Concise analysis using implication algebras for task-local memory optimisation

Leo White

University of Cambridge

20th Static Analysis Symposium

Outline

- We want to improve performance of task-based OpenMP programs by optimising their memory usage
- The analysis required for this optimisation is inherently non-monotonic
- We use a generalisation of logic programming to concisely represent this analysis
- Using the notions of stable model and stratified model from logic programming we are able to show that our analysis has a single solution and that it can be computed in polynomial time

Traditional OpenMP

- OpenMP was originally designed to provide *static parallelism* for scientific applications
- C annotated with compiler directives:

```
void fill_table( int *a ) {
  #pragma omp parallel for
    for (i = 0; i < N; i++)
      a[i] = 2 * i;
}
```

- Recently, added support for *task-based parallelism*

Task-based parallelism

- A parallel programming model based on lightweight cooperative threads – called *tasks*
- These tasks are executed by a team of *worker threads*
- Tasks can *spawn* more tasks:

```
...
#pragma omp task
  func(...);
...
```

- Tasks can also *synchronise* on the completion of the tasks that they have spawned:

```
...
#pragma omp taskwait
...
```

OpenMP tasks example

```
void postorder_traverse( struct tree_node *p ) {
    if (p->left)
        #pragma omp task // OpenMP Spawn
            postorder_traverse(p->left);
    if (p->right)
        #pragma omp task // OpenMP Spawn
            postorder_traverse(p->right);
    #pragma omp taskwait // OpenMP Sync
    process(p);
}
```

- `postorder_traverse` traverses a binary tree, recursively creating a task for each node in that tree.
- Each task waits for the tasks processing its children to finish before it processes its node

OpenMP tasks example

```
void postorder_traverse( struct tree_node *p ) {
    if (p->left)
        #pragma omp task // OpenMP Spawn
            postorder_traverse(p->left);
    if (p->right)
        #pragma omp task // OpenMP Spawn
            postorder_traverse(p->right);
    #pragma omp taskwait // OpenMP Sync
    process(p);
}
```

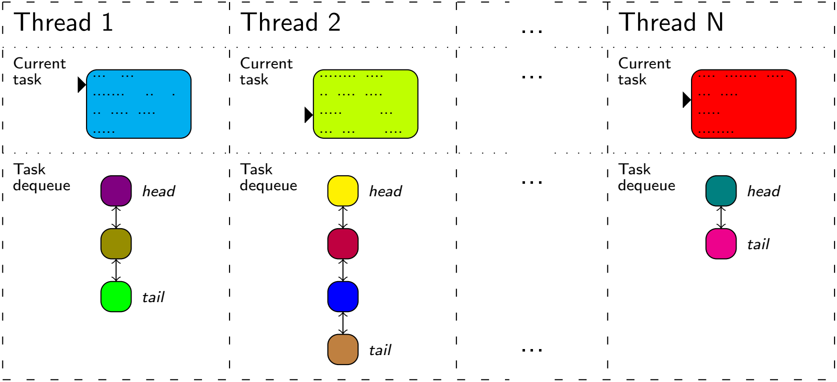
- `postorder_traverse` traverses a binary tree, recursively creating a task for each node in that tree.
- Each task waits for the tasks processing its children to finish before it processes its node

OpenMP tasks example

```
void postorder_traverse( struct tree_node *p ) {
    if (p->left)
        #pragma omp task // OpenMP Spawn
            postorder_traverse(p->left);
    if (p->right)
        #pragma omp task // OpenMP Spawn
            postorder_traverse(p->right);
    #pragma omp taskwait // OpenMP Sync
    process(p);
}
```

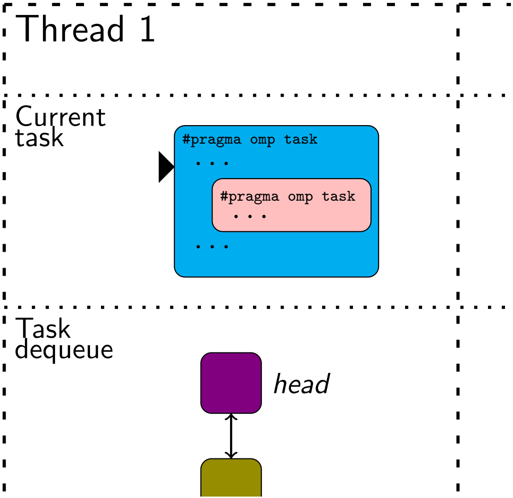
- `postorder_traverse` traverses a binary tree, recursively creating a task for each node in that tree.
- Each task waits for the tasks processing its children to finish before it processes its node

