Introduction to Parallel and Distributed Processing

Shared Memory

Jaroslaw 'Jaric' Zola

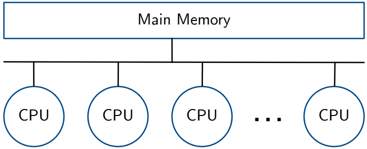
http://www.jzola.org/

Suggested Reading

- Cilk+ reference guide
  https://software.intel.com/en-us/node/522579
- General resources https://www.cilkplus.org/

Shared Memory Architecture

- A type of parallel architecture where multiple processors share main memory (e.g. modern multi-core processors)
- Software perspective: threads share address space



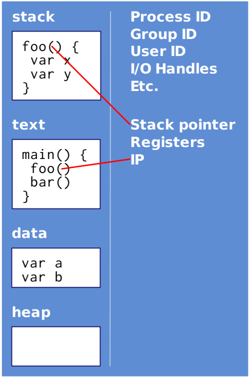
Why Shared Memory

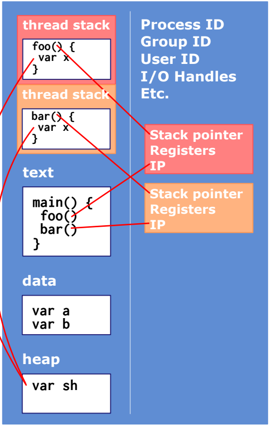
- Currently the most accessible type of parallel architecture
- Always at hand, why not use it
- Perceived easy parallelism

Why Shared Memory

Programming Shared Memory

- Create multiple threads that communicate by reading/writing shared variables



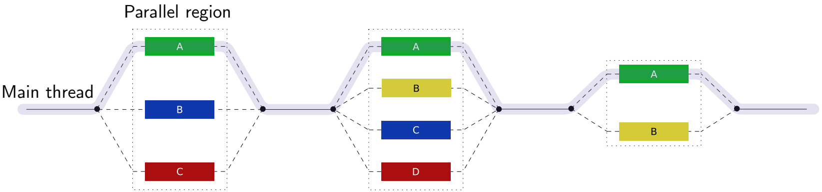


Programming Shared Memory

- Working with threads is too low-level for practical purposes
  scheduling, synchronization
- At the high level two types of parallelism:
  - Task parallelism (MIMD): pool of threads executes tasks, each task is a set of instructions on some data
  - Data parallelism: pool of threads acts on different parts of data with the same set of instructions
- Ideally, we want to describe our tasks and data transformations, without worrying about low-level realization

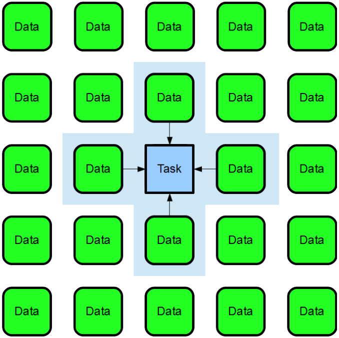
Fork-Join Pattern

- General strategy to express and execute task parallelism
- Main model for OpenMP, Cilk+, Intel TBB, etc.



Stencil Pattern

- Basic data parallel pattern: data structured (e.g. in 1D, 2D), update to data element depends on its neighborhood



Stencil

Cilk/Cilk+ API

- Provide a minimal set of C/C++ language extensions to enable shared memory parallelism (task and data)
- Focus on a parallel algorithms not implementation
- Efficient realization left to a compiler and run-time
- Comes with a powerful work-stealing scheduler and lock-free hyperobjects
- Deterministic Cilk+ code has serial semantics

Cilk+ Support

- Open source version in GCC 4.8 to 6.9, Clang/LLVM some support
- Commercial release by Intel (Parallel Studio XE, etc.)
- Using in GCC:

```
g++ -std=c++11 -fcilkplus -O3 ...
```

Spawn and Sync

- The Cilk+ standard defines three new keywords: `cilk_spawn`, `cilk_sync` and `cilk_for`
- These keywords are all what is needed to fork and join tasks

```
int fib(int n) {
  if (n < 2) return n;

  // create child task
  int a = cilk_spawn fib(n - 1);
  int b = fib(n - 2);

  // wait for all child tasks in function
  cilk_sync;

  return a + b;
} // implicit cilk_sync
```

Spawn and Sync

- Arguments passed to “spawned” function are evaluated before invocation
- Pass-by-references and pass-by-pointer arguments must be valid at least till `cilk_sync`
- Implicit `cilk_sync` follows after destructors
- It is a programmer job to avoid race conditions
  more to folow

