Mestrado em Engenharia Informática
Dissertação/Estágio
Relatório Final

Exploração de ambientes desconhecidos com Clusters Robóticos

Bruno Duarte Gouveia
bgouveia@student.dei.uc.pt

Orientadores:
Prof. Doutor Lino Forte Marques
ISR - FCTUC

Prof. Doutor Daniel Castro Silva
DEI - FCTUC

Data: 4 de Setembro 2013

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Exploração de ambientes desconhecidos com Clusters Robóticos

Resumo

Um dos problemas fundamentais de robótica móvel é a navegação, sendo este ainda mais acentuado quando o ambiente é desconhecido. Nesse âmbito foram implementados vários algoritmos entre quais a família de algoritmos de *Simultaneous Localization And Mapping* (SLAM).

A presente dissertação descreve o desenvolvimento de um estudo sobre a aplicação de uma versão distribuída em *cluster* do algoritmo de SLAM GMapping, para explorar ambientes desconhecidos, numa configuração multirobot sem uso de computação externa ao sistema, continuando a explorar a ideia de "Robotic Clusters" desenvolvida no Laboratório de Sistemas Embebidos (LSE).

Foi desenvolvido um *package* em C/C++ para a plataforma *Robot Operating System* (ROS) e aplicado a um ambiente multirobot. Foi escolhido o algoritmo *Grid Mapping* (GMapping) por ser *grid based*, podendo assim funcionar em qualquer ambiente, e por ser mais facilmente adaptado a um ambiente em *cluster*.

Numa primeira fase obteve-se um maior conhecimento sobre os conceitos de robótica móvel implementando uma plataforma móvel de raiz. Seguidamente passou-se a utilizar a plataforma *Roomba*. Testaram-se os algoritmos de localização *Adaptive Monte Carlo Localization* (AMCL) (com mapa conhecido) e GMapping (SLAM), utilizando *Laser Range Finders* e um *Kinect* como sensores, medindo o desempenho do sistema.

Numa segunda fase implementaram-se duas arquitecturas em *cluster* do algoritmo GMapping. Na primeira abordagem tentou-se manter o sistema *"stateless"* para não haver dependência entre o cliente (robot a correr o algoritmo) e os *"remote workers"* (computadores/robots a correr parte do algoritmo). Devido aos resultados obtidos com *datasets* maiores implementou-se uma arquitectura que tentasse reduzir a comunicação, tendo que manter um estado global no sistema.

A primeira abordagem produziu inicialmente bons resultados, obtendo-se um *speedup* de 1,32 sobre a versão linear do algoritmo, permitindo assim a um Eee PC 901 (Intel® Atom™ N270) correr o algoritmo em conjunto com um computador de apoio (Intel® Core™ 2 Duo T9300), obtendo um mapa correcto. Contudo com *Datasets* maiores esta abordagem chegou rapidamente ao seu limite, o tempo de serialização dos mapas era superior ao tempo ganho pela distribuição.

A segunda abordagem obteve bons resultados, mesmo em mapas maiores, resultando daí um *speedup* de 1,81 com um *Remote Worker* com 50% das partículas, usando o *Dataset* do Killian Court do Massachusetts Institute of Technology. Esta abordagem revelou-se menos limitada por haver menor troca de dados, podendo unicamente haver necessidade de serialização de mais dados na fase de *Resampling*, consoante a distribuição escolhida das partículas pelos computadores.

Palavras-Chave: Robot Cluster,GMapping, MPI, Rao-Blackwellized Particle Filter, RBPF, Robótica móvel, ROS, SLAM, ZeroMQ

Conteúdo

Glossário

AMCL	*Adaptive Monte Carlo Localization.*
AMQP	*Advanced Message Queuing Protocol.*
API	*Application Programming Interface.*
DNS	*Domain Name System.*
DP-SLAM	*Distributed Particle SLAM.*
EKF	*Extended Kalman Filter.*
Encoder	Dispositivo electromecânico, incremental ou absoluto, que faz medições de distâncias angulares ou lineares.
GCC	*GNU Compiler Collection.*
GMapping	*Grid Mapping.*
GPGPU	*General-Purpose Computing on Graphics Processing Units.*
GPU	*Graphics Processing Unit.*
HTTP	*Hypertext Transfer Protocol.*
INPROC	*In-Process (inter-thread) Communication Transport.*
JSON	*JavaScript Object Notation.*
KF	*Kalman Filter.*
KLT	*Kanade–Lucas–Tomasi feature tracker.*
LSE	Laboratório de Sistemas Embebidos.
MPI	*Message Passing Interface.*
odometria	Estimativa da posição e orientação do robot a partir da integração temporal do movimento.
PGM	*Pragmatic General Multicast.*
ponte H	Circuito electrónico para controlar motores de corrente continua a partir de um microcontrolador.
pose	Posição e orientação.
RBPF	*Rao-Blackwellised Particle Filter.*
RGB-D	*RedGreenBlue-Depth.*
ROS	*Robot Operating System.*
RS-232	Norma usada na porta série.
SCI	*Serial Command Interface.*
Servomotor	Actuador rotativo com controlo de posição.
SIFT	*Scale-Invariant Feature Transform.*
SIMD	*Single Instruction, Multiple Data.*
SISD	*Single Instruction, Single Data.*

SLAM	*Simultaneous Localization And Mapping.*
SoC	*System On a Chip.*
SSE	*Streaming SIMD Extensions.*
STL	*Standard Template Library.*
TF	*Transformation Library.*
URDF	*Unified Robot Description Format.*
VSLAM	*Visual SLAM.*

Lista de Figuras

Lista de Tabelas

1 Introdução

Os progressos realizados nas últimas décadas em robótica móvel possibilitaram a criação de vários robots que são utilizados nas mais diferentes áreas, desde robots autónomos industriais [13], aspiradores autónomos (Roomba [24]), que aspiram a casa e que depois voltam sem ajuda do utilizador à sua base, ao exemplo mais famoso nos últimos tempos, o último *Rover* autónomo da NASA, Curiosity [41], que está neste momento a explorar Marte.

Um dos problemas fundamentais da robótica móvel é a navegação, isto é, ser capaz de se mover num determinado ambiente de um local de origem para um local de destino. Muitos desses ambientes podem ser perigosos (por exemplo: um armazém onde há uma fuga de gás tóxico) ou de acesso impossível (por exemplo: sistemas de condutas de ar de um prédio). Esse problema ainda é mais complexo quando o ambiente é desconhecido. Para conseguir gerar um mapa um robot precisa de saber onde está no ambiente, e para saber onde está no ambiente precisa de um mapa.

Esse problema pode ser actualmente resolvido usando robots de diversas formas, equipados com o conjunto necessário de sensores (*Laser Range Finder*, Câmara Estereoscópica, Sonares...) correndo um dos algoritmos de *Simultaneous Localization And Mapping* (SLAM)(uma parte destes algoritmos vai ser explicada com mais detalhe no capítulo 2).

Em sistemas *Multirobot* (sistemas que utilizem dois ou mais robots para efectuarem uma determinada tarefa em conjunto) existem geralmente picos de computação, como por exemplo momentos em que o sistema precisa de fundir mapas parciais para obter um mapa global. Estas fases são geralmente processadas por um único computador/robot do sistema. Isto geralmente traz a desvantagem de esse computador ser sobredimensionado para o resto do tempo. Para colmatar esta falha o Laboratório de Sistemas Embebidos (LSE) introduziu o conceito de *cluster* robótico, que assume cada membro do sistema(robot) como um nó de um *cluster*, ou sistema distribuído. Este sistema pode correr uma versão paralela e distribuída do algoritmo usado nos momentos de picos de processamento.

1.1 Objectivos da Investigação

O objectivo desta investigação é estudar a possibilidade de correr uma versão do algoritmo GMapping [15] em ambiente de *cluster* robótico sem uso de computação externa como em [7], podendo ser usada em qualquer sítio, tentando descobrir qual a melhor arquitectura possível para o algoritmo baseando-se já nos estudos efectuados em [34] e [31]. Para tal vai-se elaborar um *package* para o ROS usando *Message Passing Interface* (MPI), baseando-se no código disponível do *package* GMapping [47] (que por sua vez é baseado na implementação GMapping do OpenSLAM [43]), validando os resultados com testes em ambiente real.

1.2 Motivação

Poder correr o algoritmo de SLAM é uma parte essencial de muitos projectos que envolvem robots, trazendo sempre o problema de ter de desenhar a plataforma usada de maneira a poder ser equipada com um computador veloz. Isto, por sua vez, traz complicações a nível do tamanho e do consumo energético do robot, o qual terá, por exemplo, de ser equipado com uma bateria de capacidade elevada para poder alimentar todo o sistema, o que leva a um aumento de peso.

Os algoritmos SLAM são computacionalmente muito dispendiosos, sobretudo para ambientes complexos, e/ou quando se usa sensores tal como câmaras estereoscópicas ou RGB-D (tal como o Kinect [35]), em que a quantidade de dados a processar é elevada.

Correndo o algoritmo em estrutura de *cluster* ia trazer mais flexibilidade na configuração dos robots. Poder-se-ia por exemplo equipar a plataforma com várias placas de baixo consumo baseadas em *System On a Chip* (SoC) ARM (tal como o *Raspberry Pi* [46] ou *ODROID-U2* [18]), as quais ocupariam menos espaço que um computador e seriam também mais baratas. Outra vantagem desta configuração é que

se poderia correr o algoritmo de SLAM com parâmetros mais exigentes (usando por exemplo um maior número de partículas) permitindo criar mapas correctos e com maior precisão.

Em sistemas multirobot poder-se-ia ter em conta a heterogeneidade, usando por exemplo vários robots pequenos (que se adaptariam mais facilmente ao ambiente a explorar), com menor capacidade de processamento e um ou vários robots equipados com computadores mais velozes para compensar.

1.3 Enquadramento

Este trabalho enquadra-se no projecto "*Robotic clusters*" do Laboratório de Sistemas Embebidos (LSE), o qual foi introduzido com o artigo [31]. O tal projecto consiste no aproveitamento da capacidade de processamento de cada elemento num ambiente multirobot, para responder a picos de computação que acontecem regularmente, sem necessitar de equipar os robots com computadores velozes. Este projecto faz parte de vários projectos elaborados no LSE que envolvem robots móveis, tais como o projecto europeu "TIRAMISU" [1] ou o projecto "*Climbing Robots with high maneuverability for inspection and maintenance of 3D human-made structures*" [2].

1.4 Estrutura

Este documento está dividido em três partes.

O primeiro capítulo contém o estado da arte onde se faz um estudo sobre aplicações multirobot, passando a seguir para alguns dos algoritmos usados em exploração e geração de mapas em ambientes desconhecidos, e finalmente é dada uma visão global sobre arquitecturas e técnicas usadas para paralelizar estes algoritmos.

No capítulo seguinte é efectuada uma análise sobre algumas das ferramentas disponíveis, detalhando a razão da escolha final destas, passando de seguida para o capítulo da arquitectura e implementação onde são descritas as duas abordagens implementadas. Finalmente no último capítulo são expostos os resultados obtidos.

[1] Mais informação disponível em http://www.isr.uc.pt/index.php/embedded-systems/projects?task=showprojectisr.show&idProject=12

[2] Mais informação disponível em http://www.isr.uc.pt/index.php/embedded-systems/projects?task=showprojectisr.show&idProject=13

2 Estado da Arte

Este capítulo é dividido em três partes. A primeira parte descreve aplicações multirobot, a segunda parte descreve os algoritmos de exploração e geração de mapas, dando uma visão global sobre alguns dos algoritmos usados. E finalmente, a terceira parte descreve as várias técnicas usadas para paralelizar os algoritmos de SLAM.

2.1 Aplicações Multirobot

Sistemas multirobot são geralmente usados em cenários de procura, em que um único robot não consegue obter resultados validos em tempo útil, devido a dimensão do espaço de procura. O que todos os cenários têm também em comum, é a necessidade de obter um mapa final o mais correcto possível. De seguida são dados três exemplos de aplicações multirobot:

Em [33] usaram um sistema multirobot para explorar um ambiente desconhecido, procurando a origem do incêndio. É um bom exemplo de um cenário em que se tem de obter a localização do que se procura o mais rápido possível, neste caso até num ambiente perigoso para o ser humano. Os robots são equipados com sensores químicos e de temperatura para obter dados sobre o ambiente, além de usarem sonares para evitar colisões. Usaram triangulação para identificar a posição da origem do incêndio.

Em [32] continuaram a explorar o uso de sistemas multirobot para procura. Neste caso procuraram fontes de odor em ambientes estruturados, podendo ser usado por exemplo num cenário em que haja uma fuga de gás num armazém ou noutro ambiente fechado. Neste caso, além de sonares, também equiparam os robots de anemómetros para medir a circulação do ar. Cada robot mantém o sistema de coordenadas local e usa grafos conexos em que cada nó contém uma característica específica do ambiente (beco sem saída, cruzamento, ...) para construir um mapa local. Sempre que um robot encontra uma nova característica insere um nó correspondente no seu mapa local e envia uma mensagem com o seu mapa aos outros robots. Usam de seguida técnicas de *graph matching* para fundir os mapas locais num mapa global.

Em [53] tentaram resolver o uso de exploração multirobot com um número elevado de elementos em ambientes sem infraestrutura de comunicação existente. Neste caso específico usaram 100 robots, dando a alcunha de *Centibots* ao sistema. Dividiram o trabalho em três fases : mapeamento, procura e protecção. Na fase de mapeamento os robots tentam verificar as suas posições relativas para obter um mapa global partilhado com precisão, tentando coordenar a exploração para que a mesma seja efectuada da maneira mais rápida possível. Na fase de procura os robots são distribuídos pelo ambiente e tentam encontrar um objecto específico. Finalmente na fase de protecção os robots tornam-se em robots de patrulha, são distribuídos pelo ambiente e tentam detectar intrusos.

2.2 Algoritmos de Exploração e Geração de Mapas

O SLAM consiste numa família de algoritmos. Pode ser dividida pelos sensores que usa e pela maneira de guardar os seus dados.

Há vertentes do SLAM que usam sensores 2D tal como o *Laser Range Finder*, que permite receber um conjunto de pontos num plano horizontal à volta do robot (ver Figura 1), que representam as distâncias a que estão os objectos, ou sensores 3D tal como por exemplo câmaras *RedGreenBlue-Depth* (RGB-D) que permitem receber uma *Pointcloud* (um conjunto de pontos no espaço, ver Figura 2) e ao mesmo tempo informação visual do ambiente.

As versões mais recentes dos algoritmos SLAM são baseadas em métodos probabilísticos que diferem sobretudo pela maneira de armazenar os dados. Dividem-se em duas classes: os algoritmos com base em

features do ambiente e os baseados em grelha. Os algoritmos baseados em *features* guardam as posições de um tipo predefinido de formas geométricas detectadas no ambiente. Devem ser facilmente detectáveis para não haver erros no mapa em caso de movimentos em ciclo do robot.

No caso dos algoritmos baseados em grelha, estes guardam um conjunto abstracto de pontos numa *occupancy grid*(ver Figura 3) que representam a posição dos robot e dos obstáculos no ambiente.

Estes algoritmos têm em comum o problema de *loop closure*, necessitando geralmente de aplicar técnicas dedicadas para o resolver. A incerteza sobre a pose(Posição e orientação) aumenta com o movimento do robot, as características do mapa registadas longe do inicio vão ter portanto maior incerteza. Uma revisita às características previamente observadas (*closing the loop*) faz com que a incerteza da posição destas e da pose do robot diminua, o problema de *loop closure* consiste em reconhecer estas características e actualizar assim os dados guardados.

Nos próximos capítulos vão ser apresentados alguns dos algoritmos existentes das duas famílias (*Feature Based* e *Grid Based*), terminando com uma tabela comparativa (ver Tabela 2) com um comentário sobre cada algoritmo.

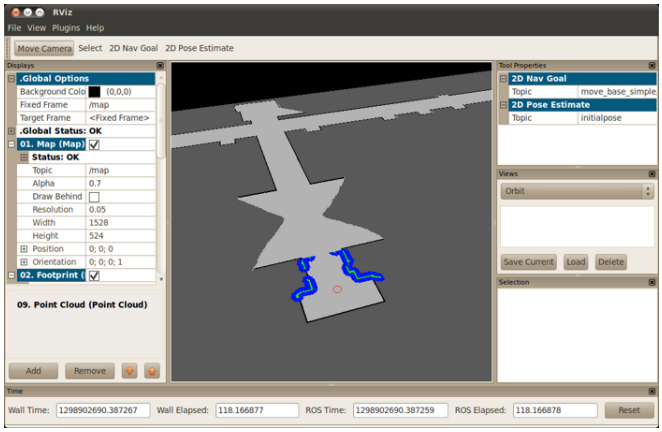

Figura 1: Screenshot da Ferramenta RViz, Laserscan visível a azul

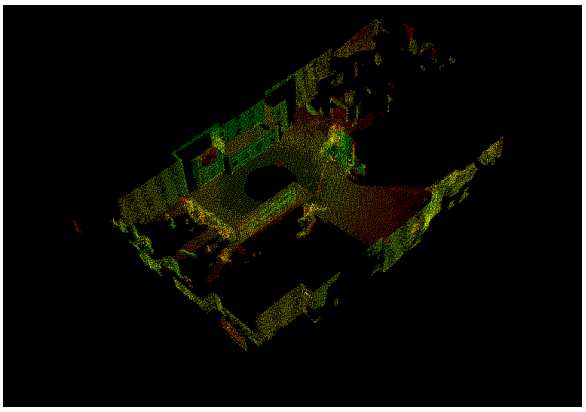

Figura 2: Exemplo Visual de PointCloud (retirado de http://www.ros.org/wiki/pointcloud_registration)

2.2.1 Feature Based SLAM

Todos os algoritmos baseados em *features* têm em comum o uso de objectos(postes, portas,...) ou características específicas do ambiente (bifurcação no caminho, cruzamento, ...) predeterminadas, chamados *Landmarks* e têm de ser facilmente detectáveis e inconfundíveis. Um erro ao detectar um *landmark* poderá levar a mapas errados, ou a maiores dificuldades na fusão de vários mapas.

No ponto seguinte são dados dois exemplos de *Feature Based SLAM*.

2.2.1.1 EKF-SLAM

O EKF-SLAM, tal como o nome indica, aplica o algoritmo *Extended Kalman Filter* (EKF) ao problema do SLAM.