Implementing Tasks



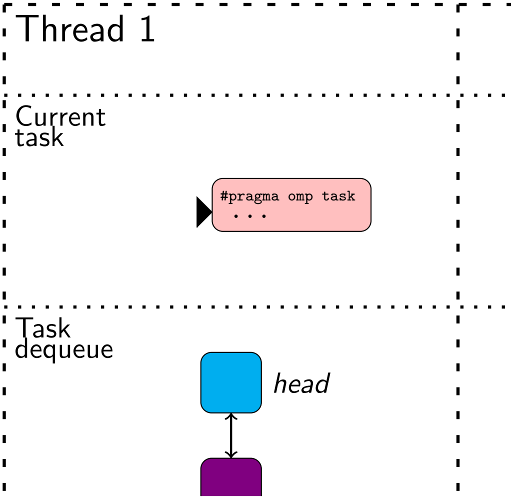
- Each worker thread has its own dequeue of tasks – improves locality and reduces contention

Implementing Tasks



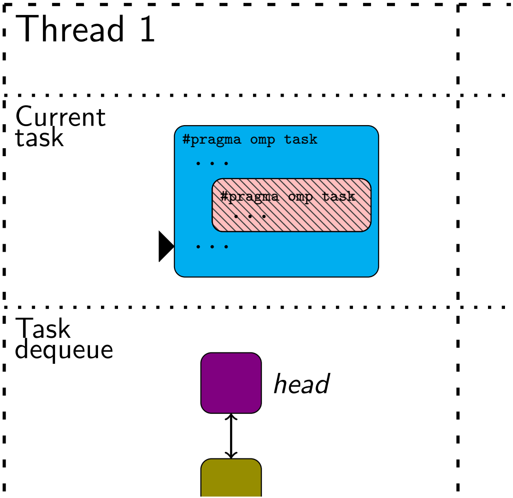
- When a thread encounters a task spawn it suspends its *current* task and pushes it onto the *head* of its dequeue

Implementing Tasks



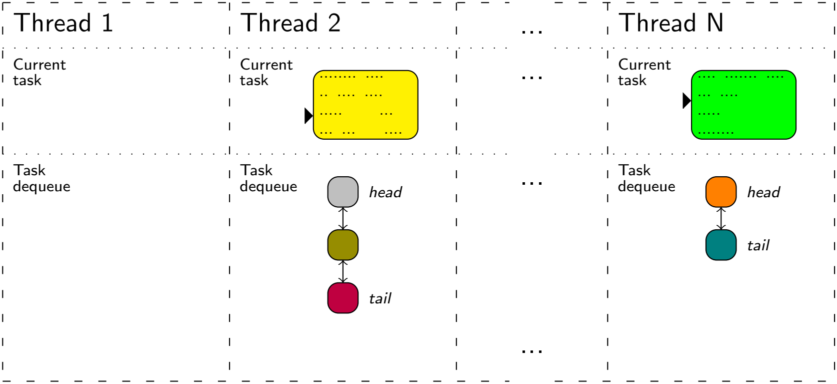
- When a thread encounters a task spawn it suspends its *current* task and pushes it onto the *head* of its dequeue
- It then executes the newly created child task

Implementing Tasks



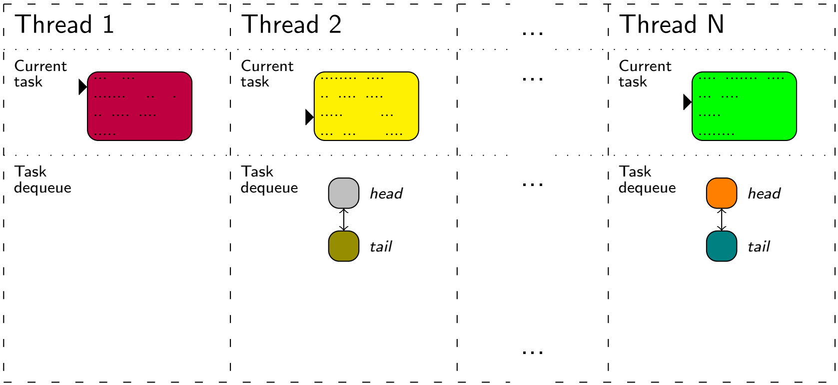
- When a thread encounters a task spawn it suspends its *current* task and pushes it onto the *head* of its dequeue
- It then executes the newly created child task
- When that task has finised it attempts to retrive the parent task from the *head* of its dequeue

