Structured parallel programming for Monte Carlo Tree Search

Mirsoleimani, S.A.

Author: Mirsoleimani, S.A.
Title: Structured parallel programming for Monte Carlo tree search
Issue Date: 2020-06-17

Bibliography

[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[AHH10] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007.

[BCC+11] Amine Bourki, Guillaume Chaslot, Matthieu Coulm, Vincent Danjean, Hassen Doghmen, Jean-Baptiste Hoock, Arpad Rimmel, Fabien Teytaud, Olivier Teytaud, Paul Vayssi, Thomas Hérault, Paul Vayssière, and Ziqin Yu. Scalability and Parallelization of Monte-Carlo Tree Search. In *Proceedings of the 7th International Conference on Computers and Games*, Lecture Notes in Computer Science (LNCS) 6515, pages 48–58, 2011.

[BG11] Petr Baudiš and Jean-Loup Gailly. Pachi: State of the Art Open Source Go Program. In *Advances in Computer Games 13*, Lecture Notes in Computer Science (LNCS) 7168, pages 24–38, 2011.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPOPP '95*, volume 30, pages 207–216. ACM Press, 1995.

[BPW+12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[CJ08] T. Cazenave and N. Jouandeau. A Parallel Monte-Carlo Tree Search Algorithm. In *Computers and Games*, Lecture Notes in Computer Science (LNCS) 5131, pages 60–71, 2008.

[Cou06] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games*, Lecture Notes in Computer Science (LNCS) 4630, pages 72–83, 2006.

[CWvdH08a] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik. Parallel Monte-Carlo Tree Search. In *the 6th International Conference on Computers and Games*, Lecture Notes in Computer Science (LNCS) 5131, pages 60–71, 2008.

[CWvdH+08b] Guillaume M. J. B. Chaslot, Mark H. M. Winands, H. J. van den Herik, Jos W. H. M. Uiterwijk, and Bruno Bouzy. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

[EM10] M. Enzenberger and M. Müller. A Lock-free Multithreaded Monte-Carlo Tree Search algorithm. In *Advances in Computer Games*, Lecture Notes in Computer Science (LNCS) 6048, pages 14–20, 2010.

[EMAS10] Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. Fuego-An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[FL11] Alan Fern and Paul Lewis. Ensemble Monte-Carlo Planning: An Empirical Study. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 58–65, 2011.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning Series. MIT Press, 2016.

[GI91] Zvi Galil and Giuseppe F. Italiano. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comput. Surv.*, 23(3):319–344, 1991.

[GS07] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *the 24th International Conference on Machine Learning*, pages 273–280. ACM Press, 2007.

[Hei01] E.A. Heinz. New self-play results in computer chess. In *Computers and Games*, Lecture Notes in Computer Science (LNCS) 2063, pages 262–276, 2001.

[HLL10] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The Cilkview scalability analyzer. *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures - SPAA '10*, pages 145–156, 2010.

[HS17] Demis Hassabis and David Silver. Alphago's next move. https://deepmind.com/blog/alphagos-next-move/, 2017.

[HT19] Ryan B Hayward and Bjarne Toft. *Hex: The Full Story*. CRC Press, 2019.

[Int13] Intel. Intel Xeon Phi Processor Competitive Performance. http://www.intel.com/content/www/us/en/benchmarks/server/xeon-phi/xeon-phi-theoretical-maximums.html, 2013.

[JR13] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Elsevier Science, 2013.

[KPVvdH13] J. Kuipers, A. Plaat, J. A. M. Vermaseren, and H. J. van den Herik. Improving Multivariate Horner Schemes with Monte Carlo Tree Search. *Computer Physics Communications*, 184(11):2391–2395, 2013.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, Lecture Notes in Computer Science (LNCS) 4212, pages 282–293, 2006.

[KUV15] J. Kuipers, T. Ueda, and J. A. M. Vermaseren. Code optimization in FORM. *Computer Physics Communications*, 189(October):1–19, 2015.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[Li13] Shou Li. Case Study: Achieving High Performance on Monte Carlo European Option Using Stepwise Optimization Framework. https://software.intel.com/en-us/articles/case-study-achieving-high-performance-on-monte-carlo-european-option-using-stepwise, 2013.

[LP98] Charles E. Leiserson and Aske Plaat. Programming Parallel Applications in Cilk. *SINEWS: SIAM News*, 31(4):6–7, 1998.

