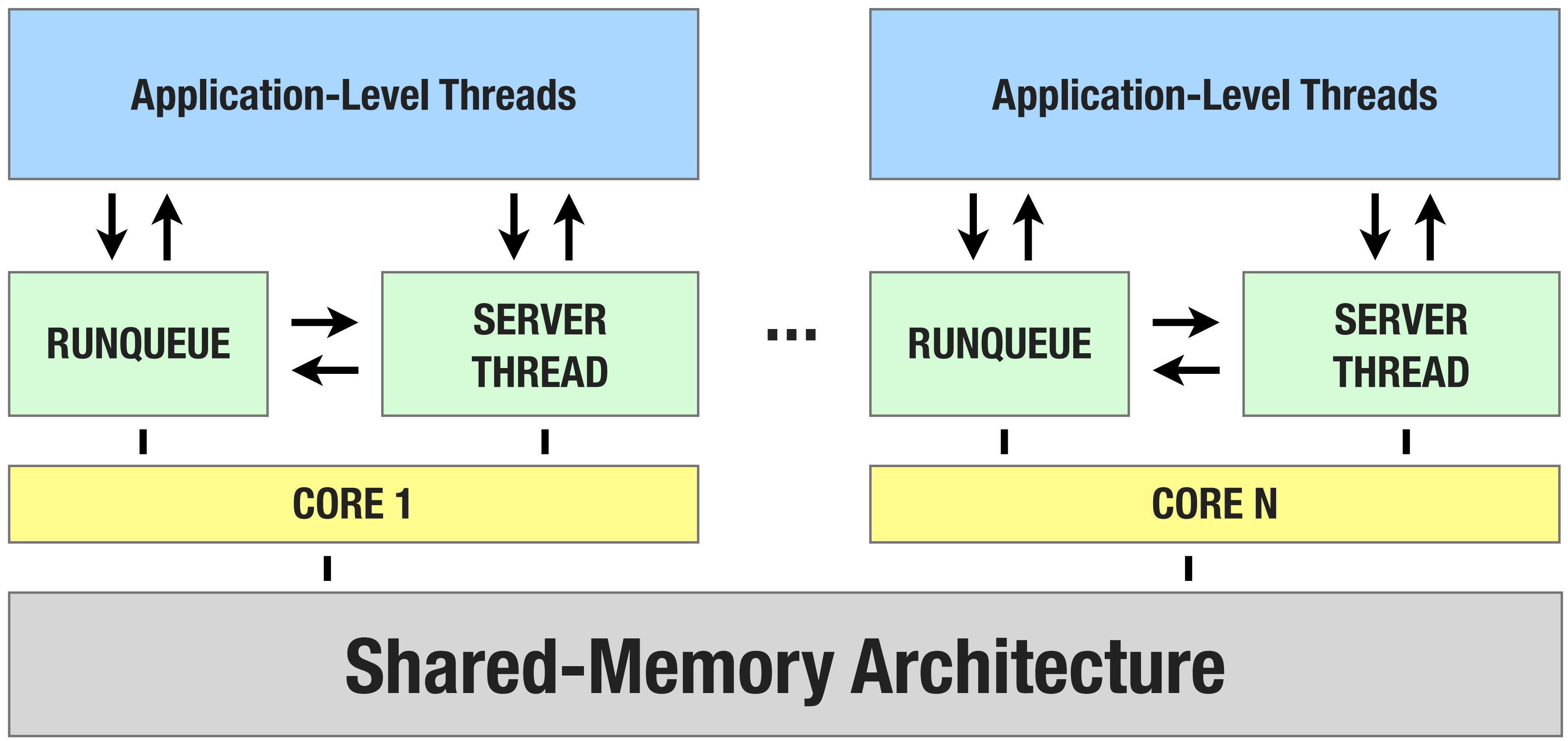
# Word Count: MapReduce

```
mr_space wordcount(char *, int);

main(void) {
   char *key;
   for (i = 0; i < n; i++)
      fork map(file[i]);
   join;

   mr_list values;
   while (mr_getkey(wordcount, &key, &values))
      fork reduce(key, values);
   join;
}

void map(char *file) {
   char *word;
   while ((word = getnextword(file)) != NULL)
      mr_put(wordcount, word, 1);
}

void reduce(char *key, mr_list values) {
   int val, count = 0;
   while (mr_getvalue(wordcount, values, &val))
      count += val;
   printf("word: %s, count: %d\n", key, count);
}
```

# Source-to-Source Translator

- Does lots of bookkeeping
- Optimizes parallel codes for annotations
  - Prunes self-recursive forked functions
  - Collapses simple nested `parallel` for loops
  - Uses equivalence classes to minimize locks used
- Adds instrumentation codes for profiling annotations
- Provides feedback for potential annotation mistakes (e.g. loop carried dependencies in parallel loops)

# Run-time System




- Application-level threads, called ***filaments***, are stackless and stateless
- Run queues and server threads per processor
- Multiple architecture and OS support

