Concise analysis using implication algebras for task-local memory optimisation

Leo White

University of Cambridge

20th Static Analysis Symposium (2013)

Joint work with Alan Mycroft

Outline

- Improve performance of task-based OpenMP programs by optimising their memory usage
- This analysis is inherently non-monotonic
- 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

Task-based parallelism in OpenMP

Stack merging

Stable models and non-monotonic analyses

Task-local memory analysis using implication algebras

Stratification

Evaluation & conclusion

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
- It recursively spawns a task for each node in the 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);
}
```

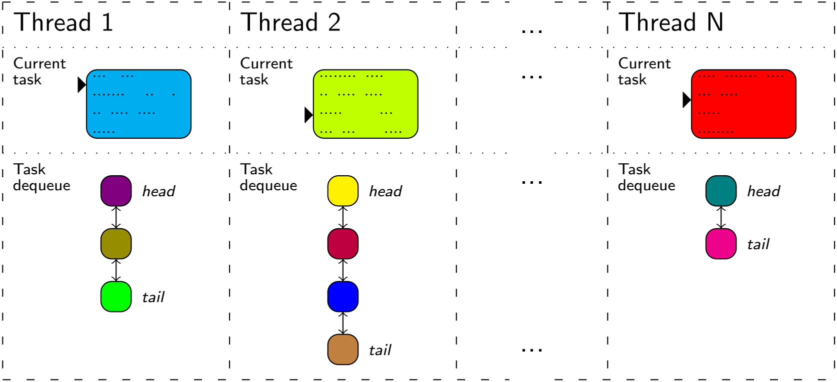
- `postorder_traverse` traverses a binary tree
- It recursively spawns a task for each node in the tree.
- Each task waits for the tasks processing its children to finish before it processes its node

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
- It recursively spawns a task for each node in the tree.
- Each task waits for the tasks processing its children to finish before it processes its node

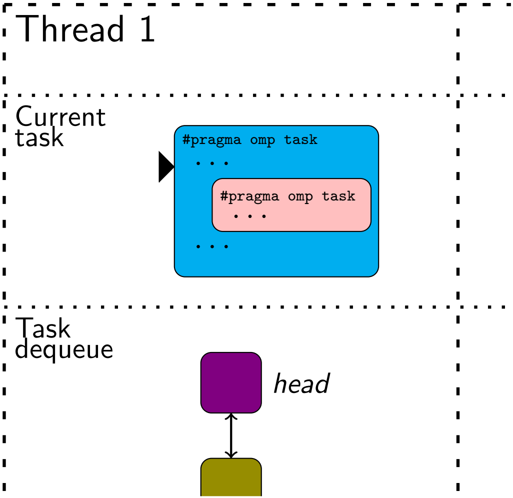
Implementing tasks: dequeues



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

Implementing tasks: procedure order

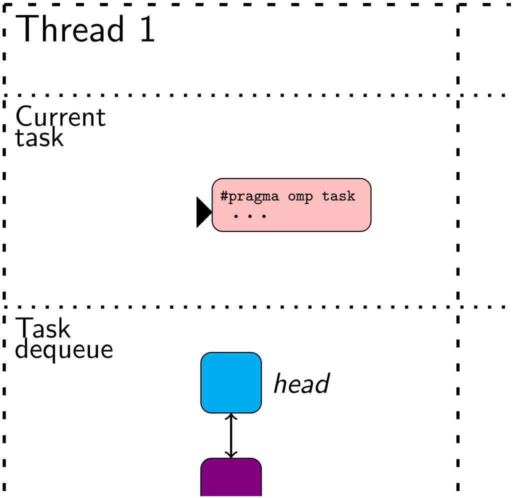
Threads execute tasks in “procedure” order:



- At a task spawn the *current* task is suspended and pushed onto the *head* of the thread’s dequeue

Implementing tasks: procedure order

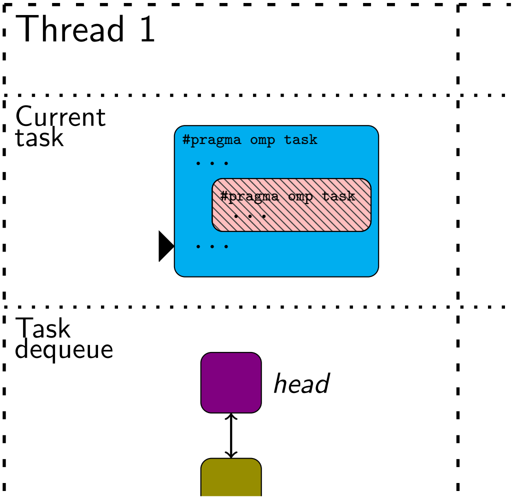
Threads execute tasks in “procedure” order:



- At a task spawn the *current* task is suspended and pushed onto the *head* of the thread’s dequeue
- The thread executes the newly created child task

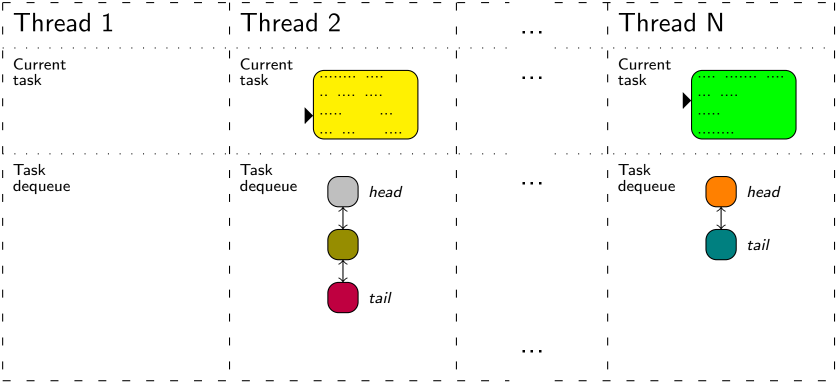
Implementing tasks: procedure order

Threads execute tasks in “procedure” order:



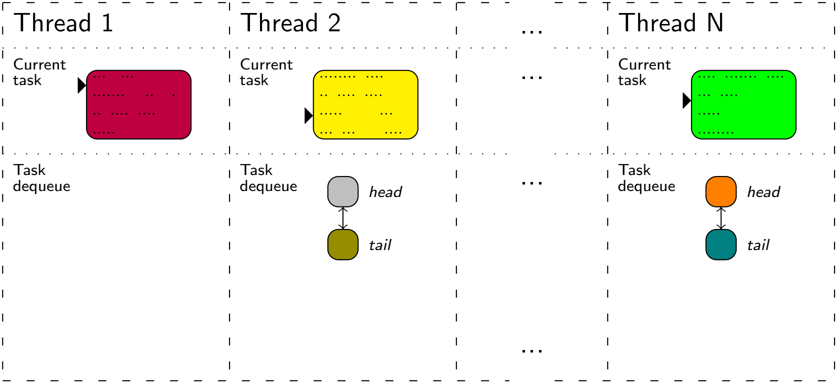
- At a task spawn the *current* task is suspended and pushed onto the *head* of the thread's dequeue
- The thread executes the newly created child task
- When that task has finished the thread trys to retrieve the parent task from the *head* of its dequeue

Implementing tasks: work stealing



- 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: work stealing



- 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

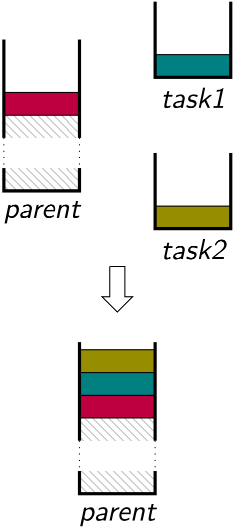
(Unguarded) stack merging