[MKK14] S. Ali Mirsoleimani, Ali Karami, and Farshad Khunjush. A Two-Tier Design Space Exploration Algorithm to Construct a GPU Performance Predictor. In *Architecture of Computing Systems–ARCS 2014*, pages 135–146. Springer, 2014.

[MPvdHV15a] S. Ali Mirsoleimani, Aske Plaat, Jaap van den Herik, and Jos Vermaseren. Parallel Monte Carlo Tree Search from Multi-core to Many-core Processors. In *ISPA 2015: The 13th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 77–83, 2015.

[MPvdHV15b] S. Ali Mirsoleimani, Aske Plaat, Jaap van den Herik, and Jos Vermaseren. Scaling Monte Carlo Tree Search on Intel Xeon Phi. In *The 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 666–673, 2015.

[MPVvdH14] S. Ali Mirsoleimani, Aske Plaat, Jos Vermaseren, and Jaap van den Herik. Performance analysis of a 240 thread tournament level MCTS Go program on the Intel Xeon Phi. In *The 2014 European Simulation and Modeling Conference (ESM'2014)*, pages 88–94. Eurosis, 2014.

[MRR12] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., 1996.

[OS12] D. O'Shea and R. Seroul. *Programming for Mathematicians*. Universitext. Springer Berlin Heidelberg, 2012.

[Rah13] Rezaur Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.

[Rei07] J. Reinders. *Intel threading building blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[RJ14] James Reinders and James Jeffers. *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, volume 4. Elsevier Science, 2014.

[RJM+15] James Reinders, Jim Jeffers, Iosif Meyerov, Alexander Sysoyev, Nikita Astafiev, and Ilya Burylov. *High Performance Parallelism Pearls*. Elsevier, 2015.

[Rob13] Arch D. Robison. Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science & Engineering*, 15(2):66–71, 2013.

[Rom01] John W. Romein. *Multigame – An Environment for Distributed Game-Tree Search*. PhD thesis, Vrije Universiteit, 2001.

[RPBS99] John Romein, Aske Plaat, Henri E. Bal, and Jonathan Schaeffer. Transposition Table Driven Work Scheduling in Distributed Search. In *The 16th National Conference on Artificial Intelligence (AAAI'99)*, pages 725–731, 1999.

[RVPvdH14] Ben Ruijl, Jos Vermaseren, Aske Plaat, and Jaap van den Herik. Combining Simulated Annealing and Monte Carlo Tree Search for Expression Simplification. *Proceedings of ICAART Conference 2014*, 1(1):724–731, 2014.

[RVW+13] A. Ramachandran, J. Vienne, R. V. D. Wijngaart, L. Koesterke, and I. Sharapov. Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi. In *2013 42nd International Conference on Parallel Processing*, pages 736–743, 2013.

[SBDD+02] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Gwendolyn Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R Clint Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002.

[SÇ12] Erik Saule and Umit V. Çatalyürek. An early evaluation of the scalability of graph algorithms on the Intel MIC architecture. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012*, pages 1629–1639, 2012.

[SCP+14] N. Sephton, P. I. Cowling, E. Powley, D. Whitehouse, and N. H. Slaven. Parallelization of Information Set Monte Carlo Tree Search. In *The 2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2290–2297, 2014.

[Seg11] Richard B. Segal. On the Scalability of Parallel UCT. In *Proceedings of the 7th International Conference on Computers and Games*, Lecture Notes in Computer Science (LNCS) 6515, pages 36–47, 2011.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.

[SKW10] Yusuke Soejima, Akihiro Kishimoto, and Osamu Watanabe. Evaluating Root Parallelization in Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):278–287, 2010.

[SP14] L. Schaefers and M. Platzner. Distributed Monte-Carlo Tree Search: A Novel Technique and its Application to Computer Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):1–15, 2014.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354, 2017.

[Suk15] Jim Sukha. Brief announcement: A compiler-runtime application binary interface for pipe-while loops. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 83–85. ACM, 2015.

[TD15] Fabien Teytaud and Julien Dehos. On the Tactical and Strategic Behaviour of MCTS When Biasing Random Simulations. *ICCA Journal*, 38(2):67–80, 2015.

[TV10] M. Tentyukov and J. A. M. Vermaseren. The multithreaded version of FORM. *Computer Physics Communications*, 181(8):1419–1427, 2010.

[TV15] Ashkan Tousimojarad and Wim Vanderbauwhede. Steal locally, share globally. *Int. J. Parallel Program.*, 43(5):894–917, 2015.