Implementing Tasks



- When a thread finishes all of its tasks – or all of them have been stolen – it tries to *steal* tasks from the *tail* of another thread's dequeue

Implementing Tasks



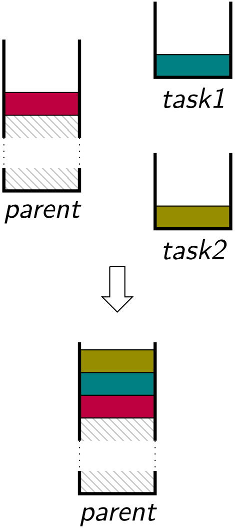
- When a thread finishes all of its tasks – or all of them have been stolen – it tries to *steal* tasks from the *tail* of another thread's dequeue

Stack merging

```
void add_tree(struct tree_node *root) {
    #pragma omp task untied
    {   tree_node *p = root;
        while (p) { left_sum += p->value;
                    p = p->left;
        }
    }
    #pragma omp task untied
    {   tree_node *q = root;
        while (q) { right_sum += q->value;
                    q = q->right;
        }
    }
    #pragma omp taskwait
}
```

Stack merging

```
void add_tree(tree_node *root) {
    #pragma omp task untied // task1
    {   tree_node *p = root;
        while (p) { left_sum += p->value;
                    p = p->left;
        }
    }
    #pragma omp task untied // task2
    {    tree_node *q = root;
        while (q) { right_sum += q->value;
                    q = q->right;
        }
    }
    #pragma omp taskwait
}
```



Stacks merged guarded

```
void postorder_traverse( struct tree_node *p ) {
    if (p->left)
      #pragma omp task
          postorder_traverse(p->left);
    if (p->right)
      #pragma omp task
          postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

Solution sets

- M , the set of merged task spawns
- $U \subseteq M$, the set of task spawns merged unguarded

$$(M, U) \sqsubseteq (M', U') \quad \Leftrightarrow \quad M \subset M' \vee (M = M' \wedge U \subseteq U')$$

- Not guaranteed to be a unique greatest safe model: we use a heuristic to pick best maximal model
- We prefer to merge spawns nearer the root of the call graph
- This is sufficient to provide a unique "best" solution

Unsafe merges

```
int fibonacci(int n)
{
    if ( n == 0 )
       return 0;
    else if ( n == 1 )
       return 1;
    else
       return ( Fibonacci(n-1) + Fibonacci(n-2) );
}