```
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
}
```

(Unguarded) 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
}
```

Since the child tasks use a bounded amount of stack space, we can allocate this space on the parent task's stack.



Guarded stack merging

```
void postorder_traverse( struct tree_node *p ) {
    if (p->left)
        #pragma omp task // task1
            postorder_traverse(p->left);
    if (p->right)
        #pragma omp task // task2
            postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

- It is not always safe to merge task2's stack with its parent.
- However, on a busy system – where task stealing is rare – these tasks would be executed in procedure order.
- It is very cheap to check at runtime whether they are being executed in procedure order, and only merge the stack in that case.
- We say that task2's stack can be *merged guarded*

Solution sets

Solutions to our analysis are pairs of sets (M, U):

- M is the set of all merged task spawns (guarded and unguarded)
- $U \subseteq M$ is the set of task spawns merged unguarded

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

- Not guaranteed to have a unique greatest safe solution: 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 safe solution

Unsafe merges

The main condition for a solution to be safe is that it must not merge two stacks which both use unbounded stack space simultaneously.

Example

```
int fibonacci(int n);

void fibs( int n, int m ) {
  #pragma omp task    // task1
      fibonacci(n);
  #pragma omp task    // task2
      fibonacci(m)
  #pragma omp taskwait
}
```

Merging task1 and task2 unguarded would not be safe

Non-monotonic analysis

This analysis is inherently *non-monotonic*:

- We prefer solutions that merge more stacks
- As more stacks are merged their sizes increase
- As the stack sizes increase the solution becomes more likely to be unsafe
- When the solution becomes unsafe we must merge fewer stacks

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 $\{\texttt{bird}(\texttt{tweety})\}$ gives $\{\texttt{fly}(\texttt{tweety})\}$
- Applying this rule to $\{\texttt{bird}(\texttt{tweety}), \texttt{penguin}(\texttt{tweety})\}$ 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:

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

This model is independent of the partitions $P_1, \ldots, P_k$ chosen.

Stratified program example

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

Stratified program example

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

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

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

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 interpretation 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* – and it fits our analysis.
- 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 generalised 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(...);
}
```

We represent the stack usage of a function by four stack size values.