O *Kalman Filter* (KF) (e EKF) faz parte da família de estimadores de estado recursiva ou também chamada de filtros gaussianos [51]. Estima o estado actual do sistema baseando-se na estimativa anterior e nos dados medidos que contêm ruído, sendo geralmente melhor que o estado unicamente descrito pelos sensores.

Algoritmo 1: Kalman Filter (retirado de [51])

1 **Algorithm Kalman_filter**$(\mu_{t-1}, \Sigma_{t-1}, u_t)$:
2 $\quad \bar{\mu}_t = A_t\mu_{t-1} + B_t u_t$
3 $\quad \bar{\Sigma}_t = A_t\Sigma_{t-1}A_t^T + R_t$
4
5 $\quad K_t = \bar{\Sigma}_t C_t^T (C_t \bar{\Sigma}_t C_t^T + Q_t)^{-1}$
6 $\quad \mu_t = \bar{\mu}_t + K_t(z_t - C_t\bar{\mu}_t)$
7 $\quad \Sigma_t = (I - K_t C_t)\bar{\Sigma}_t$
8 $\quad$ **return** μ_t, Σ_t

O algoritmo do filtro de Kalman (ver algoritmo 1) representa a estimativa no tempo t pela média μ_t e covariância Σ_t. Como entrada recebe a estimativa anterior representada por μ_{t-1} e Σ_{t-1}. Na actualização destes valores usa-se o controlo u_t e as medições z_t. Na linha 2. e 3. calcula a estimativa prevista para o tempo t $(\bar{\mu}_t, \bar{\Sigma}_t)$, incorporando o controlo u_t, a média é actualizada usando a função linear de transição de estado (linha 2.), a covariância é actualizada tendo em conta o facto de ser dependente do estado anterior, através da matriz linear A. Obtém-se depois a estimativa (μ_t, Σ_t) nas próximas linhas, incorporando as medições z_t. A variável K_t é conhecida como o "ganho de kalman", define o factor da influência das medições na estimativa.As variáveis A,B,C e Q são variáveis específicas de cada sistema modelado.

O EKF é uma melhoria do algoritmo KF na medida em que pode ser aplicado a sistemas não lineares [9], linearizando os dados não gaussianos usando a expansão de Taylor para calcular a matriz Jacobiana. O EKF-SLAM mantém uma estimativa da pose do robot e dos *landmarks* (ponto de referência) que detectou no ambiente, guardando um vector de estados (μ_t) e a sua matriz de covariância (Σ_t), linearizando o modelo do movimento do robot(controlo u_t) e das suas observações(z_t) [10]. O EKF-SLAM parte do princípio de que os *landmarks* são facilmente detectados, estáticos e distinguíveis. Caso isto não se aplique ao ambiente, o mapa gerado pelo algoritmo não converge.

Um dos grandes problemas do EKF-SLAM é a parte de actualização da matriz de covariância: sempre que é detectado um novo *landmark* no ambiente, o algoritmo tem que actualizar a matriz. Esta operação tem complexidade quadrática, o que traz problemas para ambientes com uma grande quantidade de *landmarks*.

2.2.1.2 FastSLAM

Em [36] foi introduzido o FastSLAM, é um algoritmo que aplica a noção de filtro de partículas ao SLAM, neste caso específico o *Rao-Blackwellised Particle Filter* (RBPF). Utilizando este método, conseguiram resolver o problema da complexidade quadrática da inserção de novos *landmarks*, passando agora a ser logarítmica.

Algoritmo 2: Filtro de partículas (retirado de [51])

```
1  Algorithm Particle_filter(χ_{t-1}, u_t, z_t) :
2      χ̄_t = χ_t = ∅
3      for m=1 to M do
4          sample x_t^[m] ~ p(x_t|u_t, x_{t-1}^[m])
5          w_t^[m] = p(z_t|x_t^[m])
6          χ̄_t = χ̄_t + ⟨x_t^[m], w_t^[m]⟩
7      endfor
8      for m=1 to M do
9          draw i with probability ∝ w_t^[i]
10         add x_t^[i] to χ_t
11     endfor
12     return χ_t
```

No filtro de partículas (ver algoritmo 2), cada partícula $x_t^{[m]}$ representa uma hipótese/estimativa para o estado do sistema no momento t, havendo sempre M partículas (geralmente M é um número elevado). Tal como o KF utiliza a informação das estimativas anteriores x_{t-1} e do controlo u_t, mas neste caso usa essa informação na fase de *sampling* (linha 4) para determinar se a partícula x_t pertence ao *set* resultante χ_t. Na linha 5 calcula-se o factor de importância $w_t^{[m]}$ (ou peso) de cada partícula para incorporar as medições z_t. Finalmente nas linhas 8-11 efectua-se a fase de *resampling* ou *importance sampling*: com base no valor dos pesos escolhe-se aleatoriamente M partículas do *set* temporário $\bar{\chi}_t$ com reposição, podendo portanto haver duplicadas. Usando este método eliminam-se as partículas com pesos baixos. A grande vantagem do filtro de partículas sobre o KF é de este não estar limitado a sistemas com modelo linear e gaussiano.

No caso do FastSLAM, o *sampling* passa a ser feito sobre a trajectória $x_{1:t}$ e não a pose, como no EKF-SLAM, o que torna os *landmarks* do mapa condicionalmente independentes. Cada partícula(x_t) do RBPF mantém uma estimativa da trajectória $x_{1:t}$ e K EKF (com o vector μ_t de tamanho dois que contém a posição do *landmark* e uma matriz de covariância quadrática dois por dois Σ_t) que estimam a localização de K *landmarks* condicionados a trajectória. Usando M partículas obtém-se a complexidade do algoritmo *O(MK)*. Em [36] reduziram a complexidade para *O(M log K)*, usando uma estrutura em árvore.

2.2.2 Grid Based SLAM

A grande desvantagem dos algoritmos de SLAM com base em *Features* é de não se poderem aplicar em ambientes completamente desconhecidos. Para funcionarem necessitam sempre de *Landmarks* previamente definidos que sejam inconfundíveis e facilmente identificáveis. Em ambientes abertos, com um número muito pequeno de *Landmarks*, ou com falta de *features* óbvias, também se torna difícil usar este tipo de algoritmo. Para colmatar estas falhas, introduziu-se o *Grid Based SLAM*, o qual funciona com base em *occupancy Grids*(ver Figura 3), o espaço (geralmente um plano 2D) é dividido em grelha com informação detectada em cada célula. Cada célula pode ter um dos seguintes estados: ocupado, livre ou desconhecido.

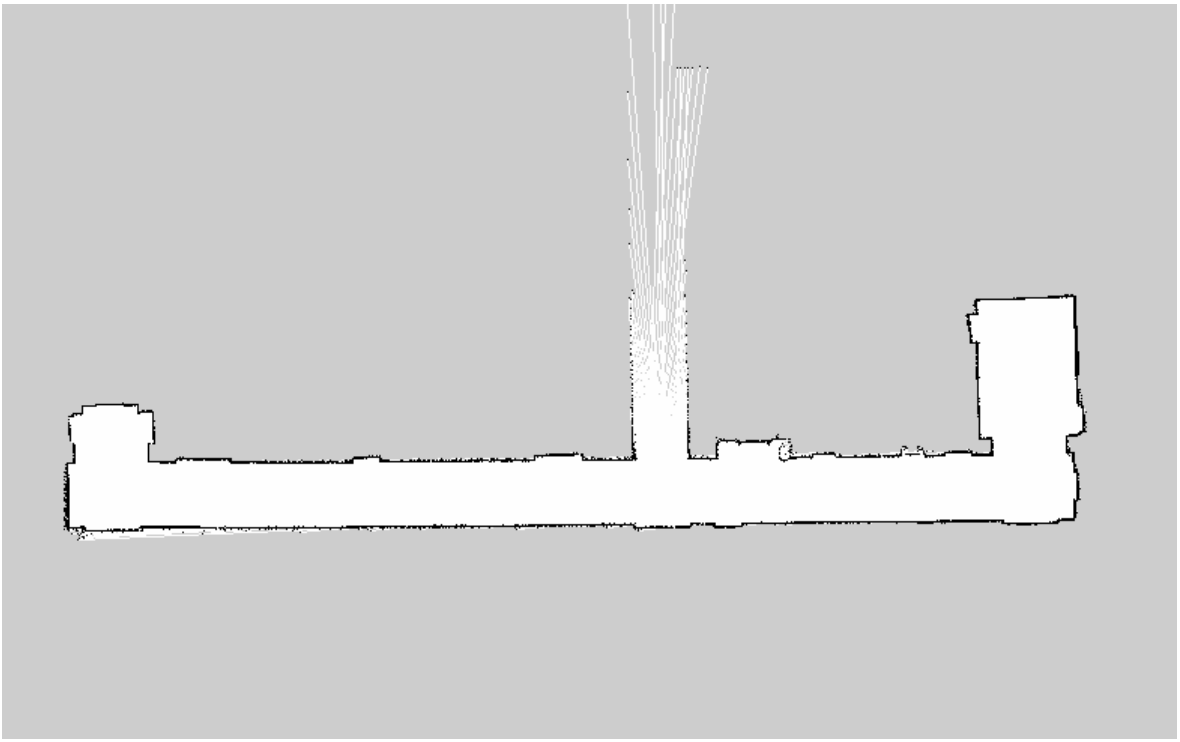

Figura 3: *Occupancy grid* gerada pelo *package* GMapping

A seguir vão ser dados dois exemplos de *Grid Based SLAM*, sendo o último o algoritmo que se vai usar neste trabalho.

2.2.2.1 DP-SLAM

Em [12] implementaram o algoritmo *Distributed Particle SLAM* (DP-SLAM), o qual assemelha-se ao FastSLAM, que também usa RBPF, mas não usa *landmarks*. Em vez disso, mantém uma lista detalhada de mapas (*Occupancy Grid*). Usa um filtro de partículas para estimar a pose do robot e as várias configurações possíveis dos mapas, e apenas um *Laser range Finder* como sensor principal.

Para reduzir o número de cópias dos mapas entre a partícula antiga e a nova obtida durante a fase de *resampling*, o DP-SLAM, além da estrutura do mapa, usa uma estrutura em árvore chamada *particle ancestry tree*. Tem como raiz uma partícula que é pai de todas as outras. Cada partícula mantém unicamente uma lista das grelhas que modificou no mapa. Para manter o número de partículas da árvore limitado, removem-se partículas desnecessárias de maneira recursiva podendo-se fundir a informação do mapa entre nó pai e filho se este for único. Em [12] observaram que não era necessário usar técnicas especiais de *loop closure* para obter bons resultados, bastava usar um número razoável de partículas.

2.2.2.2 GMapping

O algoritmo GMapping introduzido em [15], tal como o DP-SLAM, também usa o RBPF, mas mantém uma cópia individual do mapa em cada partícula.

Neste caso, em vez de tentar resolver o problema da quantidade de dados usando uma estrutura especial, tentaram reduzir o número de partículas necessárias para que o algoritmo convergisse. Para isso tiveram em conta a precisão dos sensores na fase de *Sampling* das partículas (linha 4 do Algoritmo 2) não se baseando unicamente na odometria (controlo u_t) e usam uma estratégia adaptativa de *Resampling*

(linha 9 do Algoritmo 2)), efectuando este passo unicamente quando for necessário, para manter uma boa variedade de partículas sem correr o risco de haver *particle depletion*: casos em que devido ao eliminar partículas com pesos baixos, mantendo cópias das mesmas com peso alto, não se reencontre dados para reconhecer as características do mapa por onde o robot já passou. O algoritmo vai ser explicado com mais detalhe no capítulo 3.3.

2.2.3 Resumo

A Tabela 2 contém um resumo sobre cada algoritmo descrito nos capítulos anteriores, descreve de que família são (*Feature Based ou Grid Based*) e qual o algoritmo de base que implementam (EKF ou RBPF)

SLAM	**Tipo de SLAM**	**Algoritmo**	**Comentário**
EKF-SLAM	Feature Based	EKF	Um dos primeiros algoritmos de SLAM, performance do algoritmo ultrapassado pelo algoritmo RBPF
FastSLAM	Feature Based	RBPF	Primeiro uso de RBPF em algoritmo de SLAM, continua a ser *Feature Based*, não funcionando em ambientes completamente desconhecidos (sem possibilidade de definir as *landmarks* previamente)
DP-SLAM	Grid Based	RBPF	Mantém uma única *occupancy grid* e usa estrutura especial (*ancestry tree*) para manter as partículas que guardam unicamente as modificações. O uso desta estrutura torna a versão em *cluster* mais complicada
GMapping	Grid Based	RBPF	Tenta diminuir o número de partículas necessárias através de *resampling* mais eficiente. Cada partícula mantém uma cópia única do mapa e as operações sobre estas são independentes, facilitando o paralelismo

Tabela 2: Comparação das algoritmos SLAM

2.3 Arquitecturas e Modelos de Paralelização

Um robot móvel produz uma grande quantidade de dados que muitas vezes têm de ser processados em tempo real ou o mais rápido possível. Consequentemente desde cedo se tentaram paralelizar os algoritmos usados e aproveitar arquitecturas e modelos de execução paralelos.

2.3.1 SIMD e Multithreading

Como em robótica móvel se lida constantemente com pelo menos o vector da pose e os pontos no espaço, que nos são fornecidos pelos sensores de distância, existem sempre operações de transformação sobre estes.

Em [11] aproveitaram esse facto para usar *Single Instruction, Multiple Data* (SIMD) na parte de *scan-matching* do algoritmo. Em mapas de duas dimensões, *scan-matching* consiste em obter o valor da translação e da rotação sobre um conjunto de pontos devolvido pelo *Laser Range Finder*, para este se sobrepor a um segundo conjunto detectado a partir de outra posição.

Tentaram portanto explorar o paralelismo a nível dos dados. Usando as unidades SIMD disponíveis em processadores modernos, consegue-se aplicar uma operação sobre um conjunto (vector) de dados ao mesmo tempo.

Neste caso específico usaram três instruções *Streaming SIMD Extensions* (SSE) e uma SSE3 (instruções SIMD desenvolvidas pela Intel). Estão disponíveis na maior parte dos processadores compatíveis com a arquitectura x86, tendo a primeira versão sido introduzida em 1999 com o processador Pentium III.

São usadas para efectuar a mesma transformação sobre vários vectores e para converter o resultado de números de virgula flutuante para inteiros em simultâneo. Com estas modificações obtiveram um *speedup* de 3,5 sobre a versão não-SIMD. Um facto interessante foi que descobriram em testes que usando apenas a instrução "_mm_cvtps_pi32(b)" para converter dois números de virgula flutuante (*single precision*) para inteiros de 32 bits, obtinham *Speedup* de 16,4 sobre a versão clássica (*Single Instruction, Single Data* (SISD)) da conversão. Estas optimizações podem ser usadas para qualquer algoritmo SLAM baseado em grelha, tal como o GMapping ou o DP-SLAM. São específicas dos processadores mais recentes da família x86, mas devem ser facilmente traduzidas para arquitecturas diferentes com suporte para instruções SIMD, tal como a família de processadores ARM Cortex (a partir de A8 [5] com instruções NEON [6]).

Muitas vezes não é possível paralelizar um algoritmo, o *speedup* resultante não é o desejado ou é mais pertinente correr várias vezes o mesmo algoritmo com parâmetros diferentes.

Em [28] seguiram esta última abordagem, no algoritmo que chamaram de *PF-SLAM*(*Particle Filter SLAM*). Usaram duas *threads* para correr dois algoritmos RBPF com número de partículas diferentes (20 e 100).

Este algoritmo, que foi introduzido no trabalho anterior [27], fundia a informação do *Laser Range Finder* e da câmara estereoscópica, depois de fazer o *sampling* sobre as estimativas da pose do robot (linha 4 no algoritmo 2). No caso de [28], usam unicamente o *Laser Range Finder*.

O RBPF com o menor número de partículas corre a um intervalo mais curto, e o do maior número só é corrido em certos momentos chamados *keyframes*, em que a pose do robot sofre uma grande alteração. A *thread* principal recebe o *feedback* da *thread* com maior número de partículas e actualiza a estimativa do mapa e da pose do robot. Desta maneira reduz-se em média a necessidade de recursos para correr o algoritmo, mantendo a precisão de filtros com um elevado número de partículas. Isto possibilitou correr outros algoritmos em paralelo com o SLAM. Em [28] usaram esta possibilidade para correr também *3D Mapping* noutra *thread*.

Em [34] decidiram explorar o facto de cada partícula usada no RBPF ser processada de maneira independente, não havendo troca de dados com as restantes.

Os autores introduzem duas arquitecturas paralelas para correr o algoritmo. A primeira usa uma unidade central que controla a operação de todas as unidades de processamento. Gere a parte de *resampling* do algoritmo e efectua *load balancing* para distribuir as tarefas de maneira uniforme. Existe um canal de comunicação simples entre a unidade central e cada elemento de processamento(PE). Cada PE corre o algoritmo sobre as suas partículas de maneira independente.

A segunda arquitectura envolve dois canais de comunicação (um é dedicado à unidade central e outro é partilhado pelos PE) e usa memória partilhada. A unidade central tem acesso a toda a memória e

cada PE tem acesso exclusivamente ao seu bloco. Tal como na arquitectura anterior, a unidade central é responsável por distribuir tarefas, mas desta vez a parte de *resampling* é efectuada por um conjunto de PE. Os autores não realizaram testes práticos, mas simularam os algoritmos em MATLAB. Usando a segunda arquitectura, para um algoritmo com 150 partículas obtiveram o tempo de execução mais pequeno com 10 PE, correndo o *dataset* de teste em 4,2 segundos. Com um PE demoravam mais de 20 segundos.

2.3.2 GPGPU

A evolução recente da arquitectura das placas gráficas para PC (*General-purpose computing on graphics processing units*, GPGPU) fez com que se tornasse possível disponibilizar o poder computacional destas para aplicações de uso geral (*General-Purpose Computing on Graphics Processing Units* (GPGPU)). Muito rapidamente conseguiram alcançar um desempenho teórico várias vezes superior aos processadores fabricados na mesma altura. Com a evolução das API's e linguagens de programação para *Graphics Processing Unit* (GPU), tal como o CUDA [42] ou OpenCL [26], passou a ser cada vez mais simples usar esta tecnologia para os mais diversos tipos de aplicações. Hoje em dia é usado nas mais diversas áreas, tal como por exemplo, processamento de imagens [17], simulações de fluidos [19], *brute force password cracking* [20] e *Computer Vision* [54]. Foi sobretudo nesta última área que a comunidade de robótica encontrou maior uso para GPGPU. Seguem dois exemplos recentes do uso de GPGPU em robótica.

