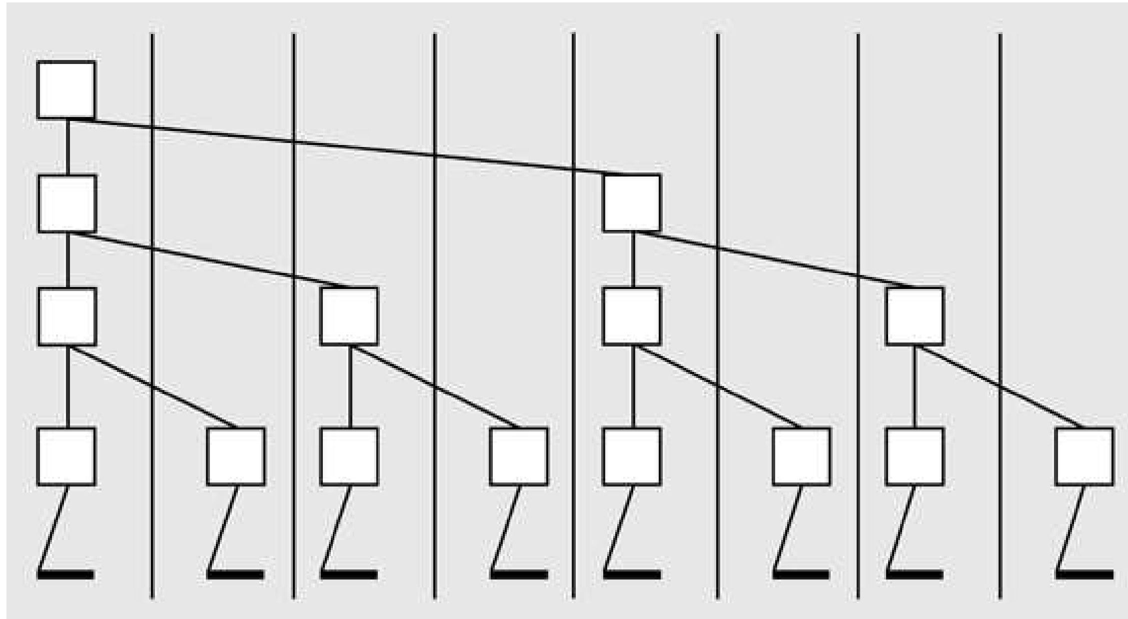
# Schwartz's Algorithm



In essence: don't assume unbounded parallelism

# Schwartz's Algorithm

```
int nodeval'[t];

forall(index in (0..t-1)) {
  int tally;

  tally = ... + ...;

  for (int stride = 1; stride < t; stride *= 2) {
    if (index % (2 * stride) == 0) {
      tally += nodeval'[index + stride];
    } else {
      nodeval'[index] = tally;
      break;
    }
  }
}
```

# Schwartz's Algorithm

```
int nodeval'[t];

forall(index in (0..t-1)) {
  int tally;

  tally = ... ⟨op⟩ ...;

  for (int stride = 1; stride < t; stride *= 2) {
    if (index % (2 * stride) == 0) {
      tally ⟨op⟩= nodeval'[index + stride];
    } else {
      nodeval'[index] = tally;
      break;
    }
  }
}
```

Assume that ⟨op⟩/ is implemented this way

# Generalized Reduce

What if you have an associate, commutative ⟨op⟩ that isn't built in?

- Implement Schwartz's algorithm yourself, or
- Use a ***generalized reduce*** template

# Generalized Reduce

```
data_t array[n];
result_t result;
tally_t nodeval'[t];

forall(index in (0..t-1)) {
  tally_t tally;

  tally = localTally(mySize(array, 0), localize(array));

  for (int stride = 1; stride < t; stride *= 2) {
    if (index % (2 * stride) == 0) {
      tally = combine(tally, nodeval'[index + stride]);
    } else {
      nodeval'[index] = tally;
      break;
    }
  }

  if (index == 0) result = reduceGen(tally);
}
```

# Generalized Reduce

```
tally_t localTally(int size, data_t[] myArray)
{
  tally_t tally = init();

  for (int i = 0; i < size; i++)
   tally = accum(tally, i, myArray);

  return tally;
}
```