[vdHPKV13] Jaap van den Herik, Aske Plaat, Jan Kuipers, and Jos Vermaseren. Connecting Sciences. In *In 5th International Conference on Agents and Artificial Intelligence (ICAART)*, volume 1, pages IS–7–IS–16, 2013.

[Ver13] J. A. M. Vermaseren. Hepgame-description of work. https://www.nikhef.nl/ form/maindir/HEPgame/HEPgame.html, 2013.

[Wei17] Eric W. Weisstein. Game of Hex. From MathWorld—A Wolfram Web Resource. http://mathworld.wolfram.com/GameofHex.html, 2017.

[Wil12] A. Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Pubs Co Series. Manning, 2012.

[Woo14] Matthew Woodcraft. Gomill Python Library. http://mjw.woodcraft.me.uk/gomill/, 2014.

[WZS+14] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. *High-Performance Computing on the Intel® Xeon Phi*. Springer International Publishing, 2014.

[YKK+11] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa. Scalable Distributed Monte-Carlo Tree Search. In *Fourth Annual Symposium on Combinatorial Search*, pages 180–187, 2011.

Appendices

A

Micro-benchmark Programs

```
double Performance(unsigned int const ITR) {
    unsigned int const SIZE = 16;
    double start_time, duration;
    int i, j;
    __declspec(aligned(64)) double a[SIZE], b[SIZE], c[SIZE];
    for (i = 0; i < SIZE; i++) {
        a[i] = b[i] = c[i] = (double) rand();
    }
#pragma omp parallel for
    for (i = 0; i < ITR; i++) {
#pragma vector aligned (a,b,c)
#pragma unroll(16)
        for (int j = 0; j < SIZE; j++)
        {   a[j] = b[j] * c[j] + a[j]; }
    }
    start_time = elapsedTime();
#pragma omp parallel for
    for (i = 0; i < ITR; i++) {
#pragma vector aligned (a,b,c)
#pragma unroll(16)
        for (int j = 0; j < SIZE; j++)
        {   a[j] = b[j] * c[j] + a[j]; }
    }
    duration = elapsedTime() - start_time;
    double gflop = ((double) 2.0 * SIZE * ITR) / 1e+9;
    double gflops = gflop / duration;
    return gflops;
}
```

Listing A.1: Micro-benchmark code for measuring performance of Xeon Phi.

```
void Bandwidth(unsigned int const ITR) {
    unsigned int const SIZE = 48 * 1000 * 1000;
    double start_time, duration;
    int i, j;
    __declspec(aligned(64)) static double a[SIZE], b[SIZE], c[SIZE];
    for (i = 0; i < SIZE; i++) {
        c[i] = 0.0f;
        a[i] = b[i] = (double) 1.0f;
    }
    for (i = 0; i < 1; i++) {
#pragma omp parallel for
        for (j = 0; j < SIZE; j++)
        {   c[j] = a[j] * b[j] + c[j]; }
    }
    start_time = elapsedTime();
    for (i = 0; i < ITR; i++) {
#pragma omp parallel for
        for (j = 0; j < SIZE; j++)
        {   c[j] = a[j] * b[j] + c[j]; }
    }
    duration = elapsedTime() - start_time;
    double gb = (SIZE * sizeof (double)) / 1e+9;
    double gbs = 4 * ITR * gb / duration;
    return gbs;
}
```

Listing A.2: Micro-benchmark code for measuring memory bandwidth of Xeon Phi.

B

Statistical Analysis of Self-play Experiments

Suppose p as true wining probability of a player [Hei01]. The value of p is estimated by $0 \leq w = x/n \leq 1$ which results from $x \leq n$ wins in a match of n games. Therefore, we may simply assume w the sample mean of a binary-valued random variable that counts two draws as a loss plus a win.
The expected value of w is $E(w) = p$ and the variance of w is $Var(w) = p(1-p)/n$. According to central limit theorem approximately, $w \approx Normal(p, p(1-p)n)$, so $(w - p)/\sqrt{p(1-p)/n} \approx Normal(0,1)$. Let $z_{\%}$ denote the upper critical value of the standard $N(0,1)$ normal distribution for any desired %-level of statistical confidence($z_{90\%} = 1.645$, $z_{95\%} = 1.96$). Then, the probability of $w - 1.96\sqrt{p(1-p)/n} \leq p \leq w + 1.96\sqrt{p(1-p)/n}$ is about 95%. Therefore, the 95% confidence interval on the true wining probability p is $[w - 1.96\sqrt{p(1-p)/n}, w + 1.96\sqrt{p(1-p)/n}]$. There are two ways to substitute the value of p which is unknown:

1. substitute p for w: $[w - 1.96\sqrt{w(1-w)/n}, w + 1.96\sqrt{w(1-w)/n}]$
2. substitute p for $1/2$ which gives wider confidence interval: $[w - 0.98\sqrt{n}, w + 0.98\sqrt{n}]$

C Implementation of GSCPM

This section will show how the GSCPM algorithm is implemented with three different threading libraries. Furthermore, the implementations for shared search tree and random number generation are explained.