Em [8] usaram CUDA para filtrar e converter as imagens recebidas da câmara estereoscópica, e desenvolveram também uma versão em CUDA dos algoritmos *Kanade–Lucas–Tomasi feature tracker* (KLT) e *Scale-Invariant Feature Transform* (SIFT). O uso de CUDA permite que mesmo com estes 2 algoritmos consigam correr o módulo de *Visual SLAM* (VSLAM) em tempo real, obtendo mais de 15 *frames* por segundo.

Em [44] implementaram o algoritmo *Iterative Closest Points* (ICP) com base numa versão do *Nearest Neighbor Search* (NNS) em CUDA. Estes dois algoritmos são muito usados na área de *Computer Vision*. Como experiência e para provar o melhor desempenho desta solução, compararam o desempenho com uma versão CPU implementada em OpenMP. Como dados de teste usaram duas *pointcloud's*, com sobreposição parcial, capturadas usando um laser scanner da marca SICK. A versão CUDA a correr numa GPU nVidia GeForce GTX280 obtinha um *Speedup* de 88 em comparação com a versão sequencial CPU e 25 com a versão OpenMP a correr num processador Intel Core2Duo 6600.

2.3.3 Cloud Computing e SOA

Em 2006 a Amazon lançou o serviço Amazon Elastic Compute Cloud (EC2) que disponibilizava o acesso a máquinas virtuais pela Internet. O utilizador pode alugar o número que quer de computadores para correr os seus programas. É portanto uma forma flexível de poder correr algoritmos computacionalmente dispendiosos sem ter de investir em hardware adicional, sobretudo quando esta necessidade só surge em certos momentos durante a execução de um programa ou serviço.

Em [7] foi implementado o *framework DAvinCi* que permite a um conjunto de robots usar *cloud computing*. Estes comunicam com um servidor central *DAvinCi* por mensagem ROS em formato *Hypertext Transfer Protocol* (HTTP), este formata os dados e envia-os para um *cluster* de nós *Hadoop* para processamento. Como exemplo de algoritmo em [7], implementaram o *FastSLAM* usando o modelo de programação *MapReduce*. É usada uma tarefa *Map* para cada partícula do algoritmo, depois é usada uma única tarefa *Reduce* para escolher a partícula com maior peso acumulado. Obtiveram *speedups* interessantes (perto de 7,3 com oito nós Hadoop a correr em comparação com 1) e puderam correr o algoritmo com 100 partículas, o que é um número bastante elevado, podendo assim gerar mapas complexos com mais precisão. Contudo não tiveram em conta a latência na comunicação que existe entre os robots e o *cluster* Hadoop, nem analisaram a influência do servidor *DAvinCi* na escalabilidade do sistema.

O cluster além de ser usado como *Platform as a service* (PaaS), também é usado como *Software as*

a service (SaaS), disponibilizando o mapa gerado pelo SLAM. Mesmo depois de uma quebra de ligação, ou no caso da activação de um novo membro, um robot pode pedir o mapa actual a qualquer momento.

Em [25] tentaram explorar mais a ideia do serviço do ponto de vista do robot. Inspiraram-se na área de *Enterprise application integration* (EAI), aplicaram os conceitos da *Service oriented Arquitecture* ao mundo da robótica. Desenvolveram um *framework* parecido ao *DAvinCi* [7], mas desta vez usaram-no exclusivamente para partilhar e introduzir conhecimento numa rede de robots. Um robot que aprende uma determinada acção pode partilhar esse conhecimento com outro que esteja num local distante.

2.3.4 Cluster/MPI

Em ambientes *multirobots*, em muitos casos não existe continuamente a necessidade de usar todos os recursos de processamento disponíveis, existindo apenas picos de computação de tempo em tempo. Para poder responder a essa necessidade, usa-se geralmente computadores com desempenho sobredimensionado para o resto da aplicação. Isto por sua vez traz problemas para uma plataforma móvel, aumenta os custos do material, pode aumentar as dimensões de cada robot, e pode reduzir a autonomia do grupo.

Em [31] tentaram eliminar este problema usando um *cluster* MPI de robots. Usam os robots para mapear um ambiente desconhecido, com o *Topological SLAM*, os quais comunicam usando uma rede *Wifi* em *Mesh*. O *Topological SLAM* é um SLAM *Feature based* que define como *Landmarks* a característica do corredor (beco sem saída, entroncamento, cruzamento, ...). Mantém essa informação em grafos conexos. Usam um algoritmo paralelo em MPI para fundir os grafos gerados por cada robot para obter um único grafo global(ver Figura 4). Usando esta técnica conseguiram usar computadores mais baratos e de baixo consumo. Outra vantagem desta técnica em comparação com a do uso de um *cluster* externo é que pode ser usada em ambientes sem ligação a Internet, precisando unicamente de rede *wireless* entre cada robot.

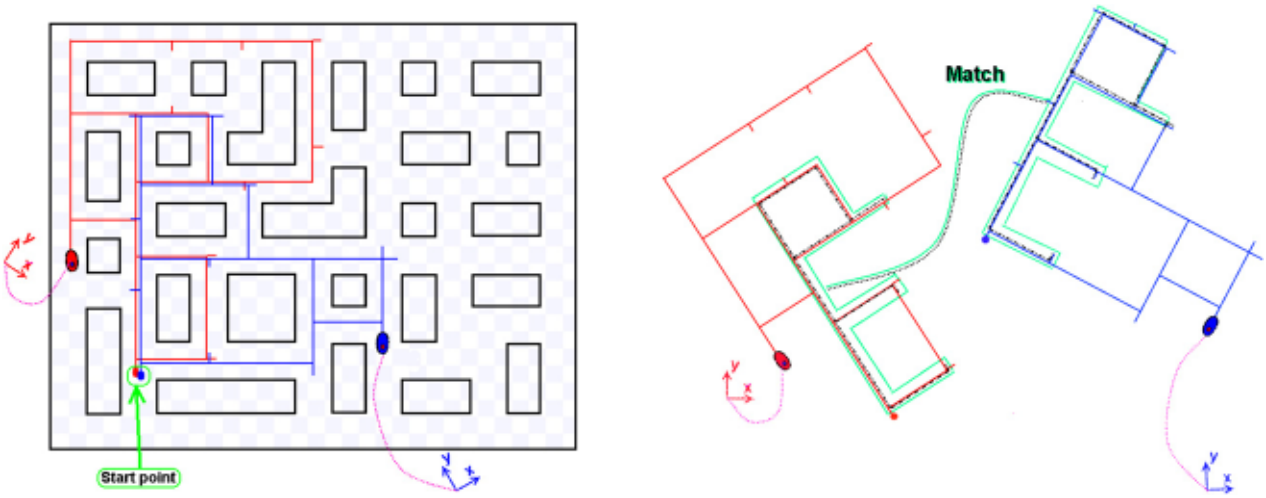

Figura 4: Fusão de Mapas

2.3.5 Resumo

A Tabela 3 contém uma comparação dos modelos paralelos utilizados em aplicações de robótica móvel. Contém o exemplo típico de aplicação de cada e um comentário breve sobre as características mais importantes.

Modelo / Arquitectura	**Aplicado a**	**Comentário**
Multithreading	SLAM baseados em filtros de partículas	O uso de *multithreading* já foi explorado várias vezes, é facilmente utilizável em robots móveis. Precisa unicamente de um processador *multicore*, o que hoje em dia é comum
SIMD	*Grid Based Slam*	A optimização é aplicável a todos os algoritmos *Grid Based* sem reestruturar o algoritmo, necessitando de um processador moderno x86 com instruções SSE, mas é facilmente adaptado a outras arquitecturas com unidades SIMD
GPGPU	VSLAM	Técnica sobretudo usada para VSLAM devido a quantidade de dados a processar.A grande desvantagem é a necessidade de usar um computador equipado com uma placa gráfica com suporte para GPGPU, o que nem sempre é prático num robot móvel
Cloud Computing	FastSLAM	Técnica interessante que remove a necessidade de computação nos robots, usando computadores externos, mas é dependente de uma rede para o exterior, podendo não ser usado em cenários que não possibilitem essa condição. Possíveis problemas de latência
Service oriented Architecture (SOA)	Infraestrutura para partilha de conhecimento	Uso apenas para partilha de conhecimento entre uma rede de robots
Cluster/MPI	*Topological SLAM* em ambiente multirobot	Uso na parte de fusão de mapas, podendo assim usar-se computadores computacionalmente mais fracos

Tabela 3: Comparação das Arquitectura/Modelos

3 Ferramentas e Implementação

Este capítulo está dividido em três partes. A primeira descreve o plano de trabalho, explicando as tarefas efectuadas, passando a seguir para as ferramentas usadas, a escolha da linguagem de programação e sistema de *clustering* e finalmente a implementação das duas arquitecturas.

3.1 Plano de Trabalho

Na Figura 5 pode-se ver o plano de trabalho.

O trabalho efectuado do ponto 1. ao ponto 6. do diagrama(ver Figura 5) foi efectuado no primeiro semestre está descrito no capítulo 3.1.1, os restantes pontos foram executados no segundo semestre e estão explicados no capítulo 3.1.2

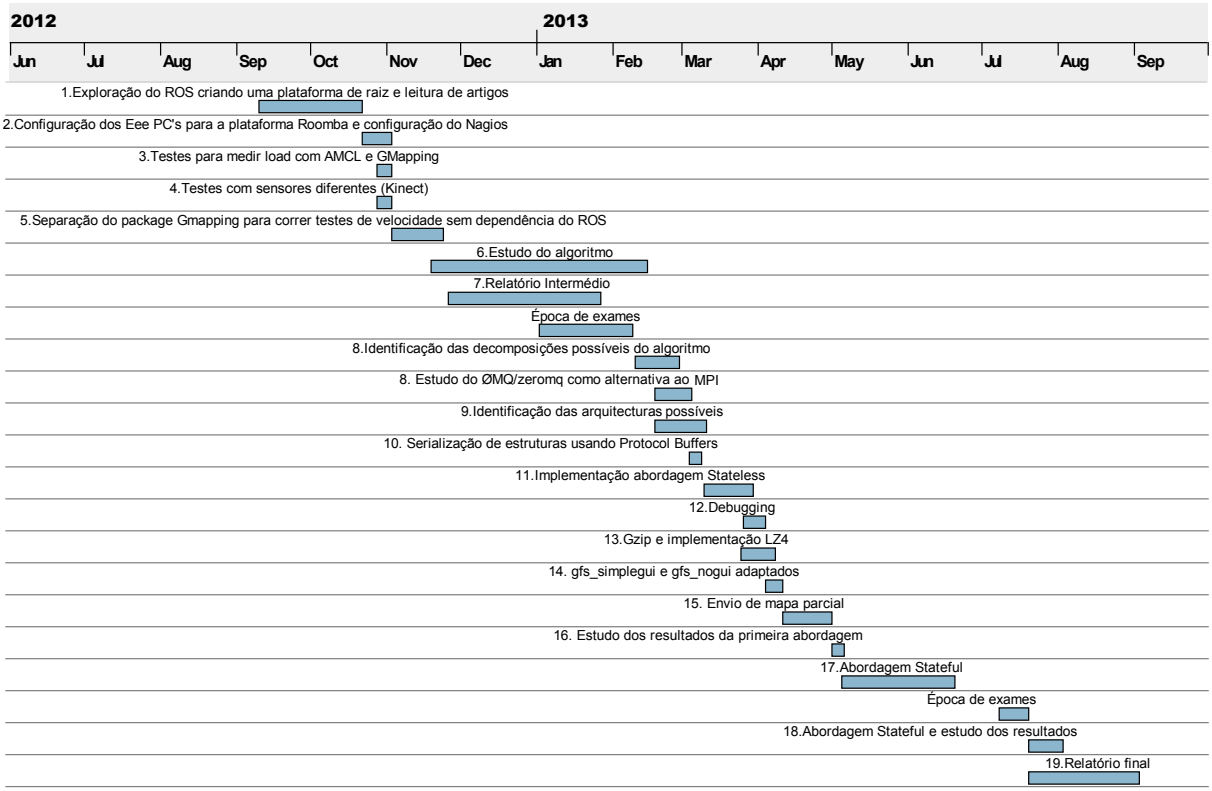

Figura 5: Diagrama de Gantt

3.1.1 Primeiro Semestre

Para obter um conhecimento aprofundado do software usado, neste caso específico o ROS, construiu-se um pequeno robot móvel, em configuração diferencial, de raiz. Este passo foi fundamental para perceber parte dos conceitos da robótica móvel, perceber como usar os sensores disponíveis, explorar o seu funcionamento e descobrir o que está disponível de base no software usado(ROS).

Fez-se um circuito impresso com um microcontrolador ARM Cortex M3 , TI Stellaris lm3s1776 [3]

[3]Mais informação disponível em http://www.ti.com/product/lm3s1776

com uma ponte H SN754410 [4] para controlar dois motores. Na placa havia também pinos e conectores para ligar um Servomotor (para controlar a direcção do sonar), dois motores de corrente contínua e dois encoders ópticos. Para comunicar com o robot usou-se uma ligação Bluetooth equipando o robot com um modulo RS-232 -> Bluetooth. O código de baixo nível (controlo dos motores e dos sensores) foi todo elaborado para esta placa, guardando o código de alto nível no PC.

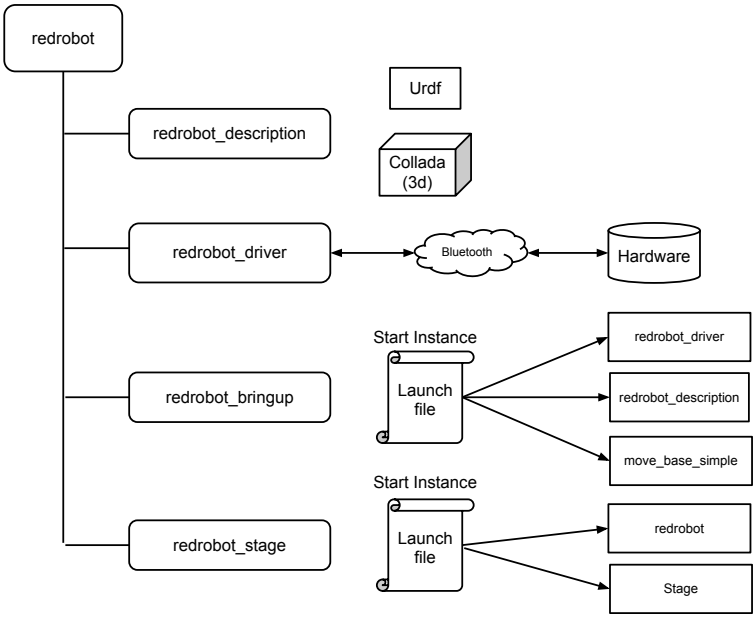

Figura 6: Estrutura da Stack "redrobot"

Juntou-se o código todo de alto nível do robot numa *stack* "redrobot" (ver Figura 6), que continha os seguintes *packages*:

- **redrobot_description:** contém a descrição do robot. Usa ficheiros *Unified Robot Description Format* (URDF) [5] para descrever os vários elementos do robot, a sua posição, opcionalmente descrevendo as suas propriedades físicas (peso, atrito,...), e como estão ligados uns aos outros, descrevendo o tipo de articulações que os ligam. Cada elemento (rodas, base, sonar e apoios) usa um ficheiro *COLLADA* [6] com o modelo 3D previamente modelado no *Blender*[7] para fins de visualização. Estes ficheiros são usados nos vários programas de visualização tal como o *Rviz* [8], ou nos simuladores tal como o *Stage* [9](ver Figura 7) e o *Gazebo* [10].

- **redrobot_driver:** contém todo o código de comunicação entre o ROS a correr no PC e o micro-controlador ARM do robot. Pede regularmente a informação sobre a odometria e os sensores (sonar) ao robot, aplica as transformações necessárias usando a *Transformation Library* (TF) [11]

[4]Mais informação disponível em http://www.ti.com/product/sn754410
[5]Mais informação disponível em http://www.ros.org/wiki/urdf
[6]Mais informação disponível em http://www.khronos.org/collada/
[7]Mais informação disponível em http://www.blender.org/
[8]Mais informação disponível em http://www.ros.org/wiki/rviz
[9]Mais informação disponível em http://playerstage.sourceforge.net/index.php?src=stage
[10]Mais informação disponível em http://gazebosim.org/
[11]Mais informação disponível em http://www.ros.org/wiki/tf

com informação da descrição do robot (redrobot_description) e publica os dados no tópicos do ROS usando os *publishers* correspondentes. Disponibiliza os tópicos necessários para controlar o robot.

- **redrobot_bringup:** contém o *launchfile* para activar o robot, inicializa um nó *redrobot_driver*, *redrobot_description* e um nó de navegação *move_base_simple* [12]
- **redrobot_stage:** contém o *launchfile* para simular o robot no software de simulação *Stage*, pode ser usado para testar algoritmos sem usar directamente o robot

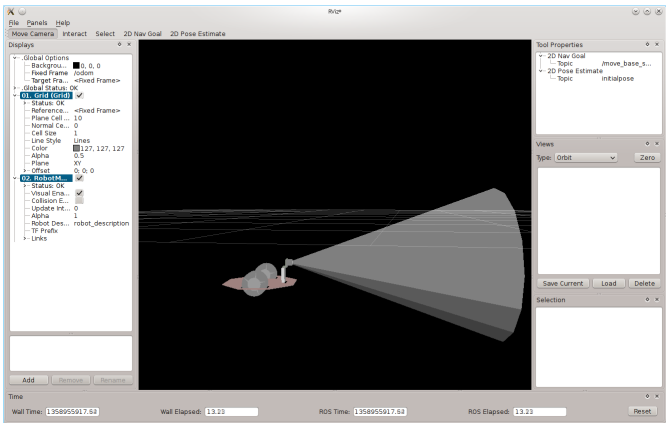

Figura 7: Visualização do robot inicial com cone do sonar no Rviz

A seguir passou-se para a plataforma que em princípio vai ser usada para o resto do trabalho, o *Roomba* da *iRobot* [24]. O roomba é um aspirador autónomo equipado com os seguintes sensores e actuadores (informação retirada de `http://www.ros.org/wiki/roomba_500_series`):

- dois *bumpers* frontais de contacto
- seis *bumpers* frontais infravermelhos
- quatro sensores infravermelhos dirigidos para o chão para detectar possíveis quedas (escadas, ...)
- receptores de dados por infravermelho
- um conjunto de botões
- um motor por roda (esquerda e direita) com Encoder
- escova principal e escova lateral
- aspirador
- um conjunto de luzes

Tem uma porta Mini-DIN (ver Figura 8) que se pode utilizar para comunicar com o sistema, ligando um adaptador RS-232. As especificações do protocolo de comunicação *Serial Command Interface* (SCI)

[12] Mais informação disponível em `http://ros.org/wiki/move_base_simple`

suportado pelo *Roomba* foram disponibilizadas pela *iRobot online* [13]. Devido a estas características e ao preço relativamente baixo [52], tornou-se numa plataforma popular para projectos amadores e de investigação [50] [30] [14].