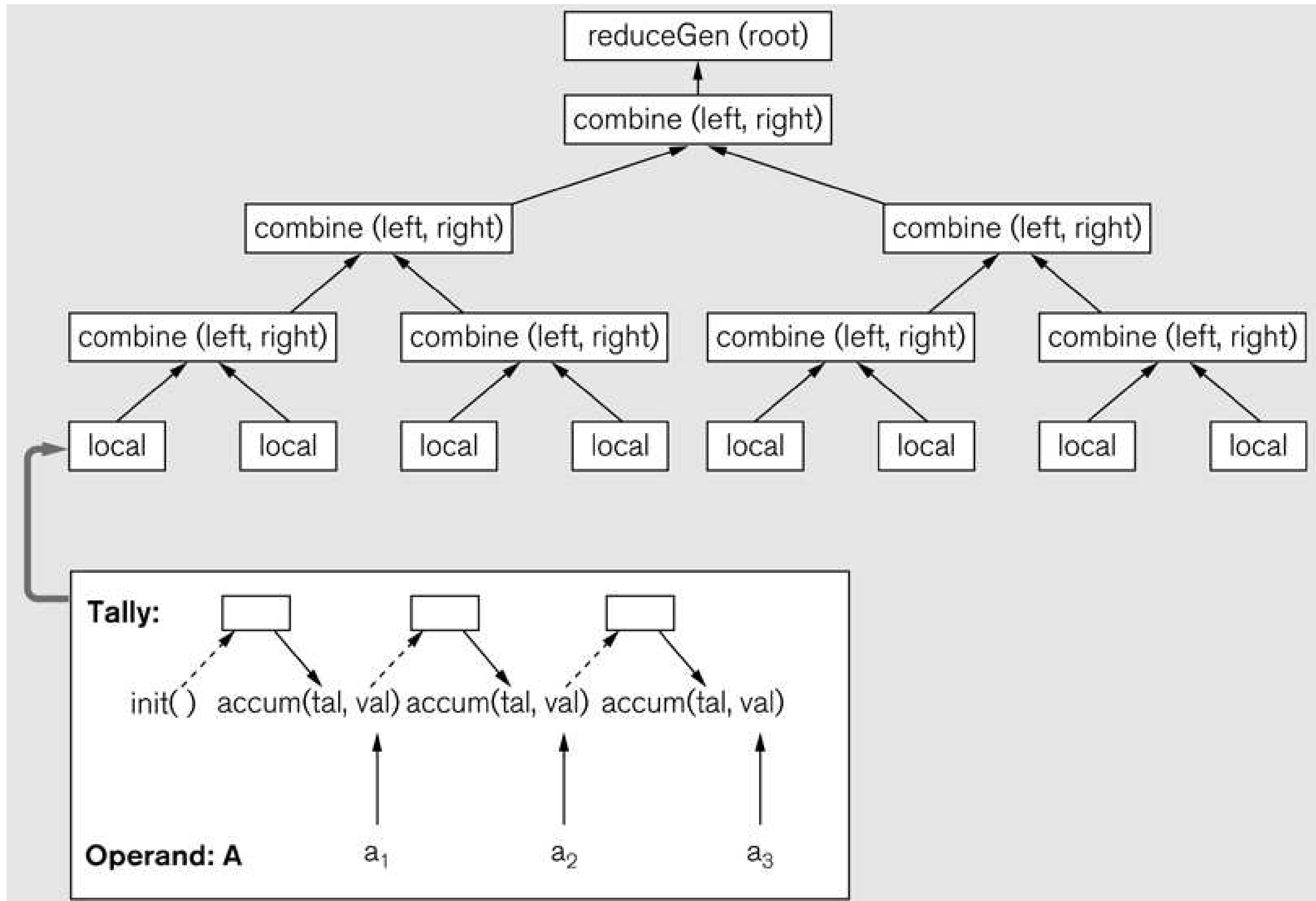
# Generalized Reduce

You provide:

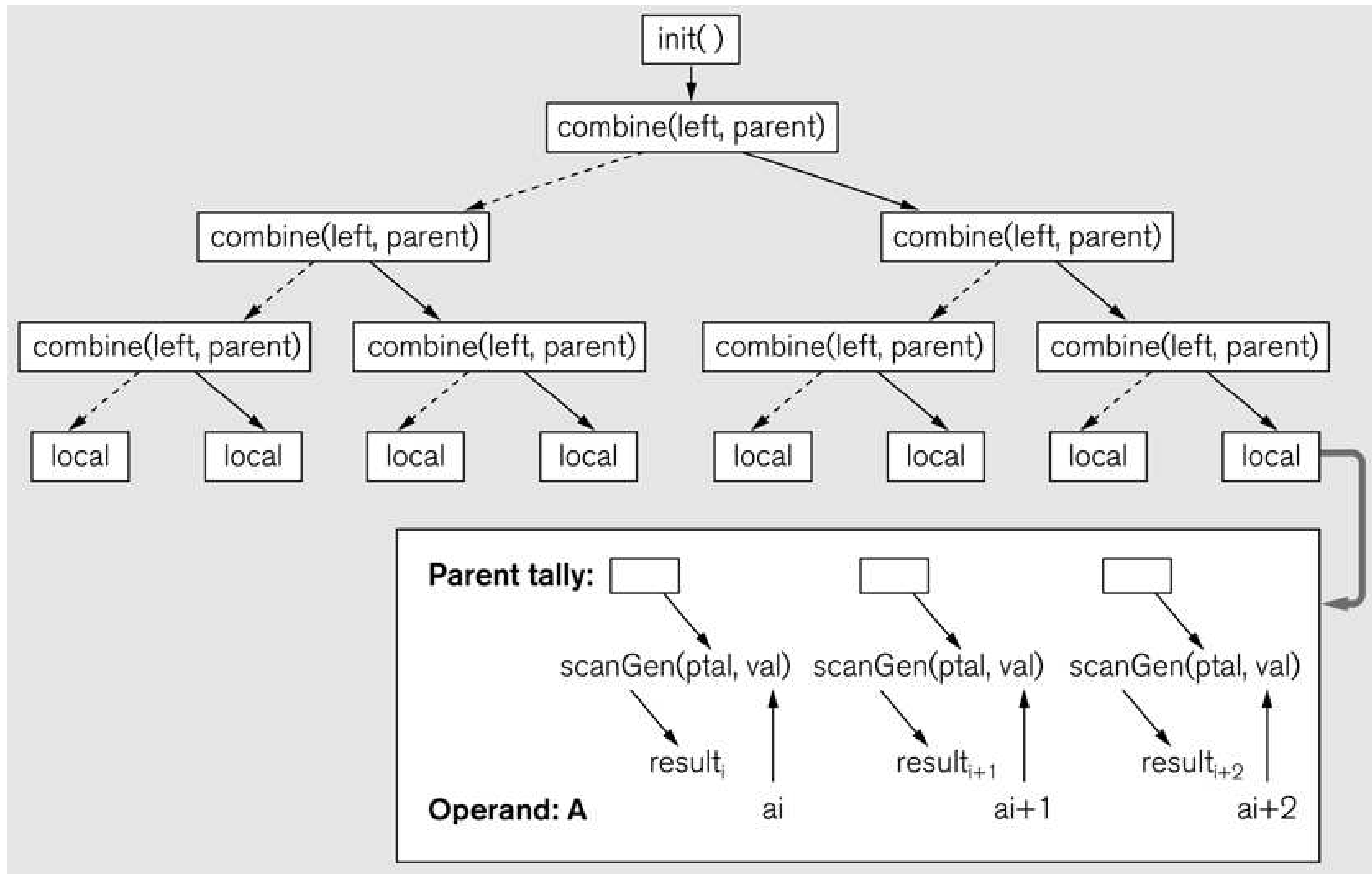
- `data_t` — type of input data
- `tally_t` — type of intermediate data
- `result_t` — type of result data

- `tally_t init()` — initialize local
- `tally_t accum(tally_t, int, data_t[])` — local
- `tally_t combine(tally_t, tally_t)` — global
- `result_t reduceGen(tally_t)` — post-process

# Generalized Reduce



# Generalized Scan



# Generalized Scan

```
data_t array[n];  result_t result[n];
tally_t nodeval'[t], pval'[t];

forall(index in (0..t-1)) {
  tally_t tally = localTally(mySize(array, 0), localize(array));

  for (int stride = 1; stride < t; stride *= 2) {
    if (index % (2 * stride) == 0) {
      pval'[index+stride] = tally; /* left total to right child */
      tally = combine(tally, nodeval'[index + stride]);
    } else {
      nodeval'[index] = tally;
      break;
    }
  }

  tally_t ptally = propagateParentTally(index);

  localScanGen(ptally, mySize(array, 0),
               localize(array), localize(result));
}
```

# Generalized Scan

```
tally_t propagateParentTally(int index)
{
   tally_t ptally;

   if (index == 0)
     ptally = init();
   else
     /* Parent supplies left total then parent */
     ptally = combine(pval'[index], pval'[index]);

   /* Propagate total from parent to right children */
   for (int stride = 1; stride < t; stride *= 2) {
     if (index % (2 * stride) == 0)
       pval'[index+stride] = ptally;
     else
      break;
   }

   return ptally;
}
```

# Generalized Scan

```
void localScanGen(tally_t ptally, int size,
                  data_t[] myArray, result_t[] myResult)
{
  for (int i = 0; i < size; i++)
    ptally = scanGen(ptally, i, myArray, myResult);
}
```

# Generalized Scan

You provide:

- `data_t` — type of input data
- `tally_t` — type of intermediate data
- `result_t` — type of result data

- `tally_t init()` — initialize local
- `tally_t accum(tally_t, int, data_t[])` — local
- `tally_t combine(tally_t, tally_t)` — global
- `tally_t scanGen(tally_t, int, data_t[], result_t[])` — finish