C.1 TBB

Listing C.1 gives a TBB implementation of GSCPM. TBB has *task_group* class for **fork-join** pattern. Method *run* marks where a fork occurs; method *wait* marks a join.

```
tbb::task_group g;
for (int t = 0; t < nTasks; t++) {
    g.run(UCTSearch(r,m));
}
g.wait();
```

Listing C.1: Task parallelism for GSCPM using TBB (*task_group*).

C.2 Cilk Plus

Two Cilk Plus implementations for GSCPM are given in Listing C.2 and C.3 . Cilk Plus has keywords for marking fork and join points. In the first implementation, the *cilk_spawn* marks the fork and the *cilk_sync* marks an explicitly join operation. The spawning tasks are within a *for* loop. A *cilk_sync* waits for all spawned calls in the loop.

```
for (int t = 0; t < nTasks; t++) {
    cilk_spawn UCTSearch(r,m);
}
cilk_sync;
```

Listing C.2: Task parallelism for GSCPM using Cilk Plus (*cilk_spawn*).

In the second implementation, the *cilk_for* construct uses recursive forking even though it looks like a loop. The *cilk_sync* (joint) at the end of the loop is implicit.

```
cilk_for (int t = 0; t < nTasks; t++) {
    UCTSearch(r,m);
}
```

Listing C.3: Task parallelism for GSCPM using Cilk Plus (*cilk_for*).

C.3 TPFIFO

In TPFIFO the tasks are put in a queue. It implements work-sharing, but the order that the tasks are executed is similar to *child stealing*. The first task that enters the queue is the first task that gets executed.

In our thread pool implementation (called TPFIFO) the task functions are executed asynchronously. A task is submitted to a FIFO task queue and will be executed as soon as one of the pool's threads is idle. *Schedule* returns immediately and there are no guarantees about when the tasks are executed or how long the processing will take. Therefore, the program waits for all the tasks to be completed.

```
for (int t = 0; t < nTasks; t++) {
    TPFIFO.schedule(UCTSearch(r,m));
}
TPFIFO.wait();
```

Listing C.4: Task parallelism for GSCPM, based on TPFIFO.

D

Implementation of 3PMCTS

In this section, we present the implementation of our 3PMCTS algorithm. In section D.1 we present the concept of *token* (when used as type name, we write *Token*). Section D.2 describes the implementation of 3PMCTS using TBB.

D.1 Definition of Token Data Type (TDT)

A token represents a path inside the search tree during the search. Algorithm D.1 presents definition for the type *Token*. It has four fields. (1) id represents a unique identifier for a token, (2) *v* represents the current node in the tree, (3) *s* represents the search state of the current node, and (4) Δ represents the reward value of the state. The definition of lock-free data structure *Node* is given in Algorithm 5.1. In Algorithm D.2, the serial UCT algorithm (which is already presented in Algorithm 2.2) is provided using token data type.

Algorithm D.1: Type definition for token.

```
type
    type id : int;
    type v : Node*;
    type s : State*;
    type Δ : int;
Token;
```

Algorithm D.2: The serial UCT algorithm using Token, with stages SELECT, EXPAND, PLAYOUT, and BACKUP.

```
Function UCTSearch(s_0)
    v_0 = create root node with state s_0;
    t_0.s = s_0;
    t_0.v = v_0;
    while within search budget do
        t_l = Select(t_0);
        t_l = Expand(t_l);
        t_l = Playout(t_l);
        Backup(t_l);

Function Select(Token t) : <Token>
    while t.v →IsFullyExpanded() do
        t.v := argmax_{v' ∈ children of v} v'.UCT(C_p);
        t.s →SetMove(t.v → move);
    return t;

Function Expand(Token t) : <Token>
    if !(t.s →IsTerminal()) then
        moves := t.s →UntriedMoves();
        shuffle moves uniformly at random;
        t.v →Init(moves);
        v' := t.v →AddChild();
        if t.v ≠ v' then
            t.v := v';
            t.s →SetMove(v' → move);
    return t;

Function Playout(Token t)
    RandomSimulation(t);
    Evaluation(t);
    return t

Function RandomSimulation(Token t)
    moves := t.s →UntriedMoves();
    shuffle moves uniformly at random;
    while !(t.s →IsTerminal()) do
        choose new move ∈ moves;
        t.s →SetMove(move);
    return t

Function Evaluation(Token t)
    t.Δ := t.s → Evalute();
    return t

Function Backup(Token t) : void
    while t.v ≠ null do
        t.v → Update(t.Δ);
        t.v := t.v → parent;
```