Simple Parallel Postorder

```
struct node {
    node* left;
    node* right;
};

node* build_tree() { ... }
int release_tree(node* p) { ... }
void process(const node* p) { ... }

void visit(const node* p) {
    if (p->left != 0) visit(p->left);
    if (p->right != 0) visit(p->right);
    process(p);
}

int main(int argc, char* argv[]) {
    node* tree = build_tree();
    visit(tree);
    return release_tree(tree);
}
```

Simple Parallel Postorder

```
struct node {
    node* left;
    node* right;
};

node* build_tree() { ... }
int release_tree(node* p) { ... }
void process(const node* p) { ... }

void visit(const node* p) {
    if (p->left != 0) cilk_spawn visit(p->left);
    if (p->right != 0) visit(p->right);
    cilk_sync;
    process(p);
}

int main(int argc, char* argv[]) {
    node* tree = build_tree();
    visit(tree);
    return release_tree(tree);
}
```

More Interesting Example

- `[f1,l1)` and `[f2,l2)` are sorted ranges

```
template <typename InIter, typename OutIter>
void xxxxx(InIter f1, InIter l1, InIter f2, InIter l2,
           OutIter res) {
  while ((f1 != l1) && (f2 != l2)) {
    *(res++) = (*f2 < *f1) ? *(f2++) : *(f1++);
  }

  std::copy(f1, l1, res);
  std::copy(f2, l2, res);

} // xxxxx
```

More Interesting Example

- `[f1,l1)` and `[f2,l2)` are sorted ranges

```
template <typename InIter, typename OutIter>
void merge(InIter f1, InIter l1, InIter f2, InIter l2,
           OutIter res) {
  while ((f1 != l1) && (f2 != l2)) {
    *(res++) = (*f2 < *f1) ? *(f2++) : *(f1++);
  }

  std::copy(f1, l1, res);
  std::copy(f2, l2, res);

} // merge
```

More Interesting Example

```
template <typename InIter, typename OutIter>
void merge_dc(InIter f1, InIter l1, InIter f2, InIter l2,
              OutIter res) {
  if ((l1 - f1) + (l2 - f2) < 5) {
    return merge(f1, l1, f2, l2, res);
  }

  InIter mid1, mid2;

  if ((l2 - f2) < (l1 - f1)) {
    mid1 = f1 + ((l1 - f1) >> 1);
    mid2 = std::lower_bound(f2, l2, *mid1);
  }
  else {
    mid2 = f2 + ((l2 - f2) >> 1);
    mid1 = std::lower_bound(f1, l1, *mid2);
  }

  merge_dc(f1, mid1, f2, mid2, res);
  merge_dc(mid1, l1, mid2, l2, res + (mid1 - f1) + (mid2 - f2));
} // merge_dc
```

More Interesting Example

```
template <typename InIter, typename OutIter>
void merge_dc(InIter f1, InIter l1, InIter f2, InIter l2,
              OutIter res) {
  if ((l1 - f1) + (l2 - f2) < MERGE_LIMIT) {
    return merge(f1, l1, f2, l2, res);
  }

  InIter mid1, mid2;

  if ((l2 - f2) < (l1 - f1)) {
    mid1 = f1 + ((l1 - f1) >> 1);
    mid2 = std::lower_bound(f2, l2, *mid1);
  }
  else {
    mid2 = f2 + ((l2 - f2) >> 1);
    mid1 = std::lower_bound(f1, l1, *mid2);
  }

  cilk_spawn merge_dc(f1, mid1, f2, mid2, res);
  merge_dc(mid1, l1, mid2, l2, res + (mid1 - f1) + (mid2 - f2));
} // merge_dc
```

More Interesting Example

```
template <typename InIter, typename OutIter>
void merge_sr(InIter f1, InIter l1, InIter f2, InIter l2,
              OutIter res) {
  while ((l1 - f1) + (l2 - f2) >= MERGE_LIMIT) {
    InIter mid1, mid2;
    if ((l2 - f2) < (l1 - f1)) {
      mid1 = f1 + ((l1 - f1) >> 1);
      mid2 = std::lower_bound(f2, l2, *mid1);
    }
    else {
      mid2 = f2 + ((l2 - f2) >> 1);
      mid1 = std::lower_bound(f1, l1, *mid2);
    }
    cilk_spawn merge_sr(f1, mid1, f2, mid2, res);
    res += (mid1 - f1) + (mid2 - f2);
    f1 = mid1;
    f2 = mid2;
  }
  merge(f1, l1, f2, l2, res);
} // merge_sr
```

Efficiency Analysis

- If n is the total number of elements to merge:

$$T_1 = 2T_1(\frac{n}{2}) + O(log(n)) = \Theta(n)$$

$$T_\infty = T_\infty(\frac{n}{2}) + O(log(n)) = \Theta(\log^2(n))$$

$$S_\infty = O\left(\frac{n}{\log^2 n}\right)$$

For Fun

- Build parallel sorting routine starting with parallel merge