Figura 8: Conector Mini-DIN (Retirado de http://www.instructables.com/file/FAT8NOTGONGHLX4)

O robot foi equipado numa primeira fase com um Eee PC 901 [14] e a seguir com um eeepc 1015PEM [15] da Asus (as características de cada PC são expostas na Tabela 4).

	Eee PC 901	**Eee PC 1015PEM**
CPU	Intel® Atom™ N270 com Hyper-Threading	Intel® Atom™ N550 com Hyper-Threading
Clock Speed	1.6GHz	1.5 GHz
# of Cores	1	2
Cache L2	512 KB	1 MB
RAM	1GB(DDR2)	1GB (DDR3)
Disco	20GB SSD	250GB 5,400rpm
SO	Ubuntu 12.04 32Bit	Ubuntu 12.04 64Bit
ROS	Fuerte	Fuerte

Tabela 4: Características dos Eee PC

Como sensor, além da odometria que o Roomba disponibiliza, usou-se um *Laser Range Finder* da marca Hokuyo (ver Figura 14).

Chegou-se a fazer testes usando um *Kinect* e o *package pointcloud_to_laserscan* [16], convertendo a *pointcloud* devolvida pelo Kinect para uma mensagem compatível com o *Laserscan*, passando a ser possível usar esta configuração como alternativa bem mais barata ao *Laser Range Finder*. Infelizmente notou-se logo a limitação desta solução, o ângulo da abertura do Kinect era pequeno e a distância devolvida pelo sensor não estava correcto, necessitando provavelmente de alguma calibração no software.

[13] Manual de especificações disponível em http://www.irobot.com/images/consumer/hacker/Roomba_SCI_Spec_Manual.pdf

[14] Mais informação disponível em http://www.asus.com/Eee/Eee_PC/Eee_PC_901/

[15] Mais informação disponível em http://www.asus.com/Eee/Eee_PC/Eee_PC_1015PEM/

[16] Mais informação disponível em http://www.ros.org/wiki/pointcloud_to_laserscan

Além destes problemas notou-se também que usava bastante mais os recursos disponíveis do computador, devido à conversão da *pointcloud*, chegando mesmo a usar perto de 100% do processador, correndo o *package* AMCL.

Não se analisou mais sensores, sobretudo os sensores 3D ou câmaras estereoscópicas, devido à natureza do algoritmo. Este é baseado em *occupancy grids* e funciona unicamente num plano. Para esse caso, o *Laser Range Finder* é a melhor solução como devolve dados com elevada precisão.

Efectuou-se a seguir uma análise do desempenho do algoritmo escolhido (GMapping) que podera ser vista no capítulo 3.3.1.

3.1.2 Segundo Semestre

O segundo semestre foi dividido nas seguintes tarefas:

- **8. Identificação das decomposições possíveis:** Nesta fase analisou-se a estrutura do código disponível no GMapping. Pretende-se mapear as várias etapas do algoritmo, tentando assim encontrar maneiras de o paralelizar, identificando também as arquitecturas possíveis para aplicar num *cluster* (por exemplo: um *master* e vários *workers*, pipeline, ...), tentando minimizar o tempo perdido na comunicação entre os elementos.

- **9. Estudo do ZeroMQ:** Depois de ter detectado o ZeroMQ como alternativa ao MPI, analisou-se em pormenor as várias filas/sockets disponibilizadas e leu-se o livro " Code Connected Volume 1" [21] que contém vários *Design Patterns* implementados em ZeroMQ.

- **10. Identificação das arquitecturas possíveis:** Ao mesmo tempo que se leu o livro " Code Connected Volume 1" [21] foi-se analisando os vários *Design Patterns* disponíveis e a sua aplicação ao algoritmo GMapping. Detectou-se logo dois possíveis, um *Design Pattern* com filas/*sockets* síncronas *Request-Reply* e o *Design Pattern* "Parallel Pipeline" que usa filas/*sockets* assíncronas "Push-Pull" para distribuir o trabalho pelos vários *Remote Workers*. Depois de vários testes descartou-se o "Parallel Pipeline" porque só fazia sentido num ambiente distribuído onde todos os elementos fossem iguais.

- **11. Serialização de estruturas usando Protocol Buffers:** Nesta fase detectaram-se as estruturas e objectos(Classes) necessárias para implementar o algoritmo num ambiente distribuído. Definiu-se o ficheiro ".proto" do *Google Protocol Buffers* com as várias estruturas e implementou-se uma classe de apoio "ProtobufHelper" externa ao código do GMapping com métodos para converter as várias estruturas (declarando-a como *Friendly Class* no código do GMapping).

- **12-13 Implementação da abordagem "Stateless" e Debugging:** A seguir ao estudo e depois de ter o código pronto para enviar estruturas pela rede, implementou-se a arquitectura "Stateless" descrita no capítulo 3.4.1, usando filas síncronas "Request-Reply". Sempre que necessário usaram-se ferramentas de *debugging* tal como o *Massif* e *Memcheck* incluídas no *Valgrind* para encontrar *memory leaks* e sítios no código que possam ser optimizados.

- **14. Gzip e implementação LZ4:** Desde cedo se notou que o tamanho dos mapas cresciam rapidamente, nesta fase usou-se o suporte de compressão de mensagens por parte da biblioteca *Protocol Buffers* e a seguir acrescentou-se suporte para o algoritmo *LZ4* visto que mostrou em vários *benchmarks* ser mais rápido.

- **15. gfs_simplegui e gfs_nogui adaptados:** Para poder usar *datasets* disponíveis na Internet de mapas maiores (Killian Court do MIT, Intel Research Center...) adaptaram-se as duas ferramentas disponíveis "gfs_nogui" e "gfs_simplgui"(ver Figura 9) do OpenSLAM que aceitam ficheiros *CARMEN* (formato usado nos *datasets* disponíveis).

- **16. Envio de mapa parcial:** Ao correr *benchmarks* com os *datasets* maiores, notou-se uma perda de velocidade quando se utilizava "Remote Workers", isto deveu-se ao tempo de serialização e envio dos mapas pela rede. Neste fase implementou-se código para enviar unicamente mapas parciais com base na pose do robot e no alcance máximo do *Laser Range Finder*.
- **17. Estudo dos resultados da primeira abordagem:** Ao longo da implementação da primeira abordagem foi-se sempre correndo *benchmarks* com o *dataset* da Willow Garage, este mostrou sempre que o algoritmo distribuído corria mais rapidamente que o da versão clássica. Nesta fase usando os *datasets* maiores notou-se que o tempo de serialização e de envio das partículas pela rede era superior ao tempo ganho por distribuir o processamento.
- **18. Abordagem Stateful:** Depois dos resultados obtidos em 17 decidiu-se implementar uma abordagem que limitasse tanto quanto possível a troca de dados. Implementou-se portanto a arquitectura descrita no capítulo 3.4.2.
- **19. Abordagem Stateful e estudo dos resultados:** Depois de uma pausa devido à época de Exames Especiais, continuou-se a Arquitectura "Stateful" e estudaram-se os resultados obtidos.

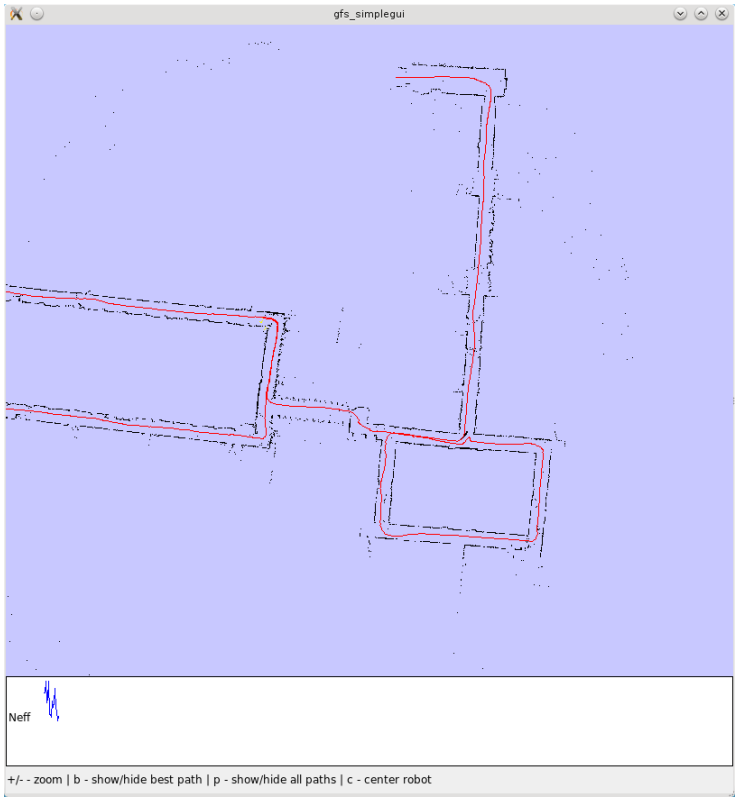

Figura 9: Ferramenta "gfs_simplgui" a correr o *dataset* mit-killiancourt.clf

3.2 Ferramentas usadas

Para desenvolver este projecto foram necessários três tipos de ferramentas, uma plataforma/*middleware* para toda a parte de robótica móvel, uma biblioteca/tecnologia para *Clustering* e finalmente uma biblioteca para serializar todo o tipo de estruturas de dados e/ou objectos necessários na comunicação entre os vários computadores.

3.2.1 ROS

Como plataforma de software principal para esta trabalho vai ser usado o ROS [45], que se tornou na plataforma *standard* na área de robótica, é *Open Souce* (BSD Licence) e é neste momento usada para maior parte dos trabalhos do LSE.

A Willow Garage identifica o ROS como sendo um *Meta-operating system*[48]. Disponibiliza todos os serviços e ferramentas que se espera de um sistema operativo completo, como por exemplo: abstracção de hardware, controlo de dispositivos de baixo nível , mecanismos de comunicação entre processos e gestão de pacotes/programas.

Corre sobre um verdadeiro sistema operativo, tendo suporte oficial para a distribuição Linux Ubuntu, com suporte experimental para outras distribuições Linux, Mac OSX e Windows.

O ROS tem como base uma arquitectura em forma de grafo(ver Figura 10), pois um sistema que use ROS usa vários processos(nós), que podem até correr em computadores diferentes. Usa o padrão *publisher-subscriber*, cada nó pode subscrever ou publicar informação(dados de sensores, comandos de velocidade,...) num tópico ou num serviço, o que torna o sistema modular, *loosely coupled* e desta maneira torna o reutilização de código existente fácil [23].

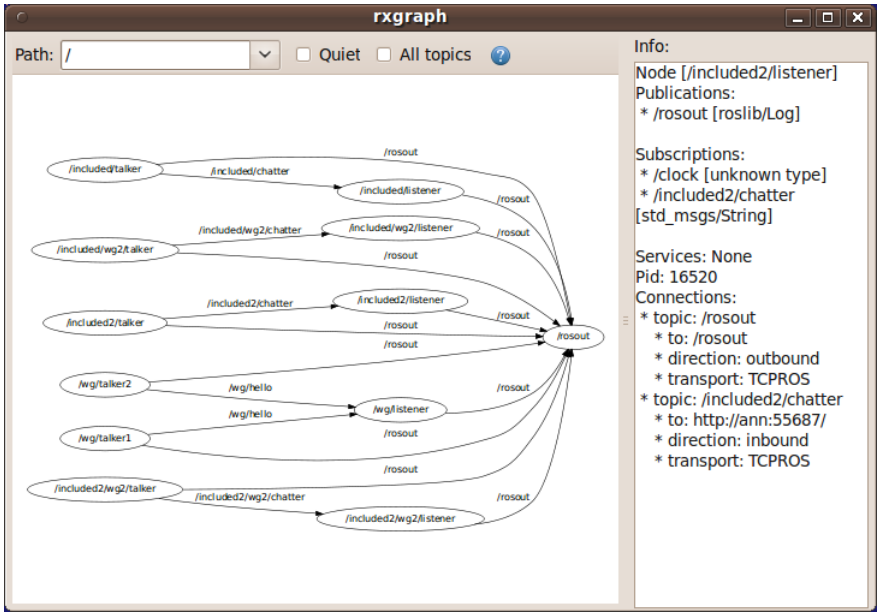

Figura 10: Arquitectura em grafo do ROS

Para a comunicação, o ROS usa um sistema *Peer-to-Peer* híbrido sobre o protocolo XML-RPC, usa um nó *Master* que funciona de maneira semelhante a um servidor *Domain Name System* (DNS). Esse nó mantém uma lista do registo de todos os elementos do sistema, sempre que necessário um nó pode pedir esta informação ao *Master* para poder efectuar uma ligação com outros nós. A partir daí a comunicação

entre nós é completamente *Peer-to-Peer*, havendo unicamente necessidade de comunicação em caso de alteração da estrutura do sistema.

Existe *Application Programming Interface* (API) em várias linguagens de programação tal como C/C++, Python e Java. Todo o código é livre e *Opensource* usando a licença BSD, e está disponível nos repositórios *Online*.

Uma grande vantagem desta plataforma é que tem uma comunidade bastante activa, com ferramentas praticas já disponíveis *online*, tal como o *site* `http://answers.ros.org/`. A Willow Garage tem conhecimento de pelo menos 90 robots a usar ROS, 28 deles com instruções disponíveis na *Wiki*. Em Abril de 2012 havia 3699 *packages* repositórios de código públicos e pelo menos uma pessoa activa(*developer*) em cada continente. Existem 439 referências do artigo "ROS: an open-source Robot Operating System" [45] no Google Scholar.

Como linguagem de programação escolheu-se o C/C++, devido ao código já existente, ao melhor desempenho geral comparativamente as outras linguagens [4] [22] [1] disponíveis, e pelo facto de se poder usar código sem modificações em várias arquitecturas (x86, ARM, ...).

3.2.2 *Clustering*

Como ferramenta/*middleware* para *clustering* existem várias alternativas. Devido a escolha de C/C++ como linguagem de programação do trabalho, descartou-se logo todos os *middlewares* das outras linguagens. O *middleware* escolhido tinha, preferencialmente, de ter mecanismos de tolerância de falhas de ligação, suporte para modificações dinâmicas na topologia do *cluster* ou então a flexibilidade necessária para poder responder a estes casos.
Como *middlewares* possíveis, comparou-se as implementações do MPI e *ZeroMQ*:

3.2.2.1 MPI

O MPI é um *standard portable* para um sistema de comunicação orientada para mensagens, foi desenvolvido por um grupo de investigadores do meio académico e industrial para funcionar num vasto leque de computadores paralelos. A estandardização envolveu mais de 80 pessoas pertencentes a 40 organizações [49], maioritariamente dos Estados Unidos e europeias.

Existem várias implementações do standard, sendo o *Open MPI* [38] e o *MPICH* [39] as mais usadas e em continuo desenvolvimento. Existem umas quantas implementações que introduzem mecanismos de tolerância de falhas (por exemplo: MPICH-V [40] e P2P-MPI [3]), infelizmente já não são desenvolvidas e implementam uma versão mais antiga do MPI. O fórum do *MPI* começou a introduzir *features* para esse caso na nova versão do MPI [37].

3.2.2.2 ZeroMQ

O ZeroMQ é uma biblioteca que tem como base a comunicação orientada para mensagens. Disponibiliza vários tipos de *sockets* com filas (*publisher-subcribe*, *request-reply*, *push-pull*, ...) que suportam vários tipos de transporte (*in-process*, *inter-process*, TCP, *multicast* e *Pragmatic General Multicast* (PGM). Existem *bindings* para mais de 30 linguagens entre as quais o C/C++, Java e Python. O "Zero" de ZeroMQ vem do facto de não ser preciso usar um "Broker". A API destas *sockets* é parecida com as *sockets* BSD, não define as estruturas a serem enviadas, portanto o ZeroMQ trabalha com *buffers* de dados. Ao contrario da maior parte das bibliotecas de *Message-Queues*, não tenta ser compatível com o *Advanced Message Queuing Protocol* (AMQP), tenta ser o mais *lightweight* possível, não implementando *features* habituais nas outras bibliotecas, como por exemplo filas com persistência. De origem suporta vários mecanismos interessantes para uso em cenários com falhas na rede de comunicação:

- o cliente pode ligar a *socket* antes do servidor estar disponível, havendo depois uma ligação automática quando este estiver disponível

- uma *socket* que perde ligação volta-se a ligar também automaticamente quando houver rede disponível
- existem vários *patterns* de comunicação disponíveis no livro "Code Connected Volume 1" [21] com mecanismos de tolerância de falhas e de *load-balancing*

O ZeroMQ disponibiliza as seguintes *sockets* organizadas em pares:

- **ZMQ_PUSH-ZMQ_PULL**(*Design Pattern Pipeline*): *Sockets* unidireccionais assíncronas clássicas com interface parecida aos BSD *sockets*, podem-se usar para fazer ligações 1 para 1, N para 1, 1 para N e N para N. No envio, a *socket* "Push" usa *Round-robin* para distribuir os dados e a *socket* "PULL" usa *Fair-Queueing* para receber os dados.
- **ZMQ_REQ-ZMQ_REP**(*Design Pattern Request-reply*): *Sockets* síncronas, ao contrário das "PUSH-PULL" são bidireccionais, permitem o mesmo tempo de ligações e usam as mesmas estratégias para receber e enviar dados. Estas *sockets* têm uma particularidade interessante, quando o a *socket* "REP" recebe uma mensagem pode unicamente responder à *socket* "REQ" que enviou os dados. Este par de *sockets* tem sempre que seguir este padrão de comunicação, não podendo, por exemplo, a *socket* "REQ" enviar duas mensagens seguidas sem receber a resposta primeiro.
- **ZMQ_PUB-ZMQ_SUB** (*Design Pattern Publisher-subscribe*): *Sockets* unidireccionais assíncronas usadas para distribuir a mesma mensagem por vários receptores, fazem portanto ligações 1 para N. A *socket* "Subscribe" permite filtrar os dados recebidos, podendo-se criar vários "canais" desta maneira.