# Run-time System

- Filament Scheduling
  - Recursive/Task filaments enqueued round-robin
  - Iterative filaments enqueued in chunks
  - Domain Decomposition filaments enqueued statically
- Synchronization
  - Join synchronization
  - In MapReduce, `mr_put` and `mr_getvalue` are lock-free, only `mr_getkey` requires locking

# Experimental Setup

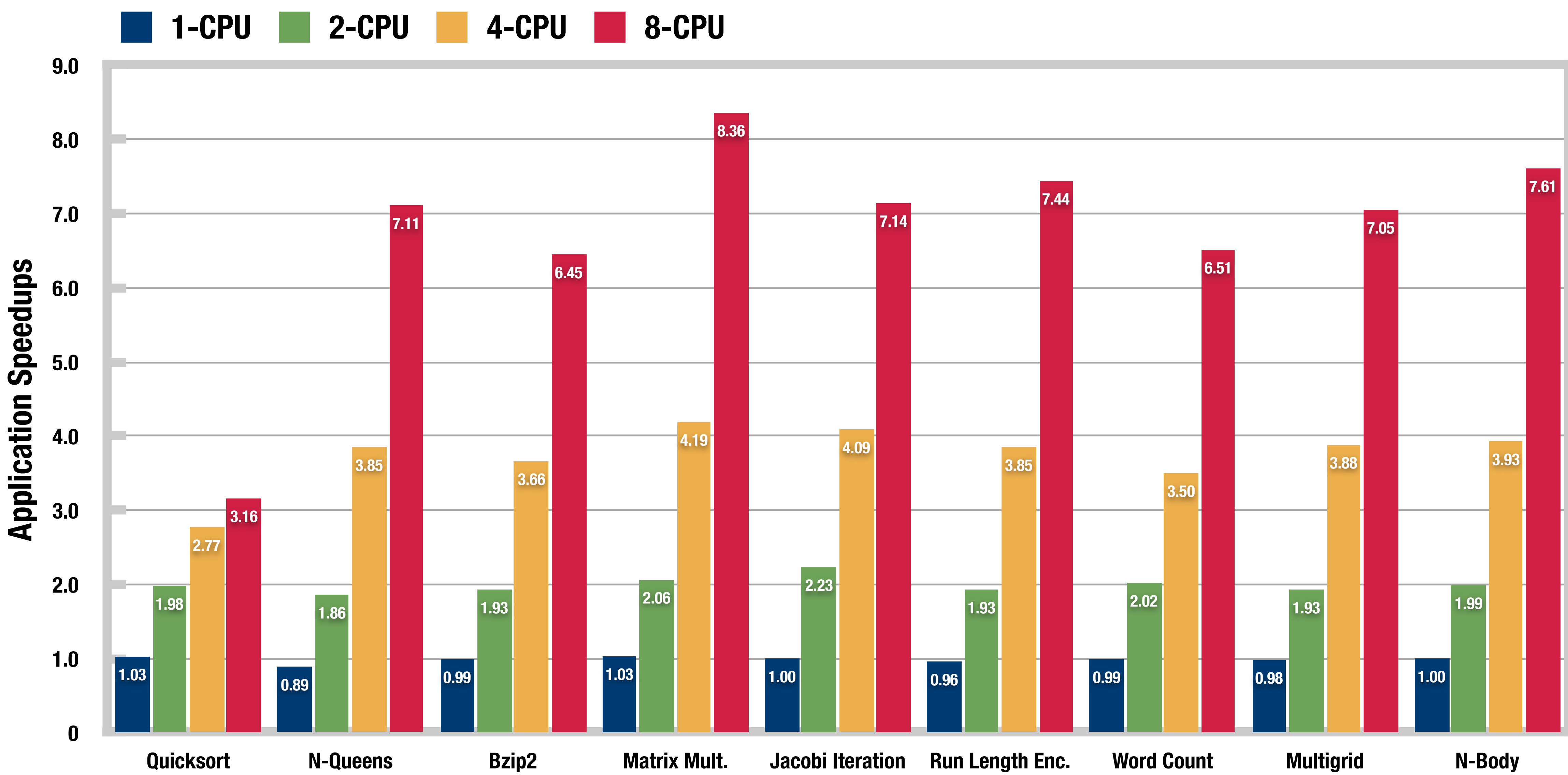
- 8-cores (two processors)
  - 2.0 GHz Intel Xeon E5405 quad-core processors
  - 12 MB L2 Cache
  - 8 GB main memory
- Ubuntu Linux 9.10, Linux Kernel 2.6.31
- gcc v4.4.1 (`gcc -O3`)

# Experimental Results

## Application Execution Times (seconds)

| Application | Parameters | Sequential | 1-CPU | Speedup |
|---|---|---|---|---|
| Quicksort | `n = 100,000,000` | 17.86 | 17.41 | 1.02 |
| N-Queens | `n = 14` | 9.29 | 10.41 | 0.89 |
| Bzip2 | `file = 256.0 MB` | 46.06 | 46.37 | 0.99 |
| Matrix Multiplication | `n = 4096x4096` | 198.51 | 193.06 | 1.03 |
| Jacobi Iteration | `grid = 1024x1024`<br>`iterations = 65536` | 277.99 | 278.04 | 0.99 |
| Run Length Encoding | `file = 4.0 GB` | 6.23 | 6.48 | 0.96 |
| Word Count | `files = 1024`<br>`file size = 2.0 MB` | 124.06 | 125.08 | 0.99 |
| Multigrid | `grid = 1024x1024`<br>`iterations = 4096` | 72.02 | 73.49 | 0.98 |
| N-Body | `bodies = 32768`<br>`iterations = 16384`<br>`tree builds = 16` | 93.10 | 93.05 | 1.00 |