void fibs( int n, int m ) {
  #pragma omp task    // Merged spawn
      fibonacci(n);
  #pragma omp task    // Merged spawn (unsafe)
      fibonacci(m)
  #pragma omp taskwait
}
```

Non-monotonic analysis

This analysis is inherently *non-monotonic*:

> We prefer solutions that merge more stacks, but as more stacks are merged their sizes increase, and as the stack sizes increase the solution becomes more likely to be unsafe

General logic programs

A general logic program is a set of rules of the form:

$$A \longleftarrow L_1, \ldots, L_k$$

- A is an *atom*
- $L_1, \ldots, L_k$ are *literals*
- A literal is either an atom B (a *positive* literal) or the negation of an atom $\neg B$ (a *negative* literal)

Interpretations

- An *interpretation* I of a logic program P is a mapping from atoms in P to boolean truth values
- We write $\hat{I}$ for the natural extension of I to literals.
- I *respects* a rule $A \longleftarrow L_1, \ldots, L_k$ iff:

$$\hat{I}(L_1) \sqcap \cdots \sqcap \hat{I}(L_k) \sqsubseteq I(A)$$

Models

- An interpretation is a *model* of a logic program P iff it respects all the rules in P
- Equivalently, they are the fixed-points of the *immediate consequence operator* T_P :

$$\big(T_P(I)\big)(A) = \bigsqcup_{(A \longleftarrow L_1,\ldots,L_k) \in P} \hat{I}(L_1) \sqcap \cdots \sqcap \hat{I}(L_k)$$

- If all the literals in P are positive then T_P is monotonic and there is a least model of P
- If P includes negative literals then T_P may be non-monotonic and there may be no least model

Non-monotonic reasoning

- Consider the rule

$$\texttt{fly}(X) \longleftarrow \texttt{bird}(X), \neg\texttt{penguin}(X)$$

- Applying this rule to $\{\texttt{bird(tweety)}\}$ gives $\{\texttt{fly(tweety)}\}$
- Applying this rule to $\{\texttt{bird(tweety)}, \texttt{penguin(tweety)}\}$ gives $\{\}$
- The addition of new facts caused us to retract a conclusion.

Stratified programs

A general logic program is stratified if it can be partitioned $P_1 \cup \cdots \cup P_k = P$ such that, for every atom A, if A is defined in P_i and used in P_j then $i \leq j$, and additionally $i < j$ if the use is negative.

A stratified program has a *standard model* M_k defined by:

$$\begin{aligned} M_1 &= \text{The least fixed point of } T_{P_1} \\ M_i &= \text{The least fixed point of } \lambda I.\ \left(T_{P_i}(I) \sqcup M_{i-1}\right) \end{aligned}$$

This model is independant of the partitions $P_1, \ldots, P_k$ choosen.

Stratified program example

$$\texttt{fly}(X) \longleftarrow \texttt{bird}(X), \neg\texttt{penguin}(X)$$
$$\texttt{bird}(X) \longleftarrow \texttt{penguin}(X)$$
$$\texttt{bird}(\texttt{tweety}) \longleftarrow$$
$$\texttt{penguin}(\texttt{skippy}) \longleftarrow$$

Stratified program example

$$P_1 = \begin{array}{l} \texttt{bird}(X) \longleftarrow \texttt{penguin}(X) \\ \texttt{bird}(\texttt{tweety}) \longleftarrow \\ \texttt{penguin}(\texttt{skippy}) \longleftarrow \end{array}$$

$$P_2 = \texttt{fly}(X) \longleftarrow \texttt{bird}(X), \neg\texttt{penguin}(X)$$

$$M_1 = \{\texttt{bird}(\texttt{tweety}), \texttt{penguin}(\texttt{skippy}), \texttt{bird}(\texttt{skippy})\}$$
$$M_2 = \{\texttt{bird}(\texttt{tweety}), \texttt{penguin}(\texttt{skippy}), \texttt{bird}(\texttt{skippy}), \texttt{fly}(\texttt{tweety})\}$$

Stable model

The *reduct* of P with respect to I:

$$\mathcal{R}_P(I) = \{\ A \longleftarrow red_I(L_1), \ldots, red_I(L_k) \mid\ (A \longleftarrow L_1, \ldots, L_k) \in P\ \}$$

$$\text{where } red_I(L) = \begin{cases} L & \text{if } L \text{ is positive} \\ \hat{I}(L) & \text{if } L \text{ is negative} \end{cases}$$

- An intepretation I is a *stable model* iff it is the least model of its own reduct
- The standard model of a stratified program is its unique stable model

Multiple stable models

- A general logic program may have multiple stable models, or none
- For example, this logic program:

$$p \longleftarrow \neg q$$
$$q \longleftarrow \neg p$$

  has two stable models: $\{p\}$ and $\{q\}$
- This does not fit with the traditional idea of logic programming, but has been used as the basis for *answer set programming*
- Answer set programming treats logic programs as a system of constraints, and computes stables models as the solutions to those constraints

Generalising logic programs to implication programs

- We generalise logic programs in two ways:
  1. We replace boolean truth values with a general *complete lattice*
  2. We extend the allowed literals to any terms from an *algebra* defined over that lattice
- Positive literals are now those whose formulas correspond to functions that are *monotonic* in the atoms they contain
- Negative literals are now those whose formulas correspond to functions that are *anti-monotonic* in the atoms they contain
- We call these generlised logic programs *implication programs*
- The ideas of stratified and stable models can be applied to implication programs just as they are for logic programs

Stack size implication algebra

Lattice:

$$\mathbb{N}^{\infty} = \mathbb{N} \cup \{\infty\}$$

Literals:

$$L ::= \neg\, L \mid {\sim}\, L \mid L + L \mid A$$

Complement:

$$\forall z \in \mathbb{N}^{\infty}. \quad \neg z \stackrel{\text{def}}{=} \begin{cases} \infty & \text{when } z = 0 \\ 0 & \text{otherwise} \end{cases}$$

Supplement:

$$\forall z \in \mathbb{N}^{\infty}. \quad {\sim} z \stackrel{\text{def}}{=} \begin{cases} 0 & \text{when } z = \infty \\ \infty & \text{otherwise} \end{cases}$$

Stack sizes

```
void foo(...)
{
#pragma omp task
    bar(...);

#pragma omp taskwait

#pragma omp task
    baz(...);
}
```

Size	*Value*
Total	$\mathit{frame}(\texttt{foo}) + (\mathit{frame}(\texttt{bar}) \sqcup \mathit{frame}(\texttt{baz}))$
Post	$\mathit{frame}(\texttt{baz})$
Pre	$\mathit{frame}(\texttt{foo}) + \mathit{frame}(\texttt{bar})$
Through	0

Implication programs

We can now generate the rules of an implication program that defines the stack sizes and analysis solutions for a particular OpenMP program.

Stack size rules example

```
void func(...)
{
    #pragma omp task    // spawn1
        ...