Existem outros tipos de *sockets* mais avançadas disponibilizados pelo ZeroMQ tal como o "ZMQ_ROUTER" e "ZMQ_DEALER", não foram explicadas aqui por serem sobretudo usadas para criar "Brokers" ou "Design Patterns" que não são interessantes para este trabalho.

3.2.2.3 Tabela comparativa

A Tabela 5 contém uma comparação do MPI com o ZeroMQ , compara as características que se achou mais importantes para este trabalho: o desempenho de cada, a usabilidade (se é simples de utilizar e se precisa de *setup* complexo), se tem mecanismos ou flexibilidade para suportar falhas e finalmente se possibilita usar topologias dinâmicas.

	performance	**user-friendly**	**fault-tolerant**	**dynamic topology**
MPI	✓	✗	✗[a]	✓[b]
ZeroMQ	✓	✓	✓	✓

[a] Existem várias implementações descontinuadas
[b] MPI spawn para introduzir novos nós noutro "world" mas que podem comunicar com o existente

Tabela 5: Comparação de Middleware para Clustering

3.2.2.4 Escolha final

Escolheu-se *ZeroMQ* para realizar o trabalho. Devido à incerteza que existe num ambiente desconhecido e à natureza do tipo de comunicação usada, achou-se importante ser possível implementar algum mecanismo de tolerância de falhas. Essa flexibilidade existe no *ZeroMQ*. No *MPI* só se começou a definir

mecanismos para estes casos a partir da versão 3 [37]. O *ZeroMQ* também tem a vantagem de funcionar sem mecanismo centralizado (comi por exemplo um *broker*) e, devido as *sockets* diversas disponíveis, suporta mais *patterns* de de comunicação. Suporta também vários protocolos de transporte, o que em certos casos (por exemplo: no caso do *pattern publisher-subscriber* usando UDP *multicast*) torna a comunicação mais rápida e o código reutilizável (uso de *sockets In-Process (inter-thread) Communication Transport* (INPROC) e TCP para comunicar respectivamente com *threads*/processos que estão a correr no mesmo computador e em computadores na mesma rede).

3.2.3 *Object Serialization*

Para facilitar na troca de informação entre os vários computadores do "cluster", foi necessário o uso de uma biblioteca/mecanismo de serialização dos dados. Mais uma vez existiam várias alternativas validas. Neste caso era necessário novamente que a biblioteca suportasse as várias arquitecturas de processador disponíveis e se possível ser compatível com outras (*endian-agnostic*), ser preferencialmente fácil de usar e sobretudo veloz. Descartou-se logo de início todos os formatos baseados em texto porque o tratamento destes costuma ser mais lento que em formatos binários.

Analisou-se as 3 seguintes bibliotecas, *Google protocol buffers* , *Boost Serialization* e *MessagePack*:

3.2.3.1 Protocol buffers

A *Protocol Buffers* é uma biblioteca de código aberto de serialização desenvolvida pela Google. É usada internamente por eles para todos os processos de serialização nas comunicações entre computadores e também para arquivar estruturas e objectos para disco. Usa ficheiros ".proto" que definem as estruturas das mensagens de maneira semelhante ao XML e JSON. De seguida usa-se o "compilador " *protoc* que gera os ficheiros de código para a linguagem de programação escolhida, estes ficheiros definem todas as estruturas e os métodos necessários para gerar os objectos serializados. A biblioteca é *endian-agnostic*, integra-se facilmente com o C++ suportando vários mecanismos da *Standard Template Library* (STL) como *streams* e serialização para *strings*. Permite comprimir as estruturas a medida que são criadas o que reduz significativamente o tamanho destas no caso por exemplo de mapas esparsos.

3.2.3.2 Boost Serialization

A biblioteca *Boost Serialization* faz parte dos algoritmos, estruturas e ferramentas disponíveis no conjunto de bibliotecas *Boost*. São bibliotecas C++ de código aberto (usam maioritariamente a licença Boost Software Licence), suportam a maior parte das arquitecturas e sistemas operativos, são *peer-reviewed* e escritas de tal maneira a encaixar bem com o código já existente da *C++ Standard Library*.

A *Boost Serialization* suporta 2 modos para serializar objectos/estruturas, um modo intrusivo em que se modifica as classes já existentes e outro não intrusivo que define classes que recebem as existentes como parâmetros. Tem suporte de origem para certas estruturas da STL, tais como o "vector" , "map" e "set", e fornece mecanismos para exportar os dados para um formato binário, texto e XML.

O formato binário não é *endian-agnostic* sem uso de código adicional.

3.2.3.3 MessagePack

Tal como a *Google Protocol Buffers* a *MessagePack* é uma biblioteca de serialização de objectos com *bindings* disponíveis para várias linguagens de programação. Usa os mesmos tipos de dados que o *JavaScript Object Notation* (JSON) (integer, nil, boolean...) e tem suporte para as estruturas *array* e *map*. Não necessita de ficheiro ".proto" como no *Google Protocol Buffers* e também é *endian-agnostic*.

3.2.3.4 Tabela comparativa

Na tabela seguinte comparam-se as características principais das várias bibliotecas de serialização: velocidade, se é fácil de utilizar (se necessita por exemplo de código complexo extra), se é "Endian Agnostic" (se sem modificar o código, se pode usar entre computadores *Little-Endian* e *Big-Endian*), se é dependente de uma linguagem de programação ou se se pode trocar mensagens entre dois programas escritos em linguagens de programação diferentes e, finalmente, se a biblioteca contém alguma funcionalidade extra (por exemplo compressão de mensagens do Protocol Buffers).

	performance	**user-friendly**	**endian agnostic**	**language agnostic**	**extras**
Protocol buffers	✓	✓	✓	✓	✓
Boost Serialization	✓	✓	✗[a]	✗	✗
MessagePack	✓	✓	✓	✓	✗

[a] Sem código extra para o formato binário, disponível para XML

Tabela 6: Comparação de bibliotecas de Serialização de dados

3.2.3.5 Escolha final

Escolheu-se *Protocol Buffers* para realizar o trabalho. A biblioteca desenvolvida pela Google é a que tem mais *features* e é das que têm a melhor performance [2]. O uso opcional de compressão podia vir a ser interessante e em conjunto com o *ZeroMQ*, que também tem *bindings* para muitas linguagens de programação, poder-se-ia usar mais tarde um sistema heterogéneo, com, por exemplo, os *Workers* escritos em Java. É uma biblioteca robusta, bastante utilizada em sistemas distribuídos e para fins de arquivos pela Google e sendo *endian-agnostic* pode ser futuramente usada noutras arquitecturas sem modificar o código.

3.3 GMapping

O GMapping é uma versão em grelha do algoritmo FastSLAM, implementa um filtro de partículas Rao BlackWellized em que cada partícula mantém o seu próprio mapa e a pose do robot. Como estrutura principal, usa uma árvore em que cada nó representa uma parte da trajectória do robot. Cada nó mantém a pose do robot, um ponteiro para a estrutura do *LaserScan* efectuado nesse momento, os pesos calculados a partir do *LaserScanMatch* (operação de optimização que tenta estimar o valor da rotação e translação entre 2 vectores da pose do robot usando as características do mapa) e um ponteiro para o nó anterior. A árvore tem portanto o número de folhas igual ao número de partículas no algoritmo, são guardadas numa estrutura "Vector' para serem mais facilmente processadas. Para obter um mapa actual a partir da partícula com melhor peso, basta atravessar a árvore da folha (partícula escolhida) até à raiz e em cada passo marcar o mapa com o dados do *LaserScan* ("RegisterScan"), marcando as células do mapa como vazio, ocupado ou desconhecido).

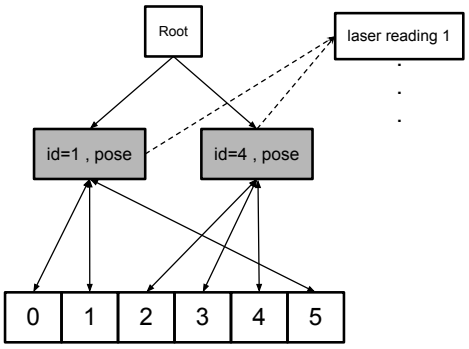

Figura 11: Estrutura em arvore (simplificada) com as trajectória possiveis

Segue a sequência de execução da Figura 13, ao receber os dados actuais da odometria e do *Laser Range Finder* executa os seguintes passos:

- Se forem os primeiros dados recebidos, regista os dados do Laser em cada partícula no mapa directamente (marca as células como vazio, ocupado ou desconhecido) e a seguir corre a função "Resample".
- Caso contrário actualiza a pose do robot em cada partícula com base no modelo de movimento, usando a odometria.
- Se a distância percorrida for maior que um limite (parâmetro do algoritmo), efectua um *LaserScanMatch* usando os dados dos sensores e corrige a pose de acordo ao mapa de cada partícula.
- Actualiza os pesos da árvore e calcula o parâmetro N_eff

$$N_eff = \frac{1}{\sum_{i=1}^{N}(\widetilde{w}^i)^2)} \tag{1}$$

  em que $\widetilde{w}^i$ é o peso normalizado da partícula i

- Corre a função "Resample"

Se o parâmetro N_eff for inferior a um limit,e a função "Resample" efectua *resampling* (ver linha 9 do Algoritmo 2), escolhendo aleatoriamente (com reposição) as partículas com base nos pesos associados, dando prioridade às que têm peso maior, podendo no passo seguinte haver várias cópias da mesma partícula(ver Figura 12).

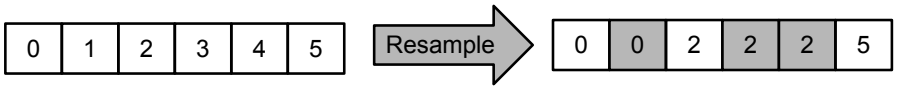

Figura 12: Resampling: algoritmo escolhe múltiplas vezes a partícula 0 e 2 , descartando a 3 e 4

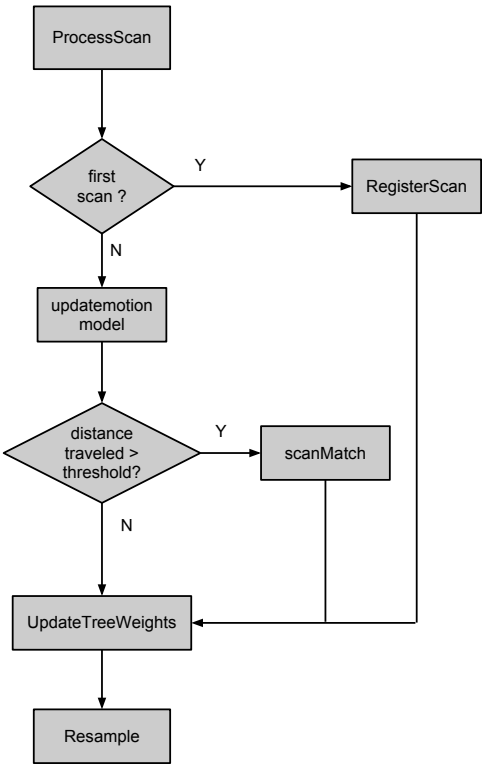

Figura 13: Sequência de execução do algoritmo

Ao analisar o algoritmo, ficou rapidamente claro que as funções "registerScan", "updatemotionModel" e "scanMatch" poderiam ser executadas de maneira paralela, havendo unicamente necessidade de sincronizar os resultados antes de chamar a função "updateTreeWeights" e de seguida a função "Resample". Estas duas últimas funções têm que ser executadas de maneira sequencial, pois alteram a árvore das trajectórias (no caso de "UpdateTreeWeight") e modificam/apagam as partículas actuais (no caso da função "Resample").

3.3.1 Análise do desempenho

Para determinar o desempenho do algoritmo GMapping instalou-se nos Eee PC's o software de monitorização de sistema *Nagios* [17] com o seu *add-on* PNP4Nagios [18] que analisa os dados de performance e constrói gráficos. Efectuou-se dois testes na arena do LSE (ver Figura 15):

- Correr o *package* AMCL [19] que usa o mapa conhecido
- Correr o *package* de SLAM GMapping [20],

[17] Mais informação disponível em http://www.nagios.org/
[18] Mais informação disponível em http://docs.pnp4nagios.org/start
[19] Mais informação disponível em http://www.ros.org/wiki/amcl
[20] Mais informação disponível em http://www.ros.org/wiki/gmapping

Os dois testes foram corridos com os parâmetros predefinidos (ver [21] e [22]), e, devido a granularidade máxima de um minuto do *Nagios* manteve-se o robot em movimento o tempo suficiente para obter gráficos no *PNP4Nagios*.

Obteve-se os resultados que podem ser vistos na Tabela 7. Pode-se ver que um Eee PC 901 não é rápido suficiente para correr o algoritmo GMapping, mas que ainda sobram recursos ao correr *AMCL* com um mapa conhecido. Do teste corrido no Eee PC 1015PEM pode-se ver que o algoritmo GMapping não está optimizado para correr em processadores *multi-core*, pois usa sobretudo os dois primeiros *cores* virtuais, notando-se sobretudo no tempo que demorava a responder a comandos de movimento e pelo facto de aparecer a seguinte mensagem no *log* do GMapping :

```
[ WARN] [1358699154.549558387]: Map update loop missed its desired
rate of 3.0000Hz...  the loop actually took 4.8561 seconds
```

	Eee PC 901 AMCL	**Eee PC 901 GMapping**	**Eee PC 1015PEM GMapping**
CPU Load Médio	40%	100%	40%
CPU Load Máximo	50%	100%	60%
Notas	Robot responsivo, responde directamente aos comandos	Robot demora perto de 40s a responder a comandos	Robot demora perto de 30s a responder ao comando, cpu load do core virtual 0 e 1 a 80% core 2 e 3 pouco usados (<40%)

Tabela 7: Resultados dos testes

Para identificar os *hotspots* do algoritmo *GMapping* fez-se um *trace* ao *package* usando o *Callgrind* da ferramenta *Valgrind* [23]. Para ser mais cómodo usou-se modo de simulação do ROS e usou-se um ficheiro *Bag* [24] disponibilizado pela *Willow Garage* [25]. Obteve-se o *callgraph* disponível na Figura 16.

Numa análise rápida pode-se ver que o algoritmo passa a maior parte do tempo nas funções *score* 52.10% (o resto das funções fazem parte de estruturas) e *updateMap* 33.76%, sendo estas candidatas a serem paralelizadas.

[21] Ficheiro de parâmetros disponível em http://isr-uc-ros-pkg.googlecode.com/svn/stacks/lse_roomba_toolbox/trunk/lse_roomba_2dnav/params/amcl_roomba.launch

[22] http://www.ros.org/wiki/gmapping?distro=fuerte#Parameters

[23] Mais informação disponível em http://valgrind.org/

[24] Mais informação disponível em http://www.ros.org/wiki/Bags

[25] Ficheiro disponível em http://pr.willowgarage.com/data/gmapping/hallway_slow_2011-03-04-21-41-33.bag

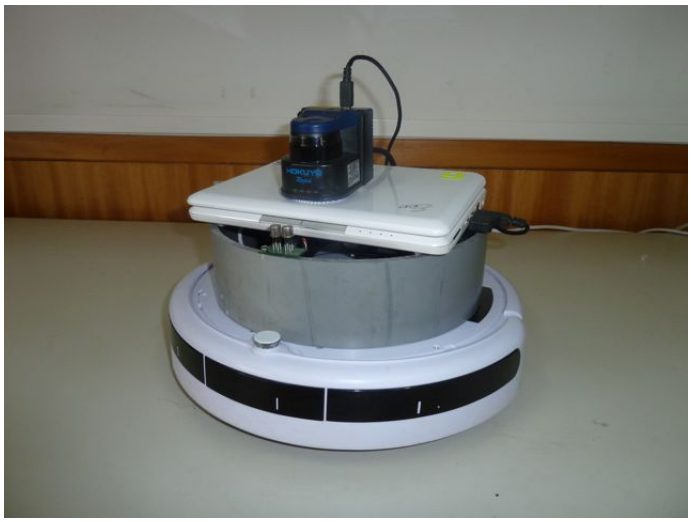

Figura 14: Roomba com Eee PC, Hokuyo e sensor de odor

Figura 15: Arena do LSE com separadores

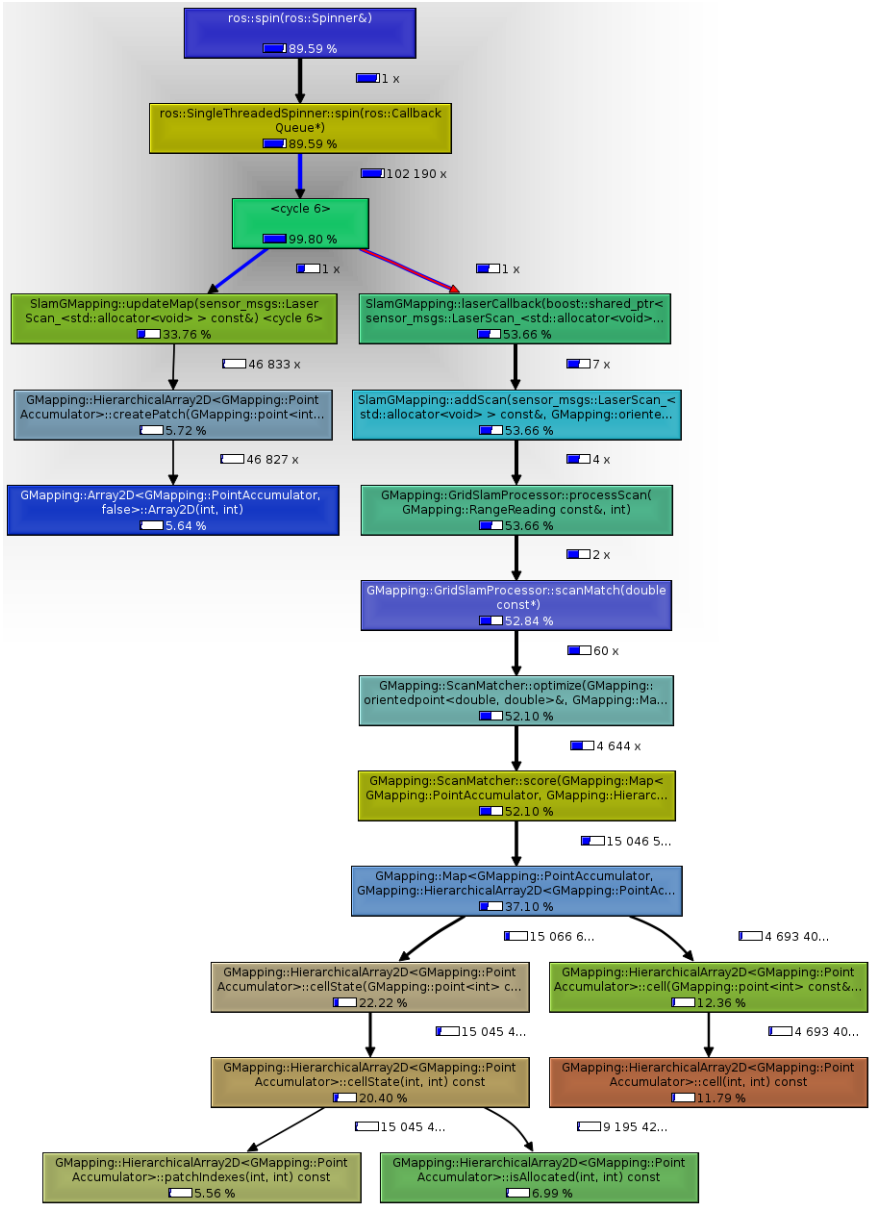

Figura 16: *Callgraph* do nó slam_gmapping

Para obter dados objectivos e quantificáveis sobre a velocidade do algoritmo, foi efectuada uma modificação do *package* GMapping para correr independentemente do ROS, o mais rapidamente possível, usando os dados anteriormente registados. Demorou 11m53.951s a correr no Eee PC 1015PEM (compilado em modo *debug*). Poderá depois ser usado para comparar a velocidade dos vários algoritmos paralelos futuramente desenvolvidos, sem ter de recorrer a *profilers* ou software de monitorização.