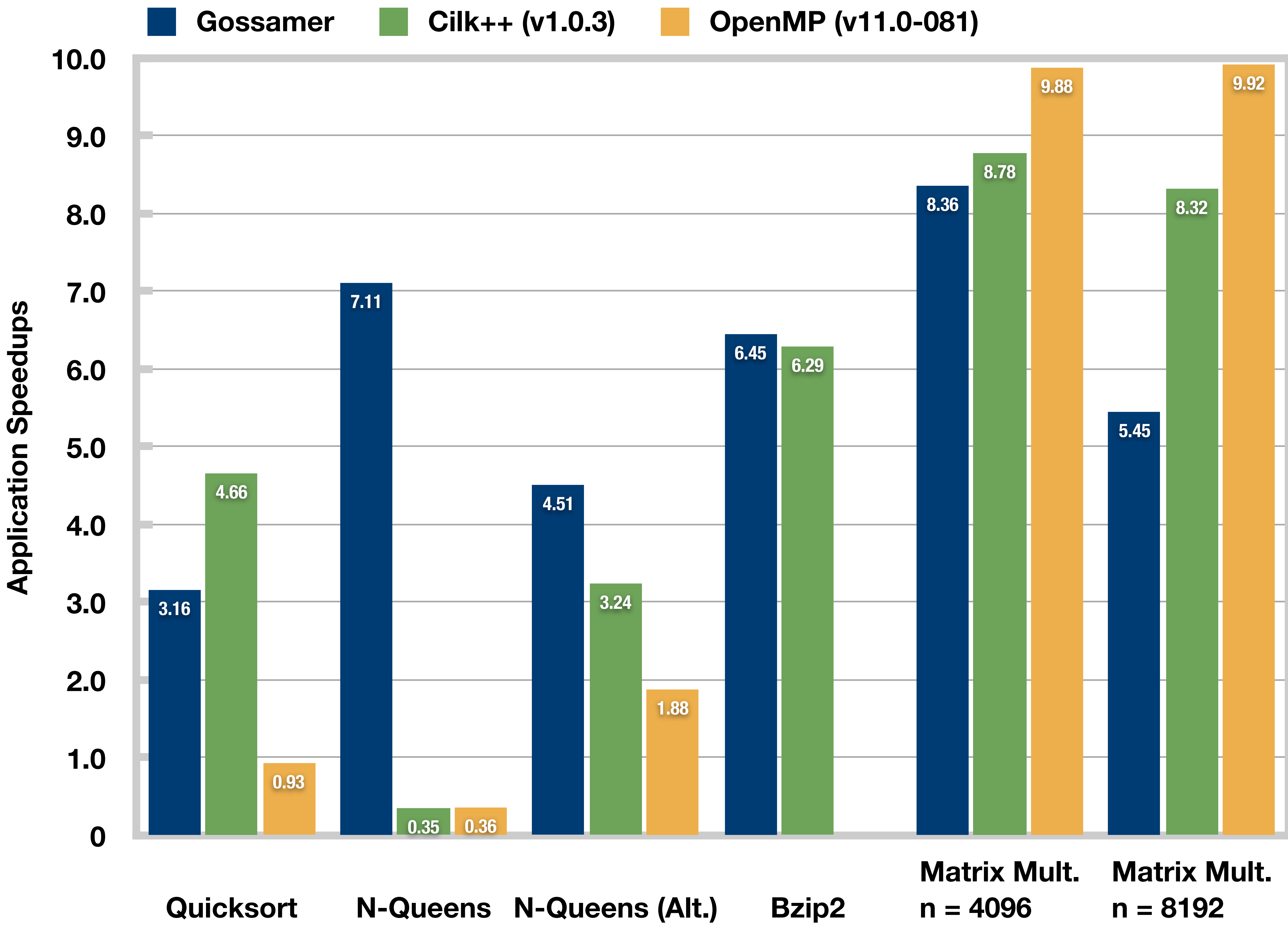
# Experimental Results: Speedups



# Related Work

- Annotation Based Approaches
  - OpenMP v3.0
  - Cilk/Cilk++
  - Unified Parallel C (UPC)
- Parallel Languages
  - Chapel
  - Fortress
  - X10
  - ZPL
- Many Others

# Related Work: Performance Comparison



# Conclusion

- **General:** covers broad domain of parallel computations
- **Simple:** abstracts the complexities of parallel programming using simple, yet powerful annotations
- **Efficient:** Good speedups through a variety of applications with low run-time overheads
- Existing applications can use the annotations with little or no modification to program structure
- Portable across many architectures. Less than 100 lines of machine-dependent assembly code per architecture.