Size	*Value*
Total-size	$\mathit{frame}(\texttt{foo}) + \max\big(\mathit{frame}(\texttt{bar}), \mathit{frame}(\texttt{baz})\big)$
Post size	$\mathit{frame}(\texttt{baz})$
Pre size	$\mathit{frame}(\texttt{foo}) + \mathit{frame}(\texttt{bar})$
Through size	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(...)
{
    sub1(...);
    sub2(...);
    sub3(...);
}
```

$$
\begin{aligned}
\mathtt{TotalSize}\langle\mathtt{func}\rangle \longleftarrow\ & \mathit{frame}(\mathtt{func}) + \mathtt{PostSize}\langle\mathtt{sub1}\rangle \\
& + \mathtt{ThroughSize}\langle\mathtt{sub2}\rangle \\
& + \mathtt{PreSize}\langle\mathtt{sub3}\rangle
\end{aligned}
$$

Safety rules example

```
...
fn( ... );

spawn: #pragma omp task
          tk( ... );
...
```

$$\mathtt{Unguarded}\langle\mathtt{spawn}\rangle \longleftarrow \mathtt{Merged}\langle\mathtt{spawn}\rangle\ ,\ \sim\mathtt{PostSize}\langle\mathtt{fn}\rangle$$

$$\mathtt{Unguarded}\langle\mathtt{spawn}\rangle \longleftarrow \mathtt{Merged}\langle\mathtt{spawn}\rangle\ ,\ \sim\mathtt{TotalSize}\langle\mathtt{tk}\rangle$$

Merging rules example

```
...
spawn: #pragma omp task
              tk( ... );
...
```

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

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

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.

Layering example

```
#pragma omp task
{
a:    #pragma omp task
           ...
b:    #pragma omp task
           ...
      #pragma omp taskwait
}
```

- The original implication program contains rules to ensure that `a` and `b` will not both be merged unguarded if they both use unbounded stack space
- However these rules do not affect the size of the parent task, because if either of the tasks uses unbounded stack space then its size is unbounded.
- So we could exclude those rules from the program and still calculate the correct size for the parent task

Layering

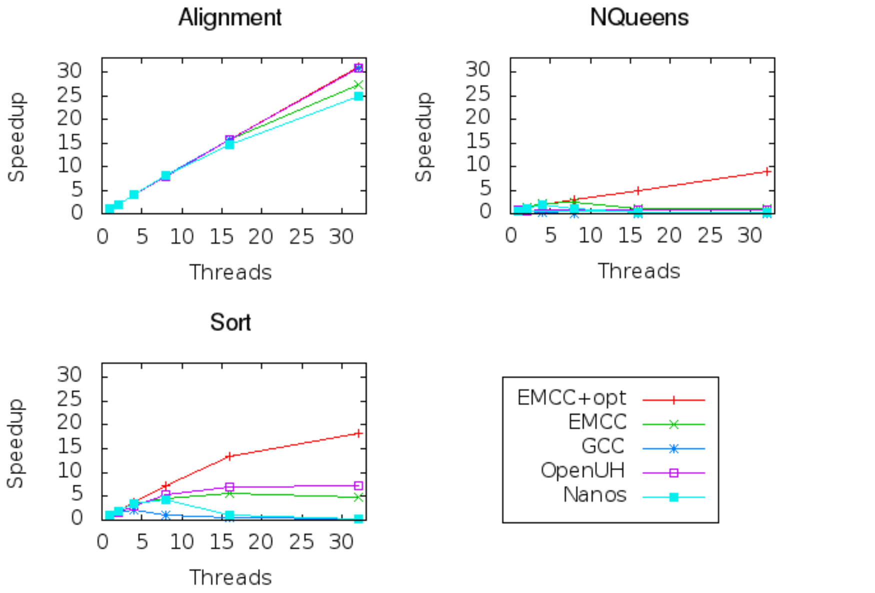
- We create a stratified program using multiple layers *layers* of the original implication program
- Each layer includes more of the rules from the original program than the previous layer
- Each layer is a more accurate approximation of the original program, and the final layer is equivalent to it

Benefits of stratification

Showing that our implication program has the same stable models as a stratified implication program allows us to:

- Show that it has a single solution
- Show that it has polynomial 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