3.4 Implementação

O facto de a parte computacionalmente intensiva do algoritmo *GMapping* ser embaraçosamente paralela, levou a poder-se escolher arquitecturas mais simples de *clustering*.

3.4.1 Arquitectura *"stateless"*

Devido à incerteza dos ambientes desconhecidos a explorar, podem existir falhas na comunicação entre os robots e condições diferentes da largura de banda disponível na rede *Wireless*. O ideal é de ter robustez, suporte de modificações na topologia do sistema e alguma forma de *loadbalancing* numa aplicação distribuída. Para obter essas características, tentou-se inicialmente implementar uma abordagem "stateless"(ver Figura 17) em que os nós de computação externos ("Remote Workers") não tentam manter um estado em conjunto com o sistema principal.

A arquitectura usa uma fila ("Workqueue") síncrona (*Request-Reply* do *ZeroMQ*) de transporte INPROC, em que cada elemento na fila contém o índice duma partícula que necessita de *Laserscanmatch*. Existe uma *thread* "Requester" por "Remote Worker" externo com acesso a esta fila, podendo também existir desta maneira *threads* locais de computação ("Local Workers") com acesso à mesma. Cada *thread* "Requester" constrói a mensagem necessária para enviar de seguida para o "Remote Worker", esperando pela resposta e alterando as estruturas locais com os dados recebidos. A comunicação entre a *thread* "Requester" e o seu "Remote Worker" é feita também através de uma fila síncrona *Request-Reply*, desta vez usando o protocolo de transporte TCP. Este tipo de fila traz também uma característica interessante: utilizando unicamente uma porta, responde sempre ao cliente que lhe enviou um *request*. Um "Remote Worker" pode ser portanto usado para processar pedidos de vários "clientes" e pode-se facilmente modificar dinamicamente o número destes no sistema. As *threads* locais "LocalWorker" acedem directamente à memória partilhada do sistema principal, correndo uma versão local do algoritmo de *Laserscanmatch*, e não necessitam portanto de serializar estruturas.

Usando este mecanismo obtém-se *loadbalancing* "gratuito", as *threads* que conseguirem processar os dados mais rapidamente vão também buscar mais vezes dados a "Workqueue". Este processo também inclui o tempo de transferência dos dados na rede, no caso de "Remote Workers". Para responder a falhas na comunicação basta simplesmente reenviar o índice para a "Workqueue" a partir de uma das *threads* "Requester", se for detectado um *timeout* no envio ou na espera da resposta. Essa partícula passa então a ser processada por um "Worker" que esteja disponível.

Para diminuir os dados transmitidos em rede decidiu-se usar compressão na mensagem enviada para os "Remote Workers", usar mapas parciais contendo unicamente a zona onde vai existir modificações e, na resposta, enviar unicamente os dados necessários: a pose, a área do mapa modificada e se houve aumento do tamanho do mapa. Para a compressão das mensagens foi usado o algoritmo LZ4 [29] visto que mostrou em *benchmarks* ser dos mais rápidos [29][16], mantendo um rácio aceitável para esta aplicação, sendo bastante melhor que o *GZIP*, já disponível no *Google Protocol Buffers*.

Para troca de informação entre os computadores utilizaram-se as seguintes mensagens:

- **WorkPackage**: Mensagem com os dados da partícula, os parâmetros usados no "LaserScanMatcher" e os dados actuais dos sensores (odometria e *Laser Range Finder*). A partícula contém o mapa e a pose actual. No capítulo 4.1 poderão ser vistos diagramas da evolução do tamanho desta mensagem (ver Figura 28 e Figura 27).
- **WorkResponse**: Mensagem de resposta, contém o índice da partícula processada, a nova pose e o peso novo obtidos pelo "LaserScanMatcher", a área que se modificou (para o registo no mapa) e opcionalmente o tamanho novo do mapa, se este cresceu. Comparativamente, a mensagem "WorkPackage" tem um tamanho pequeno (três inteiros de 32 bit, cinco a nove *doubles*, no caso de o mapa crescer).

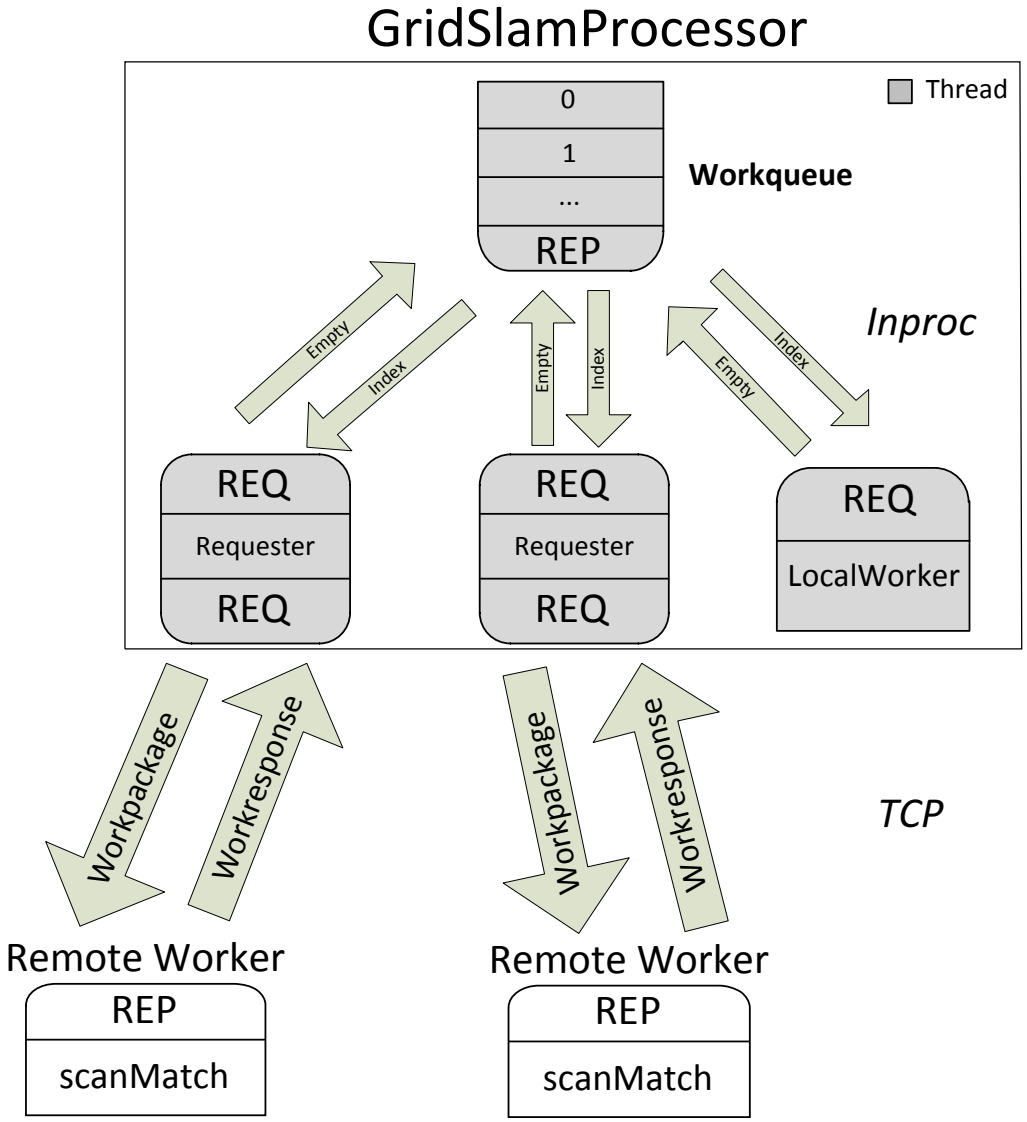

Figura 17: Arquitectura "stateless"

3.4.2 Arquitectura *"stateful"*

Devido aos resultados obtidos na arquitectura "stateless", resolveu-se implementar outra que reduzisse tanto quanto possível a troca de dados. A única possibilidade era implementar uma arquitectura que mantivesse o estado("stateful") entre os vários nós e o sistema principal.

Na nova arquitectura distribui-se as partículas do algoritmo GMapping pelos vários *Remote Workers* (ver Figura 18). Mantém-se também sempre uma cópia local sincronizada, o que permite tolerar falhas na rede. Pode-se processar localmente os dados e depois reenviar-los quando o *Remote Workers* estiver novamente disponível. Simplifica também o algoritmo em caso de *Resampling*, o sistema principal contém todas as partículas, mantém sempre a árvore das trajectórias (ver Figura 11) e pode enviar uma mensagem com todas as que faltam a um *Remote Worker*, depois de haver *Resampling*, não necessitando haver troca de informação entre *Remote Workers*.

Em vez de enviar as partículas e os dados necessários dos sensores, como na abordagem anterior, passa-se a usar uma fila/*socket PUB-SUB* que faz *broadcast* dos dados dos sensores para os vários *Workers*(locais e remotos). Esta fila usa dois tipos de transportes, INPROC para comunicar com as *threads LocalWorker* internas ao sistema principal (que tal como na abordagem anterior podem correr uma versão local do algoritmo) e TCP para comunicar com os *Remote Workers* (podendo ser trocado para PGM por razões de performance). Para manter as partículas sincronizadas, os *Remote Workers* continuam a enviar o resultado do *Laserscanmatch* para o sistema principal, desta vez usam uma fila do tipo *PUSH-PULL*, comunicam com a *thread ReplyThread* e esta modifica as partículas com os resultados obtidos.

O algoritmo segue a sequência de execução da Figura 19. É semelhante ao algoritmo tradicional (ver Figura 13), com a diferença de que várias etapas passaram a ser feitas remotamente, e ao contrário da abordagem "Stateless" passou a haver necessidade de sincronizar o algoritmo em vários momentos. Na Figura 20 pode-se ver a sequência de execução da função "Resample", no caso de haver necessidade de *Resampling* há uma troca de mensagens entre o computador local e os *Remote Workers*, para sincronizar os índices das partículas.

Em comparação com a outra abordagem perde-se infelizmente o *loadbalancing*, o utilizador tem de conhecer os computadores que vai usar e definir como vai distribuir as partículas. Uma solução possível a este problema seria que o algoritmo adaptasse o número de partículas depois de processar, por exemplo, um número específico de dados dos sensores ou então aproveitar momentos quando há *Resampling* para as redistribuir, envolvendo neste caso uma perda possível de tempo devido a ter que serializar um número maior. Outra característica que se perde é a de poder reutilizar os *Remote Workers* para vários sistemas.

Para troca de informação entre os computadores utilizaram-se as seguintes mensagens:

- **StartPackage**: Mensagem inicial enviada para os *Remote Workers*, contém os parâmetros todos do "LaserScanMatcher" e do modelo de movimentos, é enviada unicamente no início do programa.
- **Sensordata**: Mensagem com os dados dos sensores enviada a cada iteração do algoritmo. Contém os dados da odometria e opcionalmente os dados do *Laser Range Finder*.
- **WorkResponse**: Mensagem de resposta, contém o índice da partícula processada, a nova pose e o peso novo obtidos pelo "LaserScanMatcher", a área que se modificou (para o registo no mapa) e opcionalmente o tamanho novo do mapa, se este cresceu. Comparativamente, a mensagem "WorkPackage" tem um tamanho pequeno (três inteiros de 32 bit, cinco a nove *doubles*, no caso de o mapa crescer).
- **IndexMessage**: Mensagem usada no caso de *Resampling*. Contém os índices de partículas válidas e é enviada a todos os *Remote Workers*.
- **ResampleMessage**: Mensagem usada pelos *Remote Workers* na resposta à mensagem "IndexMessage", contém os índices das partículas que faltam localmente nos *Remote Workers*.
- **Particles**: Mensagem com os dados da partícula (mapa, pose e o peso) usada em caso de *Resampling*.

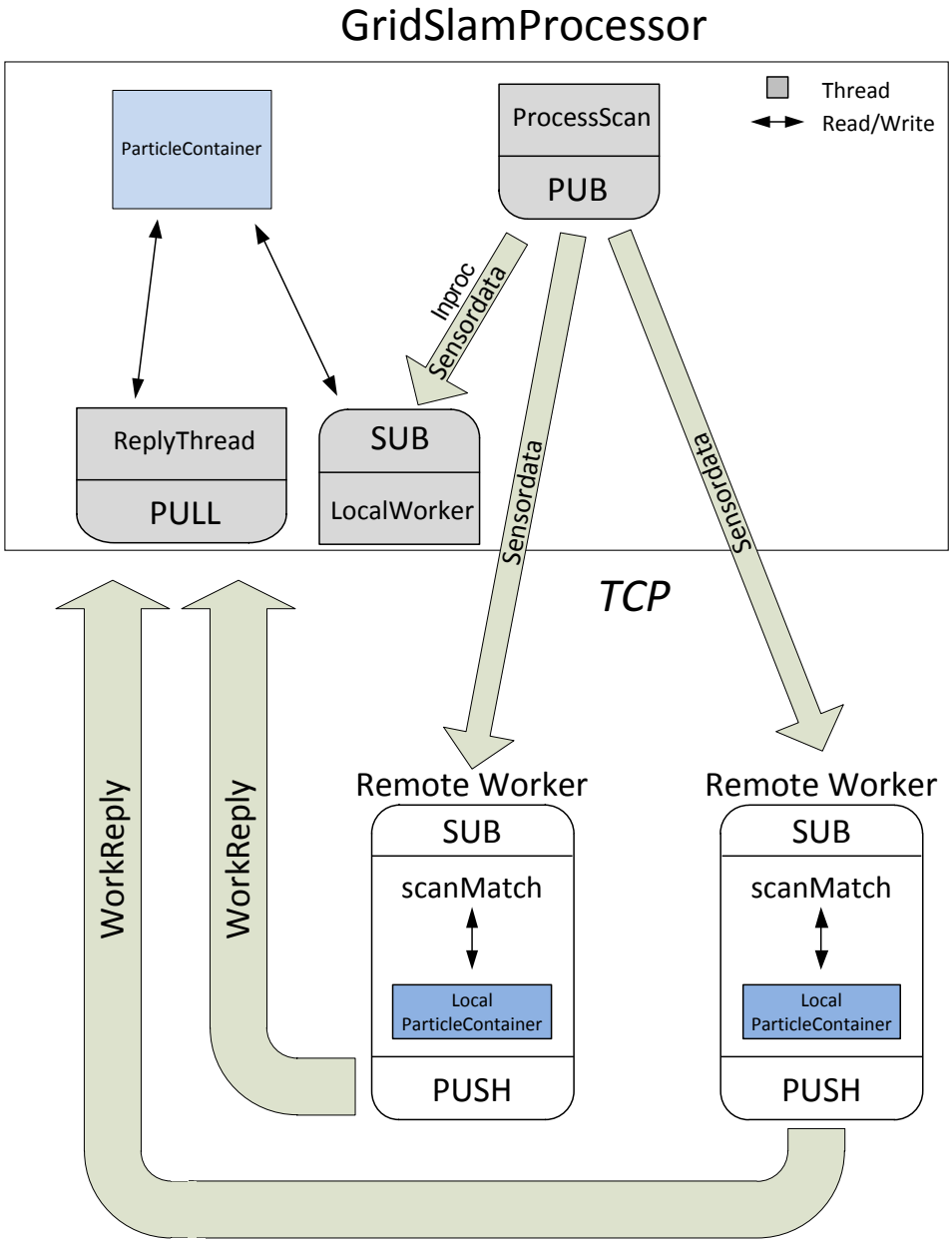

Figura 18: Arquitectura "stateful"

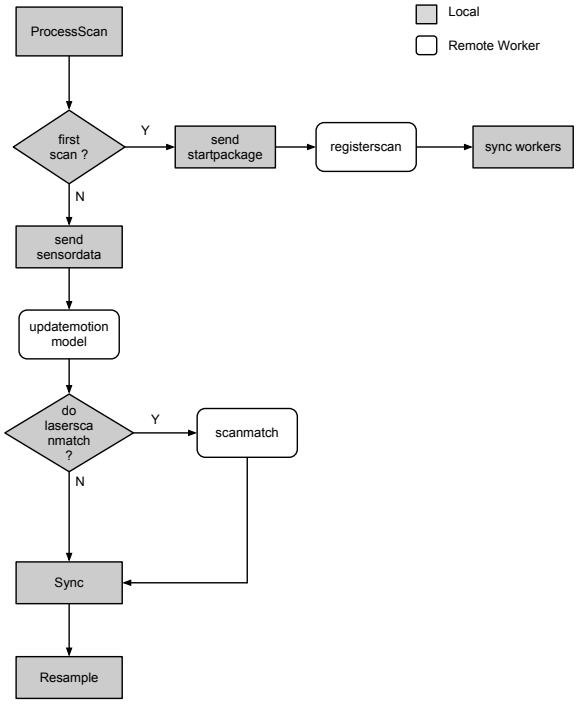

Figura 19: *Flowchart* da comunicação na função "processScan"

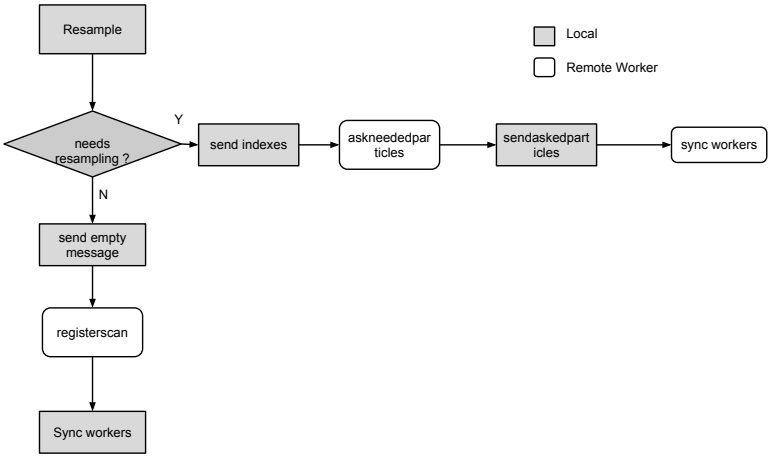

Figura 20: *Flowchart* da comunicação na função "resample"

4 Resultados

Para validar os resultados para além do *dataset* disponibilizado pela Willow Garage, que é compatível com o ROS, foram modificados os dois programas disponíveis no repositório do OpenSLAM [43], o "gfs_simplegui" e o "gfs_nogui". Permitem trabalhar com *datasets* CARMEN [26], entre os quais estão por exemplo os dados adquiridos no Killian Court do Massachusetts Institute of Technology(ver Figura 21).

