Programming and executing parallel algorithms

Thierry Gautier
thierry.gautier@inrialpes.fr
AVALON, INRIA, ENS LIP Lyon
[MOAIS, INRIA, Grenoble]
Agenda

- Context
- Model for complexity
- Illustration: prefix computation
- Program and execution on NUMA architecture
- Conclusions
Context

• Natural sequence
  1. Problem => Algorithm
  2. Algorithm => Program
  3. Program + input => output

• « Selection » process
  1. Take a « good » algorithm. Good = complexity, reliability, robust to something.
  2. Which programming language? High level? Efficiency on real machine?
  3. Which machine? Crunch, grunch… idgraph (8 GPUs - Digitalis/G5K)? How many cores? Which grain size?

➡ Complex
Algorithm on top of machine model

• **PRAM machine: Parallel Random Access Machine**
  ‣ *Unbounded* collection of processors $P_0, P_1, P_2, \ldots$
  ‣ *Unbounded* collection of *shared memory* $M[0], M[1], M[2], \ldots$
  ‣ Each $P_i$ has *unbounded* collection of registers
  ‣ *Synchronous execution*
  ‣ *Unit time memory access*
  ‣ Read conflicts & Write conflicts *EREW, CREW, CRCW models*
  ‣ ...

• **Hardware complexity**
  ‣ *Time complexity*: elapsed time for $P_0$
  ‣ *Space complexity*: number of processors in use

• **Other models**
  ‣ circuit (NC-class hierarchy)
  ‣ taking into communication latency LogP, BSP, etc
Work and depth model

• Work and Depth
  ‣ Algorithm = direct acyclic graph $G$ of operations
    - arithmetic circuit model
    - Work = number of operations in $G$
    - Depth = depth in $G$
  ‣ High level model

• Emulation on p-PRAM [Brent]
  ‣ $T_p = O(W/p + D)$

• Parallel environments that offer such guarantee with high probability
  ‣ Cilk [MIT], NESL [CM], X10, Athapascan/Kaapi, …
  ‣ need a scheduling algorithm to map high level description to the hardware
Example: Prefix computation

- Problem statement

\[
\text{let } a_0, a_1, a_2, \ldots, a_n \text{ in } K \\
\text{compute: } \pi_i = \sum_{j=0}^{i} a_j, \forall i \\
\text{e.g.:} \\
\{a_i\} = \{0, 3, 5, 7, 2, 6, -2\} \\
\{\pi_i\} = \{0, 3, 8, 9, 11, 17, 15\}
\]
A parallel prefix

\[ a[] : \]

\[ \Pi_0 \quad \Pi_1 \quad \Pi_2 \]

\[ \text{Prefix} \]

\[ \text{Partial prefix} \]

\[ \text{map}(\ast \Pi_0, \cdot) \quad \text{map}(\ast \Pi_1, \cdot) \quad \text{prefix}'(\Pi_2, \cdot) \]

\[ \Pi[] : \]
Complexity

• **Step 1: p independent ‘prefix’ on sub arrays**
  ‣ sub prefix of size $n/(p+1)$
    - sequential computation, so $W_{step1} = D_{step1} = n/(p+1) - 1$
  ‣ $W_{step1} = pn/(p+1) - 1$, $D_{step1} = n/(p+1) - 1$

• **Step 2: small prefix instance if $p \ll n$**
  ‣ prefix of size $p$
    - sequential $W_{step2} = p - 1$, $D_{step2} = p - 1$
    - parallel $W_{step2} = O(p)$, $D_{step2} = O(\log p)$

• **Step 3: independent ‘maps’ + 1 prefix on sub arrays**
  ‣ for one map of size $n/(p+1)$
    - sequential $W = n/(p+1)$, $D = n/(p+1)$
  ‣ $W_{step3} = pn/(p+1) + 1$, $D_{step3} = n/(p+1)$

At the end (seq. step 2)

\[
W \sim 2 \frac{np}{p+1} + p \\
D \sim 2 \frac{n}{p+1} + p
\]
How to make // prefix program real?

• A programming language
  ‣ OpenMP
    - task, loop: high level features for parallel programming
    - large community

• A parallel machine
  ‣ Multicore NUMA
    - Crunch/Grunch etc
NUMA parallel machine

- Multicore NUMA (crunch/grunch, ..)
  - ≠ PRAM mostly due to Non Uniform Memory Access
  - time to access data depend on
    - location of the data
    - location of the core that access it
void prefix_par( int n, long* in_a, long* out_p )
{
    int j;
    int p;
    #pragma omp parallel
    p = omp_get_num_threads();
    int k = n/(p+1);

    /* intermediate prefix */
    long partial_pref[p];

    #pragma omp parallel
    {
        /* step 1: partial prefix */
        #pragma omp for
        for (int j=0; j<p; ++j)
        {
            int start = j * k;
            int end = start + k;
            prefix_seq( start, end, in_a, out_p, 0);
        }
    }
/* step 2: sequential version */
#pragma omp single
{
    partial_pref[0] = out_p[k-1];
    for (int j=1; j<p; ++j)
        partial_pref[j] = partial_pref[j-1] + out_p[(j+1)*k-1];
}