D.2 TBB Implementation Using TDD

In our implementation for 3PMCTS, each stage (task) performs its operation on a token. We can also specify the number of in-flight tokens.

Each function constitutes a stage of the non-linear pipeline in 3PMCTS. There are two approaches for parallel implementation of a non-linear pipeline [MRR12]:

- *Bind-to-stage*: A processing element (e.g., thread) is bound to a stage and processes tokens as they arrive. If the stage is parallel, it may have multiple processing elements bound to it.
- *Bind-to-item*: A processing element is bound to a token and carries the token through the pipeline. When the processing element completes the last stage, it goes to the first stage to select another token.

```
void 3PMCTS(tokenlimit){
...
/* The routine tbb::parallel_pipeline takes two parameters.
(1) A token limit. It is an upper bound on the number of tokens that are processed simultaneously.
(2) A pipeline. Each stage is created by function tbb::make_filter. The template arguments to
make_filter indicate the type of input and output items for the filter. The first ordinary argument
specifies whether the stage is parallel or not and the second ordinary argument specifies a function
that maps the input item to the output item.
*/
    tbb::parallel_pipeline(tokenlimit,
      /* The SELECT stage is serial and mapping a special object of type tbb::flow_control, used
      to signal the end of the search, to an output token. */
      tbb::make_filter<void, Token*>(tbb::filter::serial_in_order,[&](tbb::flow_control & fc)->Token*
      {
            /* A circular buffer is used to minimize the overhead of allocating and freeing tokens
            passed between pipeline stages (it reduces the communication overhead). */
            Token* t = tokenpool[index];
            index = (index+1) % tokenlimit;
            if (within the search budget) {
                /* Invocation of the method stop() tells the tbb::parallel_pipeline that no more
                paths will be selected and that the value returned from the function should be
                ignored. */
                fc.stop();
                return NULL;
            } else {
                t = SELECT(t);
                return t
            }
        }
        ) &
        // The EXPAND stage is parallel and mapping an input token to an output token.
        tbb::make_filter<Token*, Token*>(tbb::filter::parallel,[&](Token * t){
            return EXPAND(t);
        }) &
        // The RANDOMSIMULATION stage is parallel and mapping an input token to an output token.
        tbb::make_filter<Token*, Token*>(tbb::filter::parallel,[&](Token * t){
            return RANDOMSIMULATION(t);
        }) &
        // The Evaluation stage is parallel and mapping an input token to an output token.
        tbb::make_filter<Token*, Token*>(tbb::filter::parallel,[&](Token * t){
            retun EVALUATION(t);
        }) &
        /* The  BACKUP stage has an output type of void since it is only consuming tokens,
        not mapping them. */
        tbb::make_filter<Token*, void>(tbb::filter::serial_in_order,[&](Token * t){
            return BACKUP(t);
        })
    );
...}
```

Listing D.1: An implementation of the 3PMCTS algorithm in TBB.

Our implementation for 3PMCTS algorithm is based on a bind-to-item approach. Figure 6.5 depicts a five-stage pipeline for 3PMCTS that can be implemented using TBB *tbb::parallel_pipeline* template [Rei07]. The five stages run the functions SELECT, EXPAND, RANDOMSIMULATION, EVALUATION, and BACKUP, in that order. The first (SELECT) and last stage (BACKUP) are serial in-order. They process one token at a time. The three middle stages (EXPAND, RANDOMSIMULATION, and EVALUATION) are parallel and do the most time-consuming part of the search. The EVALUATION and RANDOMSIMULATION functions are extracted out of the PLAYOUT function to achieve more parallelism. The serial version uses a single token. The 3PMCTS algorithm aims to search multiple paths in parallel. Therefore, it needs more than one in-flight *token*. Listing D.1 shows the key parts of the TBB code with the syntactic details for the 3PMCTS algorithm.