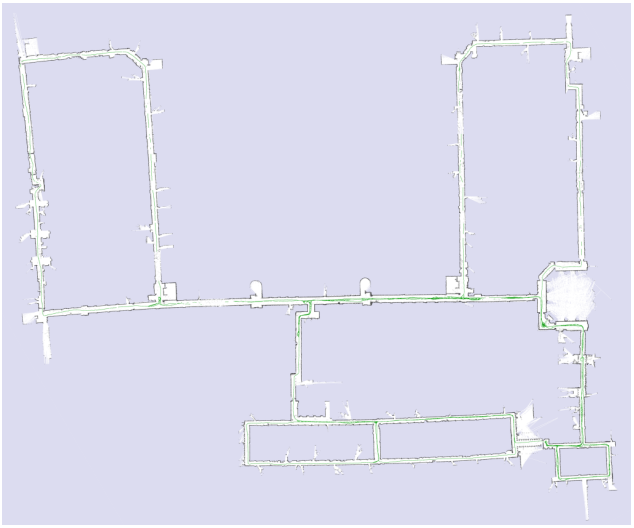

Figura 21: Mapa gerado com o *Dataset* Killian Court MIT

Todos os testes foram efectuados com os computadores descritos na Tabela 8, os resultados mostrados em diagramas são a média de 30 execuções. Sempre que possível usaram-se os parâmetros por defeito do algoritmo e não se aumentou o número de partículas, devido à memória disponível por omissão no computador de base (1GB). O Asus F8SN serviu sempre como computador de apoio ao Eee PC. Todos os algoritmos foram compilados com o *GNU Compiler Collection* (GCC) 4.7.3 usando os parâmetros de optimização "-O2 -ffast-math".

	Eee PC 901	**Asus F8SN**
CPU	Intel® Atom™ N270 com Hyper-Threading	Intel® Core™ 2 Duo T9300
Clock Speed	1.6GHz	2.5 GHz
Cache L2	512 KB	6 MB
RAM	1GB(DDR2)	3GB (DDR2)
SO	Ubuntu 12.04 32Bit	Ubuntu 12.04 64Bit
Rede	802.11g 54 Mbit/s	802.11g 54 Mbit/s
ROS	Fuerte	Fuerte

Tabela 8: Características dos computadores

[26] Disponíveis Online em http://kaspar.informatik.uni-freiburg.de/~slamEvaluation/datasets.php

4.1 Arquitectura *"stateless"*

Como se pode ver na Figura 22, esta abordagem obteve uma melhoria no tempo de execução sobre a versão local do algoritmo, chegando a um *Speedup* de 1,325 (ver Tabela 9 em Speedup Willow Garage) utilizando dois *Remote Workers* para além do *Local Worker*. Usando um único *Remote Worker*, obtendo um *Speedup* de 1,25 (poupando 30s no tempo de execução), já foi o suficiente para que o Eee PC consiga correr o algoritmo com velocidade suficiente para obter um mapa correcto. Na Figura 23 pode-se ver lado a lado os mapas que a versão distribuída e a versão local geram.

	Tempo Willow Garage	**Speedup Willow Garage**	**Tempo Intel Research Lab**	**Speedup Intel Research Lab**	**Tempo Killian Court**	**Speedup Killian Court**
Local	155,68 s	-	1513,34 s	-	2848,45 s	-
1 Remote Worker	125,32 s	1,242	1147,62 s	1,319	2715,35 s	1,049
2 Remote Workers	117,5 s	1,325	953 s	1,588	3508,8 s	0,812

Tabela 9: Tempos de execução e *Speedups* para vários *Datasets*

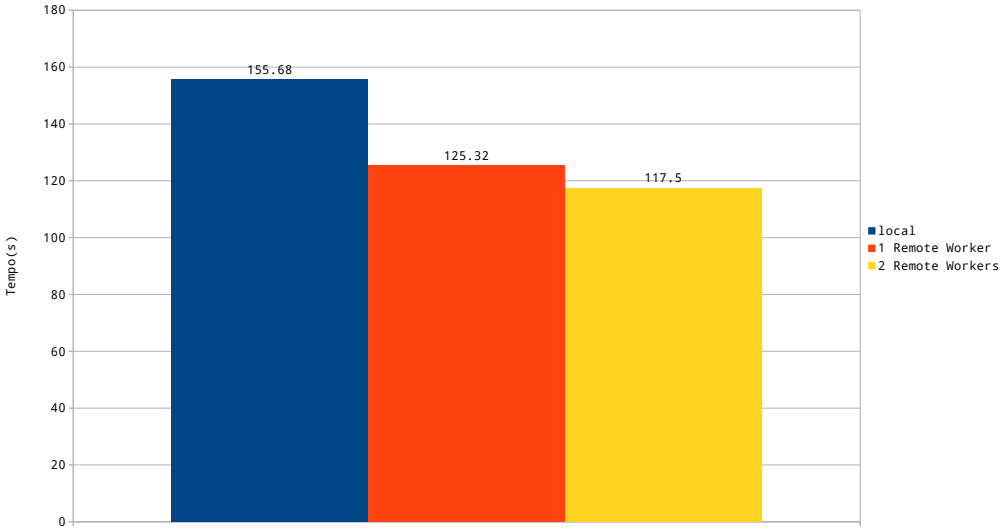

Figura 22: Tempos de execução usando o *Dataset* da Willow Garage

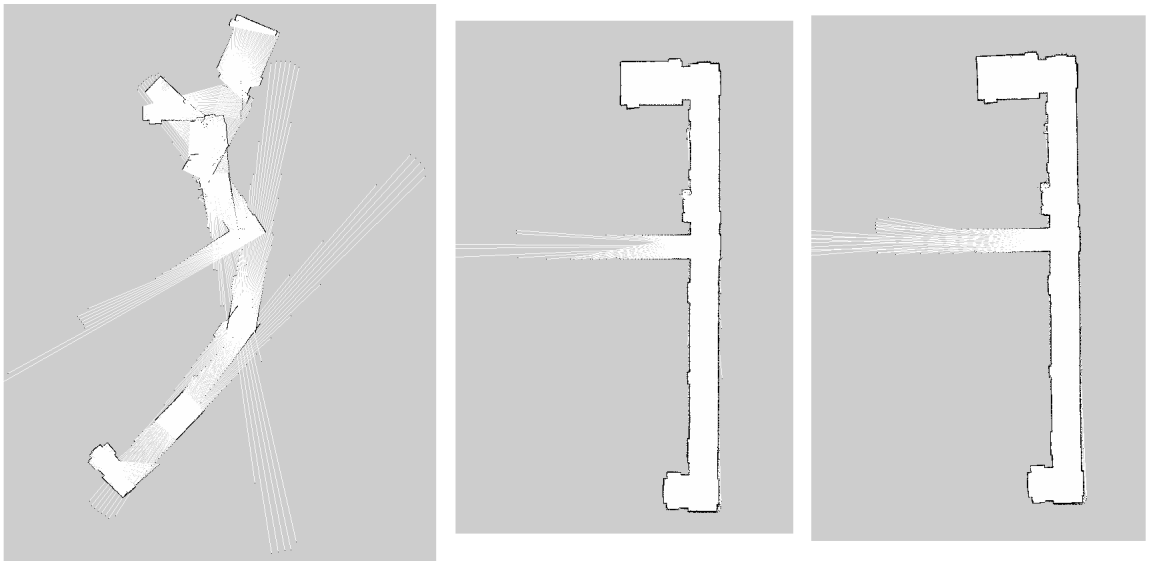

Figura 23: Mapas gerados com o GMapping: esquerda Eeepc com algoritmo original, meio Eeepc 1 *Local Worker* e 1 *Remote Worker*, direita algoritmo original no Asus F8SN

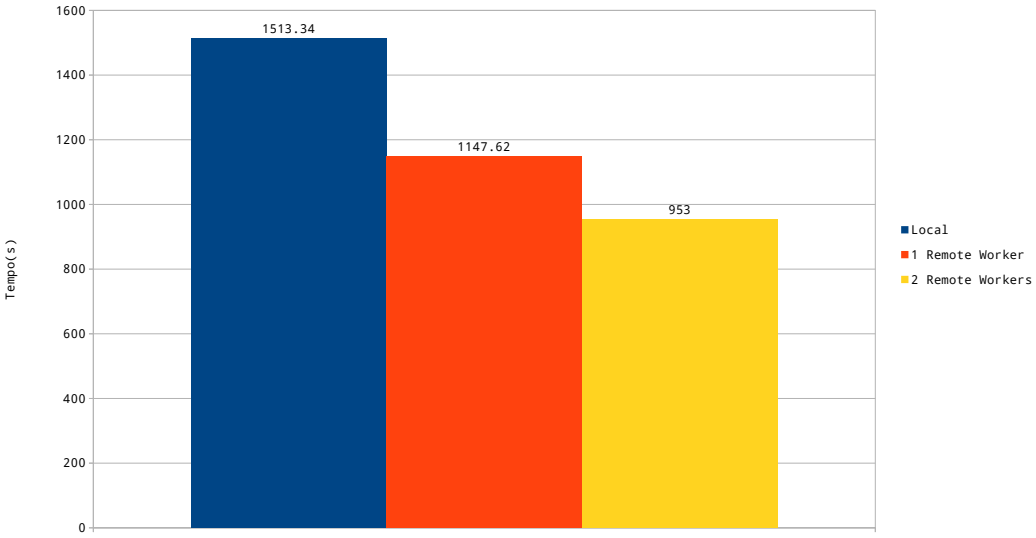

Figura 24: Tempos de execução usando o *Dataset* Intel Research Lab (Seattle)

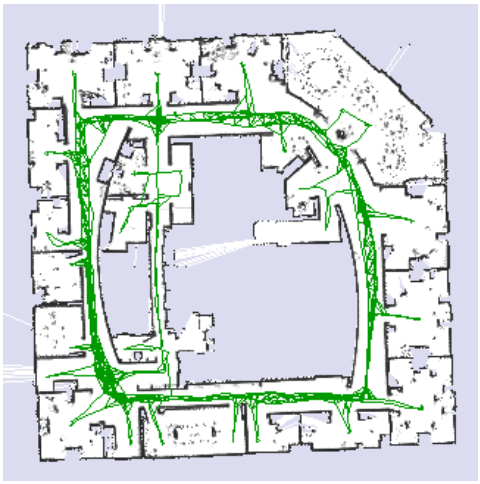

Figura 25: Mapa gerado com o *Dataset* Intel Research Lab (Seattle)

Na Figura 24 pode-se ver que mesmo para mapas do tamanho médio (ver Figura 25) se consegue obter um *Speedup* de 1,59 sobre a versão não distribuída. Contudo em mapas grandes, tal como o Killian Court do Massachusetts Institute of Technology (ver Figura 21), não se obteve uma melhoria significativa (*Speedup* de 1,05 com um '*Remote Worker*), chegando mesmo a ficar mais lento, demorando 23% mais tempo a correr na configuração com dois *Remote Workers*.

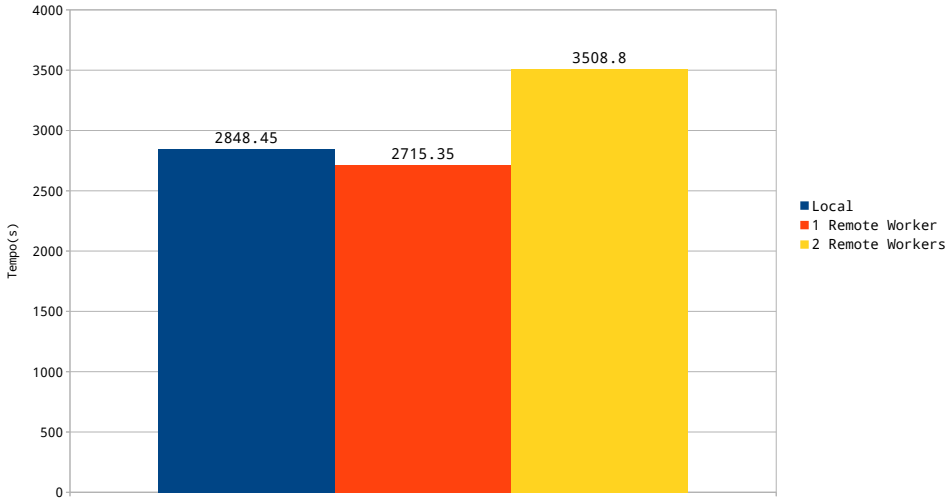

Figura 26: Tempos de execução usando o *Dataset* Killian Court

Analisando os diagramas da evolução do tamanho dos *WorkPackages* (ver Figura 27 e 28), comparativamente ao *Dataset* Intel Research Center, mesmo usando compressão e envio de mapas parciais, o algoritmo chega rapidamente a tamanhos elevados no *Dataset* Killian Court, chegando aos 138934 bytes por partícula nos primeiros 100 turnos.

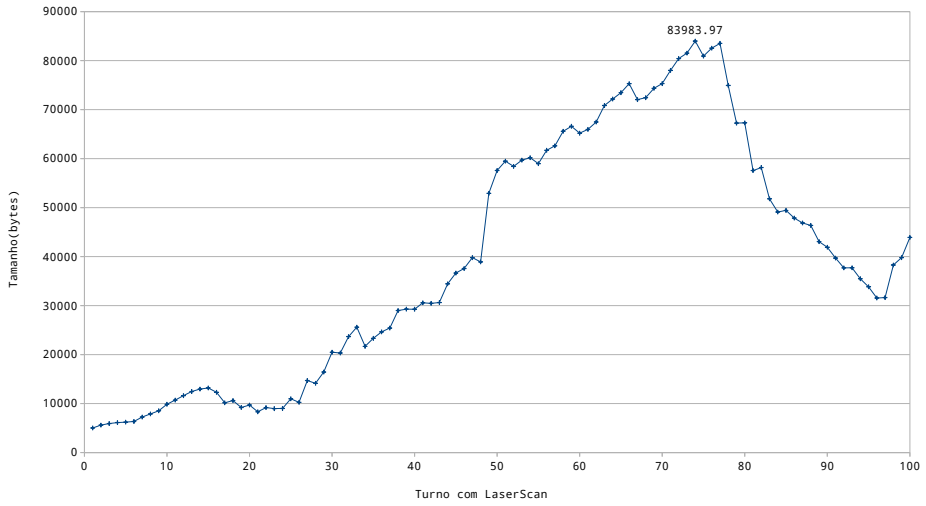

Figura 27: Evolução do tamanho dos *Workpackages* comprimidos para o *Dataset* Intel Research Lab (Seattle)

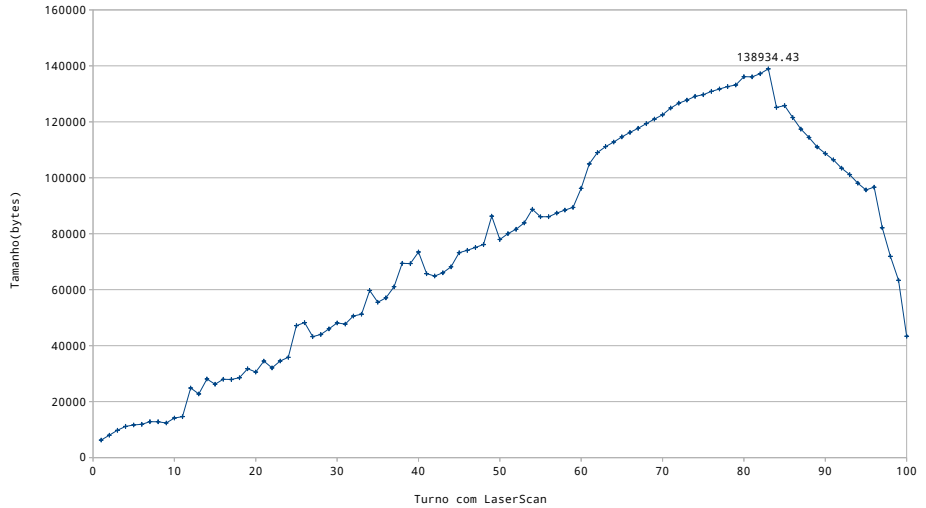

Figura 28: Evolução do tamanho dos *Workpackages* comprimidos para o *Dataset* Killian Court

4.2 Arquitectura *"stateful"*

Como se pode ver nas Figuras 29 e 30, a abordagem "Stateful" reduz consideravelmente o tamanho dos dados partilhados na rede, necessitando de enviar 1532 bytes em caso de turno com *LaserScan* ou menos de 100 bytes, caso seja um turno que só necessite da odometria. Em caso de *Resampling* poderá haver picos consoante a distribuição das partículas. No caso da partilha de 25% das partículas, houve

necessidade de maior troca de dados em 4 fases de *Resampling* (ver Figura 29). No caso em que se partilhou 50% das partículas (ver Figura 30), como não houve necessidade de trocar dados (o *Remote Worker* tinha todas as partículas "novas" localmente), os turnos com *Resampling* precisaram unicamente de trocar mensagens vazias.