/* step 3: update */
#pragma omp for
for (int j=1; j<p+1; ++j)
{
    int start = j * k;
    int end = start + k;
    if (j != p)
        map_seq( start, end, out_p, partial_pref[j-1]);
    else
        prefix_seq( start, end, in_a, out_p, partial_pref[j-1]);
}
void prefix_partask( int n, long* in_a, long* out_p )
{
    int j;
    int p;
#pragma omp parallel
    p = omp_get_num_threads();
    int k = n/(p+1);

    /* intermediate */
    long partial_pref[p];

#pragma omp parallel
#pragma omp single
{
    /* step 1: partial prefix */
    for (int j=0; j<p; ++j)
    {
        int start = j * k;
        int end = start + k;
        #pragma omp task depend(in: in_a[start:k]) depend(out: out_p[start:k])
        prefix_seq( start, end, in_a, out_p, 0);
    }
}
• But…. this program is not conform to the standard
  ‣ dependencies between non disjoints subarrays
  ‣ variable size dependencies

/* step 2: sequential version */
#pragma omp task depend(in: out_p[(j+1)*k-1:1], j=0..p) depend(out: partial_pref[0:p])
{
    partial_pref[0] = out_p[k-1];
    for (int j=1; j<p; ++j)
        partial_pref[j] = partial_pref[j-1] + out_p[(j+1)*k-1];
}

/* step 3: update */
for (int j=1; j<p+1; ++j)
{
    int start = j * k;
    int end = start + k;
    #pragma omp task depend(in: in_a[start:k], partial_pref[j-1])
                depend(out: out_p[start:k])
    if (j != p)
        map_seq( start, end, out_p, partial_pref[j-1]);
    else
        prefix_seq( start, end, in_a, out_p, partial_pref[j-1]);
}
What about performances?

• 😊

Main problems
  › grain size selection
  › affinity between data and computation
  › too early parallelism creation
Problem 1: affinity

• More complex with dynamic irregular application
void Cholesky( int N, int BS, double A[N][N] )
{
  for (size_t k=0; k < N; k += BS)
  {
    #pragma omp task depend(inout: A[k:BS][k:BS])
    LAPACKE_dpotrf( LAPACK_COL_MAJOR, 'l', BS, &A[k*N+k], N );

    for (size_t m=k+ BS; m < N; m += BS)
    {
      #pragma omp task depend(in: A[k:BS][k:BS]) depend(inout: A[m:BS][k:BS])
      cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans,
                     CblasUnit, BS, BS, 1.0, &A[k][k], N, &A[m][k], N );
    }

    for (size_t m=k+ BS; m < N; m += BS)
    {
      #pragma omp task depend(in: A[m:BS][k:BS]) depend(inout: A[m:BS][m:BS])
      cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
                    BS, BS, -1.0, &A[m][k], N, 1.0, &A[m][m], N );

      for (size_t n=k+BS; n < m; n += BS)
        #pragma omp task depend(in: A[m:BS][k:BS], A[n:BS][k:BS]) depend(inout: A[m:BS][n:BS])
        cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans, BS, BS, BS,
                      -1.0, &A[m][k], N, &A[n][k], N, 1.0, &A[m][n], N );
    }
  }
  #pragma omp taskwait
}
Effect: work time inflation

- NUMA machine SGI UV2000
  - 192 cores, 24 NUMA node,
  - dynamic scheduling

• Antagonist criteria: communication / makespan
Approaches to control affinity

• Algorithmic approach
  ‣ Rewriting code
    - low level control
    - cache oblivious algorithm
  ‣ Limitations
    - time consuming or missing feature

• Runtime approach
  ‣ scheduling heuristics to reduce makespan & communication
    - knowledge of program structure with tasks and their (data flow) dependencies
  ‣ a lot of opportunities to manage affinity task/data
    - libKOMP (http://kaapi.gforge.inria.fr)

➡ OpenMP extensions
  ‣ data and work distribution annotation on NUMA machine
    - specific scheduling heuristics
    - context: contribution to the standard K’STAR compiler (http://kstar.gforge.inria.fr)
Problem 2: cost of the parallelism

- **Runtime overhead**
  - management of parallelism
  - overhead in runtime algorithms/data structure

- **Algorithmic overhead**
  - \( n\text{-prefix}: \text{overhead parallel / sequential} \approx \frac{2p}{p+1} \)
  - arithmetic operation
  - more read/write to memory
Adaptive algorithm

• **Goals: efficient algorithms in presence of variations**
  ‣ reducing overhead
  ‣ system jitter => apparent processor speed

• **Idea: delay task creation until necessary**
  - and a dynamic scheduler!

• **Adaptive algorithm = generate parallelism (task) on demand**
  ‣ [J.L. Roch et al., TSI’05, Europar’06,’08,.. ], [PhD D. Traore 2008]
    - ~ list homomorphism \( f(a\#b) = f(a)\cdot f(b) \)
  ‣ **Runtime support development**
    - work stealing scheduler calls user level method to extract parallelism
    - when core is idle!
  ‣ **No language feature was developed**
    - How to (semi-)automatically generate the user level method?
      - class of problems?
Conclusion

- **Algorithms play important role**
  - Big complexity gain
  - Think in parallel using high level expression

- **Runtime is the link between application and hardware**
  - Important for performance portability
  - Opportunities to make low level optimizations
  - Other objective functions than pure makespan

- **OpenMP extensions**