    #pragma omp task    // spawn2
        ...

    #pragma omp task    // spawn3
        ...
}
```

$$\begin{aligned}\texttt{TotalSize}\langle\texttt{func}\rangle \longleftarrow\ & \mathit{frame}(\texttt{func}) + \texttt{PostSize}\langle \mathit{spawn1}\rangle \\ & + \texttt{ThroughSize}\langle \mathit{spawn2}\rangle \\ & + \texttt{PreSize}\langle \mathit{spawn3}\rangle\end{aligned}$$

Safety rules example

```
...
func( ... );    // call

#pragma omp task    // spawn
    task( ... );
...
```

$$\texttt{Unguarded}\langle \textit{spawn} \rangle \longleftarrow \texttt{Merged}\langle \textit{spawn} \rangle \text{ , } \sim\texttt{PostSize}\langle \textit{call} \rangle$$

$$\texttt{Unguarded}\langle \textit{spawn} \rangle \longleftarrow \texttt{Merged}\langle \textit{spawn} \rangle \text{ , } \sim\texttt{TotalSize}\langle \texttt{task} \rangle$$

Merging rules example

```
...
#pragma omp task    // spawn
    task( ... );
...
```

$$\texttt{PostSize}\langle \mathit{spawn} \rangle \longleftarrow \texttt{Merged}\langle \mathit{spawn} \rangle\ ,\ \texttt{TotalSize}\langle \texttt{task} \rangle$$

$$\texttt{ThroughSize}\langle \mathit{spawn} \rangle \longleftarrow \texttt{Unguarded}\langle \mathit{spawn} \rangle\ ,\ \texttt{TotalSize}\langle \texttt{task} \rangle$$

Stratification

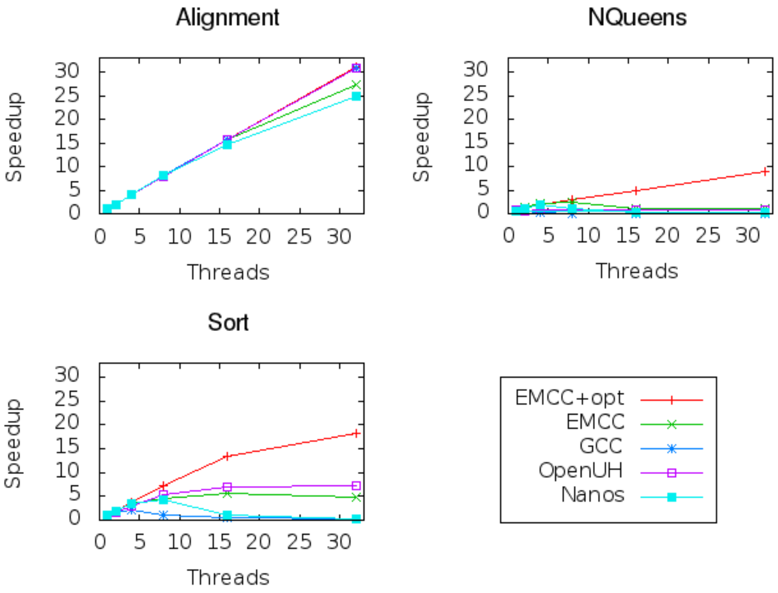
- The implication programs we generated above are not guaranteed to be stratified
- However, it is possible to construct a more complicated implication program, which has the same stable models and is stratified.
- This is done using multiple layers *layers* of the original implication program
- Each layer includes more of the rules from the original program than the previous layer

Stratification example

Benefits of stratification

- Single solution
- Polymorphic time complexity

Evaluation


Alignment
NQueens
Sort
Speedup
Threads
0
5
10
15
20
25
30
EMCC+opt
EMCC
GCC
OpenUH
Nanos

Conclusion

- Generalisations of logic programming can provide a concise way to express non-monotonic static analysis.
- The notion of stable model provides a semantics for these non-monotonic analyses
- Showing that such an analysis is equivalent to a stratified program shows that it has a single solution, and can help show it has polynomial time complexity
- Stack merging can greatly improve the performance of task-based OpenMP programs