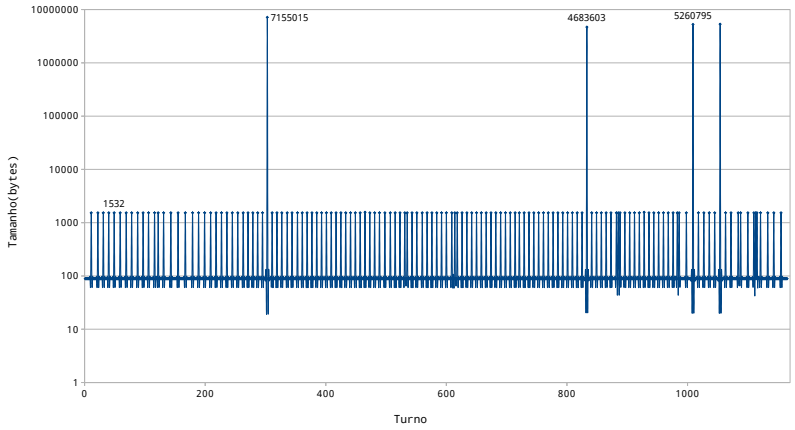

Figura 29: Evolução do tamanho dos dados recebidos pelo *Remote Worker* (25% das partículas)

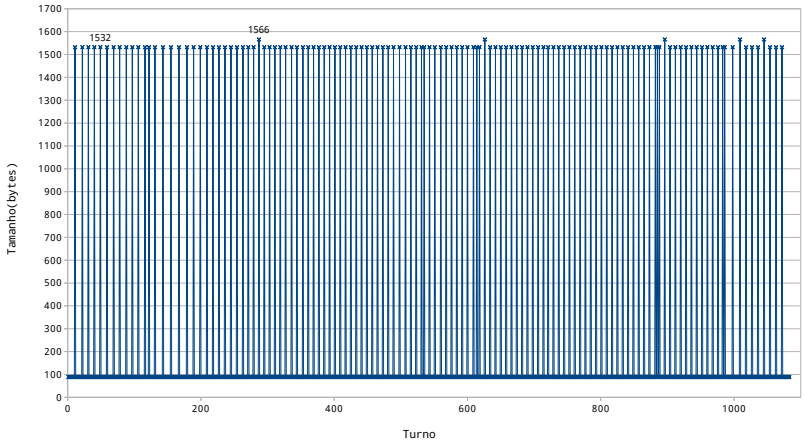

Figura 30: Evolução do tamanho dos dados recebidos pelo *Remote Worker* (50% das partículas)

Na Figura 31 podemos ver que a redução de troca de dados se traduziu por um ganho em velocidade no algoritmo, mesmo em mapas maiores tal como o do Killian Court. Podemos também perceber pelas Figuras 31 e 32, que esta abordagem, como não tem *Loadbalancing*, é muito dependente da distribuição das partículas, e que distribuindo o mesmo número de partículas por dois *Remote Workers* em vez de um, nem sempre traz vantagens devido à maior probabilidade de haver troca de dados em fase de *Resampling*.

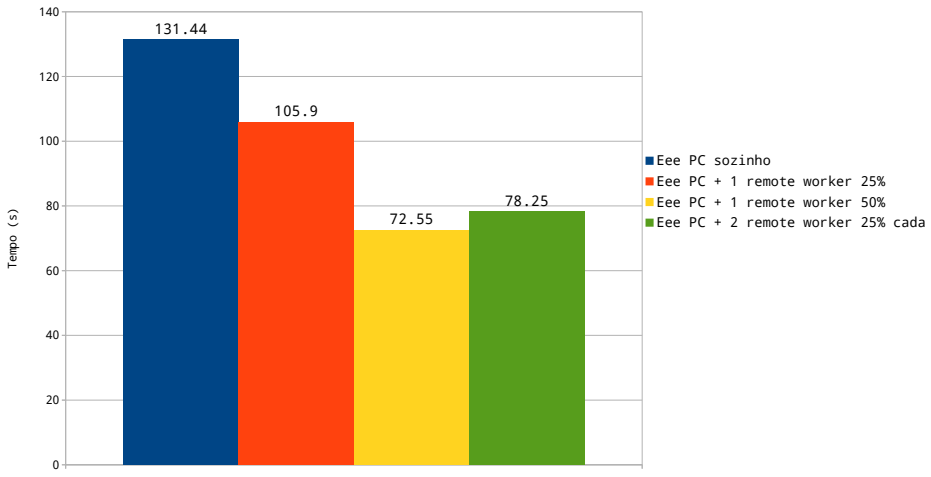

Figura 31: Tempo de execução para 100 turnos com *LaserScan* com o *dataset* Killian Court do MIT

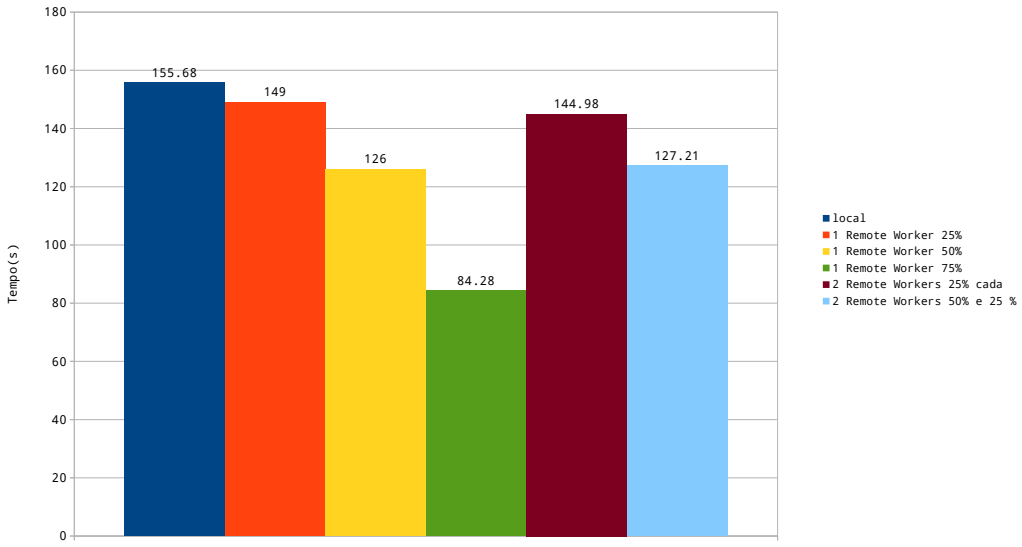

Figura 32: Tempo de execução para 100 turnos com *LaserScan* com o *dataset* da Willow Garage

5 Conclusões

Os objectivos do trabalho propostos no capítulo 1.1 foram concluídos com sucesso, faltando unicamente implementar fusão de mapas para completar o algoritmo, para o uso em ambientes multirobots que tentam explorar de maneira eficiente um ambiente desconhecido.

Implementaram-se duas abordagens distribuídas do algoritmo GMapping prontas a usar no *ROS*, uma delas implementa *Loadbalancing* e tenta ser o mais flexível possível para poder responder facilmente a falhas na rede e a mudanças na topologia do sistema, a outra tenta colmatar a maior falha da primeira que é não funcionar em mapas de elevado tamanho, perdendo para isso a flexibilidade e *Loadbalancing*.

A primeira abordagem (abordagem "stateless") obteve bons resultados para mapas pequenos e médios (*Speedups* de 1,325 e e 1,588 respectivamente), permitindo a um computador (Eee PC) com uma capacidade de processamento limitada (Intel® Atom™ N270) obter mapas correctos com o apoio de outro computador. Contudo, em mapas maiores não houve melhoria significativa, demorando mesmo mais tempo a correr o algoritmo distribuido (*Speedup* de 0,812)

A segunda abordagem (abordagem "stateful") alcançou bons resultados em mapas de tamanho elevado (*Speedup* de 1,812) e mostrou ser bastante dependente da distribuição das partículas usadas no algoritmo.

5.1 Trabalho Futuro

A implementação distribuída do algoritmo GMapping foi concluída com êxito, no entanto na versão existente não tem grande utilidade para aplicações multirobot em que é necessário obter um mapa global da maneira mais eficiente possível. Para esse fim poder-se-ia implementar um algoritmo de fusão de mapas parciais obtidos em conjunto com o GMapping, para tal poder-se-ia aproveitar o código já existente do registo dos dados do *Laser Ranger Finder* (método "registerScan" da classe "ScanMatcher") e usar um algoritmo de navegação por sectores.

A primeira abordagem introduzida no capitulo 3.1.1 poderia ser interessante num algoritmo com base num filtro de partículas com uma quantidade menor de dados, poder-se-ia por exemplo usar num algoritmo de FastSLAM topológico (Feature Based SLAM).

Outra aplicação possível seria em sistema de robots diversos, como por exemplo um conjunto de um robot aéreo (leve e fraco computacionalmente) acompanhado por um robot terrestre (computacionalmente mais veloz).

Referências

[1] The computer language benchmarks game. Disponível online em http://benchmarksgame.alioth.debian.org/u32/benchmark.php?test=all&lang=all&lang2=gpp&data=u32, Jan. 2013.

[2] java serialization benchmark. Disponível online em https://github.com/eishay/jvm-serializers/wiki, Jan. 2013.

[3] P2p-mpi. Disponível online em http://grid.u-strasbg.fr/p2pmpi/, Jan. 2013.

[4] Programming language benchmarks. Disponível online em http://attractivechaos.github.io/plb/, Jan. 2013.

[5] ARM. Arm cortex a8. Disponível online em http://www.arm.com/products/processors/cortex-a/cortex-a8.php, Nov. 2012.

[6] ARM. Neon. Disponível online em http://www.arm.com/products/processors/technologies/neon.php, Nov. 2012.

[7] R. Arumugam, V.R. Enti, Liu Bingbing, Wu Xiaojun, K. Baskaran, Foong Foo Kong, A.S. Kumar, Kang Dee Meng, and Goh Wai Kit. Davinci: A cloud computing framework for service robots. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3084 –3089, may 2010.

[8] B. Clipp, Jongwoo Lim, J.-M. Frahm, and M. Pollefeys. Parallel, real-time visual slam. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 3961 –3968, oct. 2010.

[9] Gregory Dudek and Michael Jenkin. *Computational principles of mobile robotics*. Cambridge University Press, New York, NY, USA, 2000.

[10] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *Robotics Automation Magazine, IEEE*, 13(2):99 –110, june 2006.

[11] O. El Hamzaoui and B. Steux. A fast scan matching for grid-based laser slam using streaming simd extensions. In *Control Automation Robotics Vision (ICARCV), 2010 11th International Conference on*, pages 1986 –1990, dec. 2010.

[12] Austin Eliazar and Ronald Parr. Dp-slam: Fast, robust simultaneous localization and mapping without predetermined landmarks. In *in Proc. 18th Int. Joint Conf. on Artificial Intelligence (IJCAI-03*, pages 1135–1142. Morgan Kaufmann, 2003.

[13] Engadget. Industrial robots. Disponível online em http://www.engadget.com/2008/06/04/seegrid-shows-off-autonomous-industrial-mobile-robot-system/, Nov. 2012.

[14] L.H. Erickson, J. Knuth, J.M. O'Kane, and S.M. LaValle. Probabilistic localization with a blind robot. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 1821 –1827, may 2008.

[15] G. Grisetti, C. Stachniss, and W. Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *Robotics, IEEE Transactions on*, 23(1):34 –46, feb. 2007.

[16] Hadoop. Benchmark comparativo do lz4 no bugtracker do hadoop. Disponível online em https://issues.apache.org/jira/browse/HADOOP-7657, Jan. 2013.

[17] Markus Hadwiger, Thomas Theußl, Helwig Hauser, and Eduard Gröller. Hardware-accelerated high-quality filtering on pc hardware. In *In Proc. of Vision, Modeling and Visualization 2001*, pages 105–112, 2001.

[18] Hardkernel. O-droid. Disponível online em http://www.hardkernel.com/renewal_2011/products/prdt_info.php, Nov. 2012.

[19] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[20] Hashcat. oclhashcat. Disponível online em http://hashcat.net/oclhashcat-plus/, Nov. 2012.

[21] Pieter Hintjens. *Code Connected Volume 1*. 2013.

[22] Robert Hundt. Loop recognition in c++/java/go/scala. In *Proceedings of Scala Days 2011*, 2011.

[23] Pablo Iñigo Blasco, Fernando Diaz-del Rio, Ma Carmen Romero-Ternero, Daniel Cagigas-Muñiz, and Saturnino Vicente-Diaz. Robotics software frameworks for multi-agent robotic systems development. *Robot. Auton. Syst.*, 60(6):803–821, June 2012.

[24] iRobot. Roomba. Disponível online em http://www.irobot.com/en/us/robots/home/roomba.aspx, Nov. 2012.

[25] J. Dias J. Quintas, P. Menezes. Cloud robotics: Towards context aware robotic networks. In *Proceedings of IASTED, The 16th IASTED International Conference on Robotics (Robo 2011)*, November 2011.

[26] Khronos. Opencl. Disponível online em http://www.khronos.org/opencl/, Nov. 2012.

[27] Xiuzhi Li, Songmin Jia, Wei Cui, Jinhui Fan, and Jinbo Sheng. Consistent map building by a mobile robot euipped with stereo sensor and lrf. In *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, volume 3, pages 100 –104, june 2011.

[28] Xiuzhi Li, Songmin Jia, Wei Cui, and Xiaolin Yin. Dslam: Towards a real-time robot mapping approach. In *SICE Annual Conference (SICE), 2012 Proceedings of*, pages 1859 –1863, aug. 2012.

[29] LZ4. Lz4. Disponível online em https://code.google.com/p/lz4/, Jan. 2013.

[30] P. Mandal, R. Kumar Barai, and M. Maitra. Collaborative coverage using swarm networked robot. In *Advances in Engineering, Science and Management (ICAESM), 2012 International Conference on*, pages 301 –304, march 2012.

[31] Ali Marjovi, Sarvenaz Choobdar, and Lino Marques. Robotic clusters: Multi-robot systems as computer clusters: A topological map merging demonstration. *Robotics and Autonomous Systems*, 60(9):1191–1204, 2012.

[32] Ali Marjovi and Lino Marques. Multi-robot olfactory search in structured environments. *Robot. Auton. Syst.*, 59(11):867–881, November 2011.

[33] Ali Marjovi, João Gonçalo Nunes, Lino Marques, and Aníbal de Almeida. Multi-robot exploration and fire searching. In *Proceedings of the 2009 IEEE/RSJ international conference on Intelligent robots and systems*, IROS'09, pages 1929–1934, Piscataway, NJ, USA, 2009. IEEE Press.

[34] IEEE Md. Suruz Miah, Student Member and IEEE Miodrag Bolic, Member. Parallel implementation of modified rao-blackwellised particle filter. Technical report, Computer Architecture Research Group, University of Ottawa.

[35] Microsoft. Kinect. Disponível online em http://www.microsoft.com/en-us/kinectforwindows/, Nov. 2012.

[36] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, Edmonton, Canada, 2002. AAAI.

[37] MPI. Mpi 3.0. Disponível online em http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, Jan. 2013.

[38] Open MPI. Open mpi. Disponível online em http://www.open-mpi.org, Jan. 2013.

[39] MPICH. Mpich. Disponível online em http://www.mpich.org, Jan. 2013.

[40] MPICH-V. Mpich-v. Disponível online em http://mpich-v.lri.fr/, Jan. 2013.

[41] NASA. Mars science laboratory. Disponível online em http://mars.jpl.nasa.gov/msl/, Dec. 2012.

[42] Nvidia. Cuda. Disponível online em http://www.nvidia.com/object/cuda_home_new.html, Nov. 2012.

[43] OpenSLAM. Openslam. Disponível online em http://openslam.org/gmapping.html, Nov. 2012.

[44] Deyuan Qiu, Stefan May, and Andreas Nüchter. Gpu-accelerated nearest neighbor search for 3d registration. In *Proceedings of the 7th International Conference on Computer Vision Systems: Computer Vision Systems*, ICVS '09, pages 194–203, Berlin, Heidelberg, 2009. Springer-Verlag.

[45] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[46] raspberrypi. raspberrypi. Disponível online em http://www.raspberrypi.org/, Nov. 2012.

[47] ROS. Gmapping. Disponível online em http://www.ros.org/wiki/gmapping, Nov. 2012.

[48] ROS. Ros introduction. Disponível online em http://www.ros.org/wiki/ROS/Introduction, Jan. 2013.

[49] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[50] T. Tammet, J. Vain, A. Puusepp, E. Reilent, and A. Kuusik. Rfid-based communications for a self-organising robot swarm. In *Self-Adaptive and Self-Organizing Systems, 2008. SASO '08. Second IEEE International Conference on*, pages 45 –54, oct. 2008.

[51] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[52] B. Tribelhorn and Z. Dodds. Evaluating the roomba: A low-cost, ubiquitous platform for robotics research and education. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 1393 –1399, april 2007.

[53] Regis Vincent, Dieter Fox, Jonathan Ko, Kurt Konolige, Benson Limketkai, Benoit Morisset, Charles Ortiz, Dirk Schulz, and Benjamin Stewart. Distributed multirobot exploration, mapping, and task allocation. *Annals of Mathematics and Artificial Intelligence*, 52(2-4):229–255, April 2008.

[54] C. Zach, A. Klaus, B. Reitinger, K. Karner, et al. Optimized stereo reconstruction using 3d graphics hardware. In *Workshop of Vision, Modelling, and Visualization (VMV 2003)*, pages 119